
# Salt: Combining ACID and BASE in a Distributed Database

**Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan,** ***The University of Texas at Austin***

https://www.usenix.org/conference/osdi14/technical-sessions/presentation/xie

**This paper is included in the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation.**

**October 6–8, 2014 • Broomfield, CO**

978-1-931971-16-4

**Open access to the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.**

# Salt: Combining ACID and BASE in a Distributed Database

Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang,
Navid Yaghmazadeh, Lorenzo Alvisi, Prince Mahajan

The University of Texas at Austin

**Abstract:** This paper presents Salt, a distributed database that allows developers to improve the performance and scalability of their ACID applications through the incremental adoption of the BASE approach. Salt's motivation is rooted in the Pareto principle: for many applications, the transactions that actually test the performance limits of ACID are few. To leverage this insight, Salt introduces *BASE transactions*, a new abstraction that encapsulates the workflow of performance-critical transactions. BASE transactions retain desirable properties like atomicity and durability, but, through the new mechanism of *Salt Isolation*, control which granularity of isolation they offer to other transactions, depending on whether they are BASE or ACID. This flexibility allows BASE transactions to reap the performance benefits of the BASE paradigm without compromising the guarantees enjoyed by the remaining ACID transactions. For example, in our MySQL Cluster-based implementation of Salt, BASE-ifying just one out of 11 transactions in the open source ticketing application Fusion Ticket yields a 6.5x increase over the throughput obtained with an ACID implementation.

## 1 Introduction

This paper presents *Salt*, a distributed database that, for the first time, allows developers to reap the complementary benefits of both the ACID and BASE paradigms within a single application. In particular, Salt aims to dispel the false dichotomy between performance and ease of programming that fuels the ACID vs. BASE argument.

The terms of this debate are well known [28, 30, 37]. In one corner are ACID transactions [7–9, 12–14, 36]: through their guarantees of Atomicity, Consistency, Isolation, and Durability, they offer an elegant and powerful abstraction for structuring applications and reasoning about concurrency, while ensuring the consistency of the database despite failures. Such ease of programming, however, comes at a significant cost to performance and availability. For example, if the database is distributed, enforcing the ACID guarantees typically requires running a distributed commit protocol [31] for each transaction while holding an exclusive lock on *all* the records modified during the transaction's entire execution.

In the other corner is the BASE approach (Basically-Available, Soft state, Eventually consistent) [28, 32, 37], recently popularized by several NoSQL systems [1, 15, 20, 21, 27, 34]. Unlike ACID, BASE offers more of a set of programming guidelines (such as the use of *partition local transactions* [32, 37]) than a set of rigorously specified properties and its instantiations take a variety of application-specific forms. Common among them, however, is a programming style that avoids distributed transactions to eliminate the performance and availability costs of the associated distributed commit protocol. Embracing the BASE paradigm, however, exacts its own heavy price: once one renounces ACID guarantees, it is up to developers to explictly code in their applications the logic necessary to ensure consistency in the presence of concurrency and faults. The complexity of this task has sparked a recent backlash against the early enthusiasm for BASE [22, 38]—as Shute et al. put it "Designing applications to cope with concurrency anomalies in their data is very error-prone, time-consuming, and ultimately not worth the performance gains" [38].

Salt aims to reclaim most of those performance gains while keeping complexity in check. The approach that we propose to resolve this tension is rooted in the Pareto principle [35]. When an application outgrows the performance of an ACID implementation, it is often because of the needs of only a handful of transactions: most transactions never test the limits of what ACID can offer. Numerous applications [2, 4, 5, 10, 11] demonstrate this familiar lopsided pattern: few transactions are performance-critical, while many others are either lightweight or infrequent; e.g. administrative transactions. Our experience confirms this pattern. For example, running the TPC-C benchmark [23] on a MySQL cluster, we found that, as the load increases, only two transactions take much longer to complete—a symptom of high contention; other transactions are unaffected. Similarly, we found that the ACID throughput of Fusion Ticket [6], a popular open source online ticketing application that uses MySQL as its backend database, is limited by the performance of just *one* transaction out of 11. It is tempting to increase the concurrency of those transactions by splitting them into smaller ones. Doing so, however, exposes fundamental limitations of the ACID paradigm.

One of the main attractions of the ACID paradigm is to pack in a single abstraction (the ACID transaction) the four properties that give ACID its name. This tight cou-

pling of all four properties, however, comes at the cost of little flexibility. In particular, offering atomicity and isolation at the same granularity is the very reason why ACID transactions are ill-equipped to manage effectively the tradeoff between performance and ease of programming. First, splitting an ACID transaction into several smaller transactions to increase concurrency would of course result in the loss of the all-or-nothing atomicity guarantees of the original transaction. But even more disturbing would be the resulting loss in isolation, not only for the split transaction, but for *all* transactions in the system: any transaction would be able to indiscriminately access what used to be intermediate database states protected by the guarantees of isolation, making it much harder to reason about correctness. Nonetheless, this is, in essence, the strategy adopted by the BASE approach, which for good measure also gives up consistency and durability in the bargain.

Motivated by these insights, our vision for Salt is simple: to create a database where the ACID and BASE paradigms can safely coexist within the same application. In particular, Salt enables ACID applications that struggle to meet their growing performance demands to improve their availability and scalability by incrementally "BASE-ifying" only the few ACID transactions that are performance-critical, without compromising the ACID guarantees enjoyed by the remaining transactions.

Of course, naively BASE-ifying selected ACID transactions may void their atomicity guarantees, compromise isolation by exposing intermediate database states that were previously unobservable, and violate the consistency invariants expected by the transactions that have not been BASE-ified. To enable mutually beneficial coexistence between the ACID and BASE paradigms, Salt introduces a new abstraction: *BASE transactions*.

BASE transactions loosen the tight coupling between atomicity and isolation enforced by the ACID paradigm to offer a unique combination of features: the performance and availability benefits of BASE-style partition-local transactions together with the ability to express and enforce atomicity at the granularity called for by the application semantics.

Key to this unprecedented combination is *Salt Isolation*, a new isolation property that regulates the interactions between ACID and BASE transactions. For performance, Salt Isolation allows concurrently executing BASE transactions to observe, at well-defined spots, one another's internal states, but, for correctness, it completely prevents ACID transactions from doing the same.

We have built a Salt prototype by modifying an ACID system, the MySQL Cluster distributed database [9], to support BASE transactions and Salt Isolation. Our initial evaluation confirms that BASE transactions and Salt Isolation together allow Salt to break new ground in balancing performance and ease of programming.

In summary, we make the following contributions:

- We introduce a new abstraction, *BASE transactions*, that loosens the tight coupling between atomicity and isolation to reap the performance of BASE-style applications, while at the same time limiting the complexity typically associated with the BASE paradigm.
- We present a novel isolation property, *Salt Isolation*, that controls how ACID and BASE transactions interact. Salt Isolation allows BASE transactions to achieve high concurrency by observing each other's internal states, without affecting the isolation guarantees of ACID transactions.
- We present an evaluation of the Salt prototype, that supports the view that combining the ACID and BASE paradigms can yield high performance with modest programming effort. For example, our experiments show that, by BASE-ifying just one out of 11 transactions in the open source ticketing application Fusion Ticket, Salt's performance is *6.5x* higher than that of an ACID implementation.

The rest of the paper proceeds as follows. Section 2 sheds more light on the ACID vs. BASE debate, and the unfortunate tradeoff it imposes on developers, while Section 3 proposes a new alternative, Salt, that sidesteps this tradeoff. Section 4 introduces the notion of BASE transactions and Section 5 presents the novel notion of Salt Isolation, which allows ACID and BASE transactions to safely coexist within the same application. Section 6 discusses the implementation of our Salt prototype, Section 7 shows an example of programming in Salt, and Section 8 presents the results of our experimental evaluation. Section 9 discusses related work and Section 10 concludes the paper.

## 2 A stark choice

The evolution of many successful database applications follows a common narrative. In the beginning, they typically rely on an ACID implementation to dramatically simplify and shorten code development and substantially improve the application's robustness. All is well, until success-disaster strikes: the application becomes wildly popular. As the performance of their original ACID implementation increasingly proves inadequate, developers are faced with a Butch Cassidy moment [33]: holding their current ground is untenable, but jumping off the cliff to the only alternative—a complete redesign of their application following the BASE programming paradigm—is profoundly troubling. Performance may soar, but so will complexity, as all the subtle issues that ACID handled automatically, including error handling, concurrency control, and the logic needed for consistency enforcement, now need to be coded explicitly.

```
1  // ACID transfer transaction
2  begin transaction
3    Select bal into @bal from accnts where id = sndr
4    if (@bal >= amt)
5      Update accnts set bal −= amt where id = sndr
6      Update accnts set bal += amt where id = rcvr
7  commit

9  // ACID total−balance transaction
10 begin transaction
11   Select sum(bal) from accnts
12 commit
```

(a) The ACID approach.

```
1  // transfer using the BASE approach
2  begin local−transaction
3    Select bal into @bal from accnts where id = sndr
4    if (@bal >= amt)
5      Update accnts set bal −= amt where id = sndr
6      // To enforce atomicity, we use queues to communicate
7      // between partitions
8      Queue message(sndr, rcvr, amt) for partition(accnts, rcvr)
9  end local−transaction

11 // Background thread to transfer messages to other partitions
12 begin transaction // distributed transaction to transfer queued msgs
13   <transfer messages to rcvr>
14 end transaction

16 // A background thread at each partition processes
17 // the received messages
18 begin local−transaction
19   Dequeue message(sndr, rcvr, amt)
20   Select id into @id from accnts where id = rcvr
21   if (@id ≠ 0) // if rcvr's account exists in database
22     Update accnts set bal += amt where id = rcvr
23   else // rollback by sending the amt back to the original sender
24     Queue message(rcvr, sndr, amt) for partition(accnts, sndr)
25 end local−transaction

27 // total−balance using the BASE approach
28 // The following two lines are needed to ensure correctness of
29 // the total−balance ACID transaction
30 <notify all partitions to stop accepting new transfers>
31 <wait for existing transfers to complete>
32 begin transaction
33   Select sum(bal) from accnts
34 end transaction
35 <notify all partitions to resume accepting new transfers>
```

(b) The BASE approach.

Fig. 1: A simple banking application with two implementations: (a) ACID and (b) BASE

Figure 1 illustrates the complexity involved in transitioning a simple application from ACID to BASE. The application consists of only two transactions, `transfer` and `total-balance`, accessing the *accnts* relation. In the original ACID implementation, the transfer either commits or is rolled-back automatically despite failures or invalid inputs (such as an invalid *rcvr* id), and it is easy to add constraints (such as $bal \geq amt$) to ensure consistency invariants. In the BASE implementation, it is instead up to the application to ensure consistency and atomicity despite failures that occur between the first and second transaction. And while the level of isolation (the property that specifies how and when changes to the database performed by one transaction become visible to transactions that are executing concurrently) offered by ACID transactions ensures that `total-balance` will compute accurately the sum of balances in *accnts*, in BASE the code needs to prevent explicitly (lines 30 and 31 of Figure 1(b)) `total-balance` from observing the intermediate state after the *sndr* account has been charged but before the *rcvr*'s has been credited.

It speaks to the severity of the performance limitations of the ACID approach that application developers are willing to take on such complexity.

The ACID/BASE dichotomy may appear as yet another illustration of the "no free lunch" adage: if you want performance, you must give something up. Indeed—but BASE gives virtually *everything* up: the entire application needs to be rewritten, with no automatic support for either atomicity, consistency, or durability, and with isolation limited only to partition-local transactions. Can't we aim for a more reasonable bill?

## 3 A grain of Salt

One ray of hope comes, as we noted in the Introduction, from the familiar Pareto principle: even in applications that outgrow the performance achievable with ACID solutions, not all transactions are equally demanding. While a few transactions require high performance, many others never test the limits of what ACID can offer. This raises an intriguing possibility: could one identify those few performance-critical transactions (either at application-design time or through profiling, if an ACID implementation of the application already exists) and somehow only need to go through the effort of BASE-ifying *those* transactions in order to get most of the performance benefits that come from adopting the BASE paradigm?

Realizing this vision is not straightforward. For example, BASE-ifying only the `transfer` transaction in the simple banking application of Figure 1 would allow `total-balance` to observe a state in which *sndr* has been charged but *rcvr*'s has not yet been credited, causing it to compute incorrectly the bank's holdings. The central issue is that BASE-ifying transactions, even if only a few, can make suddenly accessible to all transactions what previously were invisible intermediate database states. Protecting developers from having to worry about such intermediate states despite failures and concurrency, however, is at the core of the ease of programming offered by the transactional programming paradigm. Indeed, quite naturally, isolation (which regulates which states can be accessed when transactions execute concurrently) and atomicity (which frees from worrying about intermediate states during failures) are typically offered at the same granularity—that of the ACID transaction.

We submit that while this tight coupling of atomicity and isolation makes ACID transactions both powerful and attractively easy to program with, it also limits their ability to continue to deliver ease of programming when

performance demands increase. For example, splitting an ACID transaction into smaller transactions can improve performance, but at the cost of shrinking the original transaction's guarantees in terms of both atomicity and isolation: the all-or-nothing guarantee of the original transaction is unenforceable on the set of smaller transactions, and what were previously intermediate states can suddenly be accessed indiscriminately by all other transactions, making it much harder to reason about the correctness of one's application.

The approach that we propose to move beyond today's stark choices is based on two propositions: first, that the coupling between atomicity and isolation should be loosened, so that providing isolation at a fine granularity does not necessarily result in shattering atomicity; and second, that the choice between either enduring poor performance or allowing indiscriminate access to intermediate states by all transactions is a false one: instead, complexity can be tamed by giving developers control over who is allowed to access these intermediate states, and when.

To enact these propositions, the Salt distributed database introduces a new abstraction: *BASE transactions*. The design of BASE transactions borrows from nested transactions [41], an abstraction originally introduced to offer, for long-running transactions, atomicity at a finer granularity than isolation. In particular, while most nested transaction implementations define isolation at the granularity of the parent ACID transaction,[1] they tune the mechanism for enforcing atomicity so that errors that occur within a nested subtransaction do not require undoing the entire parent transaction, but only the affected subtransaction.

Our purpose in introducing BASE transactions is similar in spirit to that of traditional nested transactions: both abstractions aim at gently loosening the coupling between atomicity and isolation. The issue that BASE transactions address, however, is the flip side of the one tackled by nested transactions: this time, the challenge is to provide *isolation* at a finer granularity, without either drastically escalating the complexity of reasoning about the application, or shattering atomicity.

## 4 BASE transactions

Syntactically, a BASE transaction is delimited by the familiar `begin BASE transaction` and `end BASE transaction` statements. Inside, a BASE transaction contains a sequence of *alkaline* subtransactions—nested transactions that owe their name to the novel way in which they straddle the ACID/BASE divide.

When it comes to the granularity of atomicity, as we will see in more detail below, a BASE transaction provides the same flexibility of a traditional nested transaction: it can limit the effects of a failure within a single alkaline subtransaction, while at the same time it can ensure that the set of actions performed by all the alkaline subtransactions it includes is executed atomically. Where a BASE transaction fundamentally differs from a traditional nested transaction is in offering *Salt Isolation*, a new isolation property that, by supporting multiple granularities of isolation, makes it possible to control which internal states of a BASE transaction are externally accessible, and by whom. Despite this unprecedented flexibility, Salt guarantees that, when BASE and ACID transactions execute concurrently, ACID transactions retain, with respect to all other transactions (whether BASE, alkaline, or ACID), the same isolation guarantees they used to enjoy in a purely ACID environment. The topic of how Salt isolation supports ACID transactions across all levels of isolation defined in the ANSI/ISO SQL standard is actually interesting enough that we will devote the entire next section to it. To prevent generality from obfuscating intuition, however, the discussion in the rest of this section assumes ACID transactions that provide the popular *read-committed* isolation level.

Independent of the isolation provided by ACID transactions, a BASE transaction's basic unit of isolation are the alkaline subtransactions it contains. Alkaline subtransactions retain the properties of ACID transactions: in particular, when it comes to isolation, no transaction (whether ACID, BASE or alkaline) can observe intermediate states produced by an uncommitted alkaline subtransaction. When it comes to observing the state produced by a *committed* alkaline subtransaction, however, the guarantees differ depending on the potential observer.

- The committed state of an alkaline subtransaction is observable by other BASE or alkaline subtransactions. By leveraging this finer granularity of isolation, BASE transactions can achieve levels of performance and availability that elude ACID transactions. At the same time, because alkaline subtransactions are isolated from each other, this design limits the new interleavings that programmers need to worry about when reasoning about the correctness of their programs: the only internal states of BASE transactions that become observable are those at the boundaries between its nested alkaline subtransactions.
- The committed state of an alkaline subtransaction is *not* observable by other ACID transactions until the parent BASE transaction commits. The internal state of a BASE transaction is then completely opaque to ACID transactions: to them, a BASE transaction looks

[1] *Nested top-level transactions* are a type of nested transactions that instead commit or abort independently of their parent transaction. They are seldom used, however, precisely because they violate the isolation of the parent transaction, making it hard to reason about consistency invariants.

```
1   // BASE transaction: transfer
2   begin BASE transaction
3     try
4       begin alkaline−subtransaction
5         Select bal into @bal from accnts where id = sndr
6         if (@bal >= amt)
7           Update accnts set bal −= amt where id = sndr
8       end alkaline−subtransaction
9     catch (Exception e) return // do nothing
10    if (@bal < amt) return // constraint violation
11    try
12      begin alkaline−subtransaction
13        Update accnts set bal += amt where id = rcvr
14      end alkaline−subtransaction
15    catch (Exception e) //rollback if rcvr not found or timeout occurs
16      begin alkaline−subtransaction
17        Update accnts set bal += amt where id = sndr
18      end alkaline−subtransaction
19  end BASE transaction

21  // ACID transaction: total−balance (unmodified)
22  begin transaction
23    Select sum(bal) from accnts
24  commit
```

Fig. 2: A Salt implementation of the simple banking application

just like an ordinary ACID transaction, leaving their correctness unaffected.

To maximize performance, we expect that alkaline subtransactions will typically be partition-local transactions, but application developers are free, if necessary to enforce critical consistency conditions, to create alkaline subtransactions that touch multiple partitions and require a distributed commit.

Figure 2 shows how the simple banking application of Figure 1 might look when programmed in Salt. The first thing to note is what has *not* changed from the simple ACID implementation of Figure 1(a): Salt does not require any modification to the ACID `total-balance` transaction; only the performance-critical `transfer` operation is expressed as a new BASE transaction. While the complexity reduction may appear small in this simple example, our current experience with more realistic applications (such as Fusion Ticket, discussed in Section 8) suggests that Salt can achieve significant performance gains while leaving untouched most ACID transactions. Figure 2 also shows another feature of alkaline subtransactions: each is associated with an exception, which is caught by an application-specific handler in case an error is detected. As we will discuss in more detail shortly, Salt leverages the exceptions associated with alkaline subtransactions to guarantee the atomicity of the BASE transactions that enclose them.

There are two important events in the life of a BASE transaction: *accept* and *commit*. In the spirit of the BASE paradigm, BASE transactions, as in Lynx [44], are accepted as soon as their first alkaline subtransaction commits. The atomicity property of BASE transactions ensures that, once accepted, a BASE transaction will eventually commit, i.e., all of its operations will have successfully executed (or bypassed because of some exception) and their results will be persistently recorded.

To clarify further our vision for the abstraction that BASE transactions provide, it helps to compare their guarantees with those provided by ACID transactions

**Atomicity** Just like ACID transactions, BASE transactions guarantee that *either all the operations they contain will occur, or none will*. In particular, atomicity guarantees that all accepted BASE transactions will eventually commit. Unlike ACID transactions, BASE transactions can be aborted only if they encounter an error (such as a constraint violation or a node crash) *before* the transaction is accepted. Errors that occur after the transaction has been accepted do not trigger an automatic rollback: instead, they are handled using exceptions. The details of our Salt's implementation of atomicity are discussed in Section 6.

**Consistency** Chasing higher performance by splitting ACID transactions can increase exponentially the number of interleavings that must be considered when trying to enforce integrity constraints. Salt drastically reduces this complexity in two ways. First, Salt does not require all ACID transactions to be dismembered: non-performance-critical ACID transactions can be left unchanged. Second, Salt does not allow ACID transactions to observe states inside BASE transactions, cutting down significantly the number of possible interleavings.

**Isolation** Here, BASE and ACID transactions depart, as BASE transactions provide the novel *Salt Isolation* property, which we discuss in full detail in the next section. Appealingly, Salt Isolation on the one hand allows BASE transactions to respect the isolation property offered by the ACID transactions they may execute concurrently with, while on the other yields the opportunity for significant performance improvements. In particular, under Salt Isolation a BASE transaction *BT* appears to an ACID transaction just like another ACID transaction, but other BASE transactions can observe the internal states that exist at the boundaries between adjacent alkaline subtransactions in *BT*.

**Durability** BASE transactions provide the same durability property of ACID transactions and of many existing NoSQL systems: *Accepted BASE transactions are guaranteed to be durable*. Hence, developers need not worry about losing the state of accepted BASE transactions.

## 5 Salt isolation

Intuitively, our goal for Salt isolation is to allow BASE transactions to achieve high degrees of concurrency, while ensuring that ACID transactions enjoy well-defined isolation guarantees. Before taking on this challenge in earnest, however, we had to take two important preliminary steps.

The first, and the easiest, was to pick the concurrency control mechanism on which to implement Salt

| Isolation level | $\mathcal{L}$ | $\mathcal{S}$ |
|---|---|---|
| read-uncommitted | W | W |
| read-committed | W | R,W |
| repeatable-read | R,W | R,W |
| serializable | R,RR,W | R,RR,W |

Table 1: Conflicting type sets $\mathcal{L}$ and $\mathcal{S}$ for each of the four ANSI isolation levels. R = Read, RR = Range Read, W = Write.

| | ACID-R | ACID-W | alka-R | alka-W | saline-R | saline-W |
|---|---|---|---|---|---|---|
| ACID-R | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ |
| ACID-W | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| alka-R | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ |
| alka-W | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ |
| saline-R | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ |
| saline-W | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ |

Table 2: Conflict table for ACID, alkaline, and saline locks.

isolation. Our current design focuses on lock-based implementations rather than, say, optimistic concurrency control, because locks are typically used in applications that experience high contention and can therefore more readily benefit from Salt; also, for simplicity, we do not currently support multiversion concurrency control and hence snapshot isolation. However, there is nothing about Salt isolation that fundamentally prevents us from applying it to other mechanisms beyond locks.

The second step proved much harder. We had to crisply characterize what are exactly the isolation guarantees that we want our ACID transactions to provide. This may seem straightforward, given that the ANSI/ISO SQL standard already defines the relevant four isolation levels for lock-based concurrency: read-uncommitted, read-committed, repeatable read, and serializable. Each level offers stronger isolation than the previous one, preventing an increasingly larger prefix of the following sequence of undesirable phenomena: dirty write, dirty read, non-repeatable read, and phantom [18].

Where the challenge lies, however, is in preventing this diversity from forcing us to define four distinct notions of Salt isolation, one for each of the four ACID isolation levels. Ideally, we would like to arrive at a single, concise characterization of isolation in ACID systems that somehow captures all four levels, which we can then use to specify the guarantees of Salt isolation.

The key observation that ultimately allowed us to do so is that all four isolation levels can be reduced to a simple requirement: if two *operations* in different transactions[2] conflict, then the temporal dependency that exists between the earlier and the later of these operations must extend to the entire *transaction* to which the earlier operation belongs. Formally:

**Isolation.** *Let $Q$ be the set of operation types {read, range-read, write} and let $\mathcal{L}$ and $\mathcal{S}$ be subsets of $Q$. Further, let $o_1$ in $txn_1$ and $o_2$ in $txn_2$, be two operations, respectively of type $T_1 \in \mathcal{L}$ and $T_2 \in \mathcal{S}$, that access the same object in a conflicting (i.e. non read-read) manner. If $o_1$ completes before $o_2$ starts, then $txn_1$ must decide before $o_2$ starts.*

With this single and concise formulation, each of the ACID isolation levels can be expressed by simply instantiating appropriately $\mathcal{L}$ and $\mathcal{S}$. For example, $\mathcal{L} = \{write\}$ and $\mathcal{S} = \{read, write\}$ yields read-committed isolation. Table 1 shows the conflicting sets of operation types for all four ANSI isolation levels. For a given $\mathcal{L}$ and $\mathcal{S}$, we will henceforth say that two transactions are *isolated* from each other when Isolation holds between them.

Having expressed the isolation guarantees of ACID transactions, we are ready to tackle the core technical challenge ahead of us: defining an isolation property for BASE transactions that allows them to harmoniously coexist with ACID transactions. At the outset, their mutual affinity may appear dubious: to deliver higher performance, BASE transactions need to expose intermediate uncommitted states to other transactions, potentially harming Isolation. Indeed, the key to Salt isolation lies in controlling which, among BASE, ACID, and alkaline subtransactions, should be exposed to what.

### 5.1 The many grains of Salt isolation

Our formulation of Salt isolation leverages the conciseness of the Isolation property to express its guarantees in a way that applies to all four levels of ACID isolation.

**Salt Isolation.** *The Isolation property holds as long as (a) at least one of $txn_1$ and $txn_2$ is an ACID transaction or (b) both $txn_1$ and $txn_2$ are alkaline subtransactions.*

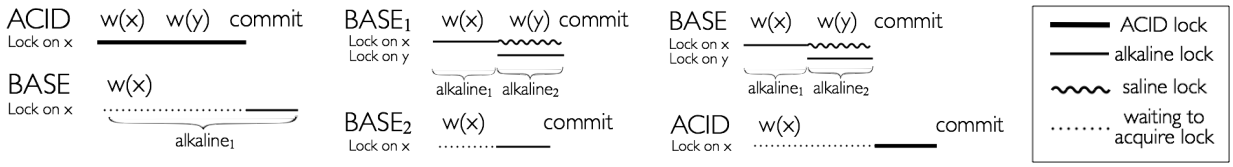
Informally, Salt isolation enforces the following constraint gradation:

- ACID transactions are isolated from all other transactions.
- Alkaline subtransactions are isolated from other ACID and alkaline subtransactions.
- BASE transactions expose their intermediate states (i.e. states produced at the boundaries of their alkaline subtransactions) to every other BASE transaction.

Hence, despite its succinctness, Salt isolation must handle quite a diverse set of requirements. To accomplish this, it uses a single mechanism—locks—but equips each type of transaction with its own type of lock: *ACID* and *alkaline* locks, which share the name of their respective transactions, and *saline* locks, which are used by BASE transactions.

**ACID locks** work as in traditional ACID systems. There are ACID locks for both read and write operations; reads

[2]Here the term *transactions* refers to both ACID and BASE transactions, as well as alkaline subtransactions.




(a) *BASE* waits until *ACID* commits. (b) $BASE_2$ waits only for $alkaline_1$... (c) ...but *ACID* must wait all of *BASE* out.

Fig. 3: Examples of concurrent executions of ACID and BASE transactions in Salt.

conflict with writes, while writes conflict with both reads and writes (see the dark-shaded area of Table 2). The duration for which an ACID lock is held depends on the operation type and the chosen isolation level. Operations in $\mathcal{L}$ require *long-term* locks, which are acquired at the start of the operation and are maintained until the end of the transaction. Operations in $\mathcal{S} \setminus \mathcal{L}$ require *short-term* locks, which are only held for the duration of the operation.

**Alkaline locks** keep alkaline subtransactions isolated from other ACID and alkaline subtransactions. As a result, as Table 2 (light-and-dark shaded subtable) shows, only read-read accesses are considered non-conflicting for any combination of ACID and alkaline locks. Similar to ACID locks, alkaline locks can be either long-term or short-term, depending on the operation type; long-term alkaline locks, however, are only held until the end of the current alkaline subtransaction, and not for the entire duration of the parent BASE transaction: their purpose is solely to isolate the alkaline subtransaction containing the operation that acquired the lock.

**Saline locks** owe their name to their delicate charge: isolating ACID transactions from BASE transactions, while at the same time allowing for increased concurrency by exposing intermediate states of BASE transactions to other BASE transactions. To that end, (see Table 2) saline locks conflict with ACID locks for non read-read accesses, but *never* conflict with either alkaline or saline locks. Once again, there are long-term and short-term saline locks: short-term saline locks are released after the operation completes, while long-term locks are held until the end of the current BASE transaction. In practice, since alkaline locks supersede saline locks, we acquire only an alkaline lock at the start of the operation and, if the lock is longterm, "downgrade" it at the end of the alkaline subtransaction to a saline lock, to be held until after the end of the BASE transaction.

Figure 3 shows three simple examples that illustrate how ACID and BASE transactions interact. In Figure 3(a), an ACID transaction holds an ACID lock on $x$, which causes the BASE transaction to wait until the ACID transaction has committed, before it can acquire the lock on $x$. In Figure 3(b), instead, transaction $BASE_2$ need only wait until the end of $alkaline_1$, before acquiring the lock on $x$. Finally, Figure 3(c) illustrates the use of saline locks. When $alkaline_1$ commits, it downgrades its lock on $x$ to a saline lock that is kept until the end of the parent BASE transaction, ensuring that the ACID and BASE transactions remain isolated.

**Indirect dirty reads** In an ACID system the Isolation property holds among any two transactions, making it quite natural to consider only direct interactions between pairs of transactions when defining the undesirable phenomena prevented by the four ANSI isolation levels. In a system that uses Salt isolation, however, the Isolation property covers only *some* pairs of transactions: pairs of BASE transactions are exempt. Losing Isolation's universal coverage has the insidious effect of introducing indirect instances of those undesirable phenomena.

The example in Figure 4 illustrates what can go wrong if Salt Isolation is enforced naively. For concreteness, assume that ACID transactions require a read-committed isolation level. Since Isolation is not enforced between $BASE_1$ and $BASE_2$, $w(y)$ may reflect the value of $x$ that was written by $BASE_1$. Although Isolation is enforced between $ACID_1$ and $BASE_2$, $ACID_1$ ends up reading $x$'s uncommitted value, which violates that transaction's isolation guarantees.

The culprit for such violations is easy to find: dirty reads can indirectly relate two transactions ($BASE_1$ and $ACID_1$ in Figure 4) without generating a direct conflict between them. Fortunately, none of the other three phenomena that ACID isolation levels try to avoid can do the same: for such phenomena to create an indirect relation between two transactions, the transactions at the two ends of the chain must be in direct conflict.

Our task is then simple: we must prevent indirect dirty reads.[3] Salt avoids them by restricting the order in which saline locks are released, in the following two ways:

**Read-after-write across transactions** A BASE transaction $B_r$ that reads a value $x$, which has been written by another BASE transaction $B_w$, cannot release its saline lock on $x$ until $B_w$ has released its own saline lock on $x$.

**Write-after-read within a transaction** An operation $o_w$ that writes a value $x$ cannot release its saline

[3] Of course, indirect dirty reads are allowed if ACID transactions require the read-uncommitted isolation level, which does not try to prevent dirty-reads.

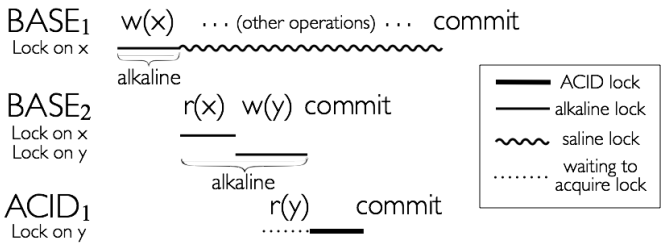



Fig. 4: $ACID_1$ indirectly reads the uncommitted value of $x$.

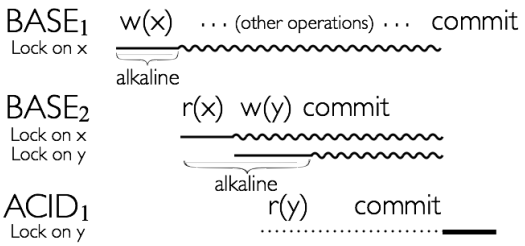



Fig. 5: How Salt prevents indirect dirty reads.

lock on $x$ until *all* previous read operations within the same BASE transaction have released their saline locks on their respective objects.[4]

The combination of these two restrictions ensures that, as long as a write remains uncommitted (i.e. its saline lock has not been released) subsequent read operations that observe that written value and subsequent write operations that are affected by that written value will not release their own saline locks. This, in turn, guarantees that an ACID transaction cannot observe an uncommitted write, since saline locks are designed to be mutually exclusive with ACID locks. Figure 5 illustrates how enforcing these two rules prevents the indirect dirty read of Figure 4. Observe that transaction $BASE_2$ cannot release its saline lock on $x$ until $BASE_1$ commits (read-after-write across transactions) and $BASE_2$ cannot release its saline lock on $y$ before releasing its saline lock on $x$ (write-after-read within a transaction).

We can now prove [43] the following Theorem for any system composed of ACID and BASE transactions that enforces Salt Isolation.

**Theorem 1.** *[Correctness] Given isolation level $\mathcal{A}$, all ACID transactions are protected (both directly and, where applicable, indirectly) from all the undesirable phenomena prevented by $\mathcal{A}$.*

**Clarifying serializability** The strongest ANSI lock-based isolation level, *locking-serializable* [18], not only prevents the four undesirable phenomena we mentioned earlier, but, in ACID-only systems, also implies the familiar definition of serializability, which requires the outcome of a serializable transaction schedule to be equal to the outcome of a serial execution of those transactions.

This implication, however, holds only if *all* transactions are isolated from all other transactions [18]; this is not desirable in a Salt database, since it would require isolating BASE transactions from each other, impeding Salt's performance goals.

Nonetheless, a Salt database remains true to the essence of the locking-serializable isolation level: it continues to protect its ACID transactions from all four undesirable phenomena, with respect to both BASE transactions and other ACID transactions. In other words, even though the presence of BASE transactions prevents the familiar notion of serializability to "emerge" from universal pairwise locking-serializability, ACID transactions enjoy in Salt the same kind of "perfect isolation" they enjoy in a traditional ACID system.

[4] A write can indirectly depend on any previous read within the same transaction, through the use of transaction-local variables.

## 6 Implementation

We implemented a Salt prototype by modifying MySQL Cluster [9], a popular distributed database, to support BASE transactions and enforce Salt Isolation. MySQL Cluster follows a standard approach among distributed databases: the database is split into a number of partitions and each partition uses a master-slave protocol to maintain consistency among its replicas, which are organized in a chain. To provide fairness, MySQL Cluster places operations that try to acquire locks on objects in a per-object queue in lock-acquisition order; Salt leverages this mechanism to further ensure that BASE transactions cannot cause ACID transactions to starve.

We modified the locking module of MySQL Cluster to add support for alkaline and saline locks. These modifications include support for (a) managing lock conflicts (see Table 2), (b) controlling when each type of lock should be acquired and released, as well as (c) a queuing mechanism that enforces the order in which saline locks are released, to avoid indirect dirty reads. Our current prototype uses the read-committed isolation level, as it is the only isolation level supported by MySQL Cluster. The rest of this section discusses the implementation choices we made with regard to availability, durability and consistency, as well as an important optimization we implemented in our prototype.

### 6.1 Early commit for availability

To reduce latency and improve availability, Salt supports *early commit* [44] for BASE transactions: a client that issues a BASE transaction is notified that the transaction has committed when its first alkaline subtransaction commits. To ensure both atomicity and durability despite

failures, Salt logs the logic for the entire BASE transaction before its first transaction commits. If a failure occurs before the BASE transaction has finished executing, the system uses the log to ensure that the entire BASE transaction will be executed eventually.

### 6.2 Failure recovery

Logging the transaction logic before the first alkaline subtransaction commits has the additional benefit of avoiding the need for managing cascading rollbacks of other committed transactions in the case of failures. Since the committed state of an alkaline subtransaction is exposed to other BASE transactions, rolling back an uncommitted BASE transaction would also require rolling back any BASE transaction that may have observed rolled back state. Instead, early logging allows Salt to roll uncommitted transactions forward.

The recovery protocol has two phases: *redo* and *roll forward*. In the first phase, Salt replays its redo log, which is populated, as in ACID systems, by logging asynchronously to disk every operation after it completes. Salt's redo log differs from an ACID redo log in two ways. First, Salt logs both read and write operations, so that transactions with write operations that depend on previous reads can be rolled forward. Second, Salt replays also operations that belong to partially executed BASE transactions, unlike ACID systems that only replay operations of committed transactions. During this phase, Salt maintains a *context* hash table with all the replayed operations and returned values (if any), to ensure that they are not re-executed during the second phase.

During the second phase of recovery, Salt rolls forward any partially executed BASE transactions. Using the logged transaction logic, Salt regenerates the transaction's query plan and reissues the corresponding operations. Of course, some of those operations may have already been performed during the first phase: the *context* hash table allows Salt to avoid re-executing any of these operations and nonetheless have access to the return values of any read operation among them.

### 6.3 Transitive dependencies

As we discussed in Section 5.1, Salt needs to monitor transitive dependencies that can cause indirect dirty reads. To minimize bookkeeping, our prototype does not explicitly track such dependencies. Instead it only tracks *direct* dependencies among transactions and uses this information to infer the order in which locks should be released.

As we mentioned earlier, MySQL Cluster maintains a per-object queue of the operations that try to acquire locks on an object. Salt adds for each saline lock a pointer to the most recent non-ACID lock on the queue. Before releasing a saline lock, Salt simply checks whether the pointer points to a held lock—an O(1) operation.

```
1  begin BASE transaction
2      Check whether all items exist. Exit otherwise.
3      Select w_tax into @w_tax from warehouse where w_id = : w_id;
4      begin alkaline−subtransaction
5          Select d_tax into @d_tax, next_order_id into @o_id from
               district where w_id = : w_id and d_id = : d_id;
6          Update district set next_order_id = o_id + 1 where w_id =
               : w_id AND d_id = : d_id;
7      end alkaline−subtransaction
8      Select discount into @discount, last_name into @name, credit
           into @credit where w_id = : w_id and d_id = : d_id and
           c_id = : c_id
9      Insert into orders values (: w_id, : d_id, @o_id, ...);
10     Insert into new_orders values (: w_id, : d_id, o_id);
11     For each ordered item, insert an order line, update stock level, and
           calculate order total
12 end BASE transaction
```

Fig. 6: A Salt implementation of the *new-order* transaction in TPC-C. The lines introduced in Salt are shaded.

### 6.4 Local transactions

Converting an ACID transaction into a BASE transaction can have significant impact on performance, beyond the increased concurrency achieved by enforcing isolation at a finer granularity. In practice, we find that although most of the performance gains in Salt come from fine-grain isolation, a significant fraction is due to a practical reason that compounds those gains: alkaline subtransactions in Salt tend to be small, often containing a single operation.

Salt's *local-transaction* optimization, inspired by similar optimizations used in BASE storage systems, leverages this observation to significantly decrease the duration that locks are being held in Salt. When an alkaline subtransaction consists of a single operation, each partition replica can locally decide to commit the transaction—and release the corresponding locks—immediately after the operation completes. While in principle a similar optimization could be applied also to single-operation ACID transactions, in practice ACID transactions typically consist of many operations that affect multiple database partitions. Reaching a decision, which is a precondition for lock release, typically takes much longer in such transactions: locks must be kept while each transaction operation is propagated along the entire chain of replicas of each of the partitions touched by the transaction and during the ensuing two-phase commit protocol among the partitions. The savings from this optimization can be substantial: single-operation transactions release their locks about one-to-two orders of magnitude faster than non-optimized transactions.[5] Interestingly, these benefits can extend beyond single operation transactions—it is easy to extend the local-transaction optimization to cover also transactions where all operations touch the same object.

[5]This optimization applies only to ACID and alkaline locks. To enforce isolation between ACID and BASE transactions, saline locks must still be kept until the end of the BASE transaction.

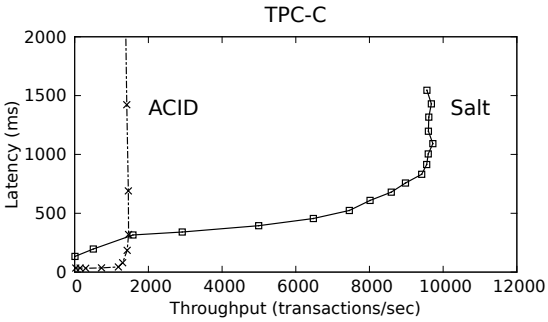



Fig. 7: Performance of ACID and Salt for TPC-C.

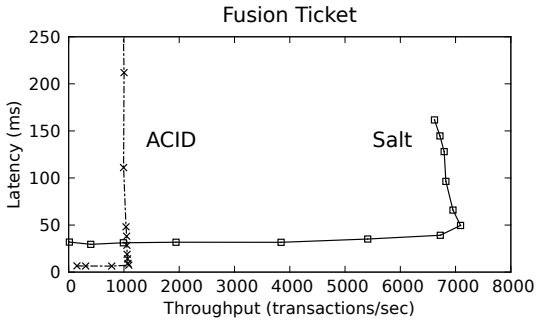



Fig. 8: Performance of ACID and Salt for Fusion Ticket.

## 7 Case Study: BASE-ifying *new-order*

We started this project to create a distributed database where performance and ease of programming could go hand-in-hand. How close does Salt come to that vision? We will address this question quantitatively in the next section, but some qualitative insight can be gained by looking at an actual example of Salt programming.

Figure 6 shows, in pseudocode, the BASE-ified version of *new-order*, one of the most heavily run transactions in the TPC-C benchmark (more about TPC-C in the next section). We chose *new-order* because, although its logic is simple, it includes all the features that give Salt its edge.

The first thing to note is that BASE-ifying this transaction in Salt required only minimal code modifications (the highlighted lines 2, 4, and 7). The reason, of course, is Salt isolation: the intermediate states of *new-order* are isolated from all ACID transactions, freeing the programmer from having to reason about all possible interleavings. For example, TPC-C also contains the *deliver* transaction, which assumes the following invariant: if an order is placed (lines 9-10), then all order lines must be appropriately filled (line 11). Salt does not require any change to *deliver*, relying on Salt isolation to ensure that *deliver* will never see an intermediate state of *new-order* in which lines 9-10 are executed but line 11 is not.

At the same time, using a finer granularity of isolation between BASE transactions greatly increases concurrency. Consider lines 5-6, for example. They need to be isolated from other instances of *new-order* to guarantee that order ids are unique, but this need for isolation does not extend to the following operations of the transaction. In an ACID system, however, there can be no such distinction; once the operations in lines 5-6 acquire a lock, they cannot release it until the end of the transaction, preventing lines 8-11 from benefiting from concurrent execution.

## 8 Evaluation

To gain a quantitative understanding of the benefits of Salt with respect to both ease of programming and performance, we applied the ideas of Salt to two applications: the TPC-C benchmark [23] and Fusion Ticket [6].
**TPC-C** is a popular database benchmark that models online transaction processing. It consists of five types of transactions: *new-order* and *payment* (each responsible for 43.5% of the total number of transactions in TPC-C), as well as *stock-level*, *order-status*, and *delivery* (each accounting for 4.35% of the total).
**Fusion Ticket** is an open source ticketing solution used by more than 80 companies and organizations [3]. It is written in PHP and uses MySQL as its backend database.

Unlike TPC-C, which focuses mostly on performance and includes only a representative set of transactions, a real application like Fusion Ticket includes several transactions—from frequently used ones such as *create-order* and *payment*, to infrequent administrative transactions such as *publishing* and *deleting-event*—that are critical for providing the required functionality of a fully fledged online ticketing application and, therefore, offers a more accurate view of the programming effort required to BASE-ify entire applications in practice.

Our evaluation tries to answer three questions:

- What is the performance gain of Salt compared to the traditional ACID approach?
- How much programming effort is required to achieve performance comparable to that of a pure BASE implementation?
- How is Salt's performance affected by various workload characteristics, such as contention ratio?

We use TPC-C and Fusion Ticket to address the first two questions. To address the third one, we run a microbenchmark and tune the appropriate workload parameters.

**Experimental setup** In our experiments, we configure Fusion Ticket with a single event, two categories of tick-

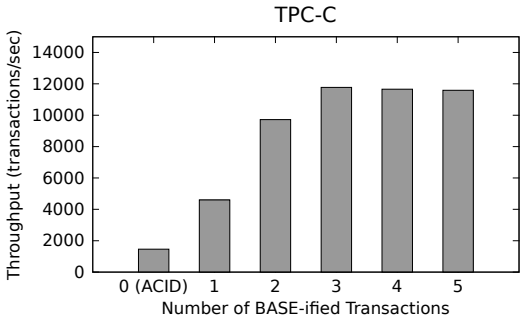



Fig. 9: Incremental BASE-ification of TPC-C.

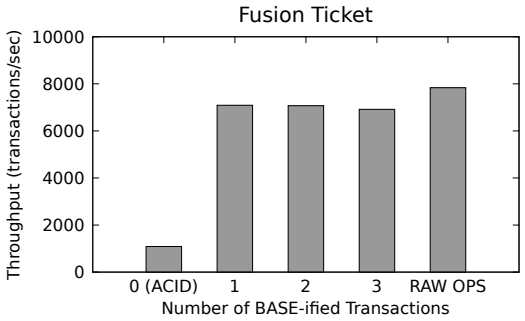



Fig. 10: Incremental BASE-ification of Fusion Ticket.

ets, and 10,000 seats in each category. Our experiments emulate a number of clients that book tickets through the Fusion Ticket application. Our workload consists of the 11 transactions that implement the business logic necessary to book a ticket, including a single administrative transaction, *delete-order*. We do not execute additional administrative transactions, because they are many orders of magnitude less frequent than customer transactions and have no significant effect on performance. Note, however, that executing more administrative transactions would have incurred no additional programming effort, since Salt allows unmodified ACID transactions to safely execute side-by-side the few performance-critical transactions that need to be BASE-ified. In contrast, in a pure BASE system, one would have to BASE-ify *all* transactions, administrative ones included: the additional performance benefits would be minimal, but the programming effort required to guarantee correctness would grow exponentially.

In our TPC-C and Fusion Ticket experiments, data is split across ten partitions and each partition is three-way replicated. Due to resource limitations, our microbenchmark experiments use only two partitions. In addition to the server-side machines, our experiments include enough clients to saturate the system.

All of our experiments are carried out in an Emulab cluster [16, 42] with 62 Dell PowerEdge R710 machines. Each machine is equipped with a 64-bit quad-core Xeon E5530 processor, 12GB of memory, two 7200 RPM local disks, and a Gigabit Ethernet port.

### 8.1 Performance of Salt

Our first set of experiments aims at comparing the performance gain of Salt to that of a traditional ACID implementation to test our hypothesis that BASE-ifying only a few transactions can yield significant performance gains.

Our methodology for identifying which transactions should be BASE-ified is based on a simple observation: since Salt targets performance bottlenecks caused by contention, transactions that are good targets for BASE-ification are large and highly-contented. To identify suitable candidates, we simply increase the system load and observe which transactions experience a disproportionate increase in latency.

Following this methodology, for the TPC-C benchmark we BASE-ified two transactions: *new-order* and *payment*. As shown in Figure 7, the ACID implementation of TPC-C achieves a peak throughput of 1464 transactions/sec. By BASE-ifying these two transactions, our Salt implementation achieves a throughput of 9721 transactions/sec—6.6$x$ higher than the ACID throughput.

For the Fusion Ticket benchmark, we only BASE-ify one transaction, *create-order*. This transaction is the key to the performance of Fusion Ticket, because distinct instances of *create-order* heavily contend with each other. As Figure 8 shows, the ACID implementation of Fusion Ticket achieves a throughput of 1088 transactions/sec, while Salt achieves a throughput of 7090 transactions/sec, 6.5$x$ higher than the ACID throughput. By just BASE-ifying *create-order*, Salt can significantly reduce how long locks are held, greatly increasing concurrency.

In both the TPC-C and Fusion Ticket experiments Salt's latency under low load is higher than that of ACID. The reason for this disparity lies in how requests are made durable. The original MySQL Cluster implementation returns to the client *before* the request is logged to disk, providing no durability guarantees. Salt, instead, requires that all BASE transactions be durable before returning to the client, increasing latency. This increase is exacerbated by the fact that we are using MySQL Cluster's logging mechanism, which—having been designed for asynchronous logging—is not optimized for low latency. Of course, this phenomenon only manifests when the system is under low load; as the load increases, Salt's performance benefits quickly materialize: Salt outperforms ACID despite providing durability guarantees.

### 8.2 Programming effort vs Throughput

While Salt's performance over ACID is encouraging, it is only one piece of the puzzle. We would like to further understand how much programming effort is required to achieve performance comparable to that of a pure BASE

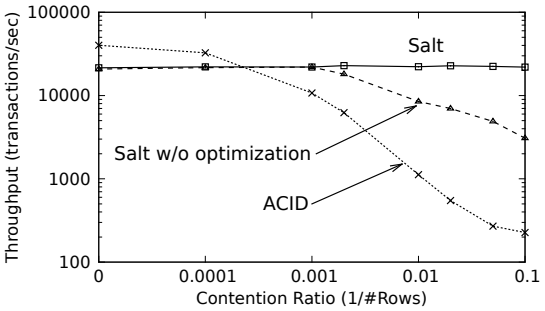



Fig. 11: Effect of contention ratio on throughput.

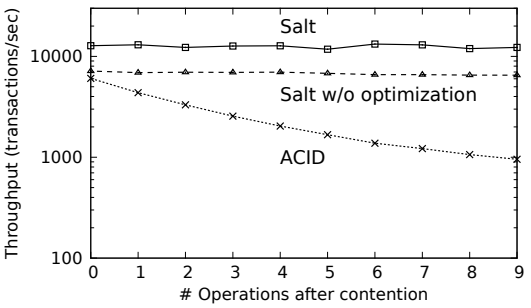



Fig. 12: Effect of contention position on throughput.

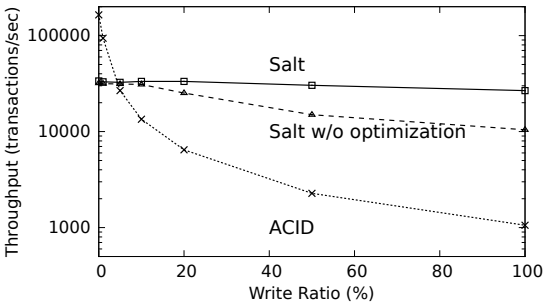



Fig. 13: Effect of read-write ratio on throughput.

implementation—i.e. where all transactions are BASE-ified. To that end, we BASE-ified as many transactions as possible in both the TPC-C and Fusion Ticket codebases, and we measured the performance they achieve as we increase the number of BASE-ified transactions.

Figure 9 shows the result of incrementally BASE-ifying TPC-C. Even with only two BASE-ified transactions, Salt achieves 80% of the maximum throughput of a pure BASE implementation; BASE-ifying three transactions actually reaches that throughput. In other words, there is no reason to BASE-ify the remaining two transactions. In practice, this simplifies a developer's task significantly, since the number of state interleavings to be considered increases exponentially with each additional transactions that need to be BASE-ified. Further, real applications are likely to have proportionally fewer performance-critical transactions than TPC-C, which, being a performance benchmark, is by design packed with them.

To put this expectation to the test, we further experimented with incrementally BASE-ifying the Fusion Ticket application. Figure 10 shows the results of those experiments. BASE-fying one transaction was quite manageable: it took about 15 man-hours—without prior familiarity with the code—and required changing 55 lines of code, out of a total of 180,000. BASE-ifying this first transaction yields a benefit of 6.5$x$ over ACID, while BASE-ifying the next one or two transactions with the highest contention does not produce any additional performance benefit.

What if we BASE-ify more transactions? This is where the aforementioned exponential increase in state interleavings caught up with us: BASE-ifying a fourth or fifth transaction appeared already quite hard, and seven more transactions were waiting behind them in the Fusion Ticket codebase! To avoid this complexity and still test our hypothesis, we adopted a different approach: we broke down all 11 transactions into raw operations. The resulting system does not provide, of course, any correctness guarantees, but at least, by enabling the maximum degree of concurrency, it lets us measure the maximum throughput achievable by Fusion Ticket. The result of this experiment is labeled RAW OPS in Figure 10. We find it promising that, even by BASE-ifying only one transaction, Salt is within 10% of the upper bound of what is achievable with a BASE approach.

### 8.3 Contention

To help us understand how contention affects the performance of Salt, we designed three microbenchmarks to compare Salt, with and without the local-transaction optimization, to an ACID implementation.

In the first microbenchmark, each transaction updates five rows, randomly chosen from a collection of $N$ rows. By tuning $N$, we can control the amount of contention in our workload. Our Salt implementation uses BASE transactions that consist of five alkaline subtransactions—one for each update.

Figure 11 shows the result of this experiment. When there is no contention, the throughput of Salt is somewhat lower than that of ACID, because of the additional bookkeeping overhead of Salt (e.g., logging the logic of the entire BASE transaction). As expected, however, the throughput of ACID transactions quickly decreases as the contention ratio increases, since contending transactions cannot execute in parallel. The non-optimized version of Salt suffers from this degradation, too, albeit to a lesser degree; its throughput is up to an order of magnitude higher than that of ACID when the contention ratio is high. The reason for this increase is that BASE transactions contend on alkaline locks, which are only held for the duration of the current alkaline subtransactions and

are thus released faster than ACID locks. The optimized version of Salt achieves further performance improvement by releasing locks immediately after the operation completes, without having to wait for the operation to propagate to all replicas or wait for a distributed commit protocol to complete. This leads to a significant reduction in contention; so much so, that the contention ratio appears to have negligible impact on the performance of Salt.

The goal of the second microbenchmark is to help us understand the effect of the relative position of contending operations within a transaction on the system throughput. This factor can impact performance significantly, as it affects how long the corresponding locks must be held. In this experiment, each transaction updates ten rows, but only one of those updates contends with other transactions by writing to one row, randomly chosen from a collection of ten shared rows. We tune the number of operations that follow the contending operation within the transaction, and measure the effect on the system throughput.

As Figure 12 shows, ACID throughput steadily decreases as the operations that follow the contending operation increase, because ACID holds an exclusive lock until the transaction ends. The throughput of Salt, however, is not affected by the position of contending operations because BASE transactions hold the exclusive locks—alkaline locks—only until the end of the current alkaline subtransaction. Once again, the local-transaction optimization further reduces the contention time for Salt by releasing locks as soon as the operation completes.

The third microbenchmark helps us understand the performance of Salt under various read-write ratios. The read-write ratio affects the system throughput in two ways: (i) increasing writes creates more contention among transactions; and (ii) increasing reads increases the overhead introduced by Salt over traditional ACID systems, since Salt must log read operations, as discussed in Section 6. In this experiment each transaction either reads five rows or writes five rows, randomly chosen from a collection of 100 rows. We tune the percentage of read-only transactions and measure the effect on the system throughput.

As Figure 13 shows, the throughput of ACID decreases quickly as the fraction of writes increases. This is expected: write-heavy workloads incur a lot of contention, and when transactions hold exclusive locks for long periods of time, concurrency is drastically reduced. The performance of Salt, instead, is only mildly affected by such contention, as its exclusive locks are held for much shorter intervals. It is worth noting that, despite Salt's overhead of logging read operations, Salt outperforms ACID even when 95% of the transactions are read-only transactions.

In summary, our evaluation suggests that, by holding locks for shorter times, Salt can reduce contention and offer significant performance improvements over a traditional ACID approach, without compromising the isolation guarantees of ACID transactions.

## 9 Related Work

**ACID** Traditional databases rely on ACID's strong guarantees to greatly simplify the development of applications [7–9, 12–14, 22, 36]. As we noted, however, these guarantees come with severe performance limitations, because of both the coarse granularity of lock acquisitions and the need for performing a two-phase commit (2PC) protocol at commit time.

Several approaches have been proposed to improve the performance of distributed ACID transactions by eliminating 2PC whenever possible. H-Store [39], Granola [24], and F1 [38] make the observation that 2PC can be avoided for transactions with certain properties (e.g. partition-local transactions). Sagas [29] and Lynx [44] remark that certain large transactions can be broken down into smaller ones without affecting application semantics. Lynx uses static analysis to identify eligible transactions automatically. Our experience with TPC-C and Fusion Ticket, however, suggests that performance critical transactions are typically complex, making them unlikely to be eligible for such optimizations. Calvin [40] avoids using 2PC by predefining the order in which transactions should execute at each partition. To determine this order, however, one must be able to predict which partitions a transaction will access before the transaction is executed, which is very difficult for complex transactions. Additionally, using a predefined order prevents the entire system from making progress when any partition becomes unavailable.

**BASE** To achieve higher performance and availability, many recent systems have adopted the BASE approach [1, 15, 20, 21, 27, 34]. These systems offer a limited form of transactions that only access a single item.

To mitigate somewhat the complexity of programming in BASE, several solutions have been proposed to provide stronger semantics. ElasTraS [25], Megastore [17], G-Store [26], and Microsoft's Cloud SQL Server [19] provide ACID guarantees within a single partition or key group. G-Store and ElasTraS further allow dynamic modification of such key groups. These systems, however, offer no atomicity or isolation guarantees across partitions. Megastore further provides the option of using ACID transactions, but in an all-or-nothing manner: either all transactions are ACID or none of them are.

## 10 Conclusion

The ACID/BASE dualism has to date forced developers to choose between ease of programming and performance. Salt shows that this choice is a false one. Using the new abstraction of BASE transactions and a mechanism to properly isolate them from their ACID counterparts, Salt enables for the first time a tenable middle ground between the ACID and BASE paradigms; a middle ground where performance can be incrementally attained by gradually increasing the programming effort required. Our experience applying Salt to real applications matches the time-tested intuition of Pareto's law: a modest effort is usually enough to yield a significant performance benefit, offering a drama-free path to growth for companies whose business depends on transactional applications.

## Acknowledgements

Many thanks to our shepherd Willy Zwaenepoel and to the anonymous reviewers for their insightful comments. Lidong Zhou, Mike Dahlin, and Keith Marzullo provided invaluable feedback on early drafts of this paper, which would not have happened without the patience and support of the Utah Emulab team throughout our experimental evaluation. This material is based in part upon work supported by a Google Faculty Research Award and by the National Science Foundation under Grant Number CNS-1409555. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

## References

[1] Apache HBase. http://hbase.apache.org/.

[2] AuctionMark. http://hstore.cs.brown.edu/projects/auctionmark/.

[3] Current users of Fusion Ticket. http://www.fusionticket.com/hosting/our-customers.

[4] Dolibarr. http://www.dolibarr.org/.

[5] E-venement. http://www.e-venement.org/.

[6] Fusion Ticket. http://www.fusionticket.org.

[7] MemSQL. http://www.memsql.com/.

[8] Microsoft SQL Server. http://www.microsoft.com/sqlserver/.

[9] MySQL Cluster. http://www.mysql.com/products/cluster/.

[10] Ofbiz. http://ofbiz.apache.org/.

[11] Openbravo. http://www.openbravo.com/.

[12] Oracle Database. http://www.oracle.com/database/.

[13] Postgres SQL. http://www.postgresql.org/.

[14] SAP Hana. http://www.saphana.com/.

[15] SimpleDB. http://aws.amazon.com/simpledb/.

[16] Utah Emulab. http://www.emulab.net/.

[17] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[18] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. ACM.

[19] Philip A Bernstein, Istvan Cseri, Nishant Dani, Nigel Ellis, Ajay Kalhan, Gopal Kakivaya, David B Lomet, Ramesh Manne, Lev Novik, and Tomas Talius. Adapting Microsoft SQL Server for Cloud Computing. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1255–1263. IEEE, 2011.

[20] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, Berkeley, CA, USA, 2006. USENIX Association.

[21] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[22] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[23] Transaction Processing Performance Council. TPC benchmark C, Standard Specification Version 5.11, 2010.

[24] James Cowling and Barbara Liskov. Granola: Low-Overhead Distributed Transaction Coordination. In *Proceedings of the 2012 USENIX Annual Technical Conference*, Boston, MA, USA, June 2012. USENIX.

[25] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTraS: an elastic transactional data store in the cloud. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[26] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 163–174. ACM, 2010.

[27] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[28] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 78–91, New York, NY, USA, 1997. ACM.

[29] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1987.

[30] Seth Gilbert and Nancy Ann Lynch. Perspectives on the CAP Theorem. Institute of Electrical and Electronics Engineers, 2012.

[31] James N Gray. *Notes on data base operating systems*. Springer, 1978.

[32] Pat Helland. Life beyond Distributed Transactions: an Apostate's

Opinion. In *Third Biennial Conference on Innovative Data Systems Research*, pages 132–141, 2007.

[33] George Roy Hill and William Goldman. Butch Cassidy and the Sundance Kid. Clip at `https://www.youtube.com/watch?v=1IbStIb9XXw`, October 1969.

[34] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44:35–40, April 2010.

[35] Michael McLure. Vilfredo Pareto, 1906 Manuale di Economia Politica, Edizione Critica, Aldo Montesano, Alberto Zanni and Luigino Bruni (eds). *Journal of the History of Economic Thought*, 30(01):137–140, 2008.

[36] Michael A Olson, Keith Bostic, and Margo I Seltzer. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[37] Dan Pritchett. BASE: An Acid Alternative. *Queue*, 6:48–55, May 2008.

[38] Jeff Shute, Mircea Oancea, Stephan Ellner, Ben Handy, Eric Rollins, Bart Samwel, Radek Vingralek, Chad Whipkey, Xin Chen, Beat Jegerlehner, Kyle Littleeld, and Phoenix Tong. F1 - The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 777–778. ACM, 2012.

[39] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[40] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

[41] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[42] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[43] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt: Combining ACID and BASE in a Distributed Database (extended version). Technical Report TR-14-10, Department of Computer Science, The University of Texas at Austin, September 2014.

[44] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 276–291. ACM, 2013.