




On the Complexity of Checking Mixed Isolation Levels for SQL Transactions

Ahmed Bouajjani¹, Constantin Enea², and Enrique Román-Calvo¹



¹ Université Paris Cité, CNRS, IRIF
{abou, calvo}@irif.fr

² LIX, École Polytechnique, CNRS and
Institut Polytechnique de Paris
cenea@lix.polytechnique.fr

Abstract. Concurrent accesses to databases are typically grouped in transactions which define units of work that should be isolated from other concurrent computations and resilient to failures. Modern databases provide different levels of isolation for transactions that correspond to different trade-offs between consistency and throughput. Quite often, an application can use transactions with different isolation levels at the same time. In this work, we investigate the problem of testing isolation level implementations in databases, i.e., checking whether a given execution composed of multiple transactions adheres to the prescribed isolation level semantics. We particularly focus on transactions formed of SQL queries and the use of multiple isolation levels at the same time. We show that many restrictions of this problem are NP-complete and provide an algorithm which is exponential-time in the worst-case, polynomial-time in relevant cases, and practically efficient.

1 Introduction

Concurrent accesses to databases are typically grouped in transactions which define units of work that should be isolated from other concurrent computations and resilient to failures. Modern databases provide different levels of isolation for transactions with different trade-offs between consistency and throughput. The strongest isolation level, *Serializability* [21], provides the illusion that transactions are executed atomically one after another in a serial order. Serializability incurs a high cost in throughput. For performance, databases provide weaker isolation levels, e.g., *Snapshot Isolation* [5] or *Read Committed* [5].

The concurrency control protocols used in large-scale databases to implement isolation levels are difficult to build and test. For instance, the black-box testing framework Jepsen [19] found a remarkably large number of subtle problems in many production databases.

In this work, we focus on testing the isolation level implementations in databases, and more precisely, on the problem of checking whether a given execution adheres to the prescribed isolation level semantics. Inspired by scenarios that arise in commercial software [22], we consider a quite generic version of the problem where transactions are formed of SQL queries and *multiple* isolation

levels are used at the same time, i.e., each transaction is assigned a possibly different isolation level (the survey in [22] found that 32% of the respondents use such “heterogeneous” configurations). Previous work [21,6] studied the complexity of the problem when transactions are formed of reads and writes on a *static* set of keys (variables), and all transactions have the *same* isolation level.

As a first contribution, we introduce a formal semantics for executions with SQL transactions and a range of isolation levels, including serializability, snapshot isolation, prefix consistency, and read committed. Dealing with SQL queries is more challenging than classic reads and writes of a *static* set of keys (as assumed in previous formalizations [11,6]). SQL insert and delete queries change the set of locations at runtime and the set of locations returned by an SQL query depends on their values (the values are restricted to satisfy **WHERE** clauses).

We define an abstract model for executions, called *history*, where every SQL query that inspects the database (has a **WHERE** clause) is associated with a set of SQL queries that wrote the inspected values. This relation is called a *write-read* relation (also known as read-from). This is similar to associating reads to writes in defining memory models. We consider two classes of histories depending on the “completeness” of the write-read relation. To define a formal semantics of isolation levels, we need a complete write-read relation in the sense that for instance, an SQL select is associated with a write *for every* possible row (identified by its primary key) in the database, even if that row is *not* returned by the select because it does not satisfy the **WHERE** clause. Not returning a row is an observable effect that needs to be justified by the semantics. Such *full* histories can not be constructed by interacting with the database in a black-box manner (a desirable condition in testing) when only the outputs returned by queries can be observed. Therefore, we introduce the class of *client* histories where the write-read concerns only rows that are *returned* by a query. The consistency of a client history is defined as the existence of an extension of the write-read to a full history which satisfies the semantics. The semantics on full histories combines axioms from previous work [6] in a way that is directed by SQL queries that inspect the database and the isolation level of the transaction they belong to. This axiomatic semantics is validated by showing that it is satisfied by a standard operational semantics inspired by real implementations.

We study the complexity of checking if a full or client history is consistent, it satisfies the prescribed isolation levels. This problem is more complex for client histories, which record less dependencies and need to be extended to full ones.

For full histories, we show that the complexity of consistency checking matches previous results in the reads and writes model when all transactions have the same isolation level [6]: polynomial time for the so-called saturable isolation levels, and NP-complete for stronger levels like Snapshot Isolation or Serializability. The former is a new result that generalizes the work of [6] and exposes the key ideas for achieving polynomial-time complexity, while the latter is a consequence of the previous results.

We show that consistency checking becomes NP-complete for client histories even for saturable isolation levels. It remains NP-complete regardless of the

expressiveness of **WHERE** clauses (for this stronger result we define another class of histories called *partial-observation*). The problem is NP-complete even if we bound the number of sessions. In general, transactions are organized in *sessions* [23], an abstraction of the sequence of transactions performed during the execution of an application (the counterpart of threads in shared memory). This case is interesting because it is polynomial-time in the read/write model [6].

As a counterpart to these negative results, we introduce an algorithm for checking consistency of client histories which is exponential-time in the worst case, but polynomial time in relevant cases. Given a client history as input, this algorithm combines an enumeration of extensions towards a full history with a search for a total commit order that satisfies the required axioms. The commit order represents the order in which transactions are committed in the database and it is an essential artifact for defining isolation levels. For efficiency, the algorithm uses a non-trivial enumeration of extensions that are *not* necessarily full but contain enough information to validate consistency. The search for a commit order is a non-trivial generalization of an algorithm by Biswas et al. [6] which concerned only serializability. This generalization applies to all practical isolation levels and combinations thereof. We evaluate an implementation of this algorithm on histories generated by PostgreSQL with a number of applications from BenchBase [12], e.g., the TPC-C model of a store and a model of Twitter. This evaluation shows that the algorithm is quite efficient in practice and scales well to typical workloads used in testing databases.

To summarize, we provide the first results concerning the complexity of checking the correctness of mixed isolation level implementations for SQL transactions. We introduce a formal specification for such implementations, and a first tool that can be used in testing their correctness.

2 Histories

2.1 Transactions

We model the database as a set of rows from an unbounded domain **Rows**. Each row is associated to a unique (primary) key from a domain **Keys**, given by the function $\text{key} : \text{Rows} \rightarrow \text{Keys}$. We consider client programs accessing the database from a number of parallel sessions, each session being a sequence of transactions defined by the following grammar:

$$\begin{aligned} \iota &\in \text{Iso} & a &\in \text{LVars} & R &\in 2^{\text{Rows}} & p &\in \text{Rows} \rightarrow \{0, 1\} & U &\in \text{Keys} \rightarrow \text{Rows} \\ \text{Transaction} &::= \text{begin}(\iota); \text{Body}; \text{commit} \\ \text{Body} &::= \text{Instr} \mid \text{Instr}; \text{Body} \\ \text{Instr} &::= \text{InstrDB} \mid a := \text{LEpr} \mid \text{if}(\text{LCond})\{\text{Instr}\} \\ \text{InstrDB} &::= a := \text{SELECT}(p) \mid \text{INSERT}(R) \mid \text{DELETE}(p) \mid \text{UPDATE}(p, U) \mid \text{abort} \end{aligned}$$

Each transaction is delimited by **begin** and **commit** instructions. The **begin** instruction defines an isolation level ι for the current transaction. The set of isolation levels **Iso** we consider in this work will be defined later. The body

contains standard SQL-like statements for accessing the database and standard assignments and conditionals for local computation. Local computation uses (transaction-)local variables from a set $LVars$. We use a, b, \dots to denote local variables. Expressions and Boolean conditions over local variables are denoted with $LExpr$ and $LCond$, respectively.

Concerning database accesses (sometimes called queries), we consider a simplified but representative subset of SQL: **SELECT**(p) returns the set of rows satisfying the predicate p and the result is stored in a local variable a . **INSERT**(R) inserts the set of rows R or updates them in case they already exist (this corresponds to **INSERT ON CONFLICT DO UPDATE** in PostgreSQL), and **DELETE**(p) deletes all the rows that satisfy p . Then, **UPDATE**(p, U) updates the rows satisfying p with values given by the map U , i.e., every row r in the database that satisfies p is replaced with $U(\text{key}(r))$, and **abort** aborts the current transaction. The predicate p corresponds to a **WHERE** clause in standard SQL.

2.2 Histories

We define a model of the interaction between a program and a database called *history* which abstracts away the local computation in the program and the internal behavior of the database. A history is a set of *events* representing the database accesses in the execution grouped by transaction, along with some relations between these events which explain the output of **SELECT** instructions.

An event is a tuple $\langle e, \text{type} \rangle$ where e is an *identifier* and type is one of **begin**, **commit**, **abort**, **SELECT**, **INSERT**, **DELETE** and **UPDATE**. \mathcal{E} denotes the set of events. For an event e of type **SELECT**, **DELETE**, or **UPDATE**, we use $\text{WHERE}(e)$ to denote the predicate p and for an **UPDATE** event e , we use $\text{SET}(e)$ to denote the map U .

We call **read** events the **SELECT** events that read the database to return a set of rows, and the **DELETE** and **UPDATE** events that read the database checking satisfaction of some predicate p . Similarly, we call **write** events the **INSERT**, **DELETE** and **UPDATE** events that modify the database. We also say that an event is of type **end** if it is either a **commit** or an **abort** event.

A *transaction log* $(t, \iota_t, E, \text{po}_t)$ is an identifier t , an *isolation level* identifier ι_t , and a finite set of events E along with a strict total order po_t on E , called *program order* (representing the order between instructions in the body of a transaction). The set E of events in a transaction log t is denoted by $\text{events}(t)$. For simplicity, we may use the term *transaction* instead of transaction log.

Isolation levels differ in the values returned by read events which are not preceded by a write on the same variable in the same transaction. We denote by $\text{reads}(t)$ the set of **read** events contained in t . Also, if t does *not* contain an **abort** event, the set of **write** events in t is denoted by $\text{writes}(t)$. If t contains an **abort** event, then we define $\text{writes}(t)$ to be empty. This is because the effect of aborted transactions (its set of writes) should not be visible to other transactions. The extension to sets of transaction logs is defined as usual.

To simplify the exposition we assume that for any given key $x \in \text{Keys}$, a transaction does not modify (insert/delete/update) a row with key x more than

once. Otherwise, under all isolation levels, only the last among multiple updates is observable in other transactions.

As expected, we assume that the minimal element of po_t is a **begin** event, if a **commit** or an **abort** event occurs, then it is maximal in po_t , and a log cannot contain both **commit** and **abort**. A transaction log without **commit** or **abort** is called *pending*. Otherwise, it is *complete*. A complete transaction log with a **commit** is *committed* and *aborted* otherwise.

A *history* contains a set of transaction logs (with distinct identifiers) ordered by a (partial) *session order* so that represents the order between transactions in the same session. It also includes a *write-read* relation wr which associates **write** events with **read** events. The **write** events associated to a **read** implicitly define the values observed (returned) by the **read** (read events do *not* include explicit values). Let T be a set of transaction logs. For every key $x \in \text{Keys}$ we consider a write-read relation $\text{wr}_x \subseteq \text{writes}(T) \times \text{reads}(T)$. The union of wr_x for every $x \in \text{Keys}$ is denoted by wr . We extend the relations wr and wr_x to pairs of transactions by $(t_1, t_2) \in \text{wr}$, resp., $(t_1, t_2) \in \text{wr}_x$, iff there exist events w in t_1 and r in t_2 , $t_2 \neq t_1$ s.t. $(w, r) \in \text{wr}$, resp., $(w, r) \in \text{wr}_x$. Analogously, we extend wr and wr_x to tuples formed of a transaction (containing a write) and a read event. We say that the transaction t_1 is *read* by the transaction t_2 when $(t_1, t_2) \in \text{wr}$. The inverse of wr_x is defined as usual and denoted by wr_x^{-1} . We assume that wr_x^{-1} is a partial function and thus, use $\text{wr}_x^{-1}(e)$ to denote the **write** event w such that $(w, e) \in \text{wr}_x$. We also use $\text{wr}_x^{-1}(e) \downarrow$ and $\text{wr}_x^{-1}(e) \uparrow$ to say that there exists a **write** w such that $(w, e) \in \text{wr}_x$ (resp. such **write** w does not exist).

To simplify the exposition, every history includes a distinguished transaction *init* preceding all the other transactions in so and inserting a row for every x . It represents the initial state and it is the only transaction that may insert as value \dagger_x (indicating that initially, no row with key x is present).

Definition 1. A history $(T, \text{so}, \text{wr})$ is a set of transaction logs T along with a strict partial session order so , and a write-read relation $\text{wr}_x \subseteq \text{writes}(T) \times \text{reads}(T)$ for each $x \in \text{Keys}$ s.t.

- the inverse of wr_x is a partial function,
- $\text{so} \cup \text{wr}$ is acyclic (here we use the extension of wr to pairs of transactions),
- if $(w, r) \in \text{wr}_x$, then $\text{value}_{\text{wr}}(w, x) \neq \perp$, where

$$\text{value}_{\text{wr}}(w, x) = \begin{cases} r & \text{if } w = \text{INSERT}(R) \wedge r \in R \wedge \text{key}(r) = x \\ \dagger_x & \text{if } w = \text{DELETE}(p) \wedge \text{wr}_x^{-1}(w) \downarrow \\ & \quad \wedge p(\text{value}_{\text{wr}}(\text{wr}_x^{-1}(w), x)) = 1 \\ U(x) & \text{if } w = \text{UPDATE}(p, U) \wedge \text{wr}_x^{-1}(w) \downarrow \\ & \quad \wedge p(\text{value}_{\text{wr}}(\text{wr}_x^{-1}(w), x)) = 1 \\ \perp & \text{otherwise} \end{cases}$$

The function wr_x^{-1} may be partial because some query may not read a key x , e.g., if the corresponding row does not satisfy the query predicate.

The function $\text{value}_{\text{wr}}(w, x)$ returns the row with key x written by the **write** event w . If w is an **INSERT**, it returns the inserted row with key x . If w is an

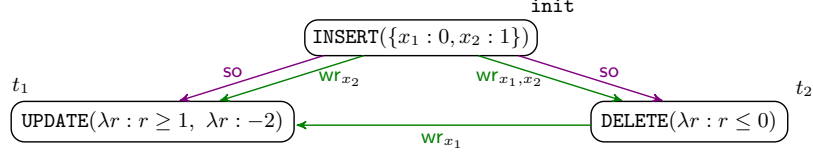


Fig. 1: An example of a history (isolation levels omitted for legibility). Arrows represent so and wr relations. Transaction init defines the initial state: row 0 with key x_1 and row 1 with key x_2 . Transaction t_2 reads x_1 and x_2 from init and deletes row with key x_1 (the only row satisfying predicate $\lambda r : r \leq 0$ corresponds to key x_1). Transaction t_1 reads x_1 from t_2 and x_2 from init , and updates only row with key x_2 as this is the only row satisfying predicate $\lambda r : r \geq 1$.

$\text{UPDATE}(\text{p}, \text{U})$ event, it returns the value of U on key x if w reads a value for key x that satisfies predicate p . If w is a $\text{DELETE}(\text{p})$, it returns the special value \dagger_x if w reads a value for key x that satisfies p . This special value indicates that the database does *not* contain a row with key x . In case no condition is satisfied, $\text{value}_{\text{wr}}(w, x)$ returns an undefined value \perp . We assume that the special values \dagger_x or \perp do not satisfy any predicate. Note that the recursion in the definition of $\text{value}_{\text{wr}}(w, x)$ terminates because wr is an acyclic relation.

Figure 1 shows an example of a history. For the UPDATE event w in t_1 , $\text{value}_{\text{wr}}(w, x_1) = \perp$ because this event reads x_1 from the DELETE event in t_2 ; while $\text{value}_{\text{wr}}(w, x_2) = -2$ as it reads x_2 from the INSERT event in init .

The set of transaction logs T in a history $h = (T, \text{so}, \text{wr})$ is denoted by $\text{tr}(h)$ and $\text{events}(h)$ is the union of $\text{events}(t)$ for every $t \in T$. For a history h and an event e in h , $\text{tr}(e)$ is the transaction t in h that contains e . We assume that each event belongs to only one transaction. Also, $\text{writes}(h) = \bigcup_{t \in \text{tr}(h)} \text{writes}(t)$ and $\text{reads}(h) = \bigcup_{t \in \text{tr}(h)} \text{reads}(t)$. We extend so to pairs of events by $(e_1, e_2) \in \text{so}$ if $(\text{tr}(e_1), \text{tr}(e_2)) \in \text{so}$. Also, $\text{po} = \bigcup_{t \in T} \text{po}_t$. We use h, h_1, h_2, \dots to range over histories.

For a history h , we say that an event r reads x in h whenever $\text{wr}_x^{-1}(r) \downarrow$. Also, we say that an event w writes x in h , denoted by w writes x , whenever $\text{value}_{\text{wr}}(w, x) \neq \perp$ and the transaction of w is *not* aborted. We extend the function value to transactions: $\text{value}_{\text{wr}}(t, x)$ equals $\text{value}_{\text{wr}}(w, x)$, where w is the maximal event in po_t that writes x .

2.3 Classes of histories

We define two classes of histories: (1) *full* histories which are required to define the semantics of isolation levels and (2) *client* histories which model what is observable from interacting with a database as a black-box.

Full histories model the fact that every read query “inspects” an entire snapshot of the database in order to for instance, select rows satisfying some predicate. Roughly, full histories contain a write-read dependency for every read and key. There is an exception which concerns “local” reads. If a transaction modifies a row with key x and then reads the same row, then it must always return the value written in the transaction. This holds under all isolation levels. In such

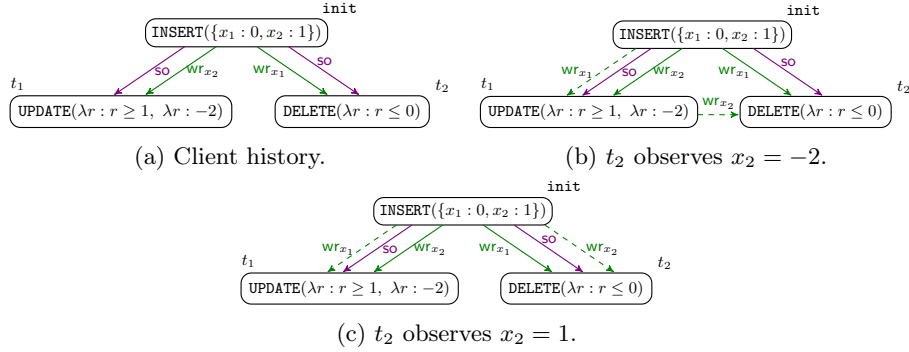


Fig. 2: Examples of a client history h and two possible extensions. The dashed edge belongs only to the extensions. The first extension is not a witness of h as t_1 writes -2 on x_2 and $\text{WHERE}(t_2)(-2) = 1$.

a case, there would be no write-read dependency because these dependencies model interference across different transactions. We say that a read r reads a key x *locally* if it is preceded in the same transaction by a write w that writes x .

Definition 2. A full history $(T, \text{so}, \text{wr})$ is a history where $\text{wr}_x^{-1}(r)$ is defined for all x and r , unless r reads x locally.

Client histories record less write-read dependencies compared to full histories, which is formalized by the *extends* relation.

Definition 3. A history $\bar{h} = (T, \text{so}, \bar{\text{wr}})$ extends another history $h = (T, \text{so}, \text{wr})$ if $\text{wr} \subseteq \bar{\text{wr}}$. We denote it by $h \subseteq \bar{h}$.

Definition 4. A client history $h = (T, \text{so}, \text{wr})$ is a history s.t. there is a full history $\bar{h} = (T, \text{so}, \bar{\text{wr}})$ with $h \subseteq \bar{h}$, and s.t. for every x , if $(w, r) \in \bar{\text{wr}}_x \setminus \text{wr}_x$ then $\text{WHERE}(r)(\text{value}_{\bar{\text{wr}}}(w, x)) = 0$. The history h' is called a witness of h .

Compared to a witness full history, a client history may omit write-read dependencies if the written values do *not* satisfy the predicate of the read query. These values would not be observable when interacting with the database as a black-box. This includes the case when the write is a **DELETE** (recall that the special value \dagger_x indicating deleted rows falsifies every predicate by convention). Figure 1 shows a full history as every query reads both x_1 and x_2 . Figure 2a shows a client history: transactions t_1, t_2 does not read x_2 and x_1 resp. Figure 2b is an extension but not a witness while Figure 2c is indeed a witness of it.

3 Axiomatic Semantics With Different Isolation Levels

We define an axiomatic semantics on histories where transactions can be assigned different isolation levels, which builds on the work of Biswas et al. [6].

3.1 Executions

An *execution* of a program is represented using a history with a set of transactions T along with a total order $\text{co} \subseteq T \times T$ called *commit order*. Intuitively, the commit order represents the order in which transactions are committed in the database.

Definition 5. An execution $\xi = (h, \text{co})$ is a history $h = (T, \text{so}, \text{wr})$ along with a commit order $\text{co} \subseteq T \times T$, such that transactions in the same session or that are read are necessarily committed in the same order: $\text{so} \cup \text{wr} \subseteq \text{co}$. ξ is called an execution of h .

For a transaction t , we use $t \in \xi$ to denote the fact that $t \in T$. Analogously, for an event e , we use $e \in \xi$ to denote that $e \in t$ and $t \in \xi$. The extension of a commit order to pairs of events or pairs of transactions and events is done in the obvious way.

3.2 Isolation Levels

Isolation levels enforce restrictions on the commit order in an execution that depend on the session order so and the write-read relation wr . An *isolation level* ι for a transaction t is a set of constraints called *axioms*. Intuitively, an axiom states that a read event $r \in t$ reads key x from transaction t_1 if t_1 is the latest transaction that writes x which is “visible” to r – latest refers to the commit order co . Formally, an axiom a is a predicate of the following form:

$$a(r) := \forall x, t_1, t_2. t_1 \neq t_2 \wedge (t_1, r) \in \text{wr}_x \wedge t_2 \text{ writes } x \wedge \text{vis}_a(t_2, r, x) \Rightarrow (t_2, t_1) \in \text{co} \quad (1)$$

where r is a read event from t .

The visibility relation of a vis_a is described by a formula of the form:

$$\text{vis}_a(\tau_0, \tau_{k+1}, x) : \exists \tau_1, \dots, \tau_k. \bigwedge_{i=1}^{k+1} (\tau_{i-1}, \tau_i) \in \text{Rel}_i \wedge \text{WrCons}_a(\tau_0, \dots, \tau_{k+1}, x) \quad (2)$$

with each Rel_i is defined by the grammar:

$$\text{Rel} ::= \text{po} \mid \text{so} \mid \text{wr} \mid \text{co} \mid \text{Rel} \cup \text{Rel} \mid \text{Rel}; \text{Rel} \mid \text{Rel}^+ \mid \text{Rel}^* \quad (3)$$

This formula states that τ_0 (which is t_2 in Eq.1) is connected to τ_{k+1} (which is r in Eq.1) by a path of dependencies that go through some intermediate transactions or events τ_1, \dots, τ_k . Every relation used in such a path is described based on po , so , wr and co using union \cup , composition of relations $;$, and transitive closure operators. Finally, extra requirements on the intermediate transactions s.t. writing a different key $y \neq x$ are encapsulated in the predicate $\text{WrCons}_a(\tau_0, \dots, \tau_k, x)$.

Each axiom a uses a specific visibility relation denoted by vis_a . $\text{vis}(\iota)$ denotes the set of visibility relations used in axioms defining an isolation level ι .

Figure 3 shows two axioms which correspond to their homonymous isolation levels [6]: *Read Committed* (RC) and *Serializability* (SER). SER states that t_2 is

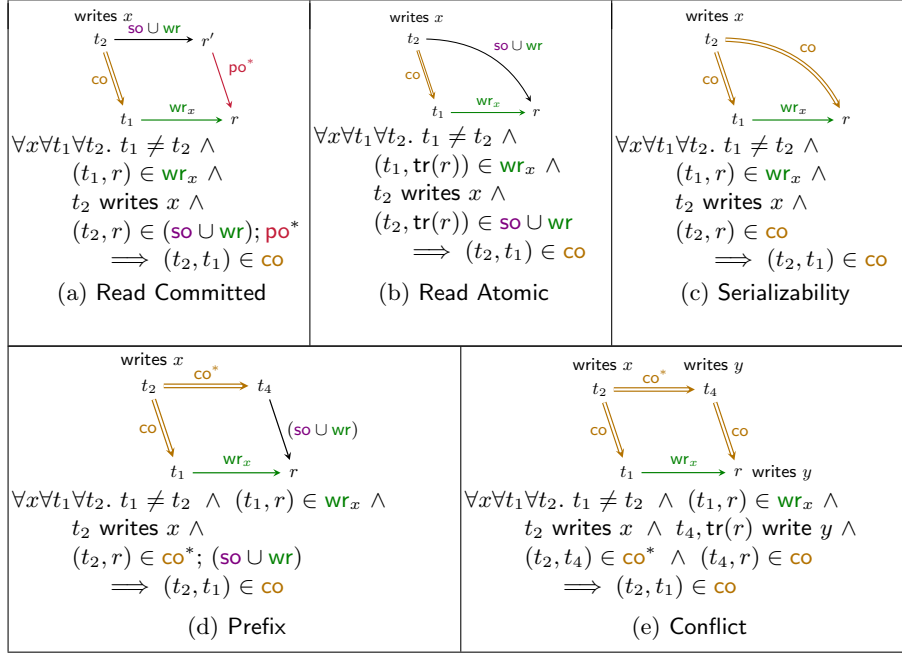


Fig. 3: Axioms defining RC, RA, SER, PC and SI isolations levels respectively. Visibility relations are “inlined” to match the definitions in [6].

visible to r if t_2 commits before r , while RC states that t_2 is visible to r if either $(t_2, r) \in \text{so}$ or if there exists a previous event r' in $\text{tr}(r)$ that reads x from t_2 . Similarly, *Read Atomic* (RA) and *Prefix Consistency* (PC) are defined using their homonymous axioms while *Snapshot Isolation* (SI) is defined as a conjunction of both Prefix and Conflict.

The *isolation configuration* of a history is a mapping $\text{iso}(h) : T \rightarrow \text{Iso}$ associating to each transaction an isolation level identifier from a set Iso .

Whenever every transaction in a history has the same isolation level ι , the isolation configuration of that history is denoted simply by ι .

Note that SER is stronger than RC: every transaction visible to a read r according to RC is also visible to r according to SER. This means SER imposes more constraints for transaction t_1 to be read by r than RC. In general, for two isolation configurations I_1 and I_2 , I_1 is *stronger than* I_2 when for every transaction t , $I_1(t)$ is stronger than $I_2(t)$ (i.e., whenever $I_1(t)$ holds in an execution ξ , $I_2(t)$ also holds in ξ). The *weaker than* relationship is defined similarly.

Given a history h with isolation configuration $\text{iso}(h)$, h is called *consistent* when there exists an execution ξ of h such that for all transactions t in ξ , the axioms in $\text{iso}(h)(t)$ are satisfied in ξ (the interpretation of an axiom over an execution is defined as expected). For example, let h be the full history in Figure 2c. If both t_1, t_2 's isolation are SER, then h is *not* consistent, i.e., every execution $\xi = (h, \text{co})$ violates the corresponding axioms. Assume for instance, that $(t_1, t_2) \in \text{co}$. Then, by axiom SER, as $(\text{init}, t_2) \in \text{wr}_{x_1}$ and t_1 writes x_1 , we

get that $(t_1, \text{init}) \in \text{co}$, which is impossible as $(\text{init}, t_1) \in \text{so} \subseteq \text{co}$. However, if the isolation configuration is weaker (for example $\text{iso}(h)(t_2) = \text{RC}$), then the history is consistent using $\text{init} <_{\text{co}} t_1 <_{\text{co}} t_2$ as commit order.

Definition 6. A full history $h = (T, \text{so}, \text{wr})$ with isolation configuration $\text{iso}(h)$ is consistent iff there is an execution ξ of h s.t. $\bigwedge_{t \in T, r \in \text{reads}(t), a \in \text{iso}(h)(t)} a(r)$ holds in ξ ; ξ is called a consistent execution of h .

The notion of consistency on full histories is extended to client histories.

Definition 7. A client history $h = (T, \text{so}, \text{wr})$ with isolation configuration $\text{iso}(h)$ is consistent iff there is a full history \bar{h} with the same isolation configuration which is a witness of h and consistent; \bar{h} is called a consistent witness of h .

In general, the witness of a client history may not be consistent. In particular, there may exist several witnesses but no consistent witness.

3.3 Validation of the semantics

To justify the axiomatic semantics defined above, we define an operational semantics inspired by real implementations and prove that every run of a program can be translated into a consistent history. Every instruction is associated with an increasing timestamp and it reads from a snapshot of the database defined according to the isolation level of the enclosing transaction. At the end of the transaction we evaluate if the transaction can be committed or not. We assume that a transaction can abort only if explicitly stated in the program. We model an optimistic approach where if a transaction cannot commit, the run blocks (modelling unexpected aborts). We focus on three of the most used isolation levels: SER, SI, RC. Other isolation levels can be handled in a similar manner. For each run ρ we extract a full history $\text{history}(\rho)$. We show by induction that $\text{history}(\rho)$ is consistent at every step.

Theorem 1. For every run ρ , $\text{history}(\rho)$ is consistent.

4 Complexity of Checking Consistency

4.1 Saturation and Boundedness

We investigate the complexity of checking if a history is consistent. Our axiomatic framework characterize isolation levels as a conjunction of axioms as in Equation (1). However, some isolation levels impose stronger constraints than others. For studying the complexity of checking consistency, we classify them in two categories, saturable or not. An isolation level is *saturable* if its visibility relations are defined without using the co relation (i.e. the grammar in Equation (3) omits the co relation). Otherwise, we say that the isolation level is *non-saturable*. For example, RC and RA are saturable while PC, SI and SER are not.

Algorithm 1 Extending an initial pco relation with necessary ordering constraints

```

1: function SATURATE( $h = (T, \text{so}, \text{wr}), \text{pco}$ )  $\triangleright$   $\text{pco}$  must be transitive.
2:    $\text{pco}_{\text{res}} \leftarrow \text{pco}$ 
3:   for all  $x \in \text{Keys}$  do
4:     for all  $r \in \text{reads}(h), t_2 \neq \text{tr}(r) \in T$  s.t.  $t_2$  writes  $x$  and  $t_2 \neq \text{tr}(\text{wr}_x^{-1}(r))$  do
5:        $t_1 \leftarrow \text{tr}(\text{wr}_x^{-1}(r))$   $\triangleright t_1$  is well defined as  $h$  is a full history.
6:       for all  $v \in \text{vis}(\text{iso}(h)(\text{tr}(r)))$  do
7:         if  $v(t_2, r, x)$  then
8:            $\text{pco}_{\text{res}} \leftarrow \text{pco}_{\text{res}} \cup \{(t_2, t_1)\}$ 
9:   return  $\text{pco}_{\text{res}}$ 

```

Algorithm 2 Checking saturable consistency

```

1: function CHECKSATURABLE( $h = (T, \text{so}, \text{wr})$ )
2:   if  $\text{so} \cup \text{wr}$  is cyclic then return false
3:    $\text{pco} \leftarrow \text{SATURATE}(h, (\text{so} \cup \text{wr})^+)$ 
4:   return true if  $\text{pco}$  is acyclic, and false, otherwise

```

Definition 8. An isolation configuration $\text{iso}(h)$ is saturable if for every transaction t , $\text{iso}(h)(t)$ is a saturable isolation level. Otherwise, $\text{iso}(h)$ is non-saturable.

We say an isolation configuration $\text{iso}(h)$ is *bounded* if there exists a fixed $k \in \mathbb{N}$ s.t. for every transaction t , $\text{iso}(h)(t)$ is defined as a conjunction of at most k axioms that contain at most k quantifiers. For example, SER employs one axiom and four quantifiers while SI employs two axioms, Prefix and Conflict, with four and five quantifiers respectively. Any isolation configuration composed with SER, SI, PC, RA and RC isolation levels is bounded. We assume in the following that isolation configurations are bounded.

Checking consistency requires computing the value_{wr} function and thus, evaluating WHERE predicates. In the following, we assume that evaluating WHERE predicates on a single row requires constant time.

4.2 Checking Consistency of Full Histories

Algorithm 2 computes necessary and sufficient conditions for the existence of a consistent execution $\xi = (h, \text{co})$ for a history h with a saturable isolation configuration. It calls SATURATE, defined in Algorithm 1, to compute a “partial” commit order relation pco that includes $(\text{so} \cup \text{wr})^+$ and any other dependency between transactions that can be deduced from the isolation configuration. A consistent execution exists iff this partial commit order is acyclic. Algorithm 2 generalizes the results in [6] for full histories with heterogeneous saturable isolation configurations.

Theorem 2. Checking consistency of full histories with bounded saturable isolation configurations can be done in polynomial time.

For bounded non-saturable isolation configurations, checking if a history is consistent is NP-complete as an immediate consequence of the results in [6]. These previous results apply to the particular case of transactions having the same isolation level and being formed of classic read and write instructions on a fixed set of variables. The latter can be simulated by SQL queries using `WHERE` predicates for selecting rows based on their key being equal to some particular value. For instance, `SELECT($\lambda r : \text{key}(r) = x$)` simulates a read of a “variable” x .

4.3 Checking Consistency of Client Histories

We show that going from full histories to client histories, the consistency checking problem becomes NP-complete, independently of the isolation configurations. Intuitively, NP-hardness comes from keys that are not included in outputs of SQL queries. The justification for the consistency of omitting such rows can be ambiguous, e.g., multiple values written to a row may not satisfy the predicate of the `WHERE` clause, or multiple deletes can justify the absence of a row.

The *width* of a history $\text{width}(h)$ is the maximum number of transactions which are pairwise incomparable w.r.t. `so`. In a different context, previous work [6] showed that bounding the width of a history (consider it to be a constant) is a sufficient condition for obtaining polynomial-time consistency checking algorithms. This is not true for client histories.

Theorem 3. *Checking consistency of bounded-width client histories with bounded isolation configuration stronger than RC and $\text{width}(h) \geq 3$ is NP-complete.*

The proof of NP-hardness uses a reduction from 1-in-3 SAT which is inspired by the work of Gibbons and Korach [16] (Theorem 2.7) concerning sequential consistency for shared memory implementations. Our reduction is a non-trivial extension because it has to deal with any weak isolation configuration stronger than RC.

The proof of Theorem 3 relies on using non-trivial predicates in `WHERE` clauses. We also prove that checking consistency of client histories is NP-complete irrespectively of the complexity of these predicates. This result uses another class of histories, called *partial-observation* histories. These histories are a particular class of client histories where events read all inserted keys, irrespectively of their `WHERE` clauses (as if these clauses were *true*).

Definition 9. A partial observation history $h = (T, \text{so}, \text{wr})$ is a client history for which there is a witness $\bar{h} = (T, \text{so}, \bar{\text{wr}})$ of h , s.t. for every x , if $(w, r) \in \bar{\text{wr}}_x \setminus \text{wr}_x$, then w deletes x .

Theorem 4. *Checking consistency of partial observation histories with bounded isolation configurations stronger than RC is NP-complete.*

The proof of NP-hardness uses a novel reduction from 3 SAT. The main difficulty for obtaining consistent witnesses of partial observation histories is the ambiguity of which delete event is responsible for each absent row.

Algorithm 3 Checking consistency of client histories

```

1: function CHECKCONSISTENCY( $h = (T, \text{so}, \text{wr})$ )
2:   let  $\text{pco} = \text{FIX}(\lambda R : \text{SATURATE}(h, R))(\text{so} \cup \text{wr})^+$ 
3:   let  $E_h = \{(r, x) \mid r \in \text{reads}(h), x \in \text{Keys.wr}_x^{-1}(r) \uparrow \text{ and } 1_x^r(\text{pco}) \neq \emptyset\}$ 
4:   let  $X_h$  = the set of mappings that map each  $(r, x) \in E_h$  to a member of  $0_x^r(\text{pco})$ 
5:   if  $\text{pco}$  is cyclic then return false
6:   else if there exists  $(r, x) \in E_h$  such that  $0_x^r(\text{pco}) = \emptyset$  then return false
7:   else if  $E_h = \emptyset$  then return EXPLORECONSISTENTPREFIXES( $h, \emptyset$ )
8:   else
9:     for all  $f \in X_h$  do
10:       $\text{seen} \leftarrow \emptyset$ ;  $h' \leftarrow h \bigoplus_{(r,x) \in E_h} \text{wr}_x(f(r, x), r)$ 
11:      if EXPLORECONSISTENTPREFIXES( $h', \emptyset$ ) then return true
12:   return false

```

5 Effectively Checking Consistency of Client Histories

The result of Theorem 3 implicitly asks whether there exist conditions on the histories for which checking consistency remains polynomial as in [6]. We describe an algorithm for checking consistency of client histories and identify cases in which it runs in polynomial time.

Consider a client history $h = (T, \text{so}, \text{wr})$ which is consistent. For every consistent witness $\bar{h} = (T, \text{so}, \text{wr})$ of h there exists a consistent execution of \bar{h} , $\xi = (\bar{h}, \text{co})$. The commit order co contains $(\text{so} \cup \text{wr})^+$ and any other ordering constraint derived from axioms by observing that $(\text{so} \cup \text{wr})^+ \subseteq \text{co}$. More generally, co includes all constraints generated by the least fixpoint of the function SATURATE defined in Algorithm 1 when starting from $(\text{so} \cup \text{wr})^+$ as partial commit order. This least fixpoint exists because SATURATE is monotonic. It is computed as usual by iterating SATURATE until the output does not change. We use $\text{FIX}(\lambda R : \text{SATURATE}(h, R))(\text{so} \cup \text{wr})^+$ to denote this least fixpoint. In general, such a fixpoint computation is just an under-approximation of co , and it is not enough for determining h 's consistency.

The algorithm we propose, described in Algorithm 3, exploits the partial commit order pco obtained by such a fixpoint computation (line 2) for determining h 's consistency. For a read r , key x , we define $1_x^r(\text{pco})$, resp., $0_x^r(\text{pco})$, to be the set of transactions that are *not* committed after $\text{tr}(r)$ and which write a value that satisfies, resp., does not satisfy, the predicate $\text{WHERE}(r)$. The formal description of both sets can be seen in Equation 4.

$$\begin{aligned}
1_x^r(\text{pco}) &= \{t \in T \mid (\text{tr}(r), t) \notin \text{pco} \wedge \text{WHERE}(r)(\text{value}_{\text{wr}}(t, x)) = 1\} \\
0_x^r(\text{pco}) &= \{t \in T \mid (\text{tr}(r), t) \notin \text{pco} \wedge \text{WHERE}(r)(\text{value}_{\text{wr}}(t, x)) = 0\} \quad (4)
\end{aligned}$$

The set $0_x^r(\text{pco})$ can be used to identify extensions that are not witness of a history. Let us consider the client history h depicted in Figure 4a. Observe that t_3 is not reading x_1 and t_5 is not reading x_2 . Table 4b describes all possible full extensions \bar{h} of h . An execution $\xi = (\bar{h}, \text{co})$ is consistent if $(t, r) \in \text{wr}_x \setminus \text{wr}_x$ implies $\text{WHERE}(r)(\text{value}_{\text{wr}}(t, x)) = 0$. This implies that extensions h_1 , h_4 , and h_7 , where

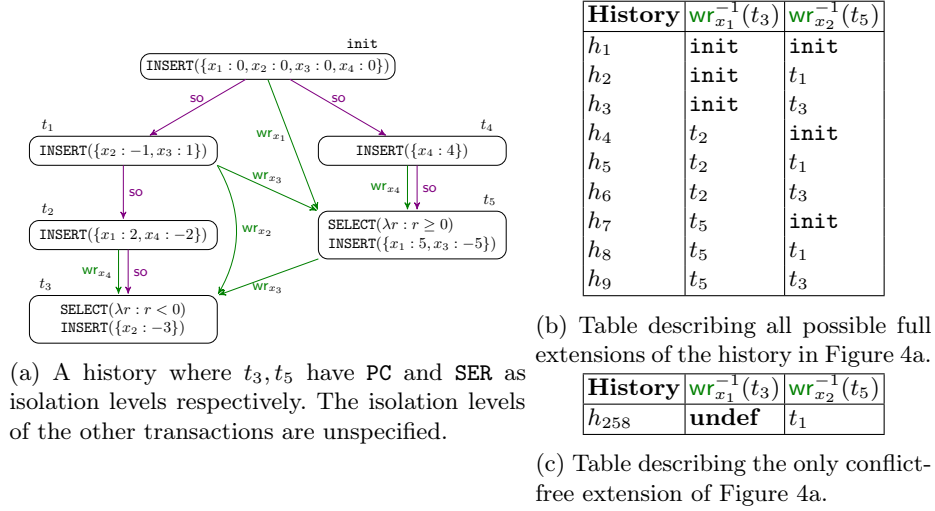


Fig. 4: Comparison between conflict-free extensions and full extensions of the history h in Figure 4a. In h , wr^{-1} is not defined for two pairs: (t_3, x_1) and (t_5, x_2) ; where we identify the single SELECT event in a transaction with its transaction. Table 4b describes all possible full extensions of h . For example, the first extension, h_1 , states that $(\text{init}, t_3) \in \text{wr}_{x_1}$ and $(\text{init}, t_5) \in \text{wr}_{x_2}$. Algorithm 3 only explore the only extension h_{258} described in Table 4c; where $\text{wr}_{x_1}^{-1}(t_3) \uparrow$ and $(t_1, t_5) \in \text{wr}_{x_2}$. The history h_{258} can be extended to histories h_2, h_5 and h_8 .

$(\text{init}, t_5) \in \overline{\text{wr}}_{x_2}$, are not witnesses of h as $\text{WHERE}(t_5)(\text{value}_{\text{wr}}(\text{init}, x_2)) = 1$. We note that $\text{init} \notin \mathcal{O}_{x_2}^{t_5}(\text{pco}) = \{t_1\}$. Also, observe that $(t_5, t_3) \in \text{wr}$; so extensions h_3, h_6 and h_9 , where $(t_3, t_5) \in \overline{\text{wr}}_{x_2}$, are not a witness of h . Once again, $t_3 \notin \mathcal{O}_{x_2}^{t_5}(\text{pco})$. In general, for every read event r and key x s.t. $\text{wr}_x^{-1}(r) \uparrow$, the extension of h where $(t, r) \in \overline{\text{wr}}_x$, $t \notin \mathcal{O}_x^r(\text{pco})$, is not a witness of h . In particular, if $\text{wr}_x^{-1}(r) \uparrow$ but $\mathcal{O}_x^r(\text{pco}) = \emptyset$, then no witness of h can exist.

The sets $\mathcal{O}_x^r(\text{pco})$ are not sufficient to determine if a witness is a consistent witness as our previous example shows: $\mathcal{O}_{x_1}^{t_3}(\text{pco}) = \{\text{init}, t_2, t_5\}$, but h_2 is not consistent. Algorithm 3, combines an enumeration of history extensions with a search for a consistent execution of each extension. The extensions are *not* necessarily full. In case $\text{wr}_x^{-1}(r)$ is undefined, we use sets $1_x^r(\text{pco})$ to decide whether the extension of h requires specifying $\text{wr}_x^{-1}(r)$ for determining h 's consistency. Algorithm 3 specifies $\text{wr}_x^{-1}(r)$ only if (r, x) is a so-called *conflict*, i.e., $\text{wr}_x^{-1}(r)$ is undefined and $1_x^r(\text{pco}) \neq \emptyset$.

Following the example of Figure 4, we observe that $1_{x_1}^{t_3}(\text{pco}) = \emptyset$, all transactions that write on x_1 write non-negative values; but instead $1_{x_2}^{t_5}(\text{pco}) = \{\text{init}\}$. Intuitively, this means that if some extension h' that does not specify $\text{wr}_{x_1}^{-1}(t_3)$ does not violate any axiom when using some commit order co , then we can extend h' , defining $\text{wr}_{x_1}^{-1}(t_3)$ as some adequate transaction, and obtain a full history \bar{h} s.t. the execution $\xi = (\bar{h}, \text{co})$ is consistent. On the other hand, specifying the write-read dependency of t_5 on x_2 matters. For not contradicting any axiom

using **co**, we may require $(\text{init}, t_5) \in \overline{\text{wr}}_{x_2}$. However, such extension is not even a witness of h as $\text{WHERE}(\text{init})(\text{value}_{\text{wr}}(\text{init}, x_2)) = 1$. This intuition holds for the particular definitions of the isolation levels that Algorithm 3 considers.

A history is *conflict-free* if it does not have conflicts. Our previous discussion reduces the problem of checking consistency of a history to checking consistency of its conflict-free extensions. For example, the history h in Figure 4a is not conflict-free but the extension h_{258} defined in Table 4c is. Instead of checking consistency of the nine possible extensions, we only check consistency of h_{258} .

Algorithm 3 starts by checking if there is at least a conflict-free extension of h (line 6). If h is conflict-free, it directly calls Algorithm 4 (line 7); while otherwise, it iterates over conflict-free extensions of h , calling Algorithm 4 on each of them (line 11).

Algorithm 4 describes the search for the commit order of a conflict-free history h . This is a recursive enumeration of consistent prefixes of histories that backtracks when detecting inconsistency (it generalizes Algorithm 2 in [6]). A *prefix* of a history $h = (T, \text{so}, \text{wr})$ is a tuple $P = (T_P, M_P)$ where $T_P \subseteq T$ is a set of transactions and $M_P : \text{Keys} \rightarrow T_P$ is a mapping s.t. (1) **so** predecessors of transactions in T_P are also in T_P , i.e., $\forall t \in T_P, \text{so}^{-1}(t) \in T_P$ and (2) for every x , $M_P(x)$ is a **so**-maximal transaction in T_P that writes x (M_P records a last write for every key).

For every prefix $P = (T_P, M_P)$ of a history h and a transaction $t \in T \setminus T_P$, we say a prefix $P' = (T_{P'}, M_{P'})$ of h is an *extension* of P using t if $T_{P'} = T_P \cup \{t\}$ and for every key x , $M_{P'}(x)$ is t or $M_P(x)$. Algorithm 4 extensions, denoted as $P \cup \{t\}$, guarantee that for every key x , if t writes x , then $M_{P'}(x) = t$.

Extending the prefix P using t means that any transaction $t' \in T_P$ is committed before t . Algorithm 4 focuses on special extensions that lead to commit orders of consistent executions.

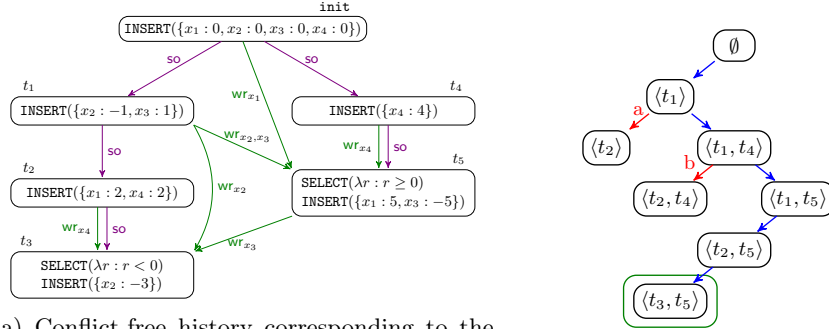
| Axiom | Predicate |
|--|---|
| Serializability, Prefix, Read Atomic, Read Committed | $\nexists x \in \text{Keys}$ s.t. t writes x , $\text{wr}_x^{-1}(r) \downarrow$ $v(\text{pco}_t^P)(t, r, x)$ holds in h and $\text{wr}_x^{-1}(r) \in T_P$ |
| Conflict | $\nexists x \in \text{Keys}, t' \in T_P \cup \{t\}$ s.t. t' writes x , $\text{wr}_x^{-1}(r) \downarrow$ $v(\text{pco}_t^P)(t', r, x)$ holds in h and $\text{wr}_x^{-1}(r) \neq M_P(x)$ |

Table 1: Predicates relating prefixes and visibility relations where pco_t^P is defined as $\text{pco} \cup \{(t', t) \mid t' \in T_P\} \cup \{(t, t'') \mid t'' \in T \setminus (T_P \cup \{t\})\}$.

Definition 10. Let h be a history, $P = (T_P, M_P)$ be a prefix of h , t a transaction that is not in T_P and $P' = (T_{P'}, M_{P'})$ be an extension of P using t . The prefix P' is a consistent extension of P with t , denoted by $P \triangleright_t P'$, if

1. P is **pco**-closed: for every transaction $t' \in T$ s.t. $(t', t) \in \text{pco}$ then $t' \in T_P$,
2. t does not overwrite other transactions in P : for every **read** event r outside of the prefix, i.e., $\text{tr}(r) \in T \setminus T_{P'}$ and every visibility relation $v \in \text{vis}(\text{iso}(h))(\text{tr}(r))$, the predicate $\text{vp}_v^P(t, r)$ defined in Table 1 holds in h .

We say that a prefix is consistent if it is either the empty prefix or it is a consistent extension of a consistent prefix.



(a) Conflict-free history corresponding to the extension h_{258} (Table 4c) of the history in Figure 4a (b) Execution of Algorithm 3 on the history in Figure 5a.

Fig. 5: Applying Algorithm 4 on the conflict-free consistent history h_{258} on the left. The right part pictures a search for valid extensions of consistent prefixes on h_{258} . Prefixes are represented by their so -maximal transactions, e.g., $\langle t_2 \rangle$ contains all transactions which are before t_2 in so , i.e., $\{init, t_1, t_2\}$. A red arrow means that the search is blocked (the prefix at the target is not a consistent extension), while a blue arrow mean that the search continues.

Figure 5b depicts the execution of Algorithm 4 on the conflict-free history Figure 5a (history h_{258} from Table 4c). Blocked and effectuated calls are represented by read and blue arrows respectively. The read arrow a is due to condition 1 in Definition 10: as t_3 enforces PC, reads x_4 from t_2 , and t_4 is visible to it ($vis_{Prefix}(t_4, t_3, x_4)$), $(t_4, t_2) \in pco$; so consistent prefixes can not contain t_2 if they do not contain t_4 . The read arrow b is due to condition 2: as t_5 enforces SER and it reads x_4 from t_4 , consistent prefixes can not contain t_2 unless t_5 is included. When reaching prefix $\langle t_3, t_5 \rangle$, the search terminates and deduces that h is consistent. From the commit order induced by the search tree we can construct the extension of h where missing write-read dependencies are obtained by applying the axioms on such a commit order. In our case, from $init <_{co} t_1 <_{co} t_4 <_{co} t_5 <_{co} t_2 <_{co} t_3$, we deduce that the execution $\xi = (h_5, co)$ is a consistent execution of h_{258} , and hence of h ; where h_5 is the history described in Table 4b.

For complexity optimizations, Algorithm 4 requires an isolation level-dependent equivalence relation between consistent prefixes. If there is transaction $t \in T$ s.t. $iso(h)(t) = SI$, prefixes $P = (T_P, M_P)$ and $P' = (T_{P'}, M_{P'})$ are *equivalent* iff they are equal (i.e. $T_P = T_{P'}, M_P = M_{P'}$). Otherwise, they are *equivalent* iff $T_P = T_{P'}$.

Theorem 5. *Let h be a client history whose isolation configuration is defined using $\{SER, SI, PC, RA, RC\}$. Algorithm 3 returns true if and only if h is consistent.*

In general, Algorithm 3 is exponential the number of conflicts in h . The number of *conflicts* is denoted by $\#conf(h)$. The number of conflicts exponent is implied by the number of mappings in X_h explored by Algorithm 3 (E_h is the set of conflicts in h). The history width and size exponents comes from the

Algorithm 4 check consistency of conflict-free histories

```

1: function EXPLORECONSISTENTPREFIXES( $h = (T, \text{so}, \text{wr}), P$ )
2:   if  $|P| = |T|$  then return true
3:   for all  $t \in T \setminus P$  s.t.  $P \triangleright_t (P \cup \{t\})$  do
4:     if  $\exists P' \in \text{seen}$  s.t.  $P' \equiv_{\text{iso}(h)} (P \cup \{t\})$  then continue
5:     else if EXPLORECONSISTENTPREFIXES( $h, P \cup \{t\}$ ) then return true
6:     else seen  $\leftarrow \text{seen} \cup (P \cup \{t\})$ 
7:   return false

```

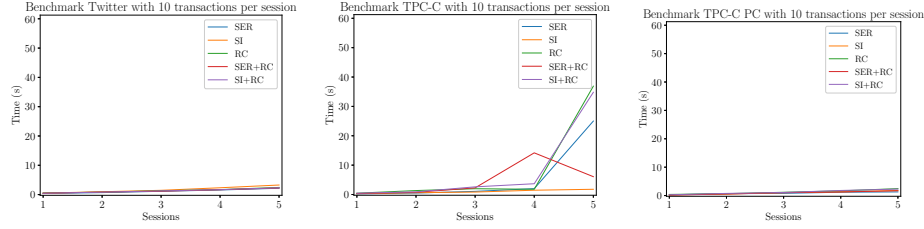


Fig.6: Running time of Algorithm 3 while increasing the number of sessions. Each point represents the average running time of 5 random clients of such size.

number of prefixes explored by Algorithm 4 which is $|h|^{\text{width}(h)} \cdot \text{width}(h)^{|\text{Keys}|}$ in the worst case (prefixes can be equivalently described by a set of **so**-maximal transactions and a mapping associating keys to sessions).

Theorem 6. *For every client history h whose isolation configuration is composed of $\{\text{SER}, \text{SI}, \text{PC}, \text{RA}, \text{RC}\}$ isolation levels, Algorithm 3 runs in $\mathcal{O}(|h|^{\# \text{conf}(h) + \text{width}(h) + 9} \cdot \text{width}(h)^{|\text{Keys}|})$. Moreover, if no transaction employs SI isolation level, Algorithm 3 runs in $\mathcal{O}(|h|^{\# \text{conf}(h) + \text{width}(h) + 8})$.*

On bounded, conflict-free histories only using **SER**, **PC**, **RA**, **RC** as isolation levels, Algorithm 3 runs in polynomial time. For instance, standard reads and writes can be simulated using **INSERT** and **SELECT** with **WHERE** clauses that select rows based on their key being equal to some particular value. In this case, histories are conflict-less (**wr** would be defined for the particular key asked by the clause, and writes on other keys would not satisfy the clause). A more general setting where **WHERE** clauses restrict only values that are immutable during the execution (e.g., primary keys) and deletes only affect non-read rows also falls in this category.

6 Experimental evaluation

We evaluate an implementation of **CHECKCONSISTENCY** in the context of the Benchbase [12] database benchmarking framework. We apply this algorithm on histories extracted from randomly generated client programs of a number of database-backed applications. We use PostgreSQL 14.10 as a database. The experiments were performed on an Apple M1 with 8 cores and 16 GB of RAM.

Implementation. We extend the Benchbase framework with an additional package for generating histories and checking consistency. Applications from

Benchbase are instrumented in order to be able to extract histories, the `wr` relation in particular. Our implementation is publicly available [9].

Our tool takes as input a configuration file specifying the name of the application and the isolation level of each transaction in that application. For computing the `wr` relation and generating client histories, we extend the database tables with an extra column `WRITEID` which is updated by every `write` instruction with a unique value. SQL queries are also modified to return whole rows instead of selected columns. To extract the `wr` relation for `UPDATE` and `DELETE` we add `RETURNING` clauses. Complex operators such as `INNER JOIN` are substituted by simple juxtaposed SQL queries (similarly to [7]). We map the result of each query to local structures for generating the corresponding history. Transactions aborted by the database (and not explicitly by the application) are discarded.

Benchmark. We analyze a set of benchmarks inspired by real-world applications and evaluate them under different types of clients and isolation configurations. We focus on isolation configurations implemented in PostgreSQL, i.e. compositions of `SER`, `SI` and `RC` isolation levels.

In average, the ratio of `SER`/`SI` transactions is 11% for Twitter and 88% for TPC-C and TPC-C PC. These distributions are obtained via the random generation of client programs implemented in BenchBase. In general, we observe that the bottleneck is the number of possible history extensions enumerated at line 9 in Alg. 3 and not the isolation configuration. This number is influenced by the distribution of types of transactions, e.g., for TPC-C, a bigger number of transactions creating new orders increases the number of possible full history extensions. We will clarify.

Twitter [12] models a social network that allows users to publish tweets and get their followers, tweets and tweets published by other followers. We consider five isolation configurations: `SER`, `SI` and `RC` and the heterogeneous `SER + RC` and `SI + RC`, where publishing a tweet is `SER` (resp., `SI`) and the rest are `RC`. The ratio of `SER` (resp. `SI`) transactions w.r.t. `RC` is 11% on average.

TPC-C [24] models an online shopping application with five types of transactions: reading the stock, creating a new order, getting its status, paying it and delivering it. We consider five isolation configurations: the homogeneous `SER`, `SI` and `RC` and the combinations `SER + RC` and `SI + RC`, where creating a new order and paying it have `SER` (respectively `SI`) as isolation level while the rest have `RC`. The ratio of `SER` (resp. `SI`) transactions w.r.t. `RC` is 88% on average.

TPC-C PC is a variant of the TPC-C benchmark whose histories are always conflict-free. `DELETE` queries are replaced by `UPDATE` with the aid of extra columns simulating the absence of a row. Queries whose `WHERE` clauses query mutable values are replaced by multiple simple instructions querying only immutable values such as unique ids and primary keys.

Experimental Results. We designed two experiments to evaluate `CHECKCONSISTENCY`'s performance for different isolation configurations increasing the number of transactions per session (the number of sessions is fixed), the number of sessions (the number of transactions per session is fixed), resp. We use a timeout of 60 seconds per history.

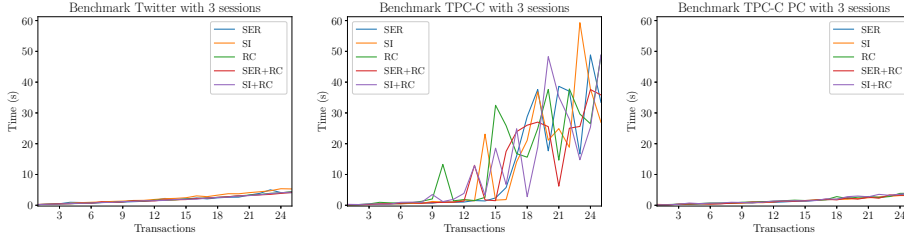


Fig. 7: Running time of Algorithm 3 increasing the number of transactions per session. We plot the average running time of 5 random clients of such size.

The first experiment investigates the scalability of Algorithm 3 when increasing the number of sessions. For each benchmark and isolation configuration, we consider 5 histories of random clients (each history is for a different client) with an increasing number of sessions and 10 transactions per session (around 400 histories across all benchmarks). No timeouts appear with less than 4 sessions. Figure 6 shows the running time of the experiment.

The second experiment investigates the scalability of Algorithm 3 when increasing the number of transactions. For each benchmark and isolation configuration, we consider 5 histories of random clients, each having 3 sessions and an increasing number of transactions per session (around 1900 histories across all benchmarks). Figure 7 shows its running time.

The runtime similarities between isolation configurations containing SI versus those without it show that in practice, the bottleneck of Algorithm 3 is the number of possible history extensions enumerated at line 11 in Algorithm 3; i.e. the number of conflicts in a history. This number is influenced by the distribution of types of transactions, e.g., for TPC-C, a bigger number of transactions creating new orders increases the number of possible full history extensions. Other isolation levels not implemented by PostgreSQL, e.g., prefix consistency PC, are expected to produce similar results.

Both experiments show that Algorithm 3 scales well for histories with a small number of writes (like Twitter) or conflicts (like TPC-C PC). In particular, Algorithm 3 is quite efficient for typical workloads needed to expose bugs in production databases which contain less than 10 transactions [6,20,18].

A third experiment compares Algorithm 3 with a baseline consisting in a naive approach where we enumerate witnesses and executions of such witnesses until consistency is determined. We consider Twitter and TPC-C as benchmarks and execute 5 histories of random clients, each having 3 sessions and an increasing number of transactions per session (around 100 histories across all benchmarks). We execute each client under RC and check the obtained histories for consistency with respect to SER.

The naive approach either times out for 35.5%, resp., 95.5% of the histories of Twitter, resp., TPC-C, or finishes in 5s on average (max 25s). In comparison, Algorithm 3 has no timeouts for Twitter and times out for 5.5% of the TPC-C histories; finishing in 1.5s on average (max 12s). Averages are computed w.r.t. non-timeout instances. The total number of executed clients is around 100. Only

one TPC-C history was detected as inconsistent, which shows that the naive approach does not timeout only in the worst-case (inconsistency is a worst-case because all extensions and commit orders must be proved to be invalid).

A similar analysis on the TPC-C PC benchmark is omitted: TPC-C PC is a conflict-free variation of TPC-C with more operations per transaction. Thus, the rate of timeouts in the naive approach increases w.r.t. TPC-C, while the rate of timeouts using Algorithm 3 decreases.

Comparisons with prior work [6,4,18,20] are not possible as they do not apply to SQL (see Section 7 for more details).

This evaluation demonstrates that our algorithm scales well to practical testing workloads and that it outperforms brute-force search.

7 Related work

The formalization of database isolation levels has been considered in previous work. Adya [2] has proposed axiomatic specifications for isolation levels, which however do not concern more modern isolation levels like PC or SI and which are based on low-level modeling of database snapshots. We follow the more modern approach in [11,6] which however addresses the restricted case when transactions are formed of reads and writes on a *static* set of keys (variables) and not generic SQL queries, and all the transactions in a given execution have the same isolation level. Our axiomatic model builds on axioms defined by Biswas et al. [6] which are however applied on a new model of executions that is specific to SQL queries.

The complexity of checking consistency w.r.t isolation levels has been studied in [21,6]. The work of Papadimitriou [21] shows that checking serializability is NP-complete while the work of Biswas et al. [6] provides results for the same isolation levels as in our work, but in the restricted case mentioned above.

Checking consistency in a non-transactional case, shared-memory or distributed systems, has been investigated in a number of works, e.g., [8,16,13,10,17,14,1,15,3]. Transactions introduce additional challenges that make these results not applicable.

Existing tools for checking consistency in the transactional case of distributed databases, e.g., [6,4,18,20] cannot handle SQL-like semantics, offering guarantees modulo their transformations to reads and writes on static sets of keys. Our results show that handling the SQL-like semantics is strictly more complex (NP-hard in most cases).

Acknowledgements

We thank the anonymous reviewers for their feedback. This work was partially supported by the Agence National de Recherche (ANR) grants “AdeCoDS” and “CENTEANES”.

References

1. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. Optimal stateless model checking under the release-acquire semantics. *Proc. ACM Program. Lang.*, 2(OOPSLA):135:1–135:29, 2018. doi:10.1145/3276505.
2. A. Adya. Weak consistency: A generalized theory and optimistic implementations for distributed transactions. Technical report, USA, 1999.
3. Pratyush Agarwal, Krishnendu Chatterjee, Shreya Pathak, Andreas Pavlogiannis, and Viktor Toman. Stateless model checking under a reads-value-from equivalence. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 341–366. Springer, 2021. doi:10.1007/978-3-030-81685-8_16.
4. Peter Alvaro and Kyle Kingsbury. Elle: Inferring isolation anomalies from experimental observations. *Proc. VLDB Endow.*, 14(3):268–280, 2020. URL: <http://www.vldb.org/pvldb/vol14/p268-alvaro.pdf>, doi:10.5555/3430915.3442427.
5. Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O’Neil, and Patrick E. O’Neil. A critique of ANSI SQL isolation levels. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, pages 1–10. ACM Press, 1995. doi:10.1145/223784.223785.
6. Ranadeep Biswas and Constantin Enea. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.*, 3(OOPSLA):165:1–165:28, 2019. doi:10.1145/3360591.
7. Ranadeep Biswas, Diptanshu Kakwani, Jyothi Vedurada, Constantin Enea, and Akash Lal. Monkeydb: effectively testing correctness under weak isolation levels. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–27, 2021. doi:10.1145/3485546.
8. Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. On verifying causal consistency. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 626–638. ACM, 2017. doi:10.1145/3009837.3009888.
9. Ahmed Bouajjani, Constantin Enea, and Enrique Román-Calvo. Artifact for “on the complexity of checking mixed isolation levels for sql transactions”, October 2024. URL: <https://github.com/Galieve/benchbase-histories>.
10. Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. The complexity of verifying memory coherence and consistency. *IEEE Trans. Parallel Distributed Syst.*, 16(7):663–671, 2005. doi:10.1109/TPDS.2005.86.
11. Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. A framework for transactional consistency models with atomic visibility. In Luca Aceto and David de Frutos-Escrig, editors, *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1-4, 2015*, volume 42 of *LIPICs*, pages 58–71. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. URL: <https://doi.org/10.4230/LIPICs.CONCUR.2015.58>, doi:10.4230/LIPICs.CONCUR.2015.58.
12. Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.*, 7(4):277–288, 2013. URL: <http://www.vldb.org/pvldb/vol7/p277-difallah.pdf>, doi:10.14778/2732240.2732246.

13. Michael Emmi and Constantin Enea. Sound, complete, and tractable linearizability monitoring for concurrent collections. *Proc. ACM Program. Lang.*, 2(POPL):25:1–25:27, 2018. doi:10.1145/3158113.
14. Florian Furbach, Roland Meyer, Klaus Schneider, and Maximilian Senftleben. Memory-model-aware testing: A unified complexity analysis. *ACM Trans. Embed. Comput. Syst.*, 14(4):63:1–63:25, 2015. doi:10.1145/2753761.
15. Phillip B. Gibbons and Ephraim Korach. On testing cache-coherent shared memories. In Lawrence Snyder and Charles E. Leiserson, editors, *Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '94, Cape May, New Jersey, USA, June 27-29, 1994*, pages 177–188. ACM, 1994. doi:10.1145/181014.181328.
16. Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM J. Comput.*, 26(4):1208–1244, 1997. doi:10.1137/S0097539794279614.
17. Alex Gontmakher, Sergey V. Polyakov, and Assaf Schuster. Complexity of verifying java shared memory execution. *Parallel Process. Lett.*, 13(4):721–733, 2003. doi:10.1142/S0129626403001628.
18. Kaile Huang, Si Liu, Zheng Chen, Hengfeng Wei, David A. Basin, Haixiang Li, and Anqun Pan. Efficient black-box checking of snapshot isolation in databases. *Proc. VLDB Endow.*, 16(6):1264–1276, 2023. URL: <https://www.vldb.org/pvldb/vol16/p1264-wei.pdf>, doi:10.14778/3583140.3583145.
19. Jepsen. Distributed systems testing, 2020. <https://jepsen.io/>.
20. Si Liu, Long Gu, Hengfeng Wei, and David A. Basin. Plume: Efficient and complete black-box checking of weak isolation levels. *Proc. ACM Program. Lang.*, 8(OOPSLA2):876–904, 2024. doi:10.1145/3689742.
21. Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979. doi:10.1145/322154.322158.
22. Andrew Pavlo. What are we doing with our lives?: Nobody cares about our concurrency control research. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, page 3. ACM, 2017. doi:10.1145/3035918.3056096.
23. Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94), Austin, Texas, USA, September 28-30, 1994*, pages 140–149. IEEE Computer Society, 1994. doi:10.1109/PDIS.1994.331722.
24. TPC. Technical report, Transaction Processing Performance Council, February 2010. URL: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.