Representation without Taxation: A Uniform, Low-Overhead, and High-Level Interface to Eventually Consistent Key-Value Stores

KC Sivaramakrishnan*
University of Cambridge, UK
sk826@cl.cam.ac.uk

Gowtham Kaki
Purdue University, USA
gkaki@purdue.edu

Suresh Jagannathan
Purdue University, USA
suresh@cs.purdue.edu

Abstract

Geo-distributed web applications often favor high availability over strong consistency. In response to this bias, modern-day replicated data stores often eschew sequential consistency in favor of weaker eventual consistency (EC) data semantics. While most operations supported by a typical web application can be engineered, with sufficient care, to function under EC, there are oftentimes critical operations that require stronger consistency guarantees. A few off-the-shelf eventually consistent key-value stores offer tunable consistency levels to address the need for varying consistency guarantees. However, these consistency levels often have poorly-defined ad hoc *semantics that is usually too low-level from the perspective of an application to relate their guarantees to invariants that must be respected by the application. Moreover, these guarantees are often defined in way that is strongly influenced by a specific implementation of the data store. While such low-level implementation-dependent solutions do not readily cater to the high-level requirements of an application, relying on ill-defined guarantees additionally complicates the already hard task of reasoning about application semantics under eventual consistency.*

In this paper, we describe Quelea*, a declarative programming model for eventually consistent data stores. A novel aspect of* Quelea *is that it abstracts the actual implementation of the data store via high-level programming and system-level models that are agnostic to a specific implementation of the data store. By doing so,* Quelea *frees application programmers from having to reason about their application in terms of low-level implementation specific data store semantics. Instead, programmers can now reason in terms of an abstract model of the data store, and develop applications by defining and composing high-level replicated data types.* Quelea *is equipped with a formal specification language that is capable of expressing precise semantics of high-level consistency guarantees (*e.g.*, causal consistency) in the abstract model. Any eventually consistent key-value store can support* Quelea *by implementing a thin shim layer and a choosen set of high-level consistency guarantees on top of its existing low-level interface. We describe one such instantiation on top of Cassandra, that includes support for causal and sequential consistency guarantees, and coordination-free transactions. We present a case study of a large web application benchmark to demonstrate* Quelea*'s practical utility.*

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*This work was done at Purdue University, USA.

1 Introduction

Eventual consistency facilitates high availability, but can lead to surprising anomalies that have been well-documented [4, 15, 18, 8, 12]. While applications can often tolerate many of these anomalies, there are some that adversely effect the user experience, and hence need to be avoided. For instance, a social network application can tolerate out-of-order delivery of unrelated posts, but causally related posts need to be delivered in causal order; *e.g.,* a comment cannot be delivered before the post itself. The view count of a video on YouTube need not necessarily reflect the precise count of the number of views, but it should not appear to be decreasing. A bank account application may not always show the accurate balance in an account, but neither should it let the balance go below zero, nor should it admit operations that would lead it to display a negative balance.

Bare eventual consistency is often too weak to ensure such high-level application invariants; stronger consistency guarantees are needed. To help applications enforce such high-level invariants, off-the-shelf replicated data stores, such as Cassandra and Riak, offer tunable consistency levels on a per-operation basis: applications can specify the consistency level for every read and write operation they perform on the data store. However, consistency levels offered by these off-the-shelf stores are often defined at a very low-level. For example, consistency levels in Cassandra and Riak assume the values of ONE, TWO, QUORUM, ALL etc., describing how many physically distributed nodes comprising the store must respond before a read or write operation is declared successful. It is not immediately apparent what permutation of these low-level consistency guarantees would let the application enforce its high-level level invariants. For instance, what should be the consistency level of reads and writes to the `posts` table to guarantee causal order delivery of posts in the aforementioned social network application?

Furthermore, the semantics of low-level consistency guarantees are not uniform across different store implementations. For instance, while QUORUM means *strict quorum* (i.e., Lamport's quorum [11]) in the case of Cassandra, it means a *sloppy quorum* [8] in Riak. Complicating matters yet further, consistency semantics is often imprecisely, or even inaccurately, defined in the informal vendor-hosted documentations. For instance, Datastax's Cassandra documentation [7] claims that one can achieve "strong consistency" with "quorum reads and writes" in Cassandra. While this claim appears reasonable superficially (because a pair of quorum operations are serialized at least at one node), it is incomplete, at best, and inaccurate at worst.[1] Another example of a low-level consistency enforcement construct with vaguely defined semantics is Cassandra's Compare-and-Set (CAS) operation, which is advertised as a "lightweight transaction" and exposed as a conditional write query (*e.g.,* `INSERT INTO users VALUES ... IF NOT EXISTS`). The addition of CAS to Cassandra was coupled with the introduction of a new consistency level named SERIAL. Surprisingly, SERIAL is not a valid query-level consistency parameter for a write (conditional or not), while the others (*e.g.,* ONE) are valid.[2] Furthermore, Cassandra accepts a new *protocol-level* consistency parameter for a CAS operation that can be set to SERIAL, but its informal description doesn't explain how this parameter interacts with the query-level consistency parameter. The only way to unravel this complexity is to understand low-level details of the operator's underlying Paxos-based implementation. Mired in this quagmire of low-level implementation details, it is easy to lose track of our original goal - ensuring the high-level semantics guarantees required by the application are met as efficiently as possible by the implementation.

In this paper, we describe QUELEA, a declarative programming framework for eventually consistent data stores that was built to address the issues discussed above. QUELEA can be realized as a thin layer on top of any off-the-shelf eventually consistent key-value store, and as such, provides a uniform implementation-independent

[1] The devil is in the details of the timestamp-based last-writer-wins conflict resolution strategy in Cassandra, which need not necessarily pick the last writer due to inevitable clock drift across nodes. [9] presents a counterexample.

[2] Given the advertised use cases for lightweight transactions (such as maintaining uniqueness of usernames), one might expect a CAS to be SERIAL by default. It is therefore unintuitive that CAS accepts a consistency parameter, at least to the developers of *cassandra-cql*, a popular Haskell library for programming with Cassandra, whose API for CAS operation incorrectly hardcodes the parameter to SERIAL. This bug has been reported and fixed.

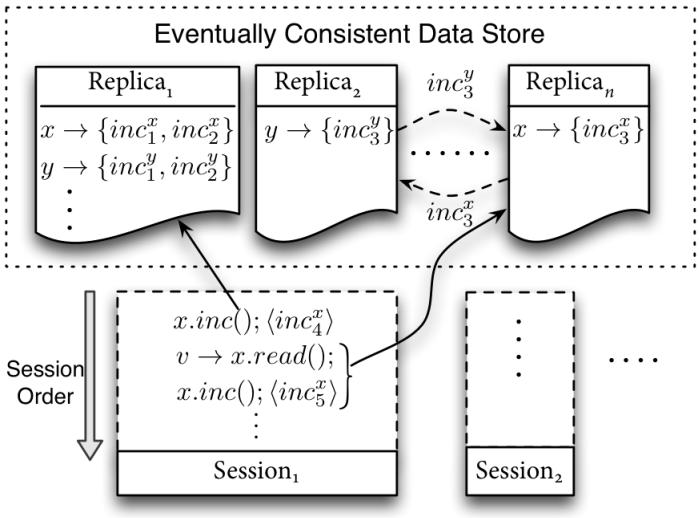


Figure 1: QUELEA system model.

interface to the data store. QUELEA programmers reason in terms of an abstract system model of an eventually consistent data store (ECDS), and any functionality offered by the store in addition to bare eventual consistency, including stronger consistency guarantees, transactions with tunable isolation levels etc., is required to have a well-defined semantics in the context of this abstract model. We show that various high-level consistency guarantees (eg., causal consistency) and various well-known isolation levels for transactions (eg., read committed) indeed enjoy such properties. QUELEA is additionally equipped with an expressive specification language that lets data store developers succinctly describe the semantics of the functionality they offer. A similar specification language is exposed to application programmers, who can declare the consistency requirements of their application as axiomatic specifications. Specifications are constructed using primitive consistency relations such as *visibility* and *session order* along with standard logical and relational operators. A novel aspect of QUELEA is that it can directly compare the specifications written by application programmers and data store developers since both are written within the same specification language, and can thus automatically map application requirements to the appropriate store-level guarantees. Consequently, QUELEA programmers can write portable code that automatically adapts to any data store that can express its functionality in terms of QUELEA's abstract system model.

Another key advantage of QUELEA is that it allows the addition of new replicated data types to the store, which obviates the need to support data types with application-specific semantics at the store level. Replicated data types (RDTs) are ordinary data structures, such as sets and counters, but with their state replicated across multiple replicas. As such, they offer useful high-level abstractions to build applications on top of weakly consistent replication. Weak consistency admits the possibility of concurrent conflicting updates to the state of the data structure at different replicas. The definition of an RDT must therefore specify its convergence semantics (i.e., *how* conflicting updates are resolved), along with its consistency properties (i.e., *when* updates become visible). QUELEA achieves a clean separation of concerns between these two aspects of the RDT definition, permitting *operational* reasoning for conflict resolution, and *declarative* reasoning for consistency. The combination of these techniques enhances overall programmability and simplifies reasoning about application correctness.

2 System Model

Figure 1 summarizes the abstract system model of a data store exposed to the QUELEA programmer. The store is a collection of *replicas*, each storing a set of *objects* $(x, y, \ldots)$ of a replicated data type. For the sake of an

example, let x and y represent objects of an *increment-only counter* replicated data type (RDT) that admits *inc* and *read* operations. The state of an RDT object is represented as the set of all updates (effectful operations, or simply *effects*) performed on the object. In Fig. 1, the state of x at replica 1 is the set $\{inc_1^x, inc_2^x\}$, where each inc_i^x denotes an *inc* effect on x.

Clients interact with the data store via concurrent *sessions*, where each session is a sequence of operations that a client invokes on any of the objects contained in the store. Note that clients have no control over which replica an operation is applied to; the data store may choose to route the operation to any replica in order to minimize latency, load balance, etc. For example, the *inc* and *read* operations invoked by the same session on the same object, may be applied on different replicas because replica 1 (to which the *inc* operation is applied, say) might be unreachable when the client invokes a subsequent *read*.

When an operation is applied to a replica, it is said to witness the state of its object at that replica. For example, *x.inc* applied to replica 1 witnesses the state of x as $\{inc_1^x, inc_2^x\}$. We say that the effects inc_1^x and inc_2^x are *visible* to the effect (inc_4^x) of *x.inc*, written logically as $\mathsf{vis}(inc_1^x, inc_4^x) \wedge \mathsf{vis}(inc_2^x, inc_4^x)$, where vis stands for the irreflexive and asymmetric visibility relation between effects over the same object. The notion of visibility is important since the result of an operation often depends on the set of visible effects[3]. For instance, a *read* on x applied to the last replica in Fig. 1 returns 1 since it only witnesses the effect (inc_3^x) of a single *x.inc* operation.

A visibility relation between two effects implies that the former operation has happened before the latter (since the latter has witnessed the effect of the former). However, visibility is not enough to capture a happens-before order between operations. As Fig. 1 demonstrates, a pair of operations from the same session, although one happens before the other, need not be visible to each other. To capture happens-before, we define an irreflexive transitive *session order* relation that relates the effects of operations arising from the same session. For example, in Fig. 1, inc_4^x and inc_5^x are in session order (written logically as $\mathsf{so}(inc_4^x, inc_5^x)$).

The effect added to a particular replica is asynchronously sent to other replicas, and eventually merged into all other replicas. Observe that this model is independent of the resolution strategy for concurrent conflicting updates, and instead preserves *every* update. Update conflicts are resolved when an operation reduces over the set of effects on an object at a particular replica. The model, however, admits all the inconsistencies associated with eventual consistency, some of which could adversely impact the usability of the application. We call such unacceptable inconsistencies as *anomalies*. Stronger consistency guarantees are needed to prevent unwanted anomalies.

In the next section we concretize, in Quelea, the *counter* application described informally above, followed by the anomalies the application admits under our model. Next, we show that strengthening the model with a few simple guarantees is enough to prevent these anomalies. We introduce a specification language that lets us naturally express such additional requirements. Finally, we show that well-known high-level consistency guarantees have precise semantics under our model, and hence can be expressed as formulas in our specification language. This makes it straightforward to compare application requirements with consistency guarantees, and determine the appropriate consistency semantics required to prevent anomalies.

3 Programming with Quelea

3.1 RDT Definition

Figure 2 shows the implementation of a counter RDT in Quelea. The data type `Ctr` represents the counter's effect type. Every RDT in Quelea is associated with an effect type that specifies the effects allowed on the objects of that type. An RDT is defined merely a list of its effects. The type of an RDT operation is an instance of the following type, written in Haskell syntax as:

```
type Operation e a r = [e] → a → (r, Maybe e)
```

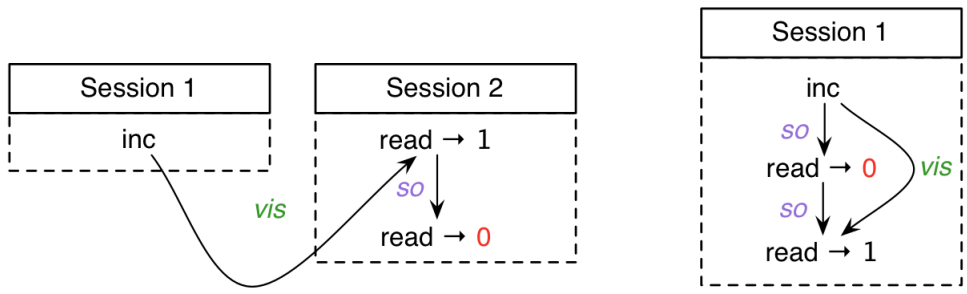
[3]We abuse the visibility relation by informally extending it to operations (including read-only operations, which produce no effects).

```haskell
-- CtrEff is the data type of counter effects.
-- Inc is its only inhabitant.
data CtrEff = Inc
-- Counter (rather, its state) is defined simply
-- as a list of counter effects.
type Counter = [CtrEff]
-- A read operation reads the value of the counter
-- by counting the number of Inc effects.
read :: Counter → () → (Int, Maybe Ctr)
read hist _ = (length hist, Nothing)
-- An inc operation simply generates an Inc effect.
inc :: Counter → () → ((), Maybe Ctr)
inc hist _ = ((), Just Inc)
```

Figure 2: Definition of a counter expressed in Quelea (Haskell syntax).

An operation on an RDT accepts a list of effects (the *history* of updates representing the state of the object at some replica), and an input argument, and returns a result along with an optional effect. While a read-only operation (eg., `read`) generates no effect (i.e., it returns `Nothing`), a write-only operation (eg., `inc`) returns a new effect.

3.2 Anomalies under Eventual Consistency



(a) Monotonic Violation (b) Missing Update

Figure 3: Anomalies possible under eventual consistency for the counter read operation.

Observe that the counter RDT does not admit a decrement operation. Therefore, the value of a counter should appear to be monotonically increasing. Indeed, this property is what makes the RDT useful to implement, for example, a video view counter on YouTube. Unfortunately, the monotonicity invariant can be violated in an eventually consistent execution.

Consider the execution shown in Figure 3a. Session 1 performs an `inc` operation on the counter, while Session 2 performs two `read` operations. The first `read` witnesses the effect of `inc` from Session 1, hence returns 1. The second `read`, however, does not witness `inc`, possibly because it was served by a replica that has not yet merged the `inc` effect. It returns 0, thus violating the monotonicity invariant.

In order for counter's value to appear monotonically increasing, the second `read` should also witness the effect of `inc`, because it was witnessed by the preceding `read`. Because eventually consistent `read` operations do not ensure this property, `read` operations need to be executed at a stronger consistency level. The choice of consistency level must guarantee that if a `read` witnesses an `inc` effect, all the subsequent `read` operations on the same counter object also witness that effect. If we let sameobj relate effects over the same object, we can

$$
\begin{array}{llll}
a, b \in \texttt{EffVar} & & \mathsf{Op} \in \texttt{OperName} & \\
\psi & \in \texttt{Spec} & ::= & \forall(a:\tau).\psi \mid \forall a.\psi \mid \pi \\
\tau & \in \texttt{EffType} & ::= & \mathsf{Op} \mid \tau \vee \tau \\
\pi & \in \texttt{Prop} & ::= & \mathsf{true} \mid R(a,b) \mid \pi \vee \pi \mid \pi \wedge \pi \mid \pi \Rightarrow \pi \\
R & \in \texttt{Relation} & ::= & \mathsf{vis} \mid \mathsf{so} \mid \mathsf{sameobj} \mid = \mid R \cup R \mid R \cap R \mid R^{+}
\end{array}
$$

Figure 4: Syntax (Context-Free Grammar) of the specification language.

formalize this requirement using visibility (vis) and session order (so) relations:

$$\forall(a:\texttt{inc}),(b,c:\texttt{read}).\ \mathsf{vis}(a,b) \ \wedge\ \mathsf{so}(b,c) \ \wedge\ \mathsf{sameobj}(b,c) \Rightarrow \mathsf{vis}(a,c)$$

The above formula is in fact a valid specification that can be associated with counter RDT operations in Quelea. Note that the specification captures the guarantees required to enforce the monotonicity invariant with respect to the abstract model of the store described in § 2. In particular, the specification does not refer to the low-level details of any specific data store. Our observation is that if consistency levels can also be specified in a similar manner, we can eliminate the need for the application programmer to understand the low-level nuances of the data store to enforce the required invariants; understanding their semantics in the abstract model is enough. In the following sections, we demonstrate that this is indeed possible. We first introduce our specification language.

3.3 Specification Language

The syntax of our specification language is shown in Figure 4. The language is based on first-order logic (FOL), and admits prenex universal quantification over typed and untyped effect variables. We use a special effect variable ($\hat{\eta}$) to denote the effect of the *current operation* - the operation for which a specification is being written. The type of an effect is simply the name of the operation (eg: `inc`) that induced the effect. We admit disjunction in types to let an effect variable range over multiple operation names. The specification $\forall(a : \tau_1 \vee \tau_2).\ \psi$ is just syntactic sugar for $\forall a.(\mathsf{oper}(a,\tau_1) \vee \mathsf{oper}(a,\tau_2)) \Rightarrow \psi$. An untyped effect variable ranges over all operation names.

The syntactic class of relations is seeded with primitive vis, so, and sameobj relations, and also admits derived relations that are expressible as union, intersection, or transitive closure of primitive relations. Commonly used derived relations are the *same object session order* ($\mathsf{soo} = \mathsf{so} \cap \mathsf{sameobj}$), *happens-before order* ($\mathsf{hb} = (\mathsf{so} \cup \mathsf{vis})^{+}$) and the *same object happens-before order* ($\mathsf{hbo} = (\mathsf{soo} \cup \mathsf{vis})^{+}$). For example, the same object session order (soo) can be used to succinctly represent the specification of counter RDT's consistency requirement:

$$\forall(a:\texttt{inc}),(b,c:\texttt{read}).\ \mathsf{vis}(a,b) \ \wedge\ \mathsf{soo}(b,c) \Rightarrow \mathsf{vis}(a,c)$$

Same object happens-before order (hbo) captures causal order among operations on same object. For example, if an `inc` is visible to a `read`, and the `read` precedes another `read` in session order (all operations on the same counter object), then the `inc` and the second `read` are related by hbo, although they may not be directly related by vis or soo. The causal relationship is transitive, via the first `read`. As such, hbo is useful to capture the causal consistency condition, which requires an operation to witness all the causally preceding operations perfomed on the same object (i.e., operations that precede the current operation in hbo). This condition is formalized in the following section.

3.4 Consistency Guarantees

To help programmers eliminate certain classes of anomalies in their applications, Terry *et al.* equip their data store Bayou [17] with four incomparable consistency levels called *session guarantees* [18]. While Terry *et al.*

realize efficient implementations of session guarantees making use of low-level properties of their store, the semantics of these guarantees can nonetheless can be captured succinctly within QUELEA's abstract model thus:

$$\begin{aligned}
\text{Read Your Writes (RYW)} &::= \forall a, b.\ \mathsf{soo}(a,b) \Rightarrow \mathsf{vis}(a,b)\\
\text{Monotonic Reads (MR)} &::= \forall a, b, c.\ \mathsf{vis}(a,b) \wedge \mathsf{soo}(b,c) \Rightarrow \mathsf{vis}(a,c)\\
\text{Monotonic Writes (MW)} &::= \forall a, b, c.\ \mathsf{soo}(a,b) \wedge \mathsf{vis}(b,c) \Rightarrow \mathsf{vis}(a,c)\\
\text{Writes Follow Reads (WFR)} &::= \forall a, b, c, d.\ \mathsf{vis}(a,b) \wedge \mathsf{vis}(c,d) \wedge (\mathsf{soo} \cup =)(b,c) \Rightarrow \mathsf{vis}(a,d)
\end{aligned}$$

Consider a Monotonic Reads (MR) session guarantee. The semantics of MR guarantees that if the effect of an operation a is visible to the effect of b, and b precedes c in (same object) session order, then a will also be made visible to c. Recall that this is precisely the guarantee required by the counter to enforce monotonicity. In fact, by restricting the bound variable a in MR's specification to range over `inc` effects, and bound variables b and c to range over `read` effects, we can easily conclude that executing `read` at an MR consistency level is sufficient to enforce the monotonicity invariant.

Like MR, the semantics of other session guarantees are categorically stated by their specifications. Read-Your-Writes (RYW), for example, guarantees that an operation (b) witnesses the effect of every preceding operation (a) in the session. A `read` operation executed at RYW consistency level therefore witnesses every previous `inc` operation from the same session. This guarantee is necessary to avoid the anomaly shown in Fig. 3b, where a `read` that succeeds an `inc` fails to witness the effect of `inc`, but a later `read` witnesses the effect. The anomaly can also be avoided by running `inc` and `read` under the QUORUM consistency level offered by some off-the-shelf key-value stores, but doing so requires non-trivial reasoning over the semantics of quorum operations (*e.g.*, strict/sloppy) and conflict resolution strategies (eg., LWW) to arrive at this conclusion. In contrast, reasoning with high-level consistency guarantees, such as MR and RYW, circumvents this complexity.

The precise characterization of guarantees as specifications facilitates the use of automatic analyses to determine if a consistency level meets application requirements. For instance, consider a data store that offers the following three consistency levels:

$$\begin{aligned}
\text{Causal Visibility (CV)} &::= \forall a, b, c.\ \mathsf{hbo}(a,b)\ \wedge\ \mathsf{vis}(b,c) \Rightarrow \mathsf{vis}(a,c)\\
\text{Causal Consistency (CC)} &::= \forall a, b, c.\ \mathsf{hbo}(a,b) \Rightarrow \mathsf{vis}(a,b)\\
\text{Strong Consistency (SC)} &::= \forall a, b.\ \mathsf{sameobj}(a,b) \Rightarrow \mathsf{vis}(a,b) \vee \mathsf{vis}(b,c)
\end{aligned}$$

It is not immediately apparent which among CV, CC and SC meet the requirements of a counter (CR). Fortunately, we can leverage the power of automated theorem provers (*e.g.*, Z3) to prove that the specification of CR is stronger than CV, but weaker than CC and SC, thus letting us deduce that counter's requirements can be met both under causal and strong consistency levels. A theorem prover can also be used to prove that among CC and SC, CC is weaker. Assuming that weaker guarantees incur lower cost to enforce availability, it is reasonable to conclude that counter's `read` operations should be executed under causal consistency to enforce monotonicity.

The analysis described informally above is formalized as a *classification scheme* [15] in QUELEA. This scheme completely automates the choice of consistency levels in QUELEA, thus eliminating the need for programmers to understand the semantics of different consistency levels. The ease of reasoning with precisely stated high-level guarantees demonstrates the advantage of exposing the functionality of the data store via QUELEA, as against the low-level ad hoc interfaces currently offered by many ECDS.

4 Transactions

Real-world applications often need to perform atomic operations that span multiple objects. An example is a `transfer` operation on bank accounts, which needs to perform a `withdraw` operation on one bank account, and a `deposit` operation on another. Transactions are usually the preferred means to compose sets of operations into a single atomic operation. However, a classical ACID transaction model requires inter-replica

coordination, leading to the loss of availability. To address this problem, several recent systems [16, 3, 1] have proposed *coordination-free transactions* that offer atomicity, remain available under network partitions, but only provide weaker isolation guarantees. Several variants of coordination-free transactions have subtly different isolation semantics and widely varying runtime overheads. Fortunately, the semantics of a large subset of such transactions can be captured elegantly in the abstract model of Quelea. Towards this end, we extend the contract language with a new term under quantifier-free propositions - txn S_1 S_2, where S_1 and S_2 are sets of effects, and which introduces a new primitive equivalence relation sametxn that holds for effects from the same transaction. txn$\{a,b\}\{c,d\}$ is just syntactic sugar for sametxn$(a,b) \wedge$ sametxn$(c,d) \wedge \neg$sametxn(a,c), where a and b considered to belong to the *current* transaction. Since atomicity is a defining characteristic of a transaction, we extend our formal model with the following axiom, which guarantees that a transaction is visible in its entirety, or it is not visible at all:

$$\forall a, b, c.\ \mathsf{txn}\{a,b\}\{c\} \wedge \mathsf{sameobj}(a,b) \wedge \mathsf{vis}(a,c) \Rightarrow \mathsf{vis}(b,c)$$

4.1 Isolation Requirements

§ 3.2 demonstrates how the consistency requirements of a counter RDT's `read` operation can be expressed in Quelea. In a similar manner, Quelea's specification language allows applications to declare isolation requirements for their transactions.

Consider an implementation of a bank account RDT in Quelea with three operations – `withdraw`, `deposit`, and `getBalance`, each with straightforward semantics. Additionally, we define two transactions – `save(amt)`, which transfers `amt` from current (c) to savings (s), and `totalBalance`, which returns the sum of the balances of individual accounts. Our goal is to ensure that `totalBalance` returns the result obtained from a consistent snapshot of the object states. The Quelea code for these transactions is given below:

```
save amt = atomically do
    b ← withdraw c amt
    -- b is true iff withdraw succeeds.
    when b $ deposit s amt
```

```
totalBalance = atomically do
    b1 ← getBalance c
    b2 ← getBalance s
    return b1 + b2
```

The `atomically` construct ensures that the effects of the operations are made visible atomically. However, atomicity itself is insufficient in this case, as it does not guarantee that both `getBalance` operations (in `totalBalance`) witness the effects of `save`. Consequently, `getBalance` on `s` may not witness the `deposit` on `s` from `save`, although `getBalance` on `c` witnesses the `withdraw` on `c`, resulting in `totalBalance` reporting an inconsistent balance.

Observe that the aforementioned anomaly can be averted by requiring that both `getBalance` operations in a `totalBalance` transaction witness the same set of `save` actions. This requirement can be captured succinctly in our specification language:

$$\forall(a,b : \texttt{getBalance}), (c : \texttt{withdraw} \vee \texttt{deposit}), (d : \texttt{withdraw} \vee \texttt{deposit}).$$
$$\mathsf{txn}\{a,b\}\{c,d\} \wedge \mathsf{vis}(c,a) \wedge \mathsf{sameobj}(d,b) \Rightarrow \mathsf{vis}(d,b)$$

The key idea in the above definition is to use the txn primitive to relate operations on different objects performed in a single transaction.

4.2 Isolation Guarantees

The isolation semantics of a transaction determines how the transaction witnesses the effects of previously committed transactions[4]. As mentioned previously, the isolation semantics of many variants of coordination-free transactions can be expressed in QUELEA's specification language. For demonstration, we pick three well-understood coordination-free transactions – Read Committed (RC) [2], Monotonic Atomic View (MAV) [1] and Repeatable Read (RR) [2], and express their isolation semantics in QUELEA.

ANSI RC isolation level guarantees that a transaction only witnesses the effects of committed transaction:

$$\forall a, b, c.\ \mathsf{txn}\{a\}\{b, c\} \wedge \mathsf{sameobj}(b, c) \wedge\ \mathsf{vis}(b, a) \Rightarrow \mathsf{vis}(c, a)$$

Note that RC is the dual of atomicity; it can be guaranteed with no additional effort if all transactions in the system are atomic.

MAV semantics ensures that if some operation in a transaction T_1 witnesses the effects of another transaction T_2, then subsequent operations in T_1 will also witness the effects of T_2:

$$\forall a, b, c, d.\ \mathsf{txn}\{a, b\}\{c, d\}\ \wedge\ \mathsf{so}(a, b)\ \wedge\ \mathsf{vis}(c, a)\ \wedge\ \mathsf{sameobj}(d, b) \Rightarrow \mathsf{vis}(d, b)$$

MAV semantics is useful for maintaining the integrity of foreign key constraints, materialized views and secondary updates [1].

ANSI RR semantics requires that the transaction witness a snapshot of the data store state. More concretely, RR requires that if an operation in transaction T_1 witnesses the effects of transaction T_2, then all the remaining operations should also witness the effects of T_2:

$$\forall a, b, c, d.\ \mathsf{txn}\{a, b\}\{c, d\}\ \wedge\ \mathsf{vis}(c, a)\ \wedge\ \mathsf{sameobj}(d, b) \Rightarrow \mathsf{vis}(d, b)$$

Note that this is precisely the semantics required by the `totalBalance` transaction to ensure that it returns a balance that reflects a consistent snapshot of the data store. Hence, it is sufficient to execute `totalBalance` at RR isolation level.

The ease of reasoning with precisely stated high-level guarantees thus extends to transactions as well. Furthermore, the ability to express the isolation requirements of an application, along with the semantics of various isolation guarantees provided by the store in the QUELEA's specification language allows us to define a classification scheme [15], similar to the one in § 3.4, to automatically map requirements to guarantees.

5 Implementation

The abstract model of QUELEA, optionally extended with a chosen set of high-level consistency and isolation guarantees, can be instantiated on top of any eventually consistent key-value store. We now describe a reference implementation of QUELEA on top of Cassandra's key-value store [10]. Our implementation supports CV, CC, and SC consistency levels (§ 3.4) for data type operations, and RC, MAV, and RR isolation levels (§ 4.2) for transactions. This functionality is implemented entirely on top of the standard interface exposed by Cassandra. From an engineering perspective, leveraging an off-the-shelf data store enables an implementation comprising roughly only 2500 lines of Haskell code, which is packaged as a library [13].

5.1 Object State

Cassandra adopts a data model similar to that of BigTable [5]. Each row is identified by a composite *primary key*, whose first component is the *partition key*, and remaining components are *clustering columns*. Like Dynamo [8], Cassandra uses consistent hashing on partition keys to map rows to machines in the cluster that maintain a

[4]It is informative to compare isolation with atomicity. While the former constrains how the current transaction witnesses the effects of other transactions, the later determines how other transactions witness the effects of the current transaction.

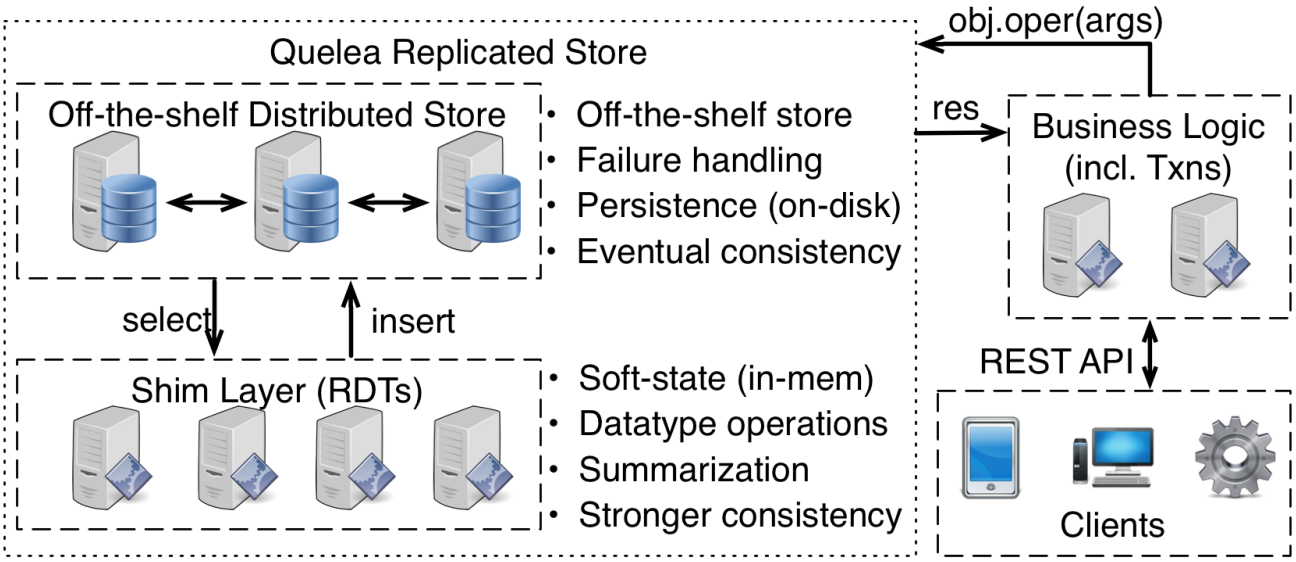


Figure 5: Implementation Model.

replica. Hence, rows with the same partition key (together referred to as a *partition*) are mapped to the same machine[5]. Within each partition, rows are clustered and sorted on the values of their clustering columns. QUELEA relies on these properties to minimize the latency of querying object state stored in Cassandra.

Recall that the state of an object in QUELEA is represented as a set of effects. An effect generated as a result of executing an effectful operation (eg., `inc` or `withdraw`) inserts a new row $(o, s, i, txn, val, deps)$, where o is the identifier of the object on which the operation is performed, s is the identifier of the session that issued the operation, and i is operation's sequence number within its session. The optional *txn* column identifies the transaction (if any). The column *val* is the value associated with the effect (eg: `Withdraw 50`). *deps* is the set of identifiers of *dependencies* of this operation and is defined as $deps(e) = \{e_1 \mid \mathsf{vis}(e_1, e) \wedge \neg(\exists e_2.\mathsf{vis}(e_1, e_2) \wedge \mathsf{vis}(e_2, e))\}$. The primary key of this row is a composite of (o, s, i), making o the partition key, and s and i clustering columns. Thus, effects on the same object belong to the same partition, minimizing the latency of reading its state. Within the partition, effects are clustered (and sorted), first on s, then on i. Consequently, range queries on s and i, such as the set of effects that precede a given effect in the same session (i.e., effects with same o and s, but lesser i), are efficient. This data model has been crafted to enable efficient implementations of consistency guarantees, such as session guarantees described in § 3.4.

5.2 Operation Consistency

The consistency semantics of replicated data types are implemented and enforced in the *shim layer* above Cassandra. The overall system architecture is shown in Fig. 5. Note that the shim layer node simply acts as a soft-state write-through cache and can safely be terminated at any point. Similarly, new shim layer nodes can be spawned on demand.

The shim layer maintains a causally-consistent in-memory state of a subset of objects in the system. A causally-consistent state of an object is a subset of effects on the object that are closed under the hbo relation. In other words, the shim layer includes an effect e over an object o, only if it also includes all effects e' over o that happened before e (i.e., $\mathsf{hbo}(e', e)$). Since all read operations are served by the shim layer, a read operation only ever witnesses a causally-consistent state of its object. Recall that this is precisely the CV consistency level

[5]Dynamo's consistent hashing maps a partition key to a *vnode*, which, in common case, maps to a single physical machine.

described in § 3.4. Thus, CV is the default consistency level in this implementation of Quelea.

The shim layer nodes periodically fetches updates from the backing store, thereby ensuring that later operations witness more recent CV-consistent state. For causally consistent operations, however, updates need to be fetched on-demand. For example, if a causally consistent operation op on o is the i^{th} operation in session s, and if the effect of $i-1^{th}$ operation in s is not present in the shim layer state of o, then a blocking read query for a row with primary key $(o, s, i-1)$ needs to be issued by the shim layer.

Strongly consistent operations are performed after obtaining a distributed global lock. Distributed lock is implemented with the help of Cassandra's conditional updates (lightweight transactions). To prevent deadlocks due to crash failures of the lock owner, the lock is leased only for a pre-determined amount of time. Lease functionality is implemented using Cassandra's support for expiring columns.

5.3 Transactions

Multi-key coordination-free transactions are implemented in Quelea by exploiting the shim layer's default CV consistency guarantee. Recall that the shim layer does not include an effect unless all its dependencies are also included. For every transaction, Quelea instantiates a special transaction marker effect m that is included as a dependence to every effect generated in the transaction. Importantly, the marker m is not inserted into the backing store until and unless the transaction finishes execution. Now, any replica which includes one of the effects from the transaction must include m, and transitively must include every effect from the transaction. This ensures atomicity and satisfies the RC requirement.

MAV semantics is implemented by keeping track of the set of transaction markers M witnessed by the transaction, and before performing an operation at some replica, ensuring that M is a subset of the transaction markers included at that replica. If not, the missing effects are synchronously fetched. RR semantics is realized by capturing an optimized snapshot of the state of some replica; each operation from an RR transaction is applied to this snapshot state. Any generated effects are added to this snapshot.

5.4 Summarization

Observe that the state of an object in Quelea grows monotonically as more effectful operations are performed on the object. Unbounded growth of state would make querying prohibitively expensive at some point. To keep state size in check, Quelea summarizes object state both in the shim layer node and the backing store, typically when the number of effects on an object crosses a tunable threshold. Quelea's summarization is similar in nature to the *major compaction* operation on *SSTables* in BigTable. While BigTable's compaction summarizes reads and writes at the storage layer, Quelea summarizes effects with application-specific semantics at the application layer. Hence, the semantics of summarization in Quelea is application-specific. Quelea therefore requires its applications to implement a special `summarize` function that is called whenever the state needs to be summarized. For example, a bank account RDT's `summarize` function may summarize a state comprising multiple `withdraw` and `deposit` effects into a single `deposit` effect, thus drastically reducing the number of effects that need to be kept track of by the shim layer and the store.

6 Evaluation

We have used Quelea to implement various applications, which include individual RDTs as well as larger applications composed of several RDTs. In order to ensure certain high-level invariants, these applications require various kinds of consistency and isolation guarantees from the data store. For instance, the microblogging application requires causal consistency for its `getTweet` operation to ensure the causal order delivery of tweets (§ 1). RUBiS [14], an eBay-like auction site, requires MAV isolation level for its `cancelBid` transaction to maintain the integrity of data in its materialized views. If an application developer were to implement these

applications on top of the default interfaces exposed by a typical off-the-shelf data store, she would either have to rely on ill-specified low-level functionality of the store, or build the required high-level functionality herself. With QUELEA, we were able to derive the high-level guarantees required by these applications by merely stating their requirements as specifications with respect to the abstract model[6].

For performance evaluation, we commissioned Amazon EC2 instances, and deployed QUELEA applications in an environment similar to the production environment of medium-scale web applications. We then ran these applications on realistic workloads constructed from standard benchmarks, such as YCSB [6] and RUBiS bidding mix. Our experiments [15] show that (a). QUELEA's expressive programming model incurs no more than 30% (resp. 20%) of latency (resp. throughput) overhead when compared to the native implementation on top of Cassandra (both run under default consistency levels) with 512 concurrent clients, and (b). with a distribution of 50% SC, 25% CC, and 25% EC operations, QUELEA incurred 41% (18%) higher latency (lower throughput) than 100% EC operations, when compared to 162% (52%) higher latency (lower throughput) incurred by 100% SC operations. These experiments demonstrate that the performance penalty of supporting QUELEA on top of an existing key-value stores is within reasonable limits, and that there is a significant performance incentive for applications to rely on QUELEA to enforce their high-level invariants rather than choosing SC for every operation.

7 Conclusion

Although modern web applications settle for eventual consistency in return for high availability, they nonetheless need stronger consistency guarantees to support small, yet significant, fraction of their functionality. The de facto interfaces exposed by well-known eventually consistent data stores are often ill-specified and too low-level from the perspective of an application developer. There currently exists a wide chasm between the high-level application-specific consistency requirements, and low-level store-specific tunable consistency levels. This paper describes QUELEA, a programming framework and a system that proposes to address this gap by lifting and standardizing the interface to an eventually consistent data store. We present many examples that demonstrate the advantage of reasoning with high-level interface exposed by QUELEA, when compared to the low-level interfaces offered by off-the-shelf data stores. We realize an instantiation of QUELEA on top of Cassandra, and illustrate the performance implications of using QUELEA, for both, applications and data stores.

References

[1] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly Available Transactions: Virtues and Limitations. *PVLDB*, 7(3):181–192, 2013.

[2] 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.

[3] Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. Eventually Consistent Transactions. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, pages 67–86, Berlin, Heidelberg, 2012. Springer-Verlag.

[4] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284, New York, NY, USA, 2014. ACM.

[5] 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*

[6] Both specifications combined span less than 5 lines. Source code available at `https://github.com/kayceesrk/Quelea/tree/master/tests`

the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[6] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[7] datastax developer blog. lightweight transactions in cassandra 2.0, 2016. URL `http://www.datastax.com/dev/blog/lightweight-transactions-in-cassandra-2-0`. accessed: 2016-02-14 2:30:00.

[8] 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.

[9] Jepsen. Cassandra, 2016. URL `https://aphyr.com/posts/294-jepsen-cassandra`. accessed: 2016-02-14 2:40:00.

[10] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Operating Systems Review*, 44(2):35–40, April 2010.

[11] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[12] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[13] Quelea. Programming with Eventual Consistency over Cassandra, 2016. URL `https://hackage.haskell.org/package/Quelea`. Accessed: 2016-02-14 12:20:00.

[14] RUBiS. Rice University Bidding System, 2014. URL `http://rubis.ow2.org/`. Accessed: 2014-11-4 13:21:00.

[15] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 413–424, New York, NY, USA, 2015. ACM.

[16] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional Storage for Geo-replicated Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[17] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

[18] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society.