Reasoning about Two-Phase Locking Concurrency Control

Author:
David Harver Pollak

Supervisor:
Prof. Philippa Gardner

Second Marker:
Dr. Mark Wheelhouse

July 24, 2017

Abstract

Transactions are the main units of execution in database systems that employ concurrency to achieve better performance. They are managed by protocols that are able to guarantee different degrees of consistency based on the application's requirements. One of the strong consistency properties is serializability, which requires the outcome of any schedule of operations, coming from concurrent transactions, to be equivalent to a serial one in terms of the effects on the database. Surprisingly, modern program logics that enable the verification of fine-grained concurrency have not been applied to the context of database transactions yet.

We present a program logic for serializable transactions that are able to manipulate a shared storage. The logic is proven to be sound with respect to operational semantics that treat transactions as real atomic blocks. Nevertheless, we are able to model the atomicity of transactions even if it is only apparent, and concurrent interleavings do actually occur. We show this by providing the first application of our logic in terms of the *Two-phase locking* (2PL) protocol which ensures serializability. We first define a formal model and semantics that fully express the two-phase locking behaviours as part of a generic software system, then we show the equivalence between its operational semantics and the truly atomic ones. This result creates the necessary link to enable client reasoning in the 2PL setting through our program logic for transactions.

Acknowledgements

I would like to thank my supervisor, Philippa Gardner, for inspiring and guiding me through the course of this project. The regular meetings we had, always led to new and interesting questions around the topic and were filled with helpful feedback. A particular thanks goes to her research team members Shale Xiong, Julian Sutherland, Azalea Raad and Andrea Cerone, in no particular order. They were true companions in this journey and offered countless hours of their time to support my work. Their insights and suggestions have been of invaluable importance and I am extremely grateful for this.

It goes without saying that I owe everything to my parents Tim and Cristina and I am thankful for all they have given me. Together with my sister Desideria, they have supported and unconditionally loved me throughout my four years at Imperial.

Finally, this section would not be complete if I did not express my eternal gratitude to my hometown football club, A.C. Fiorentina, for giving me strong vital emotions every single week of my life.

"Non exiguum temporis habemus, sed multum perdimus. Satis longa vita et in maximarum rerum consummationem large data est, si tota bene conlocaretur; sed ubi per luxum ac neglegentiam diffluit, ubi nulli bonae rei inpenditur, ultim demum necessitate cogente quam ire non intelleximus transisse sentimus. Ita est: non accipimus brevem vitam sed facimus nec inopes eiussed prodigi sumus."

— *Seneca, De Brevitate Vitae*

Contents

1. Introduction

Modern database systems make heavy use of concurrency to increase performance and therefore to support large scale operations. Programmers access the underlying data through transactions, which are self-contained programs describing a single unit of work. To manage their concurrent executions, database systems employ various techniques depending on the degree of consistency[1] that needs to be guaranteed. This choice has an impact on the performance of the system too. A high-performance database system has to inevitably weaken its consistency model at the cost of allowing the occurrence of anomalous effects. For this reason, in practice, many commercial databases provide a relatively stronger consistency guarantee to release the developers' burden when writing applications that cannot afford such behvaiours.

One of such models is shaped around the idea of serializable executions. Serializability is a strong consistency property that requires the outcome of any schedule of operations, which are grouped by transactions, to be equivalent to a serial one, where transactions are run one after the other. A typical way to implement this consistency model is through pessimistic concurrency control. *Two-phase locking* is a popular concurrency control protocol in this group. It is a blocking approach that works at the granularity of single database entries by assigning a synchronization structure to each of them, so that it can guarantee serializability without losing too much performance.

Out of the many works on reasoning about transactions and the way they are managed, few of them are compositional and work at a program logic level. On the other hand, the area of formal reasoning about shared memory concurrency has seen a noticeable development towards logic frameworks that can verify fine-grained concurrency in a compositional way. In recent years, modern concurrent program logics, based on separation logic, have introduced two fundamental notions of abstraction. The first one, namely data abstraction, makes it possible to give abstract specifications by hiding implementation details, something that is proven to be very useful for compositional reasoning in a client-module setting. Furthermore, time abstraction, also called atomicity, is a property of operations by which they appear to happen at a single and discrete moment in time.

In the world of databases, and more specifically in the context of serializable models, transactions are often coupled with the idea of atomicity, which is very similar to the atomicity in shared memory: to the programmer's eyes, transactions are seen as units of work that get executed in one step. Thus, there is a clear connection between program logics for shared memory concurrency and database transactions. This is why we formulate a program logic to reason about serializable transactions, which we believe it is the first of its kind. The focus is then shifted to an application of our logic to the setting of two-phase locking, where we prove that its semantics conform with the required atomicity. This enables users of the framework to prove the correctness of programs running in a system that adopts a flavour of this concurrency control protocol, by only having to reason atomically about blocks of code, without the complexity of concurrent interleavings.

[1] Isolation level is usually the terminology adopted for consistency in database systems.

1.1 Contributions

The main technical contributions of this project are listed below, with references to the relevant sections where they are further discussed.

- **mCAP** (Section **??**, **??**) We reformulate and extend a program logic for concurrent programs, namely CAP [1], in order to remove some of its constraints, which are hardcoded into the logic, and enable a more flexible reasoning. In fact, we change the underlying model to parametrize both the representation of machine states and of action capabilities. On top of this, we provide a new and cleaner structure for the action model that does not explicitly use interference assertions. We also considerably modify the way environment interference is modelled through the rely/guarantee relations. This is done with the goal of allowing both a thread, and the environment, to perform multiple shared region updates in one step. It follows that the repartitioning operator also has a new and extended behaviour. At the level of the programming language, we leave elementary atomic commands as a parameter to the user of the logic. Finally, we instantiate the mCAP framework into a logic for our particular needs of transactional reasoning.

- **2PL Model** (Section **??**) The details of two-phase locking are analysed and ported to a formal model that, through its constructs and structures, is able to describe a transactional software system that uses the protocol and exhibits all of the required behaviours. The main novelties introduced are related to the way we globally manage information related to locking and track the state of running transactions.

- **Operational Semantics** (Section **??**, **??**) We use the constructs introduced in the 2PL model in order to shape a set of operational semantic rules that formally express the way the protocol acts at runtime. They are provided in a small-step fashion to enable the actual interleaving between concurrent transactions. Locking is implicit and does not occur as part of a language command. Instead, we take a nondeterministic approach to locking, and for this reason the semantics can model any particular pattern of lock acquisitions and releases, as long as it complies with the two-phase rule. The mentioned rules are able to reduce programs while labelling every step of the execution with the appropriate transaction or system operation. We group all such consecutive labels into a trace, which is the main structure used to construct a proof of serializability of the operational semantics as a whole.

- **Semantics Equivalence** (Section **??**) In order to allow mCAP style reasoning on programs running under 2PL, we are required to prove its soundness with respect to the operational semantics we introduced earlier. This effort is done in two steps, as we first prove the soundness in terms of a baseline operational semantics, which does not allow any interleaving between concurrent transactions by reducing them all at once: it effectively runs transactions atomically, in complete isolation. Then, we show that any reduction that reaches a terminal state in the 2PL semantics can be replicated by the atomic one. The latter proof requires a large number of intermediate results and structures which are formally defined, and whose specific properties are proven to be sound.

Bibliography

[1] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 504–528, Berlin, Heidelberg, 2010. Springer-Verlag.