Combining Application-Level and Database-Level Monitoring to Analyze the Performance Impact of Database Lock Contention

Holger Knoche
Kiel University, Software Engineering Group
24118 Kiel, Germany
hkn@informatik.uni-kiel.de

Abstract

Database lock contention can severely impact application performance and limit scalability. This can be of particular importance when major modifications are made to transactional software, such as large refactorings or modernization projects.

In order to assess the criticality of such modifications, it is necessary to measure the current degree of database lock contention, and attribute the effects to the appropriate sections of the application. However, current monitoring tools do not provide both application-level and database-level monitoring data with sufficient detail at the same time.

In this paper, we present an approach to combine application-level and database-level monitoring to measure lock contention on a per-section basis, and present first experimental results from a prototypical implementation for PostgreSQL.

1 Introduction

Database transactions can be a limiting factor for the runtime performance of a software application. The exclusive access to data items guaranteed by the well-known ACID properties limits the achievable degree of concurrency of the application, since concurrent accesses contending for access to the same data item – and thus the same lock – are serialized by the database.

Lock contention is impedimental to performance in several ways. It can severely limit scalability, and modifications to transactional sections can cause locks to be held for a longer time, thus increasing lock contention and decreasing throughput. The latter effect is of particular importance in modernization projects towards distributed architectures such as Microservices, where remote service invocations are inserted into existing applications.

Due to its potential relevance to application performance, the ability to quantify and locate database lock contention in productive environments is crucial. However, it is not sufficient to look at the database alone. The lock contention inside the database has rather to be attributed to the parts of the applications which cause or fall victim to it.

Unfortunately, current monitoring tools do not provide the required data for this task, as these tools either work on the application level or on the database level. Application-level monitoring tools such as DynaTrace[1] can, for instance, discover that a particular database access takes a long time, but can not provide any information about the cause. Database-level monitoring tools, on the other hand, commonly only report aggregated statistics for database objects such as tables.

In this paper, we propose an approach to enable detailed analyses of lock contention by combining application-level and database-level monitoring. We show how this approach can be implemented for the popular open-source RDBMS PostgreSQL, and show first experimental results underlining its potential.

The remainder of this paper is structured as follows. In Section 2, we investigate the problem in further detail, and sketch our prototypical implementation in Section 3. Results from our experiments are presented in Section 4. Related work is discussend in Section 5, and conclusions are drawn in Section 6.

2 Problem Statement

As described in the introduction, our goal is to gather detailed data about lock contention, and attribute it to the appropriate sections inside the application. For this purpose, we need to monitor the following events: On the application level, we need to know when a transaction is started or terminated, and associate the transaction with the appropriate code section. On the database level, in addition to the start and end of a transaction, we also need to monitor when a transaction blocks due to a lock conflict or resumes once the lock is available. For the latter, we furthermore wish to know the lock object (i.e., the locked table or row).

To correlate the data from both levels, we require an identifier that is available from both the application and the database. Most databases assign transaction IDs for internal management, which may be accessible from the application. For instance, PostgreSQL provides the function `txid_current()` to retrieve the current transaction ID via SQL.

[1]`https://www.dynatrace.com`

3 Prototypical Implementation

To instrument an application, we manually inserted a piece of monitoring code at locations where transactions were started, committed or rolled back. This code determined the current transaction ID using the aforementioned function, and wrote appropriate monitoring events to a file.

For gathering the required data from the database, we employed and extended PostgreSQL's existing dynamic monitoring infrastructure (see [9], section 27.4). The PostgreSQL source code contains trace hooks, to which probes can be attached with DTrace (under Solaris) or SystemTap (under Linux). However, these hooks are deactivated by default and need to be activated at compile time.

The existing trace hooks cover a plethora of events, including transaction start, commit, and rollback as well as the begin and end of a lock wait. However, these events did not provide the necessary context data for our needs. In particular, the lock-wait events did not provide any information about the transaction in which they occurred. In addition, the transaction start and end events only provided the local transaction ID, which is different from the global, permanent ID returned by `txid_current()` and only unique per backend thread.

We therefore extended the transaction start hook to include the global transaction ID and added the backend thread ID and the local transaction ID to all hooks. This approach was chosen because the two IDs were either already available or could easily be retrieved from the process context.

4 Experimental Results

In order to evaluate and validate our approach, we conducted four experiments, which are described below. All experiments were conducted using PostgreSQL 9.5.3 on an Intel Core i7-3770K at 3.50 GHz with 16 GB RAM, a 2 TB SSHD and Gigabit Ethernet, running Ubuntu Server 16.04 LTS (Kernel 4.4) and SystemTap 3.0. The default isolation level (read committed) was used for all experiments.

For load generation, a Raspberry Pi 2 at 900 MHz with 1 GB RAM and 100 MBit Ethernet running Raspbian 8 (Kernel 4.4), Oracle JDK 1.8.0_101, and JDBC driver 9.4.1209 was used. The two machines were connected to the same Gigabit Ethernet switch.

Experiment 1: Measurement Validation

In order to validate our measurement infrastructure, we constructed the following experiment to intentionally and selectively cause lock contention. On the client, two worker threads were created, each having its own database connection. In a third thread, pairs of tasks performing database updates were simultaneously submitted to the worker threads. With a given probability, the dispatcher thread would configure the two tasks to update the same rows, thus provoking a lock conflict. Otherwise, disjoint row sets were chosen.

The transaction pairs were written to a client log together with the information whether a lock conflict was expected or not. This client log was then checked against the events produced by our implementation.

We conducted this experiment with several conflict probabilities and 10000 transaction pairs each. The implementation registered lock events precisely when expected in all cases.

Experiment 2: Transaction Duration and Lock Contention

As discussed in the introduction, an increase in transaction duration can lead to an increase in lock contention and thus in response time. In order to quantify this effect, we created a table with 100 rows in the database. The table was deliberately chosen to be small as to avoid disk I/O as much as possible. Then, we started a total of n_t update tasks with a submission frequency of λ transactions per second. Each of these tasks updated a number n_r rows in the table, which were chosen at random for each transaction. To avoid deadlocks, the updates were performed strictly in ascending order of their primary keys.

In order to analyze the effect of an increase in transaction duration, a variable delay between performing the updates and committing the transaction was inserted. A sufficient number of worker threads and database connections was chosen to prevent distortion due to clogging on the client side.

Results from a run with 64 worker threads, 10000 tasks, 5 update rows per task, and a submission frequency of 40 transactions per second are depicted in Figure 1. The left panel shows the blocking probability of a transaction, i.e. the probability that a transaction has to wait at least for one lock, depending on the duration of the variable delay. The right panel shows the mean client-observable transaction duration, together with the standard deviation, depending on the delay.

A notable observation from this experiment is that the blocking probability saturates already at about 70%. We are further investigating this observation as part of our future work.

Experiment 3: Lock Contention in the TPC-C Benchmark

To prove the general applicability of our approach in a more realistic setting, we monitored the well-known TPC-C benchmark [4] provided by HammerDB[2] with it. Since HammerDB is not available for ARM processors, it was run on an Intel Core i7-4500U notebook running Debian 8.5.0.

Using our approach, we were able to analyze the locking behaviour for each of the benchmark's client transactions (e.g., payment, new order, delivery).

[2]Version 2.2.0, `http://www.hammerdb.com`

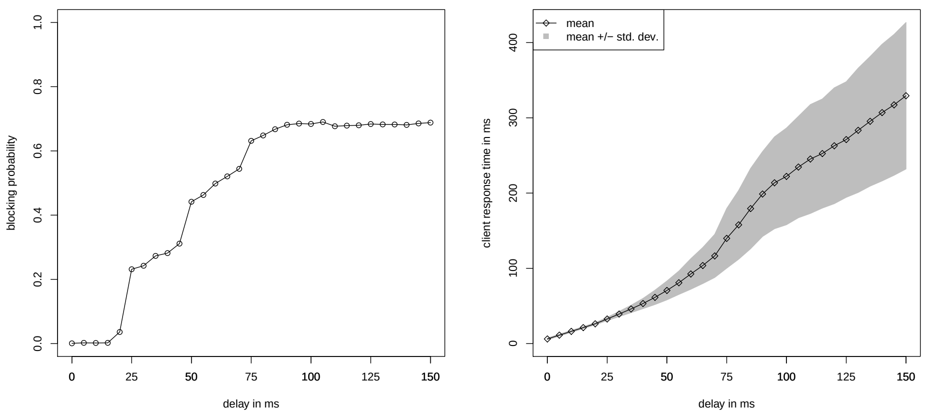

Figure 1: Delay vs. measured blocking probability and client response time for $n_r = 5$, $n_t = 10000$, and $\lambda = 40$

However, due to the specifics of PostgreSQL's locking procedure [5], lock conflicts on a row may be reported either on the transaction holding the lock or on the row itself. Therefore, for instance, we measured that payment transactions spent 69% of their wait time waiting for other payment transactions, 4% waiting for new-order transactions, and 19% waiting for rows from the warehouse table.

Nevertheless, this analysis provided a valuable insight into the causes and victims of lock contention.

Experiment 4: Monitoring Overhead

In order to roughly quantify the monitoring overhead of our approach, we ran Experiment 2 with a high submission frequency ($\lambda = 500$) against an instrumented and a non-instrumented database. We did not detect significant differences in response time, from which we conjecture that the overhead is not very large, but further research such as [7] is required to make a sound conclusion.

5 Related Work

A plethora of work and several tools are available for application-level monitoring, such as the Kieker framework [6]. Jenq et al. [1] use database monitoring to calibrate an analytic model, but consider only physical resource consumption (such as CPU and disk I/O). Elnikety et al. [2] measure client-observable transaction throughput during the TPC-W benchmark. Lock contention in shared memory is investigated in [3]. A visualization for synchronization conflicts in applications is described in [8].

6 Conclusions and Future Work

In this paper, we have presented an approach to combine application-level and database-level monitoring to analyze the performance impact of database lock contention on applications, and shown the usefulness of this approach using selected experiments.

In our future work, we intend to automate the instrumentation process and investigate the applicability of this approach to other databases. Furthermore, we intend to further investigate the dynamics of lock contention in practice and perform a detailed analysis of the performance overhead of our approach.

References

[1] B. C. Jenq, W. H. Kohler, and D. Towsley. "A queueing network model for a distributed database testbed system". In: *IEEE Trans. on Software Engineering* 14.7 (1988).

[2] S. Elnikety et al. "Predicting Replicated Database Scalability from Standalone Database Profiling". In: *Proc. 4th ECCS*. 2009.

[3] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. "Analyzing Lock Contention in Multithreaded Applications". In: *Proc. 15th PPoPP*. 2010.

[4] Transaction Processing Performance Council. *TPC Benchmark C – Standard Specification*. Feb. 2010. URL: `http://www.tpc.org`.

[5] R. Haas. *Deadlocks*. Oct. 2011. URL: `http://rhaas.blogspot.de/2011/10/deadlocks.html`.

[6] A. van Hoorn, J. Waller, and W. Hasselbring. "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis". In: *Proc. 3rd ICPE*. 2012.

[7] J. Waller and W. Hasselbring. "A Comparison of the Influence of Different Multi-Core Processors on the Runtime Overhead for Application-Level Monitoring". In: *LNCS 7303*. Springer, 2012.

[8] J. Waller et al. "SynchroVis: 3D Visualization of Monitoring Traces in the City Metaphor for Analyzing Concurrency". In: *Proc. 1st VISSOFT*. 2013.

[9] PostgreSQL Global Development Group. *PostgreSQL 9.5.3 Documentation*. 2016. URL: `https://www.postgresql.org`.