Declarative Programming over Eventually Consistent Data Stores

KC Sivaramakrishnan Gowtham Kaki Suresh Jagannathan
Purdue University
{chandras,gkaki,suresh}@cs.purdue.edu

Abstract

User-facing online services utilize geo-distributed data stores to minimize latency and tolerate partial failures, with the intention to provide a fast, always-on experience. However, geo-distribution does not come for free; application developers have to contend with weak consistency behaviors, and the lack of abstractions to composably construct high-level replicated data types, necessitating the need for complex application logic and invariably exposing inconsistencies to the user. Some commercial distributed data stores and several academic proposals provide a lattice of consistency levels, with stronger consistency guarantees incurring increased latency and throughput costs. However, correctly assigning the right consistency level for an operation requires subtle reasoning and is often an error-prone task.

In this paper, we present QUELEA, a declarative programming model for eventually consistent data stores (ECDS), equipped with a *contract* language, capable of specifying fine-grained application-level consistency properties. A *contract enforcement system* analyses contracts, and *automatically* generates the appropriate consistency protocol for the method protected by the contract. We describe an implementation of QUELEA on top of an off-the-shelf ECDS, and provide support for *coordination-free* transactions. Several benchmarks including two large web applications, illustrate the effectiveness of our approach.

1. Introduction

Many real-world web services — such as those built and maintained by Amazon, Facebook, Google, Twitter, etc. — replicate application state and logic across multiple *replicas* within and across data centers. Replication is intended not only to improve application throughput and reduce user-perceived latency, but also to tolerate partial failures without compromising overall service availability. Traditionally programmers have relied on *strong consistency* guarantees such as linearizability [Herlihy and Wing 1990] or serializability [Papadimitriou 1979] in order to build correct applications. While strong consistency is an easily stated property, it masks the reality underlying large-scale distributed systems with respect to non-uniform latency, availability and network partitions [Brewer 2000; Gilbert and Lynch 2002]. Indeed, modern web services, which aim to provide an "always-on" experience, overwhelmingly favor availability and partition tolerance over strong consistency. To this end, several *weak consistency* models such as eventual consistency, causal consistency, session guarantees, and timeline consistency have been proposed.

Under weak consistency, the developer needs to be aware of concurrent conflicting updates, and has to pay careful attention to avoid unwanted inconsistencies (e.g., negative balances in a bank account, or having an item appear in a shopping cart after it has been removed [DeCandia et al. 2007]). Oftentimes, these inconsistencies leak from the application and is witnessed by the user. Ultimately, the developer must decide the consistency level appropriate for a particular operation; this is understandably an error-prone process requiring intricate knowledge of both the application as well as the semantics and implementation of the underlying data store, which typically have only informal descriptions. Nonetheless, picking the correct consistency level is critical not only for correctness but also for scalability of the application. While choosing a weaker consistency level than required may introduce program errors and anomalies, choosing a stronger one than necessary can negatively impact program scalability and performance.

Weak consistency also hinders compositional reasoning about programs. Although an application might be naturally expressed in terms of well-understood and expressive data types such as maps, trees, queues, or graphs, geo-distributed stores typically only provide a minimal set of data types with in-built conflict resolution strategies such as last-writer-wins (LWW) registers, counters, and sets [Lakshman and Malik 2010; Sivasubramanian 2012]. Furthermore, while traditional database systems enable composability through trans-

actions, geo-distributed stores typically lack unrestricted serializable transactional access to the data. Working in this environment thus requires application state to be suitably coerced to function using only the capabilities of the store.

To address these issues, we describe QUELEA, a declarative programming model and implementation for ECDS. The key novelty of QUELEA is an expressive *contract language* to declare and verify fine-grained application-level consistency properties. The programmer uses the contract language to axiomatically specify the set of legal executions allowed over the replicated data type. Contracts are constructed using primitive consistency relations such as *visibility* and *session order* along with standard logical and relational operators. A *contract enforcement system* statically maps operations over the datatype to a particular consistency level available on the store, and provably validates the correctness of the mapping. The paper makes the following contributions:

- We introduce QUELEA, a shallow extension of Haskell that supports the description and validation of replicated data types found on ECDS. Contracts are used to specify fine-grained application-level consistency properties, and are statically analyzed to assign the most efficient and sound store consistency level to the corresponding operation.
- QUELEA supports coordination-free transactions over arbitrary datatypes. We extend our contract language to express fine-grained transaction isolation guarantees, and utilize the contract enforcement system to automatically assign the correct isolation level for a transaction.
- We provide meta-theory that certifies the soundness of our contract enforcement system, and ensures that an operation is only executed if the required conditions on consistency are met.
- An implementation of QUELEA as a transparent shim layer over Cassandra [Lakshman and Malik 2010], a well-known general-purpose data store. Experimental evaluation over a set of real-world applications, including a Twitter-like micro-blogging site and an eBay-like auction site illustrates the practicality of our approach.

The rest of the paper is organized as follows. The next section describes the system model. We describe the challenges in programming under eventual consistency, and introduce QUELEA contracts as a proposed solution to overcome these issues in § 3. § 4 provides more details on the contract language, and its mapping to the store consistency levels, along with meta-theory for certifying the correctness of the mapping. § 5 introduces transaction contracts and their classification. § 6 describes the implementation of QUELEA on top of Cassandra. § 7 discusses experimental evaluation. § 8 and 9 present related work and conclusions.

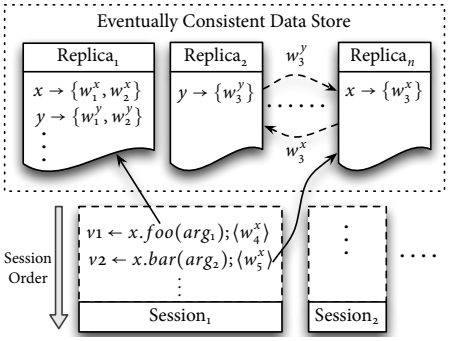


Figure 1. QUELEA system model.

2. System Model

In this section, we describe the system model and introduce the primitive relations that our contract language is seeded with. Figure 1 presents a schematic diagram of our system model. The distributed store is composed of a collection of *replicas*, each of which stores a set of *objects* $(x, y, \ldots)$. We assume that every object is replicated at every replica in the store. The state of an object at any replica is the set of all updates (*effects*) performed on the object. For example, the state of x at replica 1 is the set composed of effects w_1^x and w_2^x.

Each object is associated with a set of *operations*. The clients interact with the store by invoking operations on objects. The sequence of operations invoked by a particular client on the store is called a *session*. The data store is typically accessed by a large number of clients (and hence sessions) concurrently. Importantly, the clients are oblivious to which replica an operation is applied to; the data store may choose to route the operation to any replica in order to minimize latency, balance load, etc. For example, the operations *foo* and *bar* invoked by the same session on the same object, might end up being applied to different replicas because replica 1 (to which *foo* was applied) might be unreachable when the client invokes *bar*.

When *foo* is invoked on a object x with arguments arg_1 at replica 1, it simply *reduces* over the current set of effects at that replica on that object (w_1^x and w_2^x), produces a result $v1$ that is sent back to the client, and emits a *single* new effect w_4^x that is appended to the state of x at replica 1. Thus, every operation is evaluated over a *snapshot* of the state of the object on which it is invoked. In this case, the effects w_1^x and w_2^x are *visible* to w_4^x, written logically as $\mathsf{vis}(w_1^x, w_4^x) \wedge \mathsf{vis}(w_2^x, w_4^x)$, where vis is the visibility relation between effects. Visibility is an irreflexive and asymmetric relation, and only relates effects produced by operations on the same object. Executing a read-only operation is similar except that no new effects are produced. The effect added to a particular replica is asynchronously sent to other replicas, and eventually merged into all other replicas. Observe that this model does not assume a particular resolution strategy for concurrent conflicting updates, and instead preserves *ev-*

ery update. Update conflicts are resolved when an operation reduces over the set of effects on an object at a particular replica.

Two effects w_4^x and w_5^x that arise from the same session are said to be in *session order* (written logically as $\mathsf{so}(w_4^x, w_5^x)$). Session order is an irreflexive, transitive relation. The effects w_4^x and w_5^x arising from operations applied to the same object x are said to be under the *same object* relation, written $\mathsf{sameobj}(w_4^x, w_5^x)$. Finally, we can associate every effect with the operation that generated the effect with the help of a relation oper. In the current example, $\mathsf{oper}(w_4^x, foo)$ and $\mathsf{oper}(w_5^x, bar)$ hold. For simplicity, we assume all operation names across all object are distinct.

This model admits all the inconsistencies associated with eventual consistency. The goal of this work is to identify the precise consistency level for each operation such that application-level constraints are not violated. In the next section, we will concretely describe the challenges associated with constructing a consistent bank account on top of an ECDS. Subsequently, we will illustrate how our contract and specification language, armed with the primitive relations vis, so, sameobj and oper, mitigates these challenges.

3. Motivation

Consider how we might implement a highly available bank account on top of an ECDS, with the *integrity* constraint that the balance must be non-negative. We begin by implementing a bank account replicated data type (RDT) in QUELEA, and then describe the mechanisms to obtain the desired correctness guarantees.

3.1 RDT specification

A key novelty in QUELEA is that it allows the addition of new RDTs to the store, which obviates the need for coercing the application logic to utilize the store provided data types. In addition, QUELEA treats the convergence semantics (i.e., *how* conflicting updates are resolved) of the data type separately from its consistency properties (i.e., *when* updates become visible). This separation of concerns permits *operational* reasoning for conflict resolution, and *declarative* reasoning for consistency. The combination of these techniques enhances the programmability of the store.

Let us assume that the bank account object provides three operations: `deposit`, `withdraw` and `getBalance`, with the assumption that the withdraw fails if the account has insufficient balance. Every operation in QUELEA is of the following type, written in Haskell syntax:

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

An operation takes a list of effects (the *history* of updates to that object), and an input argument, and returns a result along with an optional effect (read-only operations return `Nothing`). The new effect (if emitted) is added to the state of the object at the current replica, and asynchronously sent to other replicas. The implementation of the bank account operations in QUELEA is given in Figure 2.

```
data Acc = Deposit Int | Withdraw Int | GetBal

getBalance :: [Acc] → () → (Int, Maybe Acc)
getBalance hist _ =
  let res = sum [x | Deposit x ← hist]
            - sum [x | Withdraw x ← hist]
  in (res, Nothing)

deposit :: [Acc] → Int → ((), Maybe Acc)
deposit hist amt = ((), Just $ Deposit amt)

withdraw :: [Acc] → Int → (Bool, Maybe Acc)
withdraw hist v =
  if sel1 $ getBalance hist () ≥ v
  then (True, Just $ Withdraw v)
  else (False, Nothing)
```

Figure 2. Definition of a bank account expressed in Quelea.

The datatype `Acc` represents the effect type for the bank account. The function `sum` returns the sum of elements in the list, and `sel1` returns the first element of a tuple. For each operation, `hist` is a *snapshot* of the state of the object at some replica. In this sense, every operation on the RDT is atomic, and thus amenable to sequential reasoning. Here, `getBalance` is a read-only operation, `deposit` always emits an effect, and `withdraw` only emits an effect if there is sufficient balance in the account. We have implemented a large corpus of RDTs for realistic benchmarks including shopping carts, auction and micro-blogging sites, etc. in a few tens of lines of code, expressed in this style.

3.1.1 Summarization

Observe that the definition of `getBalance` reduces over the *entire history* of updates to an account. If we are to realize an efficient implementation of this bank account RDT, we need a *summary* of the account history. Intuitively, the current account balance summarizes the state of an account. A bank account with the history `[Deposit 10, Withdraw 5]` is *observably equivalent* to a bank account with a single deposit operation `[Deposit 5]`; we can replace the earlier history with the latter and a client of the store would not able to tell the difference between the two.

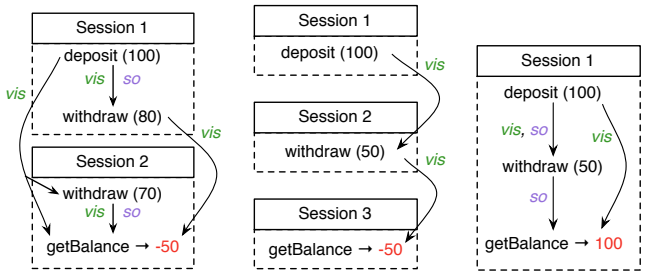
This notion of observable equivalence can be generalized to other RDTs as well. For example, a last-writer-wins register with multiple updates is equivalent to a register with only the last write. Similarly, a set with a collection of add and remove operations is equivalent to a set with a series of additions of live elements from the original set. Since the notion of observable equivalence is specific to each RDT, programmers can provide a summarization function - (`summarize`) of type `[e] -> [e]` - as a part of the RDT specification. The summarization function for the bank account is:

```
summarize hist =
  [Deposit $ sel1 $ getBalance hist ()]
```

Given a bank account history `hist`, the `summarize` function returns a new history with a single deposit of the current account balance. Our implementation invokes the summarization function associated with an RDT to reduce the size of the effect sets maintained by replicas.

3.2 Anomalies under Eventual Consistency

Our goal is to choose the correct consistency level for each of the bank account operations such that (1) the balance remains non-negative and (2) the `getBalance` operation never incorrectly returns a negative balance.



(a) Unsafe withdraw (b) Negative balance (c) Missing update

Figure 3. Anomalies possible under eventual consistency for the get balance operation.

Consider the execution shown in Figure 3(a). Assume that all operations in the figure are on the same bank account object with the initial balance being zero. Session 1 performs a `deposit` of 100, followed by a `withdraw` of 80 in the same session. The `withdraw` operation witnesses the deposit and succeeds[1]. Subsequently, session 2 perform a `withdraw` operation, but importantly, due to eventual consistency, only witnesses the `deposit` from session 1, but not the subsequent withdraw. Hence, this `withdraw` also *incorrectly* succeeds, violating the integrity constraint. A subsequent `getBalance` operation, that happens to witness all the previous operations, would report a negative balance.

It is easy to see that preventing concurrent `withdraw` operations eliminates this anomaly. This can be done by insisting that `withdraw` be executed as a strongly consistent operation. Despite this strengthening, the `getBalance` operation may still incorrectly report a negative balance to the user. Consider the execution shown in fig. 3(b), which consists of three concurrent sessions performing a `deposit`, a `withdraw`, and a `getBalance` operation, respectively, on the same bank account object. As the vis edge indicates, operation `withdraw(50)` in session 2 witnesses the effects of `deposit(100)` from session 1, concludes that there is sufficient balance, and completes successfully. However, the `getBalance` operation may only witness this successful withdraw, but not the causally preceding `deposit`, and reports the balance of negative 50 to the user.

[1] Although visibility and session order relations relate effects, we have abused the notation in these examples to relate operations, with the idea that the relations relate the effect emitted by those operations

Under eventual consistency, the users may also be exposed to other forms of inconsistencies. Figure 3(c) shows an execution where the `getBalance` operation in a session does not witness the effects of an earlier `withdraw` operation performed in the same session, possibly because it was served by a replica that has not yet merged the `withdraw` effect. This anomaly leads the user to incorrectly conclude that the `withdraw` operation failed to go through.

Although it is easy to understand the reasons behind the occurrence of the aforementioned anomalies, finding the appropriate fixes is not readily apparent. Making `getBalance` a strongly consistent operation is definitely sufficient to avert anomalies, but is it really necessary? Given the cost of enforcing strong consistency [Sivasubramanian 2012; Terry et al. 2013], it is preferable to avoid it unless there are no viable alternatives. Exploring the space of these alternatives requires understanding the subtle differences in semantics of various kinds of weak consistency alternatives.

3.3 Contracts

QUELEA helps facilitate the mapping of operations to appropriate consistency levels by letting the programmer declare application-level consistency constraints as *contracts*[2] (Figure 4) that axiomatically specify the set of allowed executions involving this operation. In the case of the bank account, any execution that does not exhibit the anomalies described in the previous section is a *well-formed* execution on the bank account object. By specifying the set of legal executions for each data type in terms of a trace of operation invocations on that type, QUELEA ensures that all executions over that type are well-formed.

In our running example, it is clear that in order to preserve the integrity constraint, the `withdraw` operation must be strongly consistent. That is, given two `withdraw` operations a and b, either a is visible to b or vice-versa. We express this application-level consistency requirement as a contract (ψ_w) over `withdraw`:

$$\forall(a : \texttt{withdraw}).\ \mathsf{sameobj}(a, \hat{\eta}) \Rightarrow a = \hat{\eta} \vee \mathsf{vis}(a, \hat{\eta}) \vee \mathsf{vis}(\hat{\eta}, a)$$

Here, $\hat{\eta}$ stands for the effect emitted by the `withdraw` operation. The syntax a : `withdraw` states that a is an effect emitted by a `withdraw` operation i.e., $\mathsf{oper}(a, \texttt{withdraw})$ holds. The contract specifies that if the current operation emits an effect $\hat{\eta}$, then for any operation a which was emitted by a `withdraw` operation, it is the case that $a = \hat{\eta}$ or a is visible to $\hat{\eta}$, or vice versa. Any execution on a bank account object that preserves the above contract for a `withdraw` operation is said to be derived from a correct implementation of `withdraw`.

For `getBalance`, we construct the following contract (ψ_{gb}):

$$\begin{aligned}&\forall(a : \texttt{deposit}), (b : \texttt{withdraw}), (c : \texttt{deposit} \vee \texttt{withdraw}).\\ &\quad (\mathsf{vis}(a, b) \wedge \mathsf{vis}(b, \hat{\eta}) \Rightarrow \mathsf{vis}(a, \hat{\eta}))\\ &\quad \wedge ((\mathsf{so} \cap \mathsf{sameobj})(c, \hat{\eta}) \Rightarrow \mathsf{vis}(c, \hat{\eta}))\end{aligned}$$

[2] QUELEA exposes the contract construction language as a Haskell library

The expression c : `deposit` $\vee$ `withdraw` states that c is an effect that was emitted either by a `deposit` or a `withdraw` operation. If a `withdraw` b is visible to `getBalance` $\hat{\eta}$, then all `deposit` operations a visible to b should also be visible to $\hat{\eta}$. Intuitively, this prevents negative balance anomalies. Our contract language provides operators to compose relations. The syntax $(R_1 \cap R_2)(a, b)$ is equivalent to $R_1(a, b) \wedge R_2(a, b)$. The last line of the above contract says that if a `deposit` or a `withdraw` operation precedes a `getBalance` operation in session order, and is applied on the same object as the `getBalance` operation, then it must be the case that the `getBalance` operation witnesses the effects of the preceding operations.

Finally, since there are no restrictions on when or how a `deposit` operation can execute, its contract is simply true.

3.4 From Contracts to Implementation

Notice that the contracts for `withdraw` and `getBalance` only express application-level consistency requirements, and make no reference to the semantics of the underlying store. To write contracts, a programmer only needs to reason about the semantics of the application under the QUELEA system model. The mapping of application-level consistency requirements to appropriate store-level guarantees is done automatically behind-the-scene. How might one go about ensuring that an execution adheres to a contract? The challenge is that a contract provides a declarative (axiomatic) specification of an execution, while what is required is an operational procedure for *enforcing* its implicit constraints.

One strategy would be to execute operations speculatively. Here, operations are tentatively applied as they are received from the client or other replicas. We can maintain a runtime manifestation of executions, and check well-formedness conditions at runtime, rolling back executions if they are ill-formed. However, the overhead of state maintenance and the complexity of user-defined contracts is likely to make this technique infeasible in practice.

We devise a static approach instead. Contracts are analyzed with the help of a theorem prover, and statically mapped to a particular store-level consistency property that the prover guarantees preserves contract semantics. We call this procedure *contract classification*. Given the variety and complexity of store level consistency properties, the idea is that the system implementer parameterizes the classification procedure by describing the store semantics in the *same* contract language as the one used to express the contract on the operations. In the next section, we describe the contract language in detail and describe the classification procedure for a particular store semantics.

4. Contract Language

4.1 Syntax

The syntax of our core contract 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 *current operation* - the operation for which a contract is being written. Notice that $\hat{\eta}$ occurs free in the contract. We will fix its scope when classifying the contracts (§ 4.4). The type of an effect is simply the name of the operation (eg: `withdraw`) that induced the effect. We admit disjunction in types to let an effect variable range over multiple operation names. The contract $\forall(a : \tau_1 \vee \tau_2).\ \psi$ 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.

$x, y, \hat{\eta} \in$ `EffVar` $\quad$ Op $\in$ `OperName`

ψ	$\in$	`Contract`	::=	$\forall(x : \tau).\psi \mid \forall x.\psi \mid \pi$
τ	$\in$	`EffType`	::=	Op $\mid \tau \vee \tau$
π	$\in$	`Prop`	::=	true $\mid R(x, y) \mid \pi \vee \pi$
			$\mid$	$\pi \wedge \pi \mid \pi \Rightarrow \pi$
R	$\in$	`Relation`	::=	vis $\mid$ so $\mid$ sameobj $\mid R^+$
			$\mid$	$R \cup R \mid R \cap R$

Figure 4. Contract language.

$\eta \in$ `Effect`	$\psi \in$	`Contract`	$\overline{\eta} \in$	`Effect Set`
A	$\in$	`EffSoup`	::=	$\overline{\eta}$
vis, so, sameobj	$\in$	`Relations`	::=	A $\times$ A
E	$\in$	`ExecState`	::=	(A,vis,so,sameobj)

Figure 5. Axiomatic execution.

Quantifier-free propositions in our contract language are conjunctions, disjunctions and implications of predicates expressing relations between pairs of effect variables. 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[3] 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* ($\mathsf{hbo} = (\mathsf{soo} \cup \mathsf{vis})^+$).

4.2 Semantics

QUELEA contracts are constraints over axiomatic definitions of program executions. Figure 5 summarizes artifacts relevant to define an axiomatic execution. We formalize an axiomatic execution as a tuple (A,vis,so,sameobj), where A, called the *effect soup*, is the set of all effects generated during the program execution, and $\mathsf{vis}, \mathsf{so}, \mathsf{sameobj} \subseteq \mathsf{A} \times \mathsf{A}$ are *visibility*, *session order*, and *same object* relations, respectively, witnessed over generated effects at run-time.

Note that the axiomatic definition of an execution (E) provides interpretations for primitive relations (eg: vis) that occur free in contract formulas, and also fixes the domain

[3] Strictly speaking, R^+ is not the transitive closure of R, as transitive closure is not expressible in FOL. Instead, R^+ in our language denotes *a* superset of transitive closure of R. Formally, R^+ is any relation R' such that forall x, y, and z, a) $R(x, y) \Rightarrow R'(x, y)$, and b) $R'(x, y) \wedge R'(y, z) \Rightarrow R'(x, z)$

of quantification to set of all effects (A) observed during the program execution. As such, E is a potential model for any first-order formula (ψ) expressible in our contract language. If E is indeed a valid model for ψ (written as $\mathsf{E} \models \psi$), we say that the execution E satisfied the contract ψ:

Definition 1. *An axiomatic execution* E *satisfies a contract* ψ *if and only if* $E \models \psi$.

4.3 Capturing Store Semantics

An important aspect of our contract language is its ability to capture store-level consistency guarantees, along with application-level consistency requirements. Similar to [Burckhardt et al. 2014], we can rigorously define a wide variety of store semantics including those that combine any subset of session and causality guarantees, and multiple consistency levels. However, for our purposes, we identify three particular consistency levels – eventual, causal, and strong, commonly offered by many distributed stores with tunable consistency, with increasing overhead in terms of latency and availability.

- **Eventual consistency**: Eventually consistent operations can be satisfied as long as the client can reach at least one replica. In the bank account example, `deposit` is an eventually consistent operation. While an ECDS typically offer *basic* eventual consistency with all possible anomalies, we assume that our store provides stronger semantics that remain highly-available [Bailis et al. 2013a; Lloyd et al. 2011]; the store always exposes a *causal cut* of the updates. This semantics can be formally captured in terms of the following contract definition:

$$\psi_{\mathsf{ec}} = \forall a, b.\ \mathsf{hbo}(a, b) \wedge \mathsf{vis}(b, \hat{\eta}) \Rightarrow \mathsf{vis}(a, \hat{\eta})$$

- **Causal consistency**: Causally consistent operations are required to see a causally consistent snapshot of the object state, including the actions performed on the same session. The latter requirement implies that if two operations o_1 and o_2 from the same session are applied to two different replicas r_1 and r_2, the second operation cannot be discharged until the effect of o_1 is included in r_2. The `getBalance` operation requires causal consistency, as it requires the operations from the same session to be visible, which cannot be guaranteed under eventual consistency. The corresponding store semantics is captured by the contract ψ_{cc} defined below:

$$\psi_{\mathsf{cc}} = \forall a.\ \mathsf{hbo}(a, \hat{\eta}) \Rightarrow \mathsf{vis}(a, \hat{\eta})$$

- **Strong Consistency**: Strongly consistent operations may block indefinitely under network partitions. An example is the total-order contract on `withdraw` operation. The corresponding store semantics is captured by the ψ_{sc} contract definition:

$$\psi_{\mathsf{sc}} = \forall a.\ \mathsf{sameobj}(a, \hat{\eta}) \Rightarrow \mathsf{vis}(a, \hat{\eta}) \ \vee\ \mathsf{vis}(\hat{\eta}, a) \ \vee\ a = \hat{\eta}$$

$$\frac{\psi \leq \psi_{\mathsf{sc}}}{\mathsf{WellFormed}(\psi)} \qquad \frac{\psi \leq \psi_{\mathsf{ec}}}{\mathsf{EventuallyConsistent}(\psi)}$$

$$\frac{\psi \not\leq \psi_{\mathsf{ec}} \quad \psi \leq \psi_{\mathsf{cc}}}{\mathsf{CausallyConsistent}(\psi)} \qquad \frac{\psi \not\leq \psi_{\mathsf{cc}} \quad \psi \leq \psi_{\mathsf{sc}}}{\mathsf{StronglyConsistent}(\psi)}$$

Figure 6. Contract classification.

4.4 Contract Classification

Our goal is to map application-level consistency constraints on operations to appropriate store-level consistency guarantees capable of satisfying these constraints. The ability to express both these kinds of constraints as contracts in our contract language lets us compare and determine if contract (ψ_{op}) of an operation (op) is weak enough to be satisfied under a store consistency level identified by the contract ψ_{st}. Towards this end, we define a binary *weaker than* relation for our contract language as following:

Definition 2. *A contract* ψ_{op} *is said to be weaker than* ψ_{st} *(written* $\psi_{op} \leq \psi_{st}$ *) if and only if* $\Delta \vdash \forall \hat{\eta}.\psi_{st} \Rightarrow \psi_{op}$.

The quantifier in the sequent binds $\hat{\eta}$ that occurs free in ψ_{st} and ψ_{op}. Context (Δ) of the sequent is a conjunction of assumptions about the nature of primitive relations. A *well-formed* axiomatic execution (E) is expected to satisfy these assumptions (i.e., $\mathsf{E} \models \Delta$).

Definition 3. *An axiomatic execution* $\mathsf{E} = (\mathsf{A},\mathsf{vis},\mathsf{so},\mathsf{sameobj})$ *is well-formed if the following axioms* (Δ) *hold:*

- *The happens-before relation is acyclic:* $\forall a.\ \neg\mathsf{hb}(a, a)$.
- *Visibility only relates actions on the same object:*
  - $\forall a, b.\ \mathsf{vis}(a, b) \Rightarrow \mathsf{sameobj}(a, b)$.
- *Session order is a transitive relation:*
  - $\forall a, b, c.\ \mathsf{so}(a, b) \ \wedge\ \mathsf{so}(b, c) \Rightarrow \mathsf{so}(a, c)$.
- *Same object is an equivalence relation:*
  - $\forall a.\ \mathsf{sameobj}(a, a)$.
  - $\forall a, b.\ \mathsf{sameobj}(a, b) \Rightarrow \mathsf{sameobj}(b, a)$.
  - $\forall a, b, c.\ \mathsf{sameobj}(a, b) \wedge \mathsf{sameobj}(b, c) \Rightarrow \mathsf{sameobj}(a, c)$.

If the contract (ψ_{op}) of an operation (op) is *weaker than* a store contract (ψ_{st}), then constraints expressed by the former are implied by guarantees provided by the latter. The completeness of first-order logic allows us to assert that any well-formed execution (E) that satisfies ψ_{st} (i.e., $\mathsf{E} \models \psi_{st}$) also satisfies ψ_{op} (i.e., $\mathsf{E} \models \psi_{op}$). Consequently, it is safe to execute operation op under a store consistency level captured by ψ_{st}.

Observe that the contracts ψ_{sc}, ψ_{cc} and ψ_{ec} are themselves totally ordered with respect to the $\leq$ relation: $\psi_{\mathsf{ec}} \leq \psi_{\mathsf{cc}} \leq \psi_{\mathsf{sc}}$. This concurs with the intuition that any contract satisfiable under ψ_{ec} or ψ_{cc} is satisfiable under ψ_{sc}, and any contract that is satisfiable under ψ_{ec} is satisfiable under ψ_{cc}. We are interested in the *weakest* guarantee (among ψ_{ec}, ψ_{cc}, and ψ_{sc}) required to satisfy the contract. We define the cor-

responding consistency level as the *consistency class* of the contract.

The classification scheme, presented formally in Figure 6, defines rules to judge the consistency class of a contact. For example, the scheme classifies the `getBalance` contract (ψ_{gb}) from § 3 as a CausallyConsistent contract, because the sequent $\Delta \vdash \psi_{\mathsf{cc}} \Rightarrow \psi_{gb}$ is valid in first-order logic (therefore, $\psi_{gb} \leq \psi_{\mathsf{cc}}$), whereas the sequent $\Delta \vdash \psi_{\mathsf{ec}} \Rightarrow \psi_{gb}$ is invalid (therefore, $\psi_{gb} \not\leq \psi_{\mathsf{ec}}$). Since we confine our contract language to a decidable subset of the logic, validity of such sequents can be decided mechanically allowing us to automate the classification scheme in QUELEA.

Along with three straightforward rules that classify contracts into consistency classes, the classification scheme also presents a rule that judges well-formedness of a contract. A contract is well-formed if and only if it is satisfiable under ψ_{sc} - the strongest possible consistency guarantee that the store can provide. Otherwise, it is considered ill-formed, and rejected statically.

4.5 Soundness of Contract Classification

We now present a meta-theoretic result that certifies the soundness of classification-based contract enforcement. To help us state the result, we define an operational semantics of the system described informally in § 2:

$$
\begin{array}{llll}
op & \in & \texttt{Operation} & \\
\tau & \in & \texttt{ConsistencyClass} & ::= \quad \mathsf{ec}, \mathsf{cc}, \mathsf{sc} \\
\sigma & \in & \texttt{Session} & ::= \quad \cdot \mid \langle op, \tau \rangle ; \sigma \\
\Sigma & \in & \texttt{Session Soup} & ::= \quad \sigma \parallel \Sigma \mid \emptyset \\
 & & \texttt{Config} & ::= \quad \mathsf{E}, \Sigma
\end{array}
$$

We model the system as a tuple E, Σ, where the axiomatic execution E captures the data store's current state, and session soup Σ is the set of concurrent client sessions interacting with the store. A session σ is a sequence of pairs composed of replicated data type operations op, tagged with the consistency class τ of their contracts (as determined by the contract classification scheme). We assume a reduction relation of form:

$$\mathsf{E}, \langle op, \tau \rangle ; \sigma \parallel \Sigma \xrightarrow{\eta} \mathsf{E}', \sigma \parallel \Sigma$$

on the system state. The relation captures the progress of the execution (from E to E') due to the successful completion of a client operation op from one of the sessions in Σ, generating a new effect η. If the resultant execution E' satisfies the store contract ψ_τ (i.e., $\mathsf{E} \models \psi_\tau$), then we say that the store has *enforced* the contract ψ_τ in the execution E'. With help of the operational semantics, we now state the soundness of contract enforcement as follows:

Theorem 4 (Soundness of Contract Enforcement)**.** *Let ψ be a well-formed contract of a replicated data type operation op, and let τ denote the consistency class of ψ as determined by the contract classification scheme. For all well-formed execution states E, E' such that $\mathsf{E}, \langle op, \tau \rangle ; \sigma \parallel \Sigma \xrightarrow{\eta} \mathsf{E}', \sigma \parallel \Sigma$, if $\mathsf{E}' \models \psi_\tau[\eta/\hat{\eta}]$, then $\mathsf{E}' \models \psi[\eta/\hat{\eta}]$*

The theorem states that if a data store correctly enforces ψ_{sc}, ψ_{cc}, and ψ_{ec} contracts in all well-formed executions, then the same store, extended with the classification scheme shown in Figure 6, can enforce all well-formed QUELEA contracts. The proof of the theorem is given in the supplementary material[4].

5. Transaction Contracts

While contracts on individual operations offer the programmer object-level declarative reasoning, real-world scenarios often involve operations that span multiple objects. In order to address this problem, several recent systems [Sovran et al. 2011; Burckhardt et al. 2012; Bailis et al. 2013a] have proposed eventually consistent transactions in order to compose operations on multiple objects. However, given that classical transaction models such as serializability and snapshot isolation require inter-replica coordination, these systems espouse *coordination-free transactions* that remain available under network partitions, but only provide weaker isolation guarantees. Coordination-free transactions have intricate consistency semantics and widely varying runtime overheads. As with operation-level consistency, the onus is on the programmer to pick the correct transaction kind. This choice is further complicated by consistency semantics of individual operations.

5.1 Syntax and Semantics Extensions

QUELEA automates the choice of assigning the correct and most efficient transaction isolation level. Similar to contracts on individual operations, the programmer associates contracts with transactions, declaratively expressing its consistency specification. We extend the contract language with a new term under quantifier-free propositions - $\mathsf{txn}\ S_1\ S_2$, where S_1 and S_2 are sets of effects, and introduce a new primitive equivalence relation $\mathsf{sametxn}$ that holds for effects from the same transaction. $\mathsf{txn}\{a, b\}\{c, d\}$ is just syntactic sugar for $\mathsf{sametxn}(a, b) \wedge \mathsf{sametxn}(c, d) \wedge \neg\mathsf{sametxn}(a, c)$, where a and b considered to belong to the *current* transaction.

We assume that operations not part of any transaction belong to their own unique transaction. While transactions may have varying isolation guarantees, we make the standard assumption that all transactions provide atomicity. Hence, we include the following axiom in Δ:

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

The semantics of this contract is illustrated in Figure 7(a).

5.2 Transactional Bank Account

In order to illustrate the utility of declarative reasoning for transactions, consider an extension of our running example to use two accounts (objects) – current (c) and savings (s).

[4] The supplementary material also provides the concrete reduction rules for enforcing the consistency classes.

Each account provides operations `withdraw`, `deposit` and `getBalance`, with the same contracts as defined previously. We consider two transactions – `save(amt)`, which transfers `amt` from current to savings, 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 the transactions is given below:

```
save amt =                          totalBalance =
  x ← $(classify ψsv)                 x ← $(classify ψtb)
  atomically x $ do                   atomically x $ do
    b ← withdraw c amt                  b1 ← getBalance c
    when b $ deposit s amt              b2 ← getBalance s
                                        return b1 + b2
```

ψ_{sv} and ψ_{tb} are the contracts on the corresponding transactions. The function `classify` assigns the contracts *statically* to one of the transaction isolation levels offered by the store; `$()` is meta-programming syntax for splicing the result into the program. The `atomically` construct invokes the enclosing operations at the given isolation level x, ensuring that the effects of the operations are made visible atomically.

While making both transactions serializable would ensure correctness, distributed stores rarely offer serializable transactions since it is unavailable and hinders scalability [Bailis et al. 2013a]. As we will see, these transactions can be satisfied with much weaker isolation guarantees. Despite the atomicity offered by the transaction, anomalies are still possible. For example, the two `getBalance` operations in `totalBalance` transactions might be served by different replicas with distinct set of committed `save` transactions. If the first(second) `getBalance` operation witness a `save` transaction that is not witnessed by the second(first) `getBalance` operation, then the balance returned will be less(greater) than the actual balance. It is not immediately apparent which weakest isolation guarantee will be sufficient to prevent the anomaly.

Instead, QUELEA requires the programmer to simply state the consistency requirement as a contract. Since we would like both the `getBalance` operations to witness the same set of `save` transactions, we define the constraint on `totalBalance` transaction ψ_{tb} as:

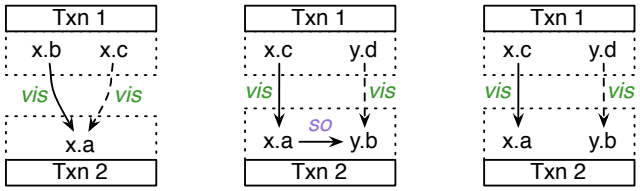
$$\psi_{tb} = \forall a : \texttt{getBalance}, b : \texttt{getBalance}, (c : \texttt{withdraw} \vee \texttt{deposit}), (d : \texttt{withdraw} \vee \texttt{deposit}). \mathsf{txn}\{a,b\}\{c,d\} \wedge \mathsf{vis}(c,a) \wedge \mathsf{sameobj}(d,b) \Rightarrow \mathsf{vis}(d,b)$$

The key idea in the above definition is that the txn primitive allows us to relate operations on different objects.

The `save` transaction only needs to ensure that the two writes it performs are made visible atomically. Since this is ensured by combining them in a transaction, `save` does not require any additional constraints, and ψ_1 is simply true.

5.3 Coordination-free Transactions

In order to illustrate the utility of transaction contract classification, we identify three well-understood coordination-free transaction semantics – Read Committed (RC) [Berenson et al. 1995], Monotonic Atomic View (MAV) [Bailis et al. 2013a] and Repeatable Read (RR) [Berenson et al. 1995], and illustrate the classification strategy. Our technique can indeed be applied to a different isolation level lattice.



(a) Atomicity (b) Monotonic Atomic View (c) Repeatable Read

Figure 7. Semantics of transaction contracts. x and y are distinct objects. The dotted line represents the visibility requested by the contracts.

A transaction with ANSI RC semantics only witnesses committed operations. Let us assume that a replica will buffer transactional updates until all the updates from the same transaction are available at that replica. Once all the updates from a transaction are available, the buffered updates are made visible to subsequent client requests. This ensure atomicity of transactions. Importantly, RC does not entail any other guarantees. As a result, a store implementing RC does not require inter-replica coordination. We can express RC as follows:

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

Notice that the above definition is the same as the atomicity guarantee of transaction described in § 5.1. The `save` is an example for RC transaction.

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. MAV semantics is useful for maintaining the integrity of foreign key constraints, materialized views and secondary updates [Bailis et al. 2013a]. In order to implement MAV, a store only needs to keep track of the set of transactions S_t witnessed by the running transaction, and before performing an operation at some replica, ensure that the replica includes all the transactions in S_t. Hence, MAV is coordination-free. MAV semantics is captured with the following contract:

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

whose semantics is illustrated in the Figure 7(b).

ANSI RR semantics requires that the transaction witness a snapshot of the data store state. Importantly, this snapshot can be obtained from any replica, and hence RR is coordination-free. An example for such a transaction is the `totalBalance` transaction. The semantics of RR is captured by the following contract:

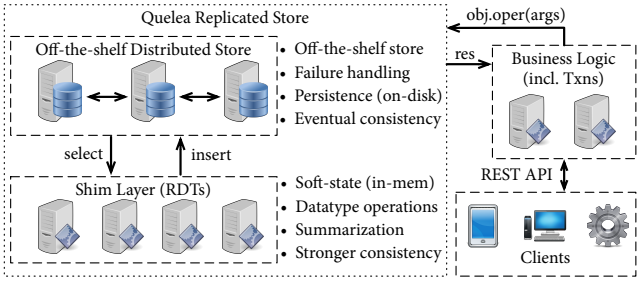


Figure 8. Implementation Model.

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

whose semantics is illustrated in the Figure 7(c).

5.4 Classification

Similar to operation-level contracts, with respect to $\leq$ relation, the coordination-free transaction semantics described here form a total order: $\psi_{\mathsf{rc}} \leq \psi_{\mathsf{mav}} \leq \psi_{\mathsf{rr}}$. The transaction classification is also similar to the operation-level contract classification presented in Figure 6; given a contract ψ on a transaction, we start from the weakest transaction contract ψ_{rc}, and progressively compare its strength to the known transaction contracts until we find a isolation level under which ψ can be safely discharged. Otherwise, we report a type error.

6. Implementation

QUELEA is implemented as a shallow extension of GHC Haskell and runs on top of Cassandra, an off-the-shelf eventually consistent distributed data (or backing) store responsible for all data management issues (i.e., replication, fault tolerance, availability, and convergence). Template Haskell is used to implement static contract classification, and proof obligations are discharged with the help of the Z3 [Z3] SMT solver. Figure 8 illustrates the overall system architecture.

Replicated data types and various consistency semantics are implemented and enforced in the *shim layer*. Our implementation supports eventual, causal, and strong consistency for data type operations, and RC, MAV, and RR semantics 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.

Each new effect in QUELEA is realized as a row insertion in Cassandra, and the state of an object is the set of all corresponding rows. The shim layer maintains a causally consistent in-memory snapshot of a subset of objects in the backing store, by explicitly tracking dependencies introduced between effects due to visibility, session and same transaction relations. Dependence tracking is similar to the techniques presented in [Bailis et al. 2013b] and [Lloyd et al. 2013]. Because Cassandra provides durability, convergence, and fault tolerance, each shim layer node simply acts as a soft-state cache, with no inter-node communication, and can safely be terminated at any point. Similarly, new shim layer nodes can be spawned on demand.

The shim layer nodes periodically fetch updates from the backing store for eventually consistent operations, and on-demand for causally consistent and strongly consistent operations. Strongly consistent operations are performed after obtaining exclusive leases on objects. The lease mechanism is implemented with the help of Cassandra's support for conditional updates and expiring columns. Cassandra does not provide general-purpose transactions. Since the transaction guarantees provided by QUELEA are coordination-free [Bailis et al. 2013a], we realize efficient implementations by explicitly tracking dependencies between operations and transactions. Importantly, the weaker isolation semantics of transactions in QUELEA permit transactions to be discharged if at least one shim layer node is reachable.

We utilize the `summarize` function (§ 3.1.1) to summarize the 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. Shim layer summarization is straight-forward; a summarization thread takes the local lock on the cached object, and replaces its state with the summarized state. The shim layer node only remains unavailable for that particular object during summarization (usually a few milliseconds). Performing summarization in the backing store is more complicated since the whole process needs to be atomic from a client's perspective, but Cassandra does not provide multi-row transactions. We have engineered and implemented a scalable summarization mechanism for the backing store that permits concurrent client operations, but nonetheless prohibits these operations from witnessing intermediate states of the summarization process.

7. Evaluation

We present an evaluation study of our implementation, report contract profiles of benchmark programs, and illustrate the performance benefits of fine-grained consistency classification on operations and transactions. We also evaluate the impact of the summarization. We have implemented the following applications, which includes individual RDTs as well as larger applications composed of several RDTs:

- **LWW register**: A last-write-wins register that provides read and write operations, where the read returns the value of the latest write.
- **DynamoDB register**: An integer register that allows eventual and strong puts and gets, conditional puts, increment and decrement operations.
- **Bank account**: Our running example.
- **Shopping list**: A collaborative shopping list that allows concurrent addition and deletion of items.

Benchmark	LOC	#T	EC	CC	SC	RC	MAV	RR
LWW Reg	108	1	2	2	2	0	0	0
DynamoDB	126	1	3	1	2	0	0	0
Bank Account	155	1	1	1	1	1	0	1
Shopping List	140	1	2	1	1	0	0	0
Online store	340	4	9	1	0	2	0	1
RUBiS	640	6	14	2	1	4	2	0
Microblog	659	5	13	6	1	6	3	1

Table 1. The distribution of classified contracts. #T refers to the number of tables in the application. The columns 4-6 (7-9) represent operations (transactions) assigned to this consistency (isolation) level.

- **Online store**: An online store with shopping cart functionality and dynamically changing item prices. The checkout process verifies that the customer only pays the accepted price.
- **RUBiS**: An eBay-like auction site [RUBiS]. The application allows users to browse items, bid for items on sale, and pay for items from a wallet modeled after a bank account.
- **Microblog**: A twitter-like microblogging site, modeled after Twissandra [Twissandra]. The application allows adding new users, adding and replying to tweets, following, unfollowing and blocking users, and fetching a user's timeline, userline, followers and following.

The distribution of contracts in these applications is given in Table 1. We see that majority of the operations and transactions are classified as eventually consistent and RC, respectively. Operation contracts are used to enforce integrity and visibility constraints on individual fields in the tables. Transactions are mainly used to consistently modify and access related fields across tables. In QUELEA, the contract classification process is completely performed at compile time and has no overheads at runtime. The proof obligations associated with contract classification is discharged through the Z3 SMT Solver. Across our benchmarks, classifying a contract took 11.5 milliseconds on average.

For our performance evaluation, we deploy QUELEA applications in *clusters*, where each cluster is composed of five fully replicated Cassandra replicas within the same datacenter. We instantiate one shim layer node co-located on the same VM as a Cassandra replica. Clients are instantiated within the same data center as the store, and run transactions. We deploy each cluster and client node on an `c3.4xlarge` Amazon EC2 instance. We call this a `1DC` configuration. For our geo-distributed experiments (`2DC`), we instantiate 2 clusters, each with five nodes, and place the clusters on US-east (Virginia) and US-west (Oregon) locations. The average inter-region latency was 85ms.

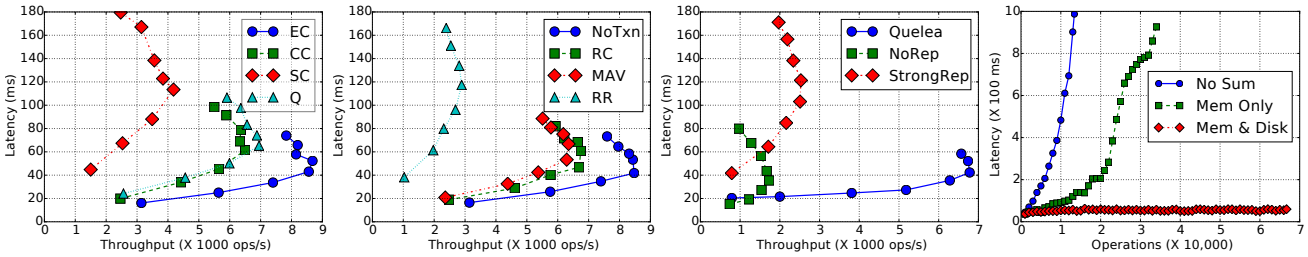
Figure 9(a) shows throughput vs. latency of operations in the bank account example as we increase the number of clients in a `1DC` configuration. Our client workload was generated using the YCSB benchmark [Cooper et al. 2010]. The benchmark uniformly chooses from 100,000 keys, where the operation spread was 25% withdraw, 25% deposit and 50% getBalance, which corresponds to the default 50:50 read:write mix in YCSB. We increased the number of clients from 128 to 1024, and each experiment ran for 180 seconds.

The lines marked EC and CC correspond to all operations (including `withdraw`) being assigned EC and CC consistency levels. These levels compromise correctness as `withdraw` has to be an SC operation. The SC line corresponds to a configuration where all operations are strongly consistent; this ensures application correctness, at the cost of performance. QUELEA corresponds to our implementation, which classifies operations based on their contract specifications. With 512 clients, the QUELEA implementation was within 41% of the latency and 18% of the throughput of EC, whereas SC operations had 162% higher latency and 52% lower throughput than EC operations. Observe that there is a point in each case after which the latency increases while the throughput decreases; these correspond to points where the store becomes saturated with client requests. In a `2DC` configuration (not shown here), the average latency of SC operations with 512 clients increased by 9.4× due to the cost of geo-distributed coordination, whereas QUELEA operations were only 2.2× slower, mainly due to the increased cost of `withdraw` operations. Importantly, the latency of `getBalance` and `deposit` remained almost the same, illustrating the benefit of fine-grained contract classification.

We compare the performance of different transaction isolation level choices in Figure 9(b) using the LWW register. The numbers were obtained under a 1DC configuration. The YCSB workload was modified to issue 10 operations per transaction, with a default 50:50 read:write mix. Each operation is assumed to be eventually consistent. NoTxn corresponds to a configuration that does not use transactions. Compared to this, RC is only 12% shower in terms of latency with 512 clients, whereas RR is 2.3X slower. The difference between RC and NoTxn is due to the meta-data overhead of recording transaction information in the object state. For RR transactions, the cost of capturing and maintaining a snapshot is the biggest source of overhead.

We also compared (not shown) the performance of EC LWW operations directly against Cassandra, which uses last-writer-wins as the only convergence semantics. While Cassandra provides no stronger-than-eventual consistency properties, QUELEA was within 30%(20%) of latency(throughput) of Cassandra with 512 clients, supporting our thesis that programmers only have to incur relatively low overhead for a more expressive programming model which provides stronger provable consistency guarantees.

Figure 9(c) compares the QUELEA implementation of RUBiS in a `1DC` configuration against a single replica (NoRep) and a strongly replicated (StrongRep) `1DC` deployment. The benchmark uses the default RUBiS bidding mix,



(a) Bank account (b) LWW transactions (c) RUBiS bidding mix (d) Impact of summarization

Figure 9. Quelea Performance.

which has 15% read-write interactions, which is representative of the auction workload. Without replication, NoRep trivially provides strong consistency. However, this deployment does not scale beyond 1750 operations per second. Strong replication offers better throughput at the cost of greater latency due to inter-replica coordination. The QUELEA deployment offers the benefit of replication, while only paying the cost of coordination when necessary.

Finally, we study the impact of summarization in Figure 9(d). We use 128 clients and a single QUELEA replica, with all clients operating on the *same* LWW register to stress test the summarization mechanism. The shim layer cache (memory) is summarized every 64 updates, while the updates in the backing store (disk) are summarized every 4096 updates. Each point in the graph represents the average latency of the previous 1000 operations. Each experiment is run for one minute. Without summarization, the average latency of operations increases exponentially to almost one second, and only 13K operations were performed in a minute. Since every operation has to reduce over the set of all previous operations, operations take increasingly more time to complete since they must contend with an ever growing set. With summarization only in memory, performance still degrades due to the cost of fetching all previous updates from the backing store into the shim layer. Fetching the latest updates from the backing store is essential for SC operations. With summarization enabled on both disk and memory, latency does not increase over time, and the implementation realizes throughput of 67K operations/minute.

8. Related Work

Operation-based RDTs have been widely studied in terms of their algorithmic properties [Shapiro et al. 2011; Burckhardt et al. 2014], and several systems utilize this model to construct distributed data structures [Lakshman and Malik 2010; Petersen et al. 1997; Balakrishnan et al. 2013]. These systems typically propose to implement the datatypes directly over a cluster of nodes, and only focus on basic eventual consistency. Hence, these systems implement custom solutions for durability and fault-tolerance. QUELEA realizes RDTs stronger consistency models on top of off-the-shelf eventually consistent distributed stores. In this respect, QUELEA is similar to [Bailis et al. 2013b] where causal consistency is achieved through a shim layer on Cassandra, which explicitly tracks and enforces dependencies between updates. However, [Bailis et al. 2013b] does not support user-defined RDTs, automatic contract classification and transactions.

Since eventual consistency alone is insufficient to build correct applications, several systems [Petersen et al. 1997; Terry et al. 2013; Li et al. 2012] propose a lattice of stronger consistency levels. Similarly, traditional database processing systems [Berenson et al. 1995] and their replicated variants [Bailis et al. 2013a] propose weaker isolation levels for performance. In these systems, the onus is on the developer to choose the correct consistency(isolation) level for operations(transactions). QUELEA relieves the developer of this burden, and instead expects contracts expressing declarative visibility requirements.

Our contract language is inspired by the axiomatic description of RDT semantics proposed by [Burckhardt et al. 2014]. While they use axioms for formal verification of correctness of an RDT implementation, we utilize them as a means for the user to express the desired consistency guarantees in the application. Similar to their work, our contract language does not incorporate real (i.e., wall-clock) time. Hence, it cannot describe store semantics such as recency or bounded-staleness guarantees offered by certain stores [Terry et al. 2013].

Several conditions have been proposed to judge whether an operation on a replicated data object needs coordination or not. [Alvaro et al. 2011] defines *logical monotonicity* as a sufficient condition for coordination freedom, and proposes a consistency analysis that marks code regions performing non-monotonic reasoning (eg: aggregations, such as `COUNT`) as potential coordination points. [Bailis et al. 2014] and [Li et al. 2014] define *invariant confluence* and *invariant safety*, respectively, as conditions for safely executing an operation without coordination. [Li et al. 2014] also proposes a program analysis that conservatively marks certain operations as *blue* (coordination not required), while marking the remaining as *red* (coordination required). Unlike QUELEA, these works focus on a coarse-grained classification of consistency as eventual or strong, and do not focus on finer-grained transaction isolation levels. However, the analyses they propose relieve programmers of the burden to tag operations with consistency levels. Indeed, we do consider au-

tomatic inference of consistency contracts from application-specific integrity constraints as the next step for QUELEA.

9. Conclusions

This paper presents QUELEA, a shallow Haskell extension for declarative programming over ECDS. The key idea underlying QUELEA's design is the automatic classification of fine-grained consistency contracts on operations and distributed transactions with respect to the consistency and isolation levels offered by the store. Our contract language is carefully crafted from a decidable subset of first-order logic, enabling the use of automated verification tools to discharge the proof obligations associated with contract classification. We realize an instantiation of QUELEA on top of off-the-shelf distributed store, Cassandra, and illustrate the benefit of fine-grained contract classification by implementing and evaluating several scalable applications.

References

P. Alvaro, N. Conway, J. Hellerstein, and W. R. Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 249–260, 2011. URL http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper35.pdf.

P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly Available Transactions: Virtues and Limitations. *PVLDB*, 7(3):181–192, 2013a.

P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013b. ACM. ISBN 978-1-4503-2037-5. doi: 10.1145/2463676.2465279.

P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination-Avoiding Database Systems. *CoRR*, abs/1402.2237, 2014. URL http://arxiv.org/abs/1402.2237.

M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 325–340, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522732.

H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. 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. ISBN 0-89791-731-6. doi: 10.1145/223784.223785.

E. Brewer. Towards Robust Distributed Systems (Invited Talk), 2000.

S. Burckhardt, D. Leijen, M. Fähndrich, and M. 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. ISBN 978-3-642-28868-5. doi: 10.1007/978-3-642-28869-2_4.

S. Burckhardt, A. Gotsman, H. Yang, and M. 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. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535848.

B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. 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. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807152.

G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. 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. ISBN 978-1-59593-591-5. doi: 10.1145/1294261.1294281.

S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002. ISSN 0163-5700. doi: 10.1145/564585.564601.

M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990. ISSN 0164-0925. doi: 10.1145/78969.78972.

A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Operating Systems Review*, 44(2):35–40, Apr. 2010. ISSN 0163-5980. doi: 10.1145/1773912.1773922.

C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. 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. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387906.

C. Li, J. a. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the Choice of Consistency Levels in Replicated Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 281–292, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-10-2. URL http://dl.acm.org/citation.cfm?id=2643634.2643664.

W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043593.

W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger Semantics for Low-latency Geo-replicated Storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 313–328, Berkeley, CA, USA, 2013. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2482626.2482657.

C. H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631–653, Oct. 1979. ISSN 0004-5411. doi: 10.1145/322154.322158.

K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 288–301, New York, NY, USA, 1997. ACM. ISBN 0-89791-916-5. doi: 10.1145/268998.266711.

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

M. Shapiro, N. Preguia, C. Baquero, and M. Zawirski. Conflict-Free Replicated Data Types. In X. Dfago, F. Petit, and V. Vil-

lain, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-24549-7. doi: 10.1007/978-3-642-24550-3_29.

S. Sivasubramanian. Amazon dynamoDB: A Seamlessly Scalable Non-relational Database Service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 729–730, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1247-9. doi: 10.1145/2213836.2213945.

Y. Sovran, R. Power, M. K. Aguilera, and J. 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. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043592.

D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based Service Level Agreements for Cloud Storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522731.

Twissandra. Twitter clone on Cassandra, 2014. URL http://twissandra.com/. Accessed: 2014-11-4 13:21:00.

Z3. High-performance Theorem Prover, 2014. URL http://z3.codeplex.com/. Accessed: 2014-11-4 13:21:00.