High-Performance Transaction Processing for Web Applications Using Column-Level Locking

Xiaodong Zhang and Jing Zhou(✉)

Shanghai Jiao Tong University, 800 Dongchuan Road, Shanghai, China
{xdzhang97,zhoujing2021}@sjtu.edu.cn

Abstract. Column-level concurrency control allows higher concurrency but also brings additional coordination overhead. Therefore, many relational database systems usually coordinate transactions at the row level. However, our observation based on real-world web applications suggests that row-level coordination can sometimes be too coarse. It can cause web applications to suffer reduced throughput due to false conflicts. To address this issue, we introduce an application-side column-level lock management system called CLL in this paper. It allows applications to choose the concurrency control granularity adaptively. With CLL, accesses to highly contended data items can now be executed in parallel without false conflicts caused by row-level coordination. Our evaluation shows that, in both synthetic and real-world workloads, CLL can help to improve performance significantly and achieve at most 64%/33% higher throughput, respectively.

Keywords: Web applications · Concurrency control · Object–Relational Mapping

1 Introduction

Web applications usually rely on the concurrency control of database management systems (DBMSs) to coordinate concurrent transactions. There is a trade-off between the degree of concurrency achieved and the coordination overhead. More precise coordination allows higher concurrency at the cost of increased management overhead (e.g., storage and computation) and complicated system architecture [9]. Nowadays, popular DBMSs provide row-level or even coarser-grained coordination [1,2,7,11] to avoid unnecessary overhead.

However, row-level coordination sometimes is not optimal. Take Broadleaf[1], a popular open-source e-commerce web framework, as an example. In Fig. 1, with row-level coordination, two transactions will conflict with each other when accessing the same row. This conflict is unnecessary since they require different columns. A recent study [12] has confirmed that some applications implement

[1] https://github.com/BroadleafCommerce/BroadleafCommerce.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
R. Chbeir et al. (Eds.): WISE 2022, LNCS 13724, pp. 186–193, 2022.
https://doi.org/10.1007/978-3-031-20891-1_14

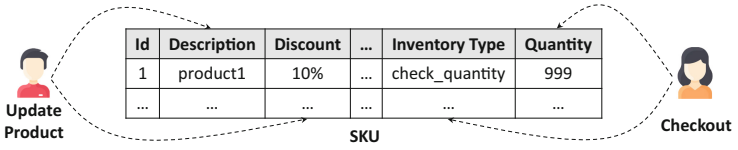

Fig. 1. In Broadleaf, transaction Checkout and Update Product access different columns of SKU.

Fig. 2. New Order and Payment transactions in TPC-C may block each other because of false conflicts. Colors distinguish columns accessed by different transactions. (Color figure online)

their own application-level coordination, bypassing the DBMSs to avoid such false conflicts. Nevertheless, in the absence of systematic design, these ad hoc coordinations are usually error-prone and can not improve performance efficiently.

Therefore, we designed CLL, an application-side column-level lock management system. It allows web applications to choose the appropriate coordination granularity for better performance while preserving correctness, i.e., transaction serializability. For SQL statements suffering from false conflicts, applications can coordinate at the finer-grained column level to improve transaction processing parallelism. As for other SQL statements, applications can still use database systems' existing mechanisms without adding any overhead.

In building CLL, two techniques are essential. First, to accurately identify the locks to be acquired, we use *Optimistic/Pessimistic Lock Location Prediction* (O/PLLP) [13] to prefetch data needed by scan SQL statements. Second, to mitigate the effect of database exclusive row locks when coordinating at the column level, we use *deferred writes* to defer the write operations until the commit phase. Our evaluation shows that CLL can bring up to 64%/33% throughput improvements respectively in TPC-C [3] and Broadleaf workloads.

2 Background

2.1 False Conflicts Caused by Row-Level Coordination

Existing DBMSs usually coordinate transactions at multiple granularities [5] for flexible concurrency control. For example, MySQL and PostgreSQL support

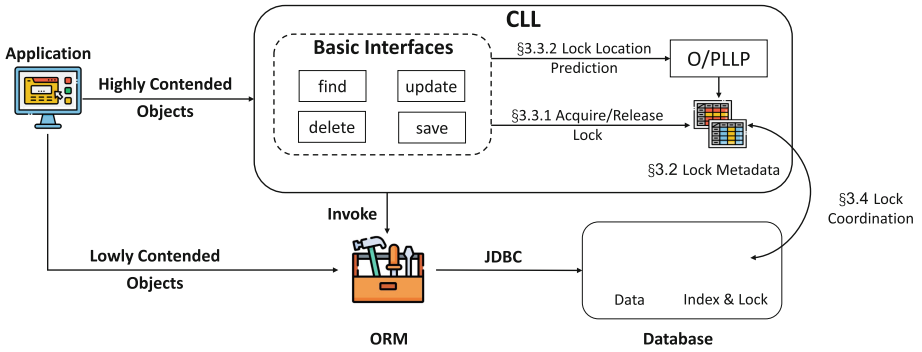

Fig. 3. The architecture of CLL

both page and row-level locks. However, among them, the finest granularity is usually the row level, which may make transactions suffer from false conflicts.

Figure 2 gives a detailed example in TPC-C. Although New Order and Payment transactions access different columns of warehouse and district tables, they may still block each other because of row-level conflicts. They are the two most frequent transactions in TPC-C. Therefore the false conflicts between them will hurt performance significantly.

2.2 Object–Relational Mapping

Object-Relational Mapping (ORM) [8] is widely used as middleware between web applications and backend storage systems. ORMs usually fetch full rows (i.e., a SELECT * FROM ... statement) from databases for simplicity [15]. Therefore DBMSs may be unaware of the needed columns, and implementing column-level locks inside database systems helps little in reducing false conflicts. This fact motivates us to design and build the column-level lock management system on the application side. Then developers can choose the granularity of concurrency control flexibly based on the applications' business logic characteristics.

3 Design and Implementation

3.1 System Overview

Figure 3 shows the architecture of CLL. SQL statements accessing highly contended data, such as ones in warehouse and district tables of TPC-C, are more likely to suffer from false conflicts. Therefore developers can utilize CLL for finer-grained coordination. As for other data, developers can directly access them through ORMs without adding unnecessary coordination overhead. CLL provides basic interfaces for CRUD, which are similar to ORMs'. With CLL, developers can specify columns to be locked for concurrency control. The lock metadata is stored in the memory of the server. To handle SQL statements scanning

multiple rows, CLL uses the Optimistic/Pessimistic Lock Location Prediction (O/PLLP) to prefetch the primary keys of the result set for locking.

3.2 Application-Maintained Data Structures

For each column in a table, CLL keeps a hashmap to store lock metadata. The key of a hashmap is the row's primary key, and the corresponding value is a read-write lock. SQL statements accessing highly contended data can acquire column-level locks to avoid false conflicts. These locks will be removed after being released by the last holder without causing a lot of storage overhead.

3.3 Identifying the Data for Locking

Handling SQL Statements Using Primary Key Equality in Conditions. Acquiring locks for SQL statements using the primary key equality in conditions is easy. According to the table and columns accessed, CLL first tries to find the primary key in the corresponding hashmap. If the key exists, the current statement can directly try to acquire the lock. The key absent means that no transaction is accessing the same column of the same row. Then CLL will atomically create the lock and get it granted for the statement.

Handling SQL Statements Using Other Conditions. We handle statements using other conditions in different ways. For statements with an index (non-primary key) equality in conditions, we use Optimistic Lock Location Prediction (OLLP) to identify the primary key of rows and get locks. OLLP will issue a non-blocking read-only query to retrieve the primary key set of required rows first. With index equality in conditions, such reconnaissance queries will not bring much overhead. Then we try to get column-level locks according to the primary keys of rows in the read/write set. After the locks are granted, we need to execute the statement again and validate whether the read/write set is the same as the reconnaissance query. If validation fails, we must retry the above procedure or abort the whole transaction.

For statements that may perform the sequential or index range scan, we apply Pessimistic Lock Location Prediction (PLLP), which acquires column locks for all rows through wildcard. The reasons are twofold. On the one hand, these statements involving many rows will bring significant overhead to column-level lock management. On the other hand, their read/write sets are likely changed after the reconnaissance query. Using OLLP may cause a lot of failed validation.

3.4 Lock Coordination

For SQL statements using column-level locks, we must prevent them from acquiring database row-level locks. Therefore we use weak isolation levels (Read Committed or Repeatable Read) for transactions. We can acquire row-level locks explicitly (e.g., such as for share/update) when the finer-grained coordination is

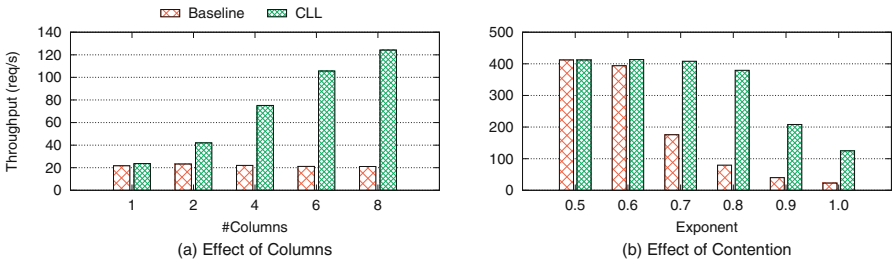

Fig. 4. The evaluation result with microbenchmark

unnecessary. However, in some databases, write SQL statements acquire exclusive row-level locks by default. To reduce the impact of these database locks, we defer write operations until the commit phase. So exclusive row-level locks will be held for little time and hardly cause write-write false conflicts.

3.5 Correctness and Consistency

CLL can be integrated with applications while preserving correctness and consistency. SQL statements using CLL are also wrapped in database transactions, so the atomicity and durability are guaranteed. Each statement will acquire locks from either database or CLL and release them until committed to guarantee the serializable isolation. When the application server crashes, we assume all ongoing transactions have failed. Therefore, the loss of column-level locks caused by a crash does not matter. After a restart, the system is still in a consistent state.

4 Evaluation

We evaluate CLL to answer the following questions: 1) In what workloads is CLL more effective? 2) How much benefit can CLL bring by avoiding false conflicts? 3) Will CLL decrease performance in workload without false conflicts?

4.1 Experimental Setup

Configuration. We build the application server based on the Spring framework with Hibernate-5.4.32 as ORM. CLL can work with DBMSs that supports weak isolation level and explicit locks, such as PostgreSQL and MySQL. We apply MySQL-8.0.25[2] for evaluation. The database and web server are deployed in independent physical machines. Both have 2×12 2.20 GHz cores (Intel Xeon Processor E5-2650 v4), 128 GiB DDR4 memory, and a 1 Gbit/s NIC.

Comparison. We compare CLL with the baseline that relies on DBMSs' row-level concurrency control. They both use a weak isolation level (Read Committed). The baseline acquires row-level locks explicitly for correctness. CLL replaces them with column-level locks when accessing highly contended data.

[2] For Broadleaf, we use MySQL-5.7.35 as suggested.

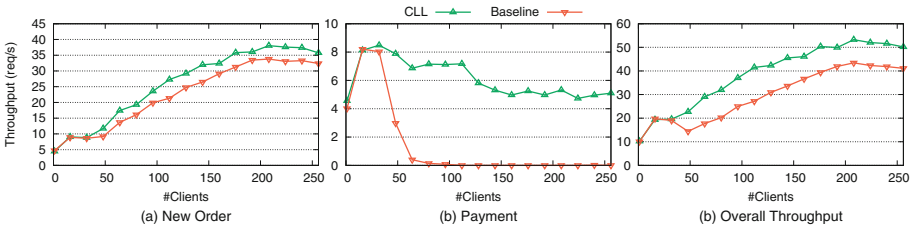

Fig. 5. The evaluation result with standard TPC-C

4.2 Microbenchmarks

The benefit of CLL is related to the contention and the number of no-overlapping column subsets accessed by transactions. To show their effect and answer the first question, we designed the following microbenchmark. A table is initialized with 100,000 rows, each of which has eight columns. Concurrent transactions read or update different columns of a row chosen from a Zipfian distribution.

The result is shown in Fig. 4. When the contention is low and most transactions access the same columns, CLL performs similar to the baseline. As the contention and the columns used increase, transactions using row-level coordination are more likely to suffer from false conflicts. Thus CLL can bring more performance benefits and achieve at most 5.4× higher throughput.

4.3 Macrobenchmarks

To answer the second question, we first compare CLL with baseline in TPC-C workload with one warehouse. Among the five transactions, the New Order (45%) and Payment(43%) can benefit from CLL. The result is shown in Fig. 5. As the number of clients increases (contention becomes higher), the Payment throughput of the baseline approaches zero, making CLL significantly better. The reason is that TPC-C specifies the upper limit of transaction response time. Under high contention, in the baseline, most Payment transactions are timed out due to false conflicts. Similarly, with CLL, the throughput of New Order transactions can be improved by at most 28% (112 clients). As for the overall throughput, CLL can achieve at most 64% (64 clients) higher than the baseline.

To answer the third question, we evaluate CLL with TPC-C New Order transaction only, in which row-level locks cause no false conflicts. As shown in Fig. 6, CLL brings a little overhead and has at most 8.9% lower throughput. Therefore, to avoid unnecessary overhead, we should not use column-level locks in workloads with little or without false conflicts.

4.4 Performance Improvement in Real-World Applications

Broadleaf is a framework used for e-commerce applications. As we mentioned, Checkout and Update Product may suffer from false conflicts caused by row-level

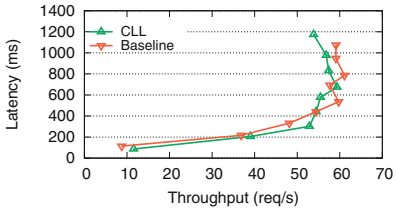

Fig. 6. The evaluation result with TPC-C New Order only (#clients from 1 to 64)

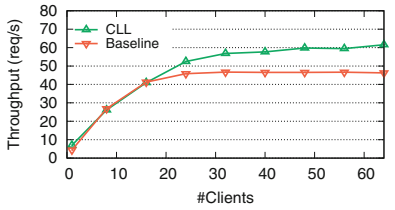

Fig. 7. The evaluation result with Broadleaf

coordination. We integrated CLL into Broadleaf with little engineering effort to address this issue. To simulate the high contention, we keep one seller updating the hottest product and many customers buying it concurrently. The result in Fig. 7 shows that CLL can improve the throughput up to 33%.

5 Related Work

Some DBMSs already provide column-level coordination. PostgreSQL [2] supports For Key Share and For No Key Update hints. Nevertheless, such column-level coordination can only be used for the primary key. Google F1 [10] fully supports column-level locks. However, it creates a separate lock column for each column to store timestamp, which may bring significant storage overhead. Furthermore, F1 is built for Google AdWords business, and it is not open-source. Our CLL can fully support column-level locks. It creates lock only when the data is accessed, thus causing little memory overhead. Finally, its design does not rely on specific databases or ORMs so that it can be deployed with most applications.

Some works focus on the optimization of concurrency control. Graefe et al. [4] proposed ghost records (logically deleted records) to avoid false conflicts caused by gap locks. Grechanik et al. [6] combine static analysis and run-time monitoring to detect and prevent database deadlocks in applications efficiently. To handle contended workloads, Wang et al. [14] designed interleaving constrained concurrency control (IC3), which allows parallel execution for transactions under contention while preserving serializability. These works optimize concurrency control in other ways rather than improving concurrency control granularity.

6 Conclusion

In this work, we propose CLL, an application-side column-level lock management system to avoid false conflicts caused by row-level concurrency control. With CLL, developers can choose finer-grained coordination granularity for highly contended access to improve parallelism. The evaluation shows that it can improve throughput significantly in both synthetic and real-world workloads.

Acknowledgement. We appreciate the anonymous reviewers for their constructive feedback and suggestions.

References

1. MySQL. https://www.mysql.com/. Accessed 18 Aug 2022
2. PostgreSQL. https://www.postgresql.org/. Accessed 18 Aug 2022
3. TPC-C Benchmark. https://www.tpc.org/tpcc/. Accessed 18 Aug 2022
4. Graefe, G.: Hierarchical locking in B-tree indexes. In: On Transactional Concurrency Control. SLDM, pp. 45–73. Springer, Cham (2019). https://doi.org/10.1007/978-3-031-01873-2_3
5. Gray, J.N., Lorie, R.A., Putzolu, G.R.: Granularity of locks in a shared data base. In: Proceedings of the 1st International Conference on Very Large Data Bases, VLDB 1975, pp. 428–451. Association for Computing Machinery, New York, NY, USA (1975). https://doi.org/10.1145/1282480.1282513
6. Grechanik, M., Hossain, B.M.M., Buy, U., Wang, H.: Preventing database deadlocks in applications. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pp. 356–366. Association for Computing Machinery, New York, NY, USA (2013)
7. Huang, D., et al.: TiDB: a raft-based HTAP database. Proc. VLDB Endow. **13**(12), 3072–3084 (2020). https://doi.org/10.14778/3415478.3415535
8. O'Neil, E.J.: Object/relational mapping 2008: hibernate and the entity data model (EDM). In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, pp. 1351–1356. Association for Computing Machinery, New York, NY, USA (2008)
9. Ries, D.R., Stonebraker, M.: Effects of locking granularity in a database management system. ACM Trans. Database Syst. **2**(3), 233–246 (1977). https://doi.org/10.1145/320557.320566
10. Shute, J., et al.: F1: a distributed SQL database that scales. Proc. VLDB Endow. **6**(11), 1068–1079 (2013). https://doi.org/10.14778/2536222.2536232
11. Taft, R., et al.: CockroachDB: the resilient geo-distributed SQL database. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD 2020, pp. 1493–1509. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3318464.3386134
12. Tang, C., et al.: Ad hoc transactions in web applications: the good, the bad, and the ugly. In: Proceedings of the 2022 International Conference on Management of Data, SIGMOD 2022, pp. 4–18. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3514221.3526120
13. Thomson, A., Diamond, T., Weng, S.C., Ren, K., Shao, P., Abadi, D.J.: Calvin: fast distributed transactions for partitioned database systems. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, pp. 1–12. Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2213836.2213838
14. Wang, Z., Mu, S., Cui, Y., Yi, H., Chen, H., Li, J.: Scaling multicore databases via constrained parallel execution. In: Proceedings of the 2016 International Conference on Management of Data, SIGMOD 2016, pp. 1643–1658. Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2882903.2882934
15. Yang, J., Subramaniam, P., Lu, S., Yan, C., Cheung, A.: How not to structure your database-backed web applications: a study of performance bugs in the wild. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, pp. 800–810. Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3180155.3180194