Lazy State Determination:
More concurrency for contending linearizable transactions

Tiago M. Vale[1], João Leitão[1], Nuno Preguiça[1], Rodrigo Rodrigues[2], Ricardo J. Dias[1], and João M. Lourenço[1]

[1]NOVA LINCS, DI, FCT, Univeridade NOVA de Lisboa
[2]INESC-ID, IST, Universidade de Lisboa

Abstract

The concurrency control algorithms in transactional systems limits concurrency to provide strong semantics, which leads to poor performance under high contention. As a consequence, many transactional systems eschew strong semantics to achieve acceptable performance. We show that by leveraging semantic information associated with the transactional programs to increase concurrency, it is possible to significantly improve performance while maintaining linearizability. To this end, we introduce the lazy state determination API to easily expose the semantics of application transactions to the database, and propose new optimistic and pessimistic concurrency control algorithms that leverage this information to safely increase concurrency in the presence of contention. Our evaluation shows that our approach can achieve up to $5\times$ more throughput with $1.5\times$ less latency than standard techniques in the popular TPC-C benchmark.

1 Introduction

Linearizable transactions provide a simple and powerful abstraction to programmers: transactions appear to complete atomically, one at a time despite executing concurrently. This property greatly simplifies developing and reasoning about concurrent applications.

In recent years, we saw a continuing interest in research on transactional systems, e.g., as transactional properties were adopted in "NoSQL" systems [11], or as the performance of distributed transactions was improved through new hardware features [13]. However, this research does not fundamentally improve the performance of transactions in the presence of their Achilles heel: contention [28]. When transactions conflict with one another, they end up executing most of their logic one at a time. To circumvent this limitation, transactional systems may resort to weaker semantics, but these often break the integrity of the applications that are built using such transactions [3].

In this paper, we show that it is possible to achieve linearizable transactions with significantly better performance than existing techniques by building on the observation that the lack of semantic information about the transaction leads to a conservative view of what is a conflict, and therefore imposes unnecessary synchronization between transactions. For example, two transactions that increase the number of items in an inventory will be treated as conflicting because they both write to the database tuple containing the total quantity. However, the semantics of those transactions do not imply a conflict, provided that the aggregated effects of both transactions are applied to the database.

To address this shortcoming, we propose lazy state determination (LSD), a novel API for defining transactions that conveys their semantics to the database. The main insight behind LSD is that by exploring the semantics of the transaction, it is possible to increase concurrency while still providing linearizability [17] (also known as strict serializability [24]). This contrasts with previous work that explores semantic infor-

1

mation to improve transaction processing [23, 26, 5], which focuses on maintaining specific application invariants under consistency models weaker than serializability.

One important challenge in our work is how to expose the semantic information without requiring programmers to significantly modify their coding practices, or make significant changes to existing applications. To this end, we realize LSD by having the READ operation return a future [4] (an opaque proxy for a value) instead of a concrete value, and materializing futures as late as possible, i.e., only when the transaction commits. To allow transactions to still be expressive with futures without resolving them, we: (a) introduce a new operation, IS-TRUE, that allows transactions to specify conditions over futures, and (b) provide operations that allow transactions to specify their updates to the database as lazily-evaluated functions that can use futures. This differs from prior works [20, 10] that need to materialize *delayed* reads as soon as they are used by the transaction logic.

This novel API allows LSD transactions to execute over an abstract database state, and resolve this abstract state as late as possible, thus increasing the chances for safely committing without breaking isolation. To demonstrate this, we modified existing optimistic and pessimistic concurrency control protocols to allow for conditional validation. The key idea of this design is to verify that the required conditions still hold when the transaction attempts to commit (in the case of optimistic concurrency control), or to use a condition lock acquired in condition mode for a certain condition c, which is only compatible with an acquisition in write mode if the value that will be written respects the condition c (in the case of pessimistic concurrency control).

We implemented and evaluated a prototype transactional key-value store that provides linearizable transactions using the LSD interface. Our evaluation shows that LSD transactions achieve up to $5\times$ more throughput with $1.5\times$ less latency than standard transactions under high contention in our experiments with the popular TPC-C benchmark [30].

In summary, we make the following contributions:

LSD API. We propose LSD, an interface to express transactions that allows the database to collect semantic information useful to achieve higher performance under contention.

Concurrency control. We propose new optimistic and pessimistic concurrency control algorithms for providing linearizability while exploring semantic information to increase

```
BEGIN
v ← READ(stock)
if v ≥ qty then
  v ← v − qty
  WRITE(stock, v)
  COMMIT
else
  ABORT
```

```
BEGIN
□ ← READ(stock)
if IS-TRUE({□ ≥ qty})
  then
    △ ← {□ − qty}
    WRITE(stock, △)
    COMMIT
else
  ABORT
```

(a) Traditional interface.   (b) LSD interface.

Figure 1: Simplified portion of TPC-C's New Order-like transaction.

concurrency in the presence of contention, using novel *condition validation* and *condition locking* techniques.

Evaluation. We implemented an LSD-compatible prototype and evaluated it using TPC-C and microbenchmarks, showing significant performance improvements.

The remainder of this paper is organized as follows. We present an overview of LSD in §2, by motivating the shortcomings of the standard interface through an example, from which we derive the LSD interface. §3 then presents LSD in detail, including the design of LSD-aware variants of OCC and 2PL. §4 describes our prototype and the results of our evaluation. §5 discusses LSD in the context of related work and §6 concludes the paper.

2 Overview

A typical database API exposes five operations: (1) BEGIN: starts a new transaction, (2) READ(*key*): returns the value of the database object identified by *key*, (3) WRITE(*key*, *val*): modifies the value of the object identified by *key* to *val*, (4) COMMIT: commit the current transaction, and (5) ABORT: aborts the current transaction.

Conceptually, a transaction is a function f that changes the database from an initial state s_i to a final state s_f, i.e. $f(s_i) = s_f$. In light of this formulation, the READ and WRITE API calls allow transactions to specify the final state (WRITE) as a function of the initial state (READ).

The pitfalls of the traditional API. Consider the example in Figure 1a that depicts a simplified portion of the TPC-C new order transaction [30], which implements the action of buying a certain quantity *qty* of items. If the item's stock $\left(v \leftarrow \text{READ}(stock)\right)$ is enough to fulfil the order $\left(v \geq qty\right)$, the stock value decreases by *qty* $\left(\text{WRITE}(stock, v - qty)\right)$.

This example illustrates how the READ/WRITE interface fails to convey the semantics of the transaction to the database, e.g., the dependencies of the transaction behavior on the values it reads, or how it computes the values it writes. From the point of view of the database, transactions are a sequence of opaque READ/WRITE operations. (This is true regardless of whether transactions execute co-located with the database, as stored procedures, or in a remote client.)

To understand how this can be a limiting factor, consider the situation where the current stock value is 42 (*stock*), and the quantity to order is 1 (*qty*). When the transaction issues the READ(*stock*) operation, the database returns the value 42. Since the database does not know what the transaction will do with the returned value, it must be conservative to account for all possible situations, e.g., the transaction only executing some operations depending on the returned value, or using the returned value to perform a computation that returns the value of a subsequent WRITE. As a consequence, 2PL must lock the *stock* object to prevent any other transaction from modifying it and invalidate any branching decision or computed value by the transaction that observed the value 42. Similarly, OCC records the read operation so that the database can check that the *stock*'s value is the same when the transaction attempts to commit; if meanwhile another transaction modifies the *stock*, transactions that observed the now-stale stock value fail to commit.

As this example shows, a central part of enforcing transaction isolation is ensuring that the state that a transaction observes (i.e., the values returned by READ operations) remains unchanged throughout its execution. Our key insight is to question whether a transaction *really* needs to observe a specific state during its execution. In other words, in our running example, does the READ(*stock*) operation really need to expose a particular state to the transaction before commit (e.g., the value 42)? With the current interface the answer is yes. Otherwise, transactions cannot have conditional branches that depend on the database state, nor perform updates to the state that are a function of that state. Going back to our example, the transaction could not check whether there is enough stock nor compute the new stock value.

Introducing LSD. In this paper, we overcome these limitations by rethinking the transactional API in order to provide linearizable transactions that allow for greater concurrency. The key observation behind LSD is that, in general, transactions do *not* need to observe a concrete state to execute most of their logic. Thus, we propose alternative semantics for the READ operation. Specifically, the READ operation should *not* expose a specific database state by returning a concrete value, but should instead return a *future* [4].

A future is an object that acts as a proxy for a value that is initially unknown. In our case, a future symbolizes the value of a specific database object. This means that the database *promises* to resolve the future's value, but does not do it right away. In particular, we want to defer evaluating futures until the transaction attempts to commit (*lazy evaluation* [18]) to maximize concurrency. (Note that the traditional semantics of the READ operation is equivalent to returning futures that are immediately resolved.) Returning to our running example, we depict this modification in Figure 1b with the future that symbolizes the stock value as □.

The proposed change to the semantics of the READ operation has a clear benefit: if a transaction does not observe a specific state, other transactions can modify it without breaking the isolation guarantees of the first transaction. However, this raises the problem of determining how can a transaction use futures. This can, in turn, be split into two main challenges. The first is how can a transaction perform conditional branching based on futures. The second is how can a transaction compute values that depend on futures. For instance, how can the logic of our example transaction decide whether it can fulfil the order if it does not know the stock value, and how can the transaction compute the new stock value? (A naive approach is to eagerly resolve futures when a transaction requires their value, but this restricts concurrency.)

To solve the first challenge, we observe that a future symbolizes the value of a particular database object. While we would like that a transaction is not able to directly observe the value of a future, we can still ask the database whether a future's value respects a certain condition. For example, the transaction can ask the database whether the stock value is greater than *qty*, and make a control flow decision depending on the database's answer.

To support this functionality we introduce a new operation, IS-TRUE(c), which, given a condition c over one (or more) futures, returns whether the condition holds or not. We show the IS-TRUE operation using the $\square \geq qty$ condition in

Figure 1b. Note that while the IS-TRUE operation effectively exposes database state to the transaction, it exposes an abstract state (the stock is greater than *qty*) rather than a concrete one (the stock is 42), which has the potential to allow for more concurrency, e.g., by allowing concurrent modifications of the stock value as long as it retains a non-negative value after all the modifications.

The second challenge is how can a transaction perform computations using futures. To solve this challenge, we observe that while a transaction cannot perform the actual computation with futures, it can define the necessary computation and let the database perform it when the transaction commits and the futures are resolved to concrete values. For example, the transaction can define that the new stock value is whatever value its future ends up resolving to minus *qty*.

To support this behavior we change the semantics of the WRITE operation so that, instead of receiving the concrete new value for an object, it receives a function that computes the concrete value when evaluated. This function has the important property that it can depend on the values of any future, since the database can resolve them. Furthermore, WRITE functions are lazily evaluated by the database when the transaction commits, so that the futures that the functions depend on may remain unresolved. In Figure 1b, we represent this function as $\{\Box - qty\}$, which is the argument of the WRITE operation.

We expect that the proposed changes to the READ and WRITE operations and the addition of the IS-TRUE operation will enable the database to provide linearizable transactions with more concurrency, potentially resulting in higher throughput and lower latency.

On the one hand we decrease the time window in which a transaction requires isolation. With the traditional interface, the transaction requires isolation from the moment when it first observes database state (with the traditional READ operation) until the transaction attempts to commit. With LSD the transaction only requires isolation during its commit operation if it does not require any specific conditions.

On the other is that we reduce the set of concurrent transactions that are forced to abort/wait when executing concurrently with some transaction to guarantee the required isolation level. Even when a transaction needs to test some condition over database objects, LSD's IS-TRUE operation still allows concurrent transactions to modify those objects as long as these modifications do not invalidate the previously asserted conditions. This contrasts with the traditional interface
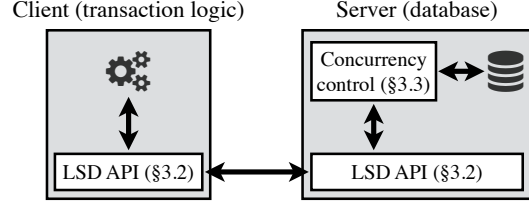

Figure 2: Overview of the system's architecture.

that prevents any modifications, whether they violate such conditions or not. This leads to lower abort rates/waiting, and hence to a higher amount of useful work.

That said, the LSD API is not a panacea. Transactions that must observe a concrete state can not reap LSD's benefits. For example, transactions that externalize values during their execution need to resolve the required futures, falling back to the standard READ semantics. However, we believe that a large class of transactions can take advantage of LSD proposed semantics.

3 LSD Design

The high-level goal of LSD is to allow databases to provide linearizable transactions with higher performance than what can typically be achieved, while minimizing changes in the way programmers specify the logic of their transactions.

3.1 Design overview

Figure 2 shows the main components of our design. Clients execute application code that interacts with the database server via transactions written using the LSD API. Note that these are logical components, meaning that our design does not make assumptions regarding the physical relationship between clients and servers, nor the physical realization of the server. For example, clients can be physically separated from the server or co-located with it (e.g., in a stored procedure), and the database may or may not be partitioned or replicated. Nevertheless, for the rest of this paper we assume that clients execute transactions and are separated from the server, which is the case in our prototype and evaluation.

4

OPERATION	DESCRIPTION
BEGIN	Starts a new transaction.
READ(key) $\rightarrow \square$	Returns $\square$, a future for the value of object key.
READ($\triangle$) $\rightarrow \square$	Evaluates the future $\triangle$ and returns $\square$, a future for the value of the object that $\triangle$ evaluates to.
IS-TRUE($\square$) $\rightarrow$ boolean	Returns whether the condition $\square$ is currently true in the database.
WRITE(key, $\square$)	Updates object key's value to the value that $\square$ will evaluate to.
WRITE($\triangle$, $\square$)	Updates the value of the object that $\triangle$ will evaluate to, to the value that $\square$ will evaluate to.
COMMIT $\rightarrow$ boolean	Attempts to commit the ongoing transaction.
ABORT	Aborts the ongoing transaction.

Figure 3: The LSD interface operations. The symbols $\square$ and $\triangle$ denote futures.

3.2 Interface

Figure 3 shows the LSD interface, which allows applications to execute transactions against the database. The BEGIN, COMMIT, and ABORT operations are the standard operations. They allow an application to start, commit, and abort a transaction, respectively. LSD introduces two changes to the standard interface: new semantics for the READ and WRITE operations, and a new IS-TRUE operation. We first describe the new READ and WRITE operation semantics, then present the IS-TRUE operation, and finally address the case when a transaction wants to access an unknown object, i.e., the object identifier is itself a future.

READ. The typical semantics of the READ operation is to return the current value of the object, which requires the concurrency control protocol to kick in as a result of exposing the database state to transactions. In contrast, LSD's READ operation returns a future for the value of a given object, instead of exposing the object's current concrete value. From the application's point of view, this future is an opaque representation of the object's value. However, the database knows how to interpret such future; in particular, it has the possibility to *resolve* the future, i.e., compute the value that the future represents, which is to *actually* read and return the object's value. Thus, informally the contract that LSD provides between the transaction and the database is the following: the transaction should use the future as if it is the actual value, and the database promises to lazily resolve the future such that, when the transaction commits, it is as if it executed with the concrete value instead of the future. The benefit of these semantics is that the concurrency control protocol only needs to intervene when the database resolves a future and *not* when a transaction issues a READ operation.

WRITE. The traditional WRITE operation receives both the identifier of the object and its new value. This interface fits well with the traditional READ operation since reads return concrete values, so if a transaction wants to modify the value of an object it can read the object, compute the modified value, and write this new value. However, since the LSD READ operation returns a future instead of a concrete value, the transaction should be able to modify and write values derived from futures, instead of concrete values. To address this, we have two choices. The first is to resolve the future so that the transaction can perform its modification. This approach goes against LSD's goal, since resolving futures exposes database state to transactions, which in turn requires the concurrency control algorithm to enforce the required isolation. The second choice, which we follow, is *defining but not performing* the computation necessary to modify the value, so that futures may remain unresolved to promote parallelism. To do so, the transaction specifies the computation it needs to do as a function that, when evaluated by the database, computes the new value for the object. For instance in our running example of Figure 1b, where a transaction wants to decrease the avaliable stock for a given item, the transaction reads the stock and obtains $\square$ (its future value), and defines the function that decreases the stock ($\square - qty$). This function is also a future: it represents the value that the transaction intends to write to the stock object. For this approach to work, the database needs to know how to evaluate such functions so that, when the transaction commits, the database can install the object's new value. To understand how this can be done, we observe that we can divide this function evaluation into two parts: resolving future reads on which the function depends on, and executing the function's logic. As discussed, the database knows how to resolve future reads. As for executing the function's logic, the idea is that we define this in a way that the database can

initially refer to the function without resolving it, but at commit time interpret and execute it. To achieve this, in our prototype, we provide transactions with a library of operations, which can be composed to create functions, e.g., SUB($\square$, qty) to decrease the stock in our example.

IS-TRUE. So far, we managed to prevent exposing database state to transactions by changing the semantics of the READ and WRITE operations. However, transactions may need to decide what to do based on the database state, as exemplified in our running example where the transaction only orders the item if there is enough stock available. As before, we want to avoid resolving the futures required to make the decision of what to do, we introduce the IS-TRUE operation, which, given a condition over the database state, returns whether the condition holds or not. This condition is a function that, as discussed for the WRITE operation, can depend on futures. In our running example, the transaction decides to decrease the stock depending on whether there is enough stock available: $\square \geq qty$. (In our prototype, we also provide transactions with operations to create conditions, e.g., GTE($\square$, qty), to check whether there is enough stock.)

Note that the IS-TRUE operation *does* expose database state to transactions, but this is inevitable if the transaction performs different actions depending on the database state. The merit of the IS-TRUE operation is that it exposes *abstract*, instead of *concrete*, state to transactions, which enables the database to maintain isolation while potentially allowing more parallelism. For instance, if the transaction of our running example attempts to purchase a quantity of 4, and the current stock is 42, the IS-TRUE operation returns $\top$. Other concurrent transactions may successfully update the stock value and commit without breaking isolation as long as the stock value remains greater or equal to 4. Enforcing the semantics of this operation requires the concurrency control protocol to either: (1) ensure that the result of the IS-TRUE operation remains valid until the transaction commits (pessimistic approach, 2PL-style), or (2) abort the transaction when it attempts to commit if the result of the condition no longer holds (optimistic approach, OCC-style). In the next section we discuss how to adapt both 2PL and OCC to support for the IS-TRUE operation. We implemented both approaches in our prototype.

Futures as keys. Up until this point, we have not discussed what happens when the transaction attempts to read or write an object whose identifier is itself a future. For reads, this situation is likely to happen when accessing objects via a secondary index. Secondary indexes are seldom kept on keys whose values are updated frequently since they tend to be expensive to modify [29]. Given this observation, we chose to resolve the future immediately when a READ operation receives a future as a parameter, in order to know which object is being read. This simplifies reasoning and implementation effort, since the alternative of maintaining "futures of futures" would require a chain of resolves at commit time. As for future identifiers in WRITE operations, we chose to keep them unresolved because transactions may write to objects whose future-keys depend on the database state. This is the case, for example, when assigning unique identifiers to keys from a monotonically increasing counter, which we believe to be a common programming idiom.[1] As such, if we resolve the future identifier immediately, we risk exposing highly-contended database state to transactions, which goes against our design goals. The price we pay for our decision is that, in the general case of distributed transactions, they may require an additional communication round with servers to commit. We discuss this aspect further in §3.4.

3.3 Concurrency control

Now we turn our attention to the impact the LSD API has on concurrency control, and discuss how to adapt two popular concurrency control protocols: OCC [19, 31] and 2PL [14, 7]. The two main elements of the LSD API that drive the adaption are: (1) futures, as the protocol needs to be aware of them to know what to do at commit time, and (2) the IS-TRUE operation, which exposes abstract database state to transactions and therefore requires concurrency control.

3.3.1 Overview

The high level idea of the adaptation of both OCC and 2PL is to maintain two extra read and write sets, which we call *future read and write sets*, to keep futures unresolved until commit time, and a *condition set* to support the IS-TRUE operation and conditions. The LSD-aware OCC and 2PL protocols differ mainly on how they handle the condition set. OCC verifies that the conditions still hold at commit time while 2PL en-

---

[1]This is the case in the popular TPC-C benchmark [30]. The fact that purchase orders have monotonically increasing identifiers not only guarantees uniqueness, but also serves to identify and compare the recency of each order.

sures that concurrent transactions that write values that invalidate active conditions cannot commit while such conditions are active.

Figures 4 and 6 show the LSD-aware OCC and 2PL protocols, respectively. The behavior of the BEGIN, READ(*key*), and WRITE operations is protocol-agnostic so we start by describing these before detailing the protocols for OCC (§3.3.2) and 2PL (§3.3.3).

BEGIN. Initializes the read/write sets, future read/write set, and condition set (*rset*, *wset*, *frset*, *fwset*, and *cset*, respectively.)

READ(*key*). Creates a future-value for *key*'s value ($\square$), add it to the future read set, and returns it. (This is a local operation).

WRITE(*key*,$\square$). Buffers $\square$, the future-value for *key*, in the write set.

WRITE($\triangle$,$\square$). Buffers $\square$, the future-value to assign the future-key $\triangle$, in the future write set.

3.3.2 Optimistic concurrency control (OCC)

In a nutshell, OCC works as follows. Each database object is associated with a version. Reads record the object identity and the observed version in the *read set*. Writes are buffered in the *write set* until the transaction attempts to commit, instead of modifying the database immediately. Then, when a transaction attempts to commit, it atomically verifies if every object in the read set is unchanged, i.e., if it is still in the same version that was read, and, if so, all buffered updates are applied, and the respective version numbers are incremented. This atomic test and change is implemented in three steps: (1) lock the write set, (2) validate the read set, and (3) perform the pending writes, if the validation was successful, and release the acquired locks.

Next, we describe the adaptations required for the remaining operations, as depicted in Figure 4.

READ($\triangle$). Resolves the future-key $\triangle$, i.e., compute its concrete value *value* and add the observed version to the read set, and then READ(*value*). (Returning a future.)

upon READ($\triangle$)
  key $\leftarrow$ KEY($\triangle$)
  $\langle value, version \rangle \leftarrow$ GET(*key*)
  rset $\leftarrow$ *rset* $\cup$ {$\langle key, version \rangle$}
  return READ(*value*)

upon IS-TRUE($\square$)
  rvalues $\leftarrow \emptyset$
  foreach *key* $\in$ KEYS($\square$) **do**
    rvalues $\leftarrow$ *rvalues* $\cup$ {$\langle key,$ GET(*key*)$\rangle$}
  result $\leftarrow$ RESOLVE($\square$, *rvalues*)
  cset $\leftarrow$ *cset* $\cup$ {$\langle \square, result \rangle$}
  return *result*

upon COMMIT
  rvalues $\leftarrow \emptyset$
  result $\leftarrow \top$
  foreach $\langle key, - \rangle \in$ *wset* **do** LOCK(*key*)
  foreach $\square \in$ *frset* **do**
    key $\leftarrow$ KEY($\square$)
    LOCK(*key*)
    rvalues $\leftarrow$ *rvalues* $\cup$ {$\langle key,$ GET(*key*)$\rangle$}
  foreach $\langle \triangle, \square \rangle \in$ *fwset* **do**
    key $\leftarrow$ RESOLVE($\triangle$, *rvalues*)
    LOCK(*key*)
    wset $\leftarrow$ *wset* $\cup$ {$\langle key, \square \rangle$}
  foreach $\langle key, version \rangle \in$ *rset* **do**
    if *version* $\neq$ VERSION(*key*) **then** *result* $\leftarrow \bot$
  foreach $\langle \square, expected \rangle \in$ *cset* **do**
    value $\leftarrow$ RESOLVE($\square$, *rvalues*)
    if *value* $\neq$ *expected* **then** *result* $\leftarrow \bot$
  if *result* $= \top$ **then**
    foreach $\langle key, \square \rangle \in$ *wset* **do**
      value $\leftarrow$ RESOLVE($\square$, *rvalues*)
      version $\leftarrow$ NEXT-VERSION(*key*)
      PUT(*key*, *value*, *version*)
  foreach *key* $\in$ *wset* $\cup$ KEYS(*frset*) **do** UNLOCK(*key*)
  return $\langle result, rvalues \rangle$

Figure 4: LSD-aware OCC protocol.

IS-TRUE($\square$). Observes the current value of each key present in the condition $\square$, i.e., each future-value over which the condition is defined, resolve $\square$ using the observed values, add the result to the condition set, and returns it.

COMMIT. As we discussed, the commit protocol executes in three steps. First, we lock the write set and the future-write set. However, the latter initially has its keys unresolved. To resolve and then lock them, we first need to resolve the future read set because it contains the future-values of READ operations that were delayed, and future-keys and future-values in the regular and future write sets are likely to depend on the future read set. $\big($e.g., a transaction reads *key* and gets future-value $\square$, which it then uses to create a future $\triangle = f(\square)$, according to some function f, that the transaction uses as a future-key — WRITE($\triangle$,...) — and/or future-value — WRITE(...,$\triangle$).$\big)$ To guarantee that we resolve the future read set consistently, we first lock the respective keys.

In the second step, we validate the read set. In addition to the read set, transactions also observe database state via conditions and the IS-TRUE operation, so we also validate each condition in the condition set using the values obtained from the future read set. In the final step, we resolve the buffered future-values, perform the writes, and release acquired locks.

To illustrate these steps, we will simulate the execution of our running example of Figure 1b. First, the transaction issues the READ operation for the item's stock. This operation is local to the client, since it merely creates the future $\square$ and returns it. Then the transaction attempts to purchase *qty* amount of items if there is enough stock. Let us assume that $qty = 10$. Since the transaction does not know the concrete value of the item's stock, it uses the IS-TRUE operation to check whether there are at least 10 items available. Assume that, in this example execution, the transaction is operating on a database state where there are at least 10 items in stock. Then, in order to maintain isolation, this condition must also hold when the transaction attempts to commit, and thus the transaction records the condition and its result in the condition set for commit-time validation. Finally, the transaction defines the necessary computation to update the stock value with the future $\triangle$, issues the WRITE with it, and attempts to commit. The COMMIT operation will then atomically resolve the stock value $\square$ to, for example, 42, verify that $42 \geq 10$, and compute the new stock value $\triangle$ to be $42 - 10 = 32$. Note that, when using standard OCC, any concurrent write to the stock value causes the transaction to abort. With LSD-aware OCC,

	R	W	R(p)	W($v:c(v)$)	W($v:\neg c(v)$)
R	✓	−	✓	−	−
W	−	−	−	−	−
R(p)	✓	−	✓	✓	−
W($v:c(v)$)	−	−	✓	−	−
W($v:\neg c(v)$)	−	−	−	−	−

Figure 5: 2PL's lock compatibility matrix. The gray cells represent the standard 2PL matrix, and LSD introduces the remaining cells. A ✓ means that acquiring the lock in row mode succeeds when the lock is in column mode.

instead, the transaction only aborts if between the time the IS-TRUE and COMMIT operations are issued the stock value changes to a value below 10.

Possible optimization. Since the IS-TRUE operations are validated at commit time to ensure isolation, it is possible to optimistically assume a specific result for an IS-TRUE operation *without* communicating with the database. Whether this behavior yields better performance or not depends on the success rate of the assumption: if the assumption is correct we save one communication round with the database, but if it is not, the transaction aborts, perhaps needlessly, and upon retry performs the IS-TRUE operation normally. We evaluate this optimization in §4.

3.3.3 2-phase locking (2PL)

2PL follows a rational opposite to OCC: instead of assuming that conflicts seldom happen, 2PL immediately acquires a lock when a transaction accesses an object to prevent conflicting transactions from breaking isolation.

The central idea of the adaptation of 2PL to LSD's IS-TRUE operation is the novel concept of a *condition lock*, which is an extension of a *read-write lock*. To understand the semantics of condition locks, we first recall that read-write locks can be acquired in either read or write mode (R or W). The semantics of read-write locks are then given by their compatibility matrix shown in gray in Figure 5. This shows that multiple readers, i.e., read-mode acquires, can proceed simultaneously, but writers are serialized. Condition locks, in turn, have two additional modes: *read condition* and *write value*. The read condition mode, R(c), associates a condition c with the lock, signaling that a transaction has observed a value that respects the condition c. Other transactions can still successfully up-

```
upon READ(△)
  key ← KEY(△)
  LOCK(key)
  value ← GET(key)
  rset ← rset ∪ {key}
  return READ(value)

upon IS-TRUE(□)
  rvalues ← ∅
  foreach key ∈ KEYS(□) do
    LOCK(key)
    rvalues ← rvalues ∪ {GET(key)}
  result ← RESOLVE(□, rvalues)
  foreach key ∈ KEYS(□) do
    ADD-CONDITION(key, ⟨□, result⟩)
    UNLOCK(key)
  cset ← cset ∪ {□}
  return result

upon COMMIT
  rvalues ← ∅
  foreach □ ∈ frset do
    key ← KEY(□)
    LOCK(key)
    rvalues ← rvalues ∪ {⟨key, GET(key)⟩}
  foreach □ ∈ cset do
    foreach key ∈ KEYS(□) do REM-CONDITION(key, □)
  set ← ∅
  foreach ⟨△, □⟩ ∈ fwset do
    key ← RESOLVE(△, rvalues)
    set ← set ∪ {⟨key, □⟩}
  writes ← ∅
  foreach ⟨key, □⟩ ∈ wset ∪ set do
    value ← RESOLVE(□, rvalues)
    writes ← writes ∪ {⟨key, value⟩}
  foreach ⟨key, value⟩ ∈ writes do
    LOCK-COMPATIBLE(key, writes)
    PUT(key, value)
  foreach key ∈ writes ∪ rset ∪ KEYS(frset) do
    UNLOCK(key)
  return ⟨⊤, rvalues⟩
```

Figure 6: LSD-aware 2PL protocol.

date a read condition-locked object by acquiring the lock in write value mode. The write value mode, $W(v)$, is aware of the value v that the transaction intends to assign to the object. If the lock is in read condition mode and the value v respects all the conditions that the lock holds, the write mode acquire succeeds. Otherwise it blocks as usual. Note that the read condition mode is a generalization of the read mode: the latter is smilar to the former with a condition that always returns false regardless of the value other transactions intend to write.

Next, we describe the adaptations required for the remaining operations, which are also summarized in Figure 6.

READ(△). Resolves the future-key $△$, i.e., compute its concrete value *value* by locking *key*, reading its value, and then READ(*value*). (Returning a future.)

IS-TRUE(□). Atomically observes the current value of each key present in the condition $□$ by locking all keys. Resolve the condition $□$ using the observed values, and downgrade the acquired locks to read condition mode using $□$ and its result.

COMMIT. Resolves the future read set by locking and performing the delayed reads. Remove the conditions installed via the IS-TRUE operations since we already resolved the future read set. Resolve all future-keys in the future write set, and all future-values in the future and concrete write set. Given that we now know the transaction's full write set, acquire the locks in write value mode, perform the writes, and release the acquired locks.

Again, to better understand these steps, we will go through the steps of the execution of our running example of Figure 1b. The transaction reads the item's stock, which is an operation local to the client. Then the transaction attempts to purchase *qty* amount of items if there is enough stock. Let us assume that $qty = 10$. The transaction uses the IS-TRUE operation to check whether there are at least 10 items available. Again, assuming that this is the case, to maintain isolation this must be also true when the transaction commits. To ensure this, a condition lock is acquired, in read condition mode, on the stock stating that its value must remain greater or equal to 10. The transaction proceeds to define the necessary computation to update the stock value with the future $△$, issues the WRITE with it, and attempts to commit. The COMMIT operation will then atomically resolve the stock value $□$ to, for example, 42, remove the condition $□ ≥ qty$ from the stock lock, and compute the new stock value $△$ to be $42 - 10 = 32$. Then

9

the transaction acquires the stock's lock in write value mode with 32, blocking only if there is any concurrent reader that installed a condition c such that $\neg c(32)$. (With standard 2PL *any* concurrent reader would cause the transaction to block.) Finally, the transaction modifies the stock to 32 and releases the locks.

3.3.4 Multi-future conditions

OCC and 2PL fundamentally differ on how they deal with the validity of conditions. OCC does not ensure that a condition asserted via the IS-TRUE operation remains valid. This is because write transactions are not aware of those conditions and can freely violate the conditions when they commit. As such, it is up to a transaction that asserts a condition to validate it when the transaction attempt to commit to ensure isolation, i.e., the burden of dealing with conditions is on the readers. In contrast, 2PL ensures that an asserted condition remains valid until the asserting transaction commits, as acquring a condition lock in write value mode will block if the value to be written violates any existing asserted condition, i.e., the burden of dealing with conditions is on the writers.

Dealing with conditions on the writer's side is more complex than on the reader's, and this complexity is exacerbated in the presence of conditions over more than one future. For example, consider two keys x and y, with values 2 and 1, respectively, read by some transaction t_1 as futures $\square$ and $\triangle$. t_1 then executes IS-TRUE($\{\square > \triangle\}$), which returns $\top$ (because $2 > 1$). Then assume that, concurrently, another transaction t_2 attempts to write 1 to x. For t_2 to acquire x's condition lock in write value mode with value 1 and commit, the procedure to acquire the condition lock in write value mode (LOCK-COMPATIBLE in Figure 6) can only grant the lock to t_1 if $1 > \triangle$ remains valid. Thus, the locking procedure must resolve $\triangle$ to check the concrete validity of $1 > 1$. To do so, there are two possibilities. If t_2 also reads y, then it has acquired a read lock on y so it can resolve $\triangle$. If not, the lock procedure needs to resolve $\triangle$ in a way that ensures transactional isolation, e.g., acquiring a read lock on y on behalf of t_2.

Given the experience in the implementation of our prototype, we argue that the IS-TRUE operation is simpler to implement, and understand, using an optimistic approach. Additionally, the experimental evaluation (§4) using our prototype shows that the LSD-aware OCC protocol performs better than the LSD-aware 2PL protocol, so we conclude that future implementations of LSD should use OCC in most cases.

3.4 Distributed transactions

So far we have discussed how to adapt both OCC and 2PL to exploit LSD in the context of a single server. However, transactions may be distributed, i.e., span multiple servers, if the database is partitioned. We now briefly sketch how to adapt 2-phase commit (2PC) [7], the most widely used distributed commit protocol, to support LSD. A more comprehensive discussion of LSD together with 2PC can be found in [32]

LSD introduces the future read and write sets, and condition set. The future write set is of particular importance, since it depends on the future read set. This means that, in general, transactions that have a non-empty future write set require an additional round of communication during 2PC's prepare phase. Each participant resolves, and returns, its portion of the future read set in the regular communication round of the prepare phase. Armed with the resolved future read set the coordinator can resolve the future write set and send it to the required participants.

It is possible to circumvent the need for the additional communication round in the prepare phase and send the future write set immediately in the first round if, for every entry in the future write set: (1) we can identify its future-key's partition without resolving it, and (2) (all) the future(s) on which the future-key depends is (are) from the same partition it belongs to. In our experiments we evaluate both cases: when LSD incurs in an additional communication round in 2PC, and when it does not.

4 Evaluation

We implemented a partitioned, transactional, key-value store prototype, including all of the previously described design with the exception of multi-future conditions. Each partition is implemented as a Thrift [2] non-blocking server, and data is stored in disk using RocksDB [15]. Clients can execute transactions using the typical API (BEGIN, READ, WRITE, COMMIT, and ABORT operations) or LSD's API which features our proposed READ, WRITE, and IS-TRUE operations. We implemented both classical OCC and 2PL, and also both their LSD-aware variants for LSD transactions. Distributed transactions commit using 2PC. We resolve deadlocks that may arise in 2PL or 2PC using the wound-wait strategy.

We conducted an experimental evaluation of our LSD prototype on a private gigabit ethernet cluster. Each server runs on a machine with a 2Ghz Intel Xeon E5-2620 processor, 32GB of RAM, and a 7200 RPM hard drive. Clients run on the various remaining machines with AMD and Intel processors, and communicate with the servers using Thrift RPCs.

Each data point reports the average of 5 runs. Our evaluation seeks to answer the following questions:

- Does LSD improve the performance of realistic applications under contention? (§4.1.1)

- What is LSD's overhead when contention is low? (§4.1.2)

- How do LSD's benefits vary across various deployment scenarios, such as with a single database, or with a partitioned database and distributed transactions? (§4.1)

- What is the impact of an increasing amount of contention with and without conditions? (§4.2)

4.1 Realistic application: TPC-C

We used the popular TPC-C benchmark [30] to assess LSD's ability to improve performance of realistic applications under contention, as well as its overhead, on different deployment scenarios. LSD was particularly helpful for the two core transactions of the workload: Payment and New Order. For example, both make use of write functions to modify client balance and stock values, and the latter also uses conditions.

We experimented with TPC-C under three different deployments: (a) a centralized database, (b) a partitioned database using an application-specific partitioning policy, and (c) a partitioned database using an application-agnostic partitioning policy. We executed TPC-C with a high and low contention workload in each deployment.

Setup. We setup each deployment as follows. The centralized database (a) uses a single server that stores the entire data. The database partitioned using an application-specific policy (b) uses 3 servers. The data associated with a particular warehouse is stored within a single server. The remaining data, such as item information, is partitioned across all servers via hashing. Finally, the database partitioned using an application-agnostic policy (c) also uses 3 servers. Data is partitioned across all servers via hashing.

4.1.1 High contention

In TPC-C, the level of contention is proportional to the number of warehouses, so we loaded the database with the minimum number of warehouses applicable to each deployment (as detailed below) and then executed TPC-C with an increasing number of clients. Figures 7a, 7b, and 7c, compare the throughput, measured in committed transactions per second (x axis), and the corresponding average transaction execution latency, measured in in milliseconds (y axis), of OCC and 2PL with and without LSD.

Centralized deployment (Figure 7a). We loaded the database with 1 warehouse. The LSD-aware OCC variant achieved a peak throughput of ≈ 1K committed transactions per second with an average latency of ≈ 70 ms, which amounts to $\approx 6.5\times$ higher throughput and $\approx 2.5\times$ lower latency than standard OCC under the same load. The LSD-aware 2PL variant achieved a peak throughput of ≈ 850 committed transactions per second with an average latency of ≈ 80 ms, which amounts to $\approx 2.5\times$ higher throughput and $\approx 1.5\times$ lower latency than standard 2PL under the same load.

Partitioned deployment using application-specific policy (Figure 7b). We loaded the database with 3 warehouses. Data was partitioned across the servers by warehouse, i.e., each server hosts a single warehouse. This scenario allows for the presence of distributed transactions. Distributed LSD transactions commit using the regular 2PC protocol, i.e., without incurring in the additional communication rounds discussed in §3.4, thanks to the application-specific partitioning policy. The LSD-aware OCC variant achieved a peak throughput of ≈ 2K committed transactions per second with an average latency of ≈ 50 ms, which amounts to $\approx 5\times$ higher throughput and $\approx 1.5\times$ lower latency than standard OCC under the same load. The LSD-aware 2PL variant achieved a peak throughput of ≈ 1.5K committed transactions per second with an average latency of ≈ 60 ms, which amounts to $\approx 1.5\times$ higher throughput and $\approx 1.3\times$ lower latency than standard 2PL under the same load.

Partitioned deployment using application-agnostic policy (Figure 7c). We loaded the database with a single warehouse, and all data is partitioned across the servers using hashing. By using an application-agnostic partitioning policy, such
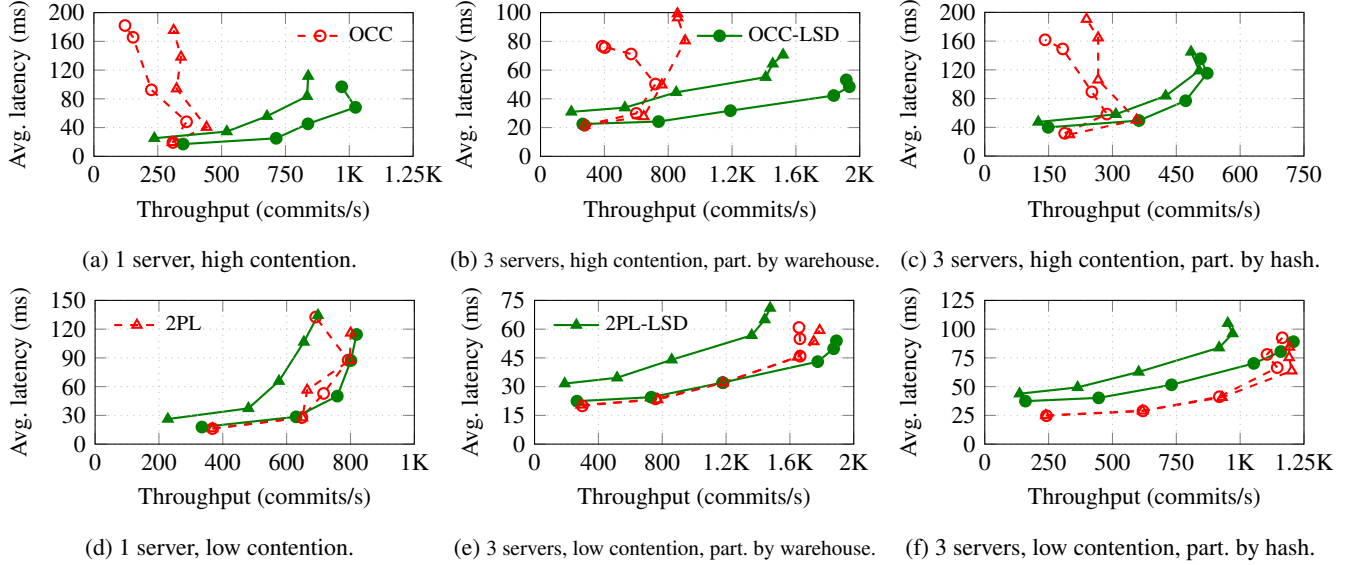
Figure 7: Performance of TPC-C on a workload using: 1 server with high (a) and low (b) contention; 3 servers with partitioning by warehouse (b,e); and 3 servers with partitioning by hash (c,f).

as hashing, distributed LSD transactions may need an additional communication round to commit using 2PC. This is the case for the New-Order transaction, which comprises almost half of the workload. Despite the additional communication round, the LSD-aware OCC variant achieved a peak throughput of ≈ 500 committed transactions per second with an average latency of ≈ 120 ms, which amounts to ≈ 2.8× higher throughput and ≈ 1.3× lower latency than standard OCC under the same load. The LSD-aware 2PL variant achieved a peak throughput of ≈ 500 committed transactions per second with an average latency of ≈ 120 ms, which amounts to ≈ 1.8× higher throughput and ≈ 1.3× lower latency than standard 2PL under the same load.

Discussion. This workload highlights the benefits of LSD. For example, under the standard interface semantics, any two concurrent New-Order transactions conflict if: (a) they operate on the same district (conflicting accesses to the district's order identifier counter), or (b) they order the same item (conflicting accesses to the item's stock). Under OCC only one of the concurrent transactions commits and the other aborts. Under 2PL one of the transactions queues behind the other when it attempts to acquire the lock held by the other. In both

cases one of the transactions prevents the other from executing, leading to an effective serialization of their execution. With LSD, New-Order transactions delay their accesses to the district's order identifier counter until commit time, so these accesses do *not* result in aborts under OCC, nor queueing during transaction execution under 2PL. Furthermore, any two New-Order transactions that order the same item only conflict if both attempt to buy the entire remaining stock. LSD's benefits translate in practice to higher throughput and lower latency under contention due to less aborts (blocks) under OCC (2PL). For example, in the data point where LSD transactions achieve their peak throughput on Figure 7a, ≈ 92% of OCC transactions abort, whereas this number drops to ≈ 8% with the LSD-aware variant.

It is worth noting that our LSD-aware 2PL implementation incurs in higher overhead than its OCC counterpart. While there still may be room for optimization of our prototype, the LSD-aware 2PL has fundamentally more overhead than its OCC counterpart because condition locks are a more complex technique than condition validation. The combination of this higher overhead of LSD-aware 2PL with the fact that unlike the usual OCC implementations, our LSD-aware variant of OCC presents low abort rate under high contention, leads to to a somewhat surprising result: the LSD-aware variant of

12

OCC performs better than its 2PL counterpart under high contention.

4.1.2 Low contention

In the previous section, we evaluated LSD using a TPC-C workload with high contention, which is the type of workload that LSD benefits. In this section we describe our evaluation of LSD in the opposite scenario: a TPC-C workload with low contention. Specifically, we increased the number of warehouses in the workload from 1 to 32.

In both the centralized (Figure 7d) and partitioned deployments using the application-specific policy (Figure 7e), we observe that the LSD-aware OCC variant incurred in marginal overhead. In the partitioned deployment using the application-agnostic policy (Figure 7f), the overhead becomes more pronounced (≈ 1.25–$1.5\times$) due to the additional communication round needed to commit some distributed transactions. However, at high load the LSD-aware variant managed to achieve similar to better performance. In contrast, the LSD-aware 2PL exhibits worse performance than either protocol using the standard interface.

We conclude that the LSD-aware OCC protocol is not only the best of the LSD variants, but also the best solution when either using a single database or a partitioned database with a partitioning scheme that allows for committing distributed transactions without incurring in additional communication rounds. Even with additional communication rounds, LSD is able to reap better performance under contention, while still providing competitive performance when contention is low.

4.2 Microbenchmarks

In this section we report on microbenchmark results that show the effect of specific workload characteristics on LSD.

Contention without conditions. We start by analyzing the effect of contending read-modify-write operations. To do so, we loaded the database with as many private counters as there were clients, and a single shared counter—the "hot" counter. Transactions consisted of an increment of either the hot counter, according to some probability p, or the respective private counter, with probability $1 - p$. We executed the microbenchmark for various values of p, ranging from 0% (no contention) to 100% (all transactions contend).

Figure 8a plots the measured throughput as a function of the parametrized contention. The LSD-aware protocols are not affected by the parameter because the increments are delayed until commit time, whereas the throughput of the OCC and 2PL protocols decreases when contention increases, as expected, due to aborts in OCC (Figure 8d), and transactions blocking when attempting to read the value of the hot counter in 2PL. At 100% contention, LSD's throughput is $\approx 5\times$ higher that 2PL and $\approx 30\times$ more than OCC.

Even when every transaction only increments its own private counter, the LSD-aware variants still perform better than their standard counterparts due to the fact that the LSD's READ operation does not communicate with the database (it creates the respective future locally). LSD transactions incur in less communication rounds than standard transactions, which translated into an $\approx 1.3\times$ increase in throughput.

Contention with conditions. We now analyze the effect of contention in the presence of conditions asserted with the IS-TRUE operation. Like in the previous microbenchmark, we loaded the database with a set of private counters and a single hot counter. These counters are initialized with a parametrized value n, and a parametrized percentage of transactions access the hot counter while the remaining access their private counter. The logic of the transactions consisted of decrementing the value of the counter if it remained greater than zero, or restoring the its initial value otherwise. Unlike the previous experiment, in this one we could control the contention that LSD transactions experienced on the condition: the smaller the initial value of the counters, the higher the contention, i.e., the condition "the counter remains greater than zero" changes at a rate of $\frac{1}{n}$, where n is the parameterized initial value for the counters.

Figures 8b, 8c, 8e, and 8f, depict the throughput and abort percentage of each protocol. For a scenario with no contention for either LSD or the standard interface, i.e., each transaction only accesses its private counter, the LSD variants incur in an overhead of ≈ 1.1–$1.25\times$ when compared to their standard counterparts (Figure 8b). This overhead comes from the additional work performed by the IS-TRUE operation, which is not extracting additional parallelism in this experiment because there is no contention. We also plot a version of the LSD-aware OCC (OCC-LSD+) that assumes the counter's value remains greater than zero after the decrement, i.e. it speculates the outcome of the IS-TRUE operation without contacting the database, as discussed in §3.3.2. The effectiveness of
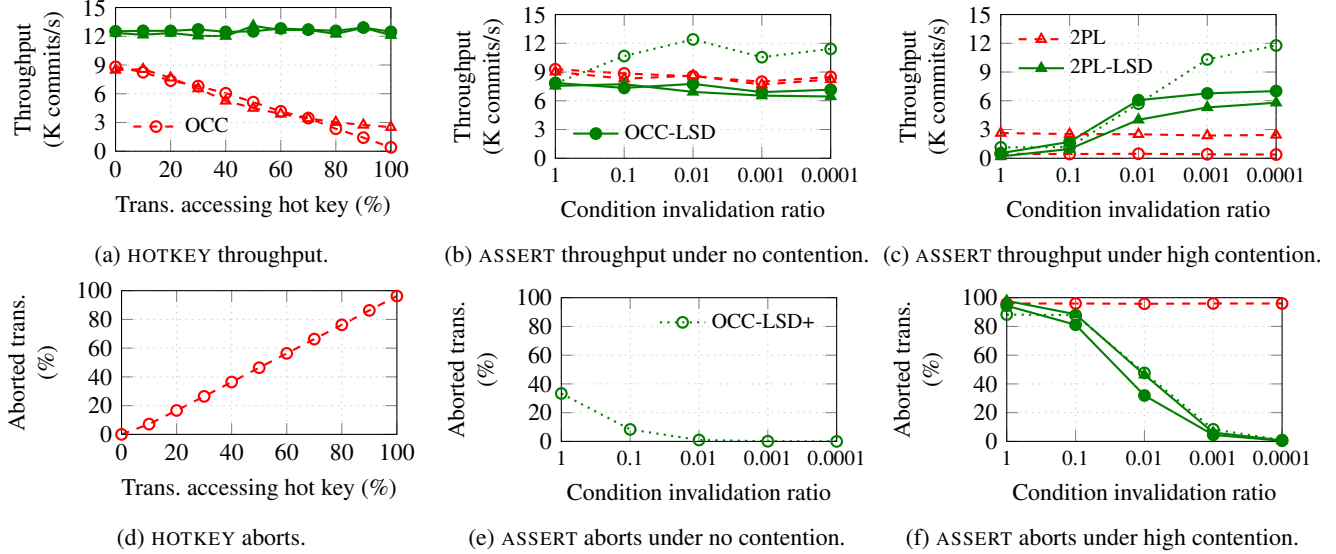
Figure 8: Throughput and aborts on the HOTKEY (a,d) and ASSERT microbenchmarks with no (b,e) and high (c,f) contention.

the LSD+ variant depends on the success of its speculation. As expected, the results in Figure 8b show that the throughput of the LSD+ variant increased when we decreased the condition invalidation ratio, increasing throughput up to $\approx 1.3\times$ that of the standard protocols. The throughput increases because the number of aborts due to failed speculation decreases, as shown in Figure 8e. Only the LSD+ variant aborts in this experiment because each transaction accesses its own private counter.

Next, we examined the situation where all transactions access the hot counter. This is the worse case scenario for the standard transactions, whereas LSD transactions can still extract parallelism if the concurrent modifications to the counter do not keep invalidating the condition. Figure 8c reports the observed throughput as a function of the condition invalidation ratio. The performance of standard transactions is unaffected by the condition invalidation ratio because standard transactions only deal with concrete values when accessing the counter, so all concurrent transactions conflict: OCC suffers from a high percentage of aborts (Figure 8f) while 2PL suffers from a "queueing" effect when acquiring the lock in the READ operation. Note that in this experiment the results for 2PL are optimal somewhat inflated, because we disabled deadlock prevention for 2PL since transactions only access a single key. With LSD, on the other hand, throughput increased

as there was more available parallelism to exploit, i.e., updates to the counter that would not make its value fall below 1. In particular, as the abort percentage decreased (Figure 8f), the LSD-aware variant of OCC (resp. 2PL) achieved up to $\approx 17\times$ (resp. $\approx 2\times$) more throughput than its standard counterpart (Figure 8c). The LSD+ variant was able to further boost the throughput gains to $\approx 30\times$ the performance of OCC.

5   Related work

Futures and lazy evaluation.   Sloth [10] uses futures and lazy evaluation to reduce the number of network round trips in database-backed applications. It batches queries at the client until any of the batched query results are needed by the client logic, at which point the batch is sent to the database. LSD's futures achieve the same goal, but LSD goes further by using futures and lazy evaluation to push application semantics and computation into the database, for the concurrency control protocol to extract more parallelism. Faleiro et al. [16] propose a lazy transaction execution engine. Transactions must be stored procedures, and the system acknowledges transactions as committed without executing them. When some transaction needs to observe state that would have been written

by some delayed transaction, the delayed transaction and its dependencies are executed. In contrast, LSD allows for both stored procedures and the most prevalent client-server execution model (a recent study shows that stored procedures account for less than 10% of the transactions in most database deployments, and that there are few deployments where all transactions are stored procedures [25]), and uses futures and lazy evaluation to extract additional read-write parallelism by refining what constitutes a conflict in terms of conditions. Other proposals [8, 12] also delay computation over contended objects until commit but only *if* the values of the objects are not used anywhere else in the transaction's logic. LSD's holistic design of futures and lazy evaluation do not impose this restriction.

Performance under contention. Doppel [22] replicates contended objects across workers to allow parallel commutative updates to each replica, at the expense of preventing the execution of transactions that need to read or perform different update operations. LSD extracts read-write parallelism using futures, conditions, and lazy evaluation instead. ROCOCO [21] requires programmers to organize transaction logic in pieces that access one or more objects stored on a single partition. Developers must therefore be aware of the partitioning policy, and the code is tied to a particular policy. All transactions need to be known in advance to perform complex static analysis. Callas [34] automates Salt's methodology [33] by requiring transactions to be known in advance to perform static analysis to expose intermediate states to other transactions. LSD improves performance without requiring changes to transaction logic and to know transactions beforehand.

Other systems explore the semantics of applications to maintain correctness under non-linearizable executions. Escrow transactions [23] and the demarcation protocol [6] maintain global invariants. The homeostatis protocol [26] allows distributed databases to execute transactions without coordination across partitions under certain conditions identified by static analysis. In contrast, LSD improves performance under contention while providing linearizability.

Performance. Silo [31] refine OCC to improve the performance of in-memory databases. FaRM [13] exploits new hardware functionality in partitioned databases. LSD redefines what constitutes a conflict so it is complementary to these proposals. Sinfonia [1] proposes a restricted form of transactions called minitransactions, whose execution can be piggybacked in the 2PC protocol at the expense of expressiveness, e.g., it is impossible to perform a read-modify-write operation in a single minitransaction. LSD does not impose these restrictions on expressiveness, and yet LSD transactions that do not observe state before attempting to commit are also piggybacked in 2PC. The IS-TRUE operation resembles warranties [20] but LSD's design with futures and lazy evaluation extracts concurrency even in cases where warranties are not helpful, such as the example of Figure 1.

Concolic execution. LSD's approach can be interpreted as concolic execution [27, 9] of transactions. LSD's futures are similar to symbolic values, and transactions collect constraints to the possible concrete values futures will resolve to using the IS-TRUE operation. LSD develops these concepts in the context of concurrency control to improve performance while still maintaining transactional isolation.

6 Conclusion

This paper presented LSD, a refined interface for database transactions. By allowing transactions to execute their logic over an abstract state and specifying their intent more clearly to the database, the concurrency control protocol can make more informed. As a consequence, LSD enables high-performance linearizable transactions even under high contention.

References

[1] AGUILERA, M., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: A new paradigm for building scalable distributed systems. In *SOSP* (2007).

[2] APACHE SOFTWARE FOUNDATION. Apache Thrift. thrift.apache.org, Nov. 2016.

[3] BAILIS, P., FEKETE, A., FRANKLIN, M. J., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Feral concurrency control: An empirical investigation of modern application integrity. In *SIGMOD* (2015).

[4] BAKER JR., H., AND HEWITT, C. The incremental garbage collection of processes. *SIGPLAN Not. 12*, 8 (1977).

[5] BALEGAS, V., DUARTE, S., FERREIRA, C., RO-DRIGUES, R., PREGUIÇA, N., NAJAFZADEH, M., AND SHAPIRO, M. Putting consistency back into eventual consistency. In *EuroSys '15* (2015).

[6] BARBARÁ-MILLÁ, D., AND GARCIA-MOLINA, H. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *VLDBJ 3*, 3 (1994).

[7] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. 1987.

[8] CACHOPO, J., AND RITO-SILVA, A. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program. 63*, 2 (2006).

[9] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI* (2008).

[10] CHEUNG, A., MADDEN, S., AND SOLAR-LEZAMA, A. Sloth: Being lazy is a virtue (when issuing database queries). In *SIGMOD* (2014).

[11] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *SOSP* (2007).

[12] DIEGUES, N., AND ROMANO, P. Bumper: Sheltering transactions from conflicts. In *SRDS* (2013).

[13] DRAGOJEVIĆ, A., NARAYANAN, D., NIGHTINGALE, E., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: distributed transactions with consistency, availability, and performance. In *SOSP* (2015).

[14] ESWARAN, K., GRAY, J., LORIE, R., AND TRAIGER, I. The notions of consistency and predicate locks in a database system. *CACM 19*, 11 (1976).

[15] FACEBOOK. RocksDB. rocks.org, Nov. 2016.

[16] FALEIRO, J. M., THOMSON, A., AND ABADI, D. J. Lazy evaluation of transactions in database systems. In *SIGMOD* (2014).

[17] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492.

[18] HUDAK, P. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv. 21*, 3 (1989).

[19] KUNG, H., AND ROBINSON, J. On optimistic methods for concurrency control. *TODS 6*, 2 (1981).

[20] LIU, J., MAGRINO, T., ARDEN, O., GEORGE, M. D., AND MYERS, A. C. Warranties for faster strong consistency. In *NSDI* (2014).

[21] MU, S., CUI, Y., ZHANG, Y., LLOYD, W., AND LI, J. Extracting more concurrency from distributed transactions. In *OSDI* (2014).

[22] NARULA, N., CUTLER, C., KOHLER, E., AND MORRIS, R. Phase reconciliation for contended in-memory transactions. In *OSDI* (2014).

[23] O'NEIL, P. E. The escrow transactional method. *TODS 11*, 4 (1986).

[24] PAPADIMITRIOU, C. The serializability of concurrent database updates. *JACM 26*, 4 (1979).

[25] PAVLO, A. What are we doing with our lives?: Nobody cares about our concurrency control research. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), SIGMOD '17, pp. 3–3.

[26] ROY, S., KOT, L., BENDER, G., DING, B., HOJJAT, H., KOCH, C., FOSTER, N., AND GEHRKE, J. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *SIGMOD* (2015).

[27] SEN, K., MARINOV, D., AND AGHA, G. CUTE: A concolic unit testing engine for c. In *ESEC-FSE* (2005).

[28] STONEBRAKER, M. The NoSQL discussion has nothing to do with SQL. Communications of the ACM Blog, Nov. 2009.

[29] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD* (2012).

[30] TPC. TPC-C. `tpc.org/tpcc`, Nov. 2016.

[31] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *SOSP* (2013).

[32] VALE, T. M. *Executing Requests Concurrently in State Machine Replication*. PhD thesis, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Portugal, May 20018.

[33] XIE, C., SU, C., KAPRITSOS, M., WANG, Y., YAGH-MAZADEH, N., ALVISI, L., AND MAHAJAN, P. Salt: Combining ACID and BASE in a distributed database. In *OSDI* (2014).

[34] XIE, C., SU, C., LITTLEY, C., ALVISI, L., KAPRIT-SOS, M., AND WANG, Y. High-performance ACID via modular concurrency control. In *SOSP* (2015).