1. INTRODUCTION

introduction

2. DEPENDENCY GRAPHS AND SERIALIZABILITY ANOMALIES

an overview of previous work on anomaly detection

3. EVENTUAL SERIALIZABILITY

In this section, we present the formal notion of *Eventual Serializability*, ES, and intuitively argue about its usefulness. Consequently, we will introduce our ES anomaly detection algorithm as an extension to a previously known static analysis technique [2]. We will finally present a handful of canonical examples from the literature and discuss how they all implicitly conform to our notion of correctness.

3.1 Definitions

In this section, we assume (for the brevity of this write-up) the SQL language and all definitions and operational rules from [2] are already presented to the readers.

Given an abstract execution $\chi = (\Sigma, \mathsf{vis}, \mathsf{ar})$, where Σ is a set of transaction instances, $\mathsf{vis} \subseteq \Sigma \times \Sigma$ is an anti-symmetric, irreflexive relation and $\mathsf{ar} \subseteq \Sigma \times \Sigma$ is a total, anti-symmetric order on Σ such that $\mathsf{vis} \subseteq \mathsf{ar}$, we define the *write set* of Σ as: $[\Sigma]_{\mathtt{wri}} = \{\mathtt{wri}(r, f, n) \mid \sigma \vdash \mathtt{wri}(r, f, n),\ \sigma \in \Sigma, n \in \mathbb{Z}, r \in \mathcal{R}, f \in \mathtt{Fields}\}$, which is the set of all writes to any field and record occured in any of the executed transactions.

Assuming a total order of operations inside a transaction, txno, and a mapping `TXN()` from operations to their enclosing transaction, we define a total order of writes within an execution as follows:

$$w_1 \xrightarrow{\mathsf{wrio}} w_2 \Leftrightarrow \mathtt{TXN}(w_1) \xrightarrow{\mathsf{ar}} \mathtt{TXN}(w_2) \vee w_1 \xrightarrow{\mathsf{txno}} w_2$$

We now define the *eventual state* of the given execution, denoted by $\vec{\chi}^s$, as the set of all writes performed on the database and the absolute order between them, which is in accordance with the fact that read operations do not affect the final state[1]. Formally:

$$\vec{\chi}^s = ([\Sigma]_{\mathtt{wri}}, \mathsf{wrio})$$

Now we are ready to define ES, as our notion of application correctness:

Definition 1. An abstract execution $\chi_1 = (\Sigma, \mathsf{vis}, \mathsf{ar})$ is said to be eventually serializable, if there exists another abstract execution χ_2 which is serializable and has the same eventual state, i.e. $\chi_2 \models \Psi_{Ser}$ and $\vec{\chi_1}^s = \vec{\chi_2}^s$.

Note that χ_2 in the above definition can theoretically include additional read-only transactions or altered transactions containing additional read operations. However, since serializability is more difficult to be achieved for executions with extra transactions or operations, in order to prove a given execution eventually serializable, the above definition intuitively guides us to first strip certain transactions and operations off the original execution and then prove classic serializability for the trimmed execution.

[1]read operations might affect writes indirectly by providing them with values to be written; however, those values are already captured in the write operations and it suffices to only include write operations in our definition

3.2 Dangling Reads

In this part, we will introduce the notion of *dangling reads* as the ones whose sole purpose is to inform users about the current state of the database, meaning that the values they return are not used anywhere in the program (inside the following statements or path conditions). For example, the `getBalance` transaction in a banking application contains a single read operation on the current balance of a given account, whose output is simply printed on the user's screen but is not used anywhere else in the transaction.

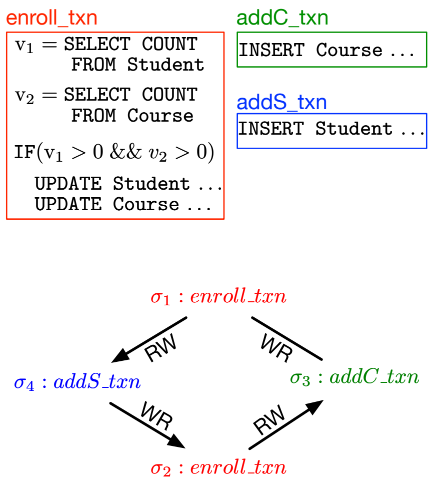
We define a mapping $\zeta : \mathtt{rd} \mapsto \mathtt{bool}$, to keep track of read operations that are dangling (which are mapped to `false`). Such a mapping can be straightforwardly generated by a static analysis of the input code (similar to the dead-code-elimination step in any basic compiler) and in fact, we will present our extended version of such an algorithm in the rest of this section.

Following [2], in order to detect eventual serializability violations, we are going to build dependency graphs form abstract executions and look for cycles. However, as we explained above, certain operations are harmless against ES and thus, cycles containing such operations must be excluded from our analysis. We define a cycle C to be *effectful*, if and only if, $\forall (r : \mathtt{rd}) \in C.\ \zeta(r)$.

Theorem 1. Given an abstract execution $\chi = (\Sigma, \mathsf{vis}, \mathsf{ar})$, if there is no effectful cycle in the dependency graph G_χ then χ is eventually serializable.

3.2.1 conditionally dangling reads

The presented notion of dangling reads is already good enough to exclude many practically harmless violations from our analysis; however, we can further augment read operations with *conditions* under which they will become dangling. For example, consider the following transactions (top) in the dependency graph (bottom), from the courseware application [1]:



enroll_txn
```
v1 = SELECT COUNT
       FROM Student
v2 = SELECT COUNT
       FROM Course
IF(v1 > 0 && v2 > 0)
  UPDATE Student ...
  UPDATE Course ...
```

addC_txn
```
INSERT Course ...
```

addS_txn
```
INSERT Student ...
```

Figure 1: a harmless violation

The situation in fig.1 is clearly a violation of classic serializability: σ_1 witnesses effects of σ_3 but not σ_4 and σ_2 witnesses effects of σ_4 but not σ_3, which creates a cycle as is depicted. Moreover, a naive analysis looking for dangling reads would similarly consider the given case a violation of ES, since values read by both reads in enroll_txn are used later inside the transaction and neither is seemingly dangling.

However, as acute readers may have noticed, the values read in enroll_txn are subsequently used only under certain conditions. In other words, the read operations can potentially become dangling depending on the values they return. Following this observation, the serializability violation in fig.1 becomes harmless against ES as the guard ($v_1 > 0$ && $v_2 > 0$) can never be satisfied according to other conditions present in the scenario.

Our goal in this paper, similar to [2], is to construct a formula in FOL for a given transactional program $\mathbb{T}$ and a consistency specification Ψ, such that any valid abstract execution χ of $\mathbb{T}$ under Ψ and its dependency graph G_χ is a satisfying model of the formula. Consequently, we can extend ζ to map each read operation to an FOL formula (instead of a boolean), specifying the conditions under which the read values become useful; these conditions can be used later in our final formulation for a more accurate analysis. As like before, $\zeta(\mathtt{rd}) = \mathtt{true}$ (never dangling) and $\zeta(\mathtt{rd}) = \mathtt{false}$ (always dangling) are the two extreme possible values of the map ζ.

Before presenting our algorithm to statically construct ζ for a given transactional SQL program $\mathcal{T}$, we first define $\mathsf{src}(\mathsf{c})$ to be the set of all read operations from which the values *used* in c are being sourced. Note that a value can be used in any statement and not just in writes, e.g. inside the WHERE clause of a SELECT. If the values used in c are all constants or the input parameters of the transaction, $\mathsf{src}(\mathsf{c})$ would trivially be empty[2].

Algorithm 1 Recursive construction of ζ, given a transactional program $\mathcal{T}$, initially called with $\varphi = \mathtt{true}$

```
procedure ExtractDanglig(c,φ)
    match c with
        case: $c_1 \in \mathtt{Stmts}(\mathcal{T})$
            for rd in $\mathsf{src}(c_1)$ do
                $\zeta(\mathtt{rd}) \leftarrow \zeta(\mathtt{rd}) \lor \varphi$
            end
        case: $c_1 ; c_2$
            ExtractDanglig($c_1$,$\varphi$)
            ExtractDanglig($c_2$,$\varphi$)
        case: IF $\phi$ THEN $c_1$ ELSE $c_2$
            ExtractDanglig($c_1$,$\varphi \land \phi$)
            ExtractDanglig($c_2$,$\varphi \land \neg\phi$)
        case: FOREACH $v_1$ IN $v_2$ DO $c_1$ END
            ExtractDanglig($c_1$,$\varphi$)
end procedure
```

Assuming that ζ initially maps all read operations to false, alg.1 recursively populates ζ, constructing a map from each read operation in $\mathtt{Stmts}(\mathcal{T})$ to a formula made of the disjunction of IF conditionals enclosing statements which are supplied by a value from that read.

3.3 FOL Encoding

In this section, we present our FOL encoding of transactional program $\mathbb{T}$, as a formula parametric over a consistency specification Ψ, such that any valid execution of $\mathbb{T}$ under Ψ and its dependency graph is a satisfying model of the formula. We base our work on all of the definitions and vocabulary from [2] and present our slightly modified version of their rules relating transactional programs to dependency relations.

As the result of choosing ES as our notion of program correctness, which is weaker than classic serializability, for each pair of transaction types $\mathcal{T}_1, \mathcal{T}_2 \in \mathbb{T}$ and a dependency type $\mathcal{R} \in \{\mathsf{WR}, \mathsf{RW}\}$, we modified the rules populating $\eta^{\mathcal{R}\rightarrow,\mathcal{T}_1,\mathcal{T}_2}$ and $\eta^{\rightarrow\mathsf{WR},\mathcal{T}_1,\mathcal{T}_2}$ (parametrized over statements c_1 and c_2, exactly one of which must be a SELECT), in a way to additionally incorporate the conditions under which the SELECT statement is not dangling. For example, our version of the rule that constructs the necessary entailments of an RW dependency relation is presented in the fig.2:

RW-Update-Select

$$\frac{\begin{gathered}\mathsf{c}_1 \equiv \mathtt{UPDATE\ SET\ f} = \mathtt{e}_\mathsf{c}\ \mathtt{WHERE}\ \phi_1 \\ \mathsf{c}_2 \equiv \mathtt{SELECT\ f\ AS\ v\ WHERE}\ \phi_2 \\ \mathsf{c}_1 \in \mathtt{Stmts}(\mathcal{T}_1) \quad \mathsf{c}_2 \in \mathtt{Stmts}(\mathcal{T}_2) \quad \Gamma(t_1) = \mathcal{T}_1 \quad \Gamma(t_2) = \mathcal{T}_2\end{gathered}}{\begin{aligned}\eta^{\mathsf{RW}\rightarrow,\mathcal{T}_2,\mathcal{T}_1}_{\mathsf{c}_2,\mathsf{c}_1}(t_2,t_1) = (\exists r.\ & \mathcal{V}(\llbracket \Lambda_{\mathcal{T}_1}(\mathsf{c}_1)\rrbracket_{t_1}) \land \mathcal{V}(\llbracket \phi_1 \rrbracket_{t_1,r}) \land \\ & \mathcal{V}(\llbracket \Lambda_{\mathcal{T}_2}(\mathsf{c}_2)\rrbracket_{t_2}) \land \mathcal{V}(\llbracket \phi_2 \rrbracket_{t_2,r}) \land \\ & \mathtt{Alive}(r,t_1) \land \mathsf{RW}(r,\mathsf{f},t_2,t_1) \land \mathcal{V}(\llbracket \zeta_{\mathcal{T}_2}(\mathsf{c}_2)\rrbracket_{t_2}))\end{aligned}}$$

Figure 2: a modified encoding rule

3.4 Examples in the Literature

to be done.

4. ROT EXTRACTION

transaction chopping procedure.

5. APPLICABILITY AND PREVALENCE

review of all benchmarks verified and prevalence of extractable ROTs.

6. EVALUATION

safety and performance evaluation

7. RELATED WORKS

detailed explanation please.

8. CONCLUSIONS

conclusions and future works

9. REFERENCES

[1] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 371–384, New York, NY, USA, 2016. ACM.

[2]we skip the formal algorithm for constructing $\mathsf{src}(\mathsf{c})$ in this paper, as it is very straightforward using a simple data-flow analysis

[2] K. Nagar and S. Jagannathan. Automated detection of serializability violations under weak consistency. `https://github.com/Kiarahmani/deSQLifier/blob/master/documents/Concure18.pdf/`, Concur 2018. [Privately Online; accessed 18-June-2018].