Compas'2022 : Parallélisme/ Architecture/ Système
MIS/UPJV - Amiens France, 5-8 juillet 2022

Towards a correct, high-performance database backend

Saalik Hatia(1), Annette Bieniusa(2), Carla Ferreira(3), Gustavo Petri(4), Marc Shapiro(1)

(1) LIP6–Sorbonne-Université & Inria, Paris, France
(2) TU Kaiserslautern, Germany
(3) Universidade NOVA de Lisboa, Portugal
(4) ARM, Cambridge, United Kingdom

Résumé
This paper describes the design and implementation of a full-featured, high-performance geo-distributed database backend that is correct by construction. We develop hand-in-hand an operational semantics, key invariants, and a reference implementation. Our baseline is a simplistic backend that supports concurrent transactions storing and retrieving versions in an unbounded memory. Later steps add a single feature, such as persistence, journaling, caching, checkpointing, or sharding, to both the model and the implementation. We first prove (in Coq) that the baseline model satisfies its invariants and show (by testing) that the reference implementation satisfies the model. For each additional feature, we prove a simulation relation between the base and extended model, and similarly for the reference implementation. This paper focuses on the reference implementation: its relation between the model, and how we instrument the code with asserts and on the litmus tests.

Mots-clés : système réparti, base de données géo-distribuée, vérification formelle, test

1. Introduction

Modern online applications use a distributed database management system (DDBMS), whose storage backend provides users with available and consistent data. The purpose of a backend is conceptually simple: to store and retrieve shared data. However, for performance and fault-tolerance reasons, the internals of a typical database backend are very complex, with many moving parts that interact in hard-to-understand ways. Existing backends are designed in a manual and ad-hoc manner; inevitably, they have bugs that impact the consistency and integrity of data.

Our research claim is that, for a complex system such as a database backend, following a formal model helps the developer avoid bugs, and is not in conflict with good performance. In this work, we aim to to build a database backend that is correct by design, while including some important optimisations (e.g., caching) and properties (e.g., fault tolerance).

Formalising a backend that provide users with fast read and writes, data safety and geo-replication is hard. Any attempt to verify at once some monolithic specification that includes all the interesting properties is probably doomed to failure. Instead, we propose the an incremental approach, decomposing the system into a set of small, orthogonal modules and features.

Our starting baseline is a bare-bones backend, restricted to its simplest function, i.e., transactions reading and writing data versions. We formalise its operational semantics and specify its

invariants. We prove (in Coq) that the semantics satisfies the invariants. We also provide a reference implementation (hand-written in Java), and use litmus test cases to check that it follows the semantics and does not violate the invariants.
Each design step adds a single feature, e.g., a cache or logging, with its associated operational semantics and invariants. We use simulation (or bisimulation) to show that the formal model is equivalent without and with the feature, modulo the added constraints. We take a similar approach with the reference implementation. Furthermore, we compare performance without and with the feature.
Our ultimate aim is to show that the final design, including all the features, is both correct (by simulation of the baseline) and has comparable performance to ad-hoc database backends with similar features.
This paper gives some background on the formal model and focuses on the system part, i.e., how the reference implementation follows the model, how we instrument the invariants, and how we conduct the testing.

2. Background

2.1. Transactions and timestamps

A client of the backend executes transactions. Classically, a transaction is a sequence of reads and updates, bounded by a begin and an end. The transaction either aborts with no effects; or it commits and all its commits become visible atomically. Formally, the choice between abort and commit is non-deterministic (in practice it depends on the application's invariants and on external events, such as sufficient resources being available).
We formalise the mutual ordering of transactions with abstract *timestamps*. Committing a transaction i assigns it a *commit timestamp* Ct_i; for atomicity, its updates are all labelled with this same timestamp. Conversely, a transaction j has a *snapshot timestamp* (or dependency timestamp) Dt_j. Transaction i precedes transaction j if $Ct_i \leq Dt_j$. Transaction j *depends on* all transactions i such i precedes j, meaning that a read that j performs includes all the updates with a label less or equal to Dt_j. The set of such preceding updates is called the *snapshot* of transaction j.
For space reasons, this paper considers a single model, Transactional Causal Plus Consistency (TCC+), i.e., causal consistency with transactions and convergence [1, 4]. Under TCC+, timestamps form a partial order (consistent with happened-before), and a snapshot must be a causally-consistent cut. However, our formalisation supports different transaction models (e.g., serialisable or snapshot-isolation), which differ only by timestamp ordering being partial or total, and by constraints on beginning and committing a transaction.
A transaction t has the following attributes:

- τ_t: unique transaction identifier.
- Dt_t: dependency timestamp.
- ε_t: effect map, records the updates made by the transaction. A write updates the effect map.
- R_t: read set, records the objects read by the transaction. A Read updates the read set.
- Ct_t: commit timestamp, assigned if and when the transaction commits.
- $State_t$: status, either *not_started*, *live* or *terminated*.

In what follows, we omit the subscript if it is obvious from the context.

	Precondition	Postcondition
$startTransaction()$	$State = null$ model-specific	$State = live$
$update(key, u)$	$State = live$	$\varepsilon' = \varepsilon \cup \{(key, u)\}$
$read(key)$ $\rightarrow r$	$State = live$	$R' = R \cup \{key\}$ $r = lookup(\varepsilon, key, Store, Dt)$
$commit()$	$State = live$ model-specific	$State = terminated$ $Store' = Store \cup \varepsilon$ $Dt < Ct$
$abort()$	$State = live$	$State = terminated$

Table 1: Unbounded-Memory Version store: pre- and post-conditions

2.2. Object-versions

For generality, we consider that each update (or, more precisely, the associated commit) creates a new version of the updated object.[1] As mentioned above, each such version is labelled with the commit timestamp of the corresponding transaction. A particular version of a particular object maps the pair (key, Ct), where *key* identifies the object, to the corresponding value, which for the purposes of this paper is an untyped "*blob*." In the above, *key* is the unique identifier or key of the object, and *Ct* is the commit timestamp of the transaction that writes this object-version.

3. Unbounded Version Store

We start with an intuitive simple in-memory key-value store. The implementation uses the Java `MultiMap<Key, Value>` for storing the object versions. Following Section 2.1, a Java Transaction object has attributes `transactionID`, `EffectMap`, `ReadSet`, etc. In addition, to speed up processing, we explicitly store the dependency graph of transactions.
The code is instrumented with Google Guava's *checkArgument* library[2] to perform assertions, both to check arguments, and to make sure that the invariants specified in the model are not violated, as we explain next, and as summarised in Table 1.
For instance, an assert checks that the commit timestamp of a committed transaction is greater than its dependency timestamp. To ensure that a client runs a sequence of well-formed transactions, asserts check that every transactional operation (read, write, commit or abort requests) takes place within a *live* transaction, and that only one transaction is live at a time.
Creating a transaction checks that its dependency timestamp `Dt` is valid, by calling a procedure that is specific to the consistency model. For instance, in the TCC+ model, it checks that `Dt` forms a causally-consistent snapshot.
To update a key, the system adds the update to the effect buffer associated with the transaction.[2]
Reading a key retrieves the key's value. There are two cases. If the transaction has previously updated the key, it returns the value in the effect buffer. Otherwise, it fetches the value corresponding to the transaction's snapshot from the store, looking up the most recent version of the key in the store whose timestamp is less or equal to *Dt*. If the snapshot contains concurrent object-versions (i.e., written by concurrent commits), they are merged [3]. If there is no pre-

[1] This is standard for databases that use Multi-Version Concurrency Control (MVCC), but our model does not mandate MVCC in the implementation.
[2] To simplify the notation, our postcondition assumes a key is updated only once per transaction.

ceding update in the store, it returns a default value. Finally, the system adds the key to the read-set R of the transaction.
When the client terminates the transaction, the system checks whether the effects of ε are valid according to the consistency model (under TCC+, this is always the case). The transaction commits only if this is true; this sets a commit timestamp, and moves the updates from the effect buffer into the store, tagged by the tagged by the transaction's commit timestamp. If not, the transaction aborts, by simply moving to the terminated state. A transaction may also abort non-deterministically.

4. Bounded-Memory Version Store

Our next step will be to impose a bound on the size of the store.
All invariants from the Unbounded memory are valid in this system. We create a global value called M_{limit} that contains the size limit for the *Store*. We also keep track of current memory utilization represented by M_{used}. To simplify our model we only consider the size of the *Store*, running transactions does not affect the invariant. Based on these two variables we introduce a new system invariant:

- $\mathsf{M}_{\mathsf{used}} \leq \mathsf{M}_{\mathsf{limit}}$

Every time a transaction commits M_{used} is updated to ensure that the size of the *Store* does not grow beyond the limit set by the system.
If the threshold is reached, the system has two possibilities: either to abort future transactions, or to delay their commitment, until M_{used} decreases.
For M_{used} to decreases the system needs to perform an eviction of Objet-Versions. For an eviction to be safe, the system must ensure that every running transaction's snapshot returns the same Object-Version for every *key* before and after any eviction. To uphold the new system invariants, we keep track of all the dependencies of running transactions called *RunningTr* but also all the commit timestamps of finished transactions *CommitTr*.
We then introduce a new timestamp called Minimum Dependency *MinDt*. *MinDt* represents the oldest snapshot any running transaction is reading from. When a transaction τ commits or aborts, it is removed from *RunningTr*. If $Dt = MinDt$ and no other transaction is reading from *Dt* the system advances *MinDt* to the next oldest snapshot used in *RunningTr*.
Eviction of Object-Version is done through a Garbage Collection. The system performs a lookup on every key that is part of the snapshot *MinDt*. Every Object-version in the *Store* that has a commit timestamp that is lower than *MinDt* and cannot be returned by a $lookup(key, Store, minDt)$ is evicted.
If at the end of the Garbage Collection the memory used is still higher than all the running transactions are aborted and we advance MinDt to allow additional garbage collection.

5. Testing

To simplify testing, every step of the implementation is built with the same client interface.
We start by executing a predetermined sequence of transactions and verify that the general behavior of the backend is correct. Once this done we run randomized tests on both systems to check if the general behavior remains correct outside predefined scenarios.
The next step is to check if our invariants are upheld by our implementation. Some invariants are directly inlined in the different functions. For others like checking if every transactional

operation takes place within a *live* transaction, we run each operation at least once outside a *live* transaction and expect our system to crash.
We want to show in our testing methodology that the Unbounded Version Store simulates the Bounded-Memory Version Store. The Bounded-Memory version has a constraint on the minimum snapshot allowed to be used by the system. So we execute the same sequence of transaction on both versions, where for every transaction the dependency snapshot is always higher than *minDt* and expect to see the same results. Then we execute a second sequence of transaction that has dependencies lower the minDt and show the differences between the two traces. One execution should succeed where in the Bounded-Memory Version Store some transaction should abort.
Finally, we plan on performing performance experiments to show that our design has comparable performance to ad-hoc database backends.

6. Conclusion

We have written several steps of the operational semantics, we have implemented the first partial versions of our design. Later we plan on adding the following features to the database: persistency, Unbounded Journal, Bounded journal with checkpoints and finally a cache.
For each feature we will complete the operational semantics by adding the necessary invariants for maintaining safety. Followed by a corresponding implementation matching our specification. Through testing we show that our implementation simulates our operational semantics. And that every step of the implementation simulate the following within the added constraints added by the new feature. Finally, we will provide a translation of the model in Coq.

References

[1] Akkoorath (D. D.), Tomsic (A. Z.), Bravo (M.), Li (Z.), Crain (T.), Bieniusa (A.), Preguiça (N.) et Shapiro (M.). – Cure: Strong semantics meets high availability and low latency. – In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, pp. 405–414, Nara, Japan, juin 2016.

[2] Google. – Guava.

[3] Shapiro (M.), Preguiça (N.), Baquero (C.) et Zawirski (M.). – Conflict-free replicated data types. – In Défago (X.), Petit (F.) et Villain (V.) (édité par), *Int. Symp. on Stabilization, Safety, and Security of Dist. Sys. (SSS), Lecture Notes in Comp. Sc.*, volume 6976, pp. 386–400, Grenoble, France, octobre 2011. Springer-Verlag.

[4] Toumlilt (I.), Sutra (P.) et Shapiro (M.). – Highly-available and consistent group collaboration at the edge with Colony. – In *Int. Conf. on Middleware (MIDDLEWARE)*, p. ??, Québec, Canada (online), décembre 2021. ACM/IFIP.