OLTPShare: The Case for Sharing in OLTP Workloads

Robin Rehrmann [1, 3]
r.rehrmann@sap.com

Carsten Binnig [2]
carsten.binnig@cs.tu-darmstadt.de

Alexander Böhm[1]
alexander.boehma@sap.com

Kihong Kim[4]
ki.kim@sap.com

Wolfgang Lehner [3]
wolfgang.lehner@tu-dresden.de

Amr Rizk[2]
amr.rizk@kom.tu-darmstadt.de

[1]SAP SE, Germany
[3]TU Dresden, Germany

[2]TU Darmstadt, Germany
[4]SAP Labs Korea

ABSTRACT

In the past, resource sharing has been extensively studied for OLAP workloads. Naturally, the question arises, why studies mainly focus on OLAP and not on OLTP workloads? At first sight, OLTP queries – due to their short runtime – may not have enough potential for the additional overhead. In addition, OLTP workloads do not only execute read operations but also updates. In this paper, we address query sharing for OLTP workloads. We first analyze the sharing potential in real-world OLTP workloads. Based on those findings, we then present an execution strategy, called OLTPShare that implements a novel batching scheme for OLTP workloads. We analyze the sharing benefits by integrating OLTPShare into a prototype version of the commercial database system SAP HANA. Our results show for different OLTP workloads that OLTPShare enables SAP HANA to provide a significant throughput increase in high-load scenarios compared to the conventional execution strategy without sharing.

PVLDB Reference Format:
Robin Rehrmann, Carsten Binnig, Alexander Böhm, Kihong Kim, Wolfgang Lehner, Amr Rizk. OLTPShare: The Case for Sharing in OLTP Workloads. *PVLDB*, 11 (12): 1769-1780, 2018.
DOI: https://doi.org/10.14778/3229863.3229866

1. INTRODUCTION

Motivation: Until now, the topic of sharing query resources for OLAP workloads has been studied and many different techniques such as materialized views [15], multi-query optimization [16], as well as shared scans [6] and shared plans [12] have already been proposed. The question is, why existing work has mainly focused on OLAP workloads and has not yet investigated the resource sharing potential of OLTP workloads.

In OLAP workloads, the sharing potential arises from the fact that long-running and complex queries with multiple joins need to be executed. To that end, the aforementioned sharing techniques for OLAP workloads all try to reduce the query overhead by avoiding to re-execute common expensive sub-expressions for each incoming query. Materialized views, on one hand, pre-compute the results of common sub-expressions a priori and store the results as derived tables. On the other hand, in the context of multi-query optimization, shared plans and shared scans try to merge incoming queries that have sharing potential at runtime by using some form of batching. That way, common sub-expressions in a batch of queries only need to be executed once.

At first sight, sharing in OLTP workloads does not seem to be beneficial, because of small (touching only a few tables) and short running (only a few milliseconds) individual point queries. Another challenge is the fact that OLTP workloads need to execute not only read, but write operations as well. Especially, write-heavy workloads render sharing techniques such as materialized views useless due to the high overhead of keeping the latter up-to-date.

It is also commonly not well understood if there is enough sharing potential between different transactions to make resource sharing by merging query fragments attractive, since there is typically no overlap between individual statements.

Thus, is this the end of this paper? To answer this question we first analyzed more than 7000 real-world OLTP workloads of the CMU database application catalog [23][1]. An interesting first observation is that 89% of all these workloads use less than 10 distinct READ statement strings in all their transactions. These READ statements make about 80% of a com-

[1]The details of the analysis are discussed in Figure 2.

Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3229866

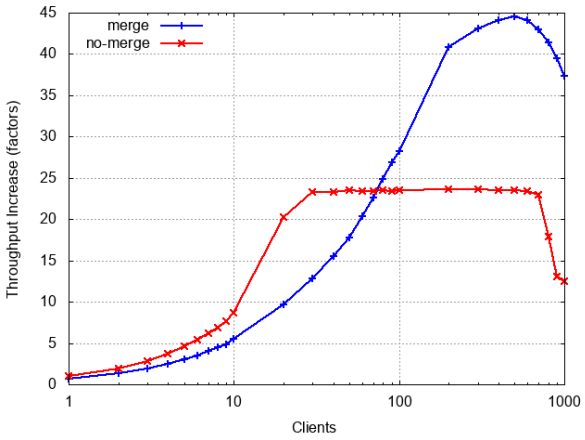


Figure 1: Throughput increase of our micro-benchmark

mon OLTP workload [9]. Even more interestingly, 50% of all the transactions are composed of a distinct single-key lookup query of the form `SELECT atts FROM table WHERE key=?` that can take different input key parameters.

To assess whether sharing of resources in OLTP workloads generally makes sense or not, we first consider a best-case scenario consisting of a single statement type. For this, we run a simple synthetic benchmark using only single-key lookup queries (which is the most common statement type, according to the analysis above). For the benchmark, we use only a single table with 10 million rows where each query randomly selects a parameter for the key.

In order to analyze the sharing potential, we extended the commercial database SAP HANA so that incoming queries can be batched within a queue. We use in total 5 execution threads on a 10 core machine (which had other threads that were reserved for other purposes such as session handling). To merge queued queries, the execution threads poll the queue in regular intervals and merge all queued queries into a single statement; this results in a merged statement with IN-lists. That merged statement is then compiled and executed. For the efficient execution of merged statements, we leverage the fact that SAP HANA, as most modern DBMSs, offers a bulk lookup interface for indexes which is used to support queries with IN-lists. The results of our motivating benchmark comparing the execution strategy with merging and without merging (i.e., conventional SAP HANA) are depicted in Figure 1. The y-axis shows the throughput increase of the system when scaling up the number of clients.

The reported increase in Figure 1 uses the throughput of running the workload with a single client where merging is disabled as a baseline. We observe that merging has a negative impact for a relative small number of clients where the DBMS still has enough resources. For example, with 30 clients, merging introduces an overhead that halves the throughput. However, at this point the resources of the conventional DBMS are already saturated: the throughput of the no-merging approach stagnates while the throughput of the merging strategy continues to increase up to 300 clients and achieves a throughput that is up to 2× higher than the non-merging execution. As expected in both cases, the throughput drops as soon as too many clients are connected.

Although we know that this workload is far from being realistic, it shows that sharing may have a huge potential in OLTP workloads to better scale a DBMS in high-load scenarios.

Contributions: In this paper, we systematically investigate a considerable body of OLTP workloads and present an execution strategy, called OLTPShare, that implements a query sharing scheme for OLTP workloads. In a nutshell, the execution strategy batches all incoming transactions using a queue-based approach, similar to the setup described before. In OLTPShare, we only merge READ statements of single-statement transactions, since they provide a high benefit as shown before and can be efficiently as well as effectively merged. We also noticed that these types of workloads are common for many web-based application scenarios that use the database system as a key-value store and implement most of the application logic within the application server. In contrast to the simple setup used in our previous benchmark, OLTPShare maintains multiple query queues - one for each distinct statement type. In order to limit the number of queues, we only batch the most frequent statement types. Obviously, the presence of OLTPShare does not prevent the processing of arbitrary transactions including updates as well as multi-statement transactions. For those cases, OLTPShare efficiently separates these types of transactions from the "mergable" ones and executes them without merging.

An important aspect is to find an optimal batching strategy for the mergeable part of the workload. To do so, we leverage a queuing-model that allows OLTPShare to find a batching strategy that aims to maximize the throughput for the given workload and a given system setup (number of queues and available execution threads). The queuing model takes statistics about the workload (e.g., arrival rate, average execution latencies) into account and derives an assignment of execution threads to queues as well as an optimal poll interval per queue.

We show for a number of different OLTP workloads that OLTPShare is capable of providing a significant throughput increase in high-load scenarios. The overall performance increase for all of these workloads is in the range of the numbers reported before even for workloads that include update statements and queries that do not only execute single-key lookups.

To put it into a nutshell we make the following main contributions in this paper:

- To the best of our knowledge, this is the first paper to analyze resource sharing opportunities for pure OLTP workloads.
- Moreover, we provide an implementation of our execution strategy called OLTPShare in SAP HANA.
- We use an analytical queuing-model which allows OLTPShare to derive an optimal batching strategy for a given workload.
- Using our prototypical implementation in SAP HANA, we show an extensive evaluation using different OLTP workloads such as the YCSB and the TATP benchmark as well as micro-benchmarks to evaluate the accuracy of our analytical model.

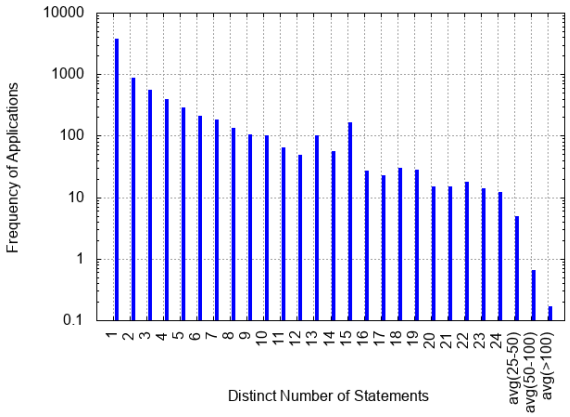


Figure 2: The frequency of number of distinct statement strings in OLTP applications of CMDBAC.

Outline: The remainder of this paper is structured as follows: Section 2 shows the results of our analysis of real-world OLTP workloads from the CMU Application Catalog [23]. We then give an overview of OLTPShare in Section 3 and discuss the details of the queuing model in Section 4. Thereafter, Section 5 details the implementation of OLTPShare, while Section 6 presents our evaluation. Finally, Section 7 discusses preceding work before we conclude our paper in Section 8.

2. WORKLOAD ANALYSIS

In this section, we analyze typical OLTP applications regarding two important questions:

1. How many distinct query statements are used by a typical OLTP application?
2. Which statement patterns are most frequently used?

The motivation of the first question is to identify the sharing potential in real OLTP workloads: if the number of distinct statements is low then the sharing potential in general may be considered high. Furthermore, as motivated in the introduction, merging single-statement read-only transactions bears some high sharing potential without increasing the complexity of the multi-query framework within the DBMS. Therefore, the aim of the second question is to see if these types of transactions are dominant in many real-world OLTP applications or if the workload in reality is far more complex and typically consists of multi-statement transactions or exhibits a high update ratio.

2.1 Real-World Workloads

The Carnegie Mellon Database Application Catalog (CMDBAC) [23] lists a wide range of open-source real-world database applications crawled from on-line source code repositories. The CMDBAC deploys and executes these applications and lists some important statistics of those applications on its website. By the time, we have analyzed the results in CMDBAC, there were 7383 projects, which could be deployed and executed without any failure.

We analyzed the statistics of those 7383 projects which included a log of all executed transactions. In order to be able to extract all distinct queries from those transaction logs, we first extracted the query and update statements and replaced all parameter values within the extracted statements by question marks. Figure 2 depicts the resulting distribution of distinct statements per application.

The x-axis shows the number of distinct statement strings found in the log after replacing the parameter values by a placeholder. The y-axis shows how many applications have that number of distinct statement strings. We observe that approximately 3.600 applications have a workload that consists of only one distinct statement string.

Overall, by analyzing our results we found out that 89% of all applications in CMDBAC have 10 distinct statement strings or less and 50% of the applications only use a single distinct statement string. Furthermore, it is important to note that all these applications only use single-statement transactions and more than 80% of all statements are simple key-based lookup queries. This underpins the significant sharing potential in many of those applications.

We are aware that CMDBAC only lists more simple web-based applications and does not focus on enterprise-level applications. It is therefore worthwhile to mention that another study [9] has already analyzed real-world OLTP workloads of enterprise-level applications. Interestingly, the study showed similar results: more than 80% of all transactions of those OLTP applications are single-statement read-only queries and more than 50% are simple key-based lookup queries.

2.2 Synthetic Workloads

In addition to actual workloads from many open source projects, we also studied the query types of synthetic benchmark scenarios, specifically, the YCSB, TATP, and TPC-C Benchmark. The overall goal of analyzing the synthetic benchmarks is to see how well these benchmarks align with the real-world applications with regard to their sharing potential and thus to justify that the findings obtained in our experimental section reflect the results that could be obtained with real-world workloads as well.

The Yahoo! Cloud Serving Benchmark (YCSB) [3] simulates cloud applications, targeting typical No-SQL workloads. The benchmark consists of six workloads, each being a mix of four query types (READ, SCAN, INSERT and UPDATE) with a specified ratio of different query types running at the same time, e.g. workload A consists of 50% READ and 50% UPDATE queries. All queries of a kind are the same, i.e., all read queries are the same, which translates to all read queries have sharing potential. The analysis of the number of distinct read queries therefore perfectly fits our observation in Figure 2, which shows that the largest group of applications only exhibits a single query type, obviously with varying parameter values during runtime.

The Telecommunication Application Transaction Processing Benchmark (TATP) [17] simulates a telco application and consists of four tables, five distinct READ statements, three distinct UPDATE statements plus one insert and one delete statement. The ratio of read statements to update statements is 80:20. We can see that with a workload consisting of 80%

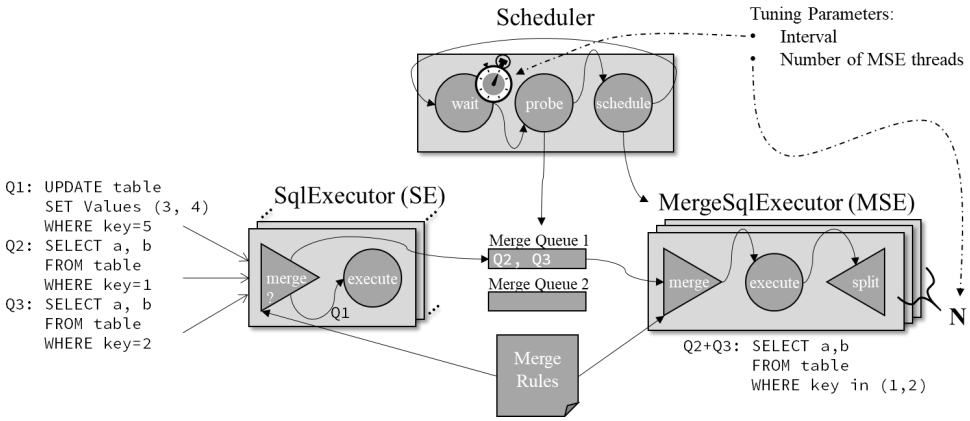


Figure 3: Overview of the OLTPShare Execution Strategy in SAP HANA

read statements, of which there are only five distinct statement strings, we have a large sharing potential also in this benchmark. This matches the results of [9], who showed that real-world scenarios consist of workloads with approximately 80% read queries.

Finally, the TPC-C Benchmark [19] is probably the best known and most accepted benchmark in the database community, when it comes to the evaluation of OLTP systems. According to its specification, the benchmark consists of five distinct multi-statement transactions with overall 19 distinct READ, nine distinct UPDATE and four distinct INSERT statements. Two of the five transactions – namely the Order-Status and the Stock-Level transactions – are read-only, the latter consisting of only two distinct READ statements. Compared to the other workloads analyzed before, this benchmark has a more complex workload. For that reason, we excluded the TPC-C from our experimental evaluation, because it contains range READ statements, which we currently do not support. We leave the extensions and evaluation of TPC-C for future work.

2.3 Discussion

Since merging queries in OLTP workloads is only beneficial when there is sharing potential, we analyzed the real-world workloads of 7383 open source projects, gathered by CMDBAC. We show that 89% **of the open-source on-line projects in CMDBAC** are running a workload of **at most 10 distinct statement strings**. Based on the work of Krueger et al. [9], we also see that 80% of a common enterprise application workload consists of read-only queries. Krueger et al also show that most of those read-only queries are simply single-point lookup queries. In addition, we showed that important OLTP benchmarks, such as the YCSB, TATP or the TPC-C benchmark, follow a similar pattern and also typically use only a small set of distinct statement strings. As a result, we conclude that OLTP workloads have a huge yet unexploited potential for sharing. The next sections discuss, how we are going to efficiently leverage this potential.

3. SYSTEM OVERVIEW

After having motivated the sharing potential of typical OLTP applications, we now present the main building blocks of our novel evaluation strategy called OLTPShare that leverages the sharing potential in OLTP workloads. The main idea of the OLTPShare execution strategy is depicted in Figure 3.

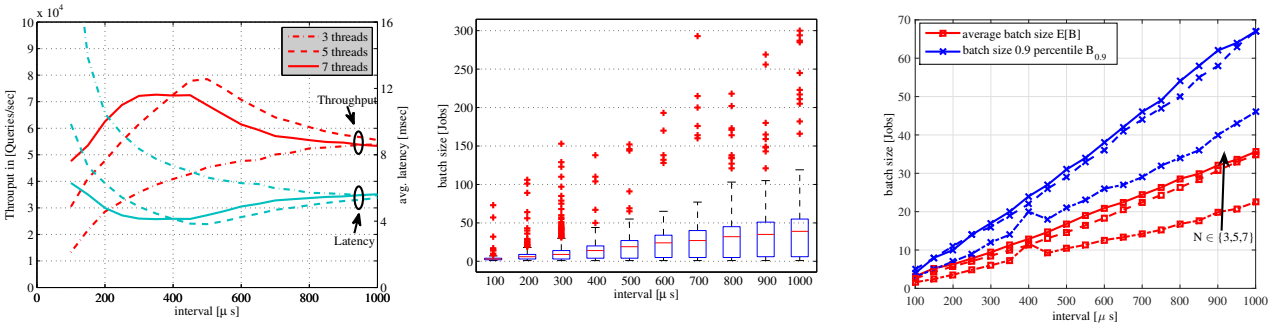
In a first step, we need to decide whether an incoming statement of a client is mergeable or not. In SAP HANA, clients always communicate directly with a dedicated SqlExecutor (SE) thread that – in the conventional version without sharing – is responsible for executing a statement of a transaction and return the result set or error state. Within our OLTPShare execution strategy, we extended the SE-thread such that it first checks whether the incoming statement is mergeable or not.

- In case it is not mergeable, the SE-thread directly executes the statement and returns the result to the client.
- In case the statement is considered mergeable, it is admitted to a dedicated merge queue for the given statement type.

Every merge queue is polled by a scheduler thread on a regular basis. If it finds elements in the queue, which are not yet processed, it instructs one MergeSqlExecutor (MSE) thread of a pool of MSE-threads to unload the unprocessed statements. The algorithm for scheduling MSE-threads is a simple Round-Robin implementation. An instructed MSE-thread merges the unprocessed statements of the merge queue into a rewritten statement, executes this statement, and finally splits the result (i.e., one for each dedicated incoming statement). The individual result sets are then sent back to the individual clients.

In the following, we discuss the individual steps required for both extensions to get a better overview on the relevant questions – the merge decision in an SE-thread and the merged execution in an MSE-thread. Afterwards, the Sections 4 and 5 discuss some more details with regard to two aspects:

1. Section 4 presents a model to find an optimal configuration of OLTPShare (e.g., to define the poll interval) given a particular workload for maximizing the overall throughput.



(a) Throughput and Latency **(b)** Batch sizes for $N = 5$ **(c)** Average batch size and $B_{0.9}$

Figure 4: Measured Statistics (throughput, latency, batch sizes) for different number of threads N and various interval lengths I.

2. Section 5 provides more in-depth implementation details of how the merge decision and the merged execution can be efficiently implemented in SAP HANA without introducing a too high overhead.

3.1 Merge Decision

As mentioned before, the SE-thread first checks whether an incoming statement is mergeable or not. This is done in two steps. In a first step, since OLTPShare currently only supports the shared execution of single-statement transactions, the SE-thread checks whether the statement is part of such a single-statement transaction or if it is part of a multi-statement transaction. In case it is part of a single-statement transaction, the SE-thread then checks whether or not the statement can be merged with other statements that use the same statement string (but with potentially different parameter values). The SE-thread therefore consults a list of merge rules whereas each merge rule specifies a *source statement pattern* (without parameters) that describes individual statement patterns. For example, the rule to merge single-key lookup queries would consist of the statement pattern `SELECT atts FROM table WHERE key = ?`. The details of how the merge decision using merge rules can be efficiently executed is described in Section 5.

The list of merge rules in OLTPShare can be extended. In order to define an optimal set of rules, we first identify the most frequent statement strings. Based on such analysis, the merge rules can be implemented for those statement strings and added to OLTPShare. For our experiments, we implemented merge rules for a variety of different OLTP benchmarks. Since the number of distinct statement strings is typically low, the manual effort is negligible. In future, we plan to extend OLTPShare to automatically generate those merge rules and thus even be able to dynamically adjust them at runtime.

3.2 Merged Execution

In the current prototype, OLTPShare provides a merge queue for each merge rule; i.e., within a queue we only find queries of the same type, and therefore the same distinct statement string – however, with potentially different parameters values. If a workload has too many distinct statements, we also support that multiple merge rules are mapped to one queue in order to not have too many merge queues in total. In that case, the merge queue may contain a mix of statement strings. An MSE-thread then applies the individual merge rules sequentially and merges only to the subset which matches the currently considered rule.

For the merged execution, one or several MSE-threads are assigned to a queue depending on the statement distribution in the workload. The MSE-threads poll all the batched statements from their dedicated merge queue. For every queue, OLTPShare defines a fixed interval (e.g., every $100\mu s$) at which one of the MSE-threads gets access to the queue. Section 4 outlines how to set the optimal number of MSE-threads and the poll interval for a queue to maximize the overall throughput.

After polling, all statements in the queue are merged into a merged statement string instantiating a *target statement pattern* for the particular merge rule. For the example above, the target statement pattern used to merge all single-key lookup queries $Q_1, Q_2, \ldots, Q_N$ into a single statement would look like `SELECT atts from table WHERE key in (?1, ?2, ..., ?N)` which is using an IN-list expression. Referring to the running example, the result of merging the two statements is also illustrated in Figure 3.

Finally, the merged statement is executed by the MSE and the result is split up to obtain a separate result for each input statement. Thereafter, the individual results are returned to the corresponding client. The details of executing those statements and splitting the results efficiently is discussed in Section 5.

4. QUEUING MODEL

In this section, we provide a queuing model to capture the impact of the parameters that control the throughput in the OLTPShare execution strategy, i.e.,

- the number of MSE-threads N that dequeue a merge queue and
- the polling interval I in which a merge queue is inspected.

In the following, we first discuss the basic queuing model with only a single queue where all available N MSE-threads are assigned to this particular queue. To that end, the main problem in the basic model is to find the poll interval to maximize the throughput. Afterwards, we extend the basic model to multiple queues. As an additional dimension, we do not only derive the poll interval per queue but also compute an assignment of MSE-threads to queues – i.e., how the overall N MSE-threads are divided into different queues.

In the context of OLTPShare, we currently use the model offline to determine the assignment of the MSE-threads to the merge queues and the polling interval per merge queue. This is a realistic assumption if the workload does not change over time, which is the case in many OLTP applications we see at SAP. In the future, we plan to extend our implementation to also apply the model dynamically at runtime to adjust the configuration to potential workload changes.

4.1 Basic Model

Figure 4 shows the basic measured statistics where these two parameters N and I are apparent. Here, we observe first the impact of the number of MSE-threads N on the system throughput. Interestingly, the impact of the inspection interval I is not monotonic. This can be intuitively explained as follows: too small intervals lead to smaller batch sizes that do not leverage the sharing potential, while too large intervals introduce unnecessary waiting times for the arriving jobs.

Figure 4 (b) and (c) therefore show the batch sizes for different polling intervals I. Note the persistent variability of the batch size B with an increasing interval length. Hence, waiting for larger intervals I does not necessarily entail a higher sharing potential in the cases where the batch size is small. This effect surely depends on the type of workload at hand, i.e., the statistical characteristics of the incoming statement stream.

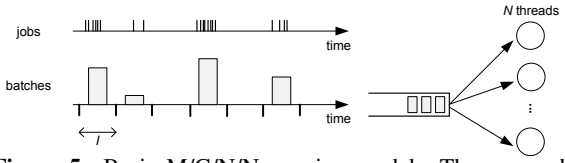
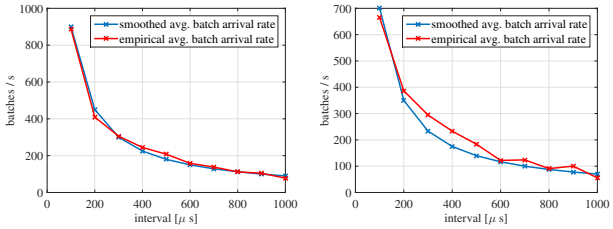


Figure 5: Basic M/G/N/N queuing model. The queue depicted is only needed for an M/M/N approximation of the batch waiting time.

To roughly estimate the appropriate interval length I that maximizes the throughput for a given workload, we consider the queuing-model as depicted in Figure 5 as an insensitive M/G/N/N queuing-system. In this model, batches arrive according to some Poisson process with rate λ_I. Figure 6 shows how the empirically measured batch arrival rate depends on the interval I as well as on a simple smoothing function that is used for the model. In a real workload the arrival rate depends on the number of clients that submit statements to the database system. We assume that the service times of batches at single threads are independently and identically distributed with some distribution $\mathcal{F}$ and that these have a mean service time



(a) $N = 5$ **(b)** $N = 7$

Figure 6: Smoothing of the empirically measured batch arrival rate for a varying number of threads N.

s_I, which represents the execution time of a merged query in the queue.

For the basic model, we assume that the system has N available threads and we are particularly interested in finding the polling interval I that maximizes the throughput. In order to maximize the throughput, we first compute the probability $p_B(I)$ of an incoming batch seeing that all threads are busy, given the interval length I. Therefore, we need to know the average batch service time and average batch sizes for given interval lengths I.

In the current prototype of OLTPShare, we therefore rely on a calibration run that executes a given workload under different intervals I using N MSE-threads. It is important to note that a queue only contains statements of a single type; making it possible to empirically obtain reliable average batch service times s_I and average batch sizes $\mathsf{E}[B]$ for given interval lengths I in advance. For notational purposes, we omit the reference to I in $\mathsf{E}[B]$.

Thus, given the empirically obtained average batch service times s_I and average batch sizes $\mathsf{E}[B]$ for given interval lengths I, we are particularly interested in minimizing the fraction of batches $p_B(I)\mathsf{E}[B]$ that are not immediately processed by a thread upon arrival, hence, reducing the system throughput. We derive the probability $p_B(I)$ similar to Erlang's loss formula [2] as

$$p_B(I) = \frac{\rho_I^N / N!}{\sum_{i=0}^{N} \rho_I^i / i!} \tag{1}$$

with $\rho_I = \lambda_I \cdot s_I$ depending on the interval length I. Note that $p_B(I)$ can be calculated efficiently in an iterative manner. The idea of maximizing the throughput is to find an I that minimizes $p_B(I)\mathsf{E}[B]$.

4.2 Extended Model

In the extended model, compared to the basic model we support multiple queues. The problem statement is that we want to distribute the N MSE-threads to M queues and approximately find an interval I_m for each queue $m \in M$ to maximize the overall throughput. We denote the number of assigned threads of queue m using N_m where $N = \sum_{m \in M} N_m$.

In this paper, we assume that the number of queues M and available MSE-threads N is fixed. Typically, N is given by the current DBMS configuration and M is derived from the

workload and represents the number of top-M statement types that are merged using a merging queue for each statement type.

To find the optimal configuration (i.e., N_m and I_m for each queue $m \in M$), we again execute calibration runs but this time for all M queues (and their statement types) and $1...N$ threads for each queue. It is important to note that each of the $m \in M$ queues again only contains statements of a single statement type, making it possible to empirically obtain reliable batch service times s_I and average batch sizes $\mathsf{E}[B]$ for given interval lengths I_m and a number of MSE-threads N_m in advance.

Using these empirically obtained statistics, we enumerate all possible distributions of N threads to M queues. For each of those distributions, we derive the optimal poll interval I_m for each of the queues $m \in M$ and compute the estimated throughput for that queue as discussed in the previous section. In order to pick the best overall configuration (i.e., N_m and I_m for each queue $m \in M$), we use the one which globally maximizes the sum of the estimated throughput over all queues.

5. IMPLEMENTATION

In order to demonstrate the feasibility of our sharing approach, we implemented our OLTPShare execution strategy within the core engine of the commercial SAP HANA database system [5]. In the following, we discuss the implementation details of the different phases of OLTPShare as discussed in Section 3.

5.1 Queuing and Merging

Clients of SAP HANA send a statement string via a socket connection to a dedicated SqlExecutor thread (SE) assigned to that connection. The SE-thread first receives the statement, compiles it and then computes a statement fingerprint from the statement's predicates and accessed tables to match it to a merge rule that could be applied to this statement. If the statement's fingerprint could be matched to a merge rule, the statement is added to the merge queue dedicated to the matching merge rule.

An incoming query is added to a merge queue using a work element that contains not only the query parameters but also the connection identifier (i.e., the client) which submitted the result as well as an expression required to post-process the result that stems from executing a merged query. The details of post-processing are explained below.

Each merge queue is polled by a pool of N_m MergeSqlExecutor threads (MSE) using the poll interval I_m as defined in Section 4; i.e., after a timespan of I_m one of the N_m MSE-threads polls from the queue m. The admission of an MSE-thread to a queue is controlled by the scheduler. Once an MSE-thread gets access to the queue, it polls all work elements from the queue that are present at the time of admission. During the merging, new work elements can be added by SE-threads (those will then be polled by the next MSE-thread).

All work elements polled by an MSE-thread are combined using the merge rule into a merged plan and executed by the MSE-thread. The details of the execution are discussed next.

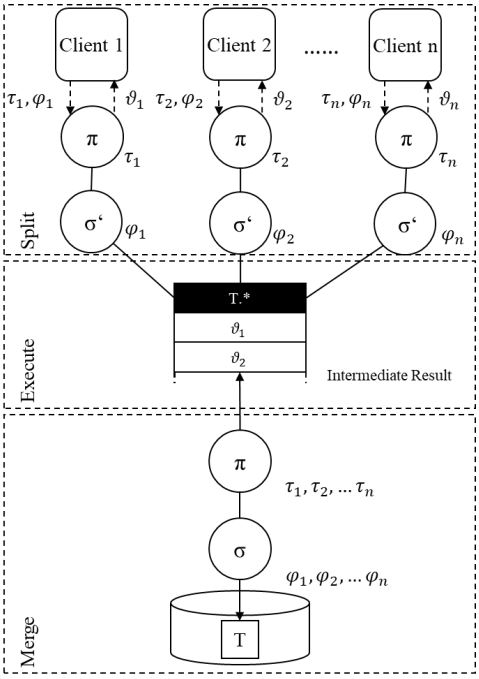


Figure 7: A Merged Query Plan.

5.2 Merged Execution

The basic idea for executing the merged plan is shown in Figure 7. Upon execution, the result of the merged plan is written into a temporary table holding the intermediate result. Once the intermediate result is created, we apply a post-filtering step for each client in order to filter out the relevant subset of the overall result.

The problem is that in general the result of typical merged queries is a superset of the union of all individual results, i.e., if ϑ_i was the result for Client$_i$, then

$$\bigcup_{i=1}^{n} \vartheta_i \subseteq \vartheta_{1 \cup \cdots \cup n}.$$

E.g., if we merged two queries with filters `f=1 and g=2` and `f=3 and g=4` into `f in (1, 3) and g in (2, 4)`, then the intermediate temporary table could also contain a row with `<f=1, g=4>` which was not requested by any of the two queries. Receiving more rows in our intermediate result could arguably hurt the performance, since the unnecessary result rows have to be materialized within a temporary table.

The above example could have also been re-written as `(f=1 and g=2) or (f=3 and g=4)` resulting in the exact union of the results of all merged queries. However, the evaluation of the expression during execution is more expensive than an IN-list which can be efficiently evaluated using a batch index lookup. Hence, tolerating some redundancy is often beneficial and providing a cost-based decision with respect to the alternatives is planned for future extensions of the framework. Furthermore, as we have seen within the workload analysis, many OLTP queries use a filter predicate on unique columns (very often primary keys) thus avoiding this effect. From our current experience, the problem of irrelevant result rows within

the intermediate result is either not existent or negligible in terms of performance issues.

In the post-filtering step, we have to split the result of the merged plan for the different clients. In order to split the intermediate result generated by the merged plan and identify the correct result rows, we iterate over all the rows of the temporary table holding the intermediate result set and evaluate the client's filter condition and projection list. Only positively evaluated rows and required columns are assigned to the result set of the individual client.

One could argue that iterating multiple times over the intermediate results for different clients is too much overhead. However, since in OLTP environments the intermediate results are rather small, the overhead due to the repeated iterations is rather negligible. Still, evaluating an expression during execution and additionally during post-processing is redundant work. We address this issue and identify a possible solution in the subsequent section.

5.3 Potential Optimizations

Evaluating the same filter expressions in the post-processing step again obviously introduces some redundancy. Filter expressions can become arbitrarily complex and their repeated evaluation adds additional overhead that could be avoided. In future, we thus plan to simplify this process such that the filter expression evaluation does not have to be additionally performed in the post processing step: the main idea is – similar to the approach taken in [7] – to augment the temporary table holding the intermediate result by an additional column holding a bitmap to identify the clients to which the particular row corresponds. This allows to scan the result set once and directly write each row to the respective output buffers.

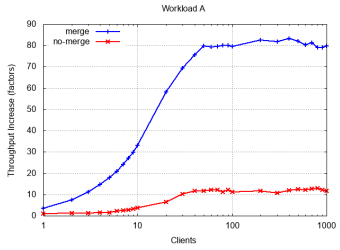
6. EXPERIMENTAL EVALUATION

In this section, we report the results of our experimental evaluation of the OLTPShare execution strategy in SAP HANA compared to the conventional execution without merging.

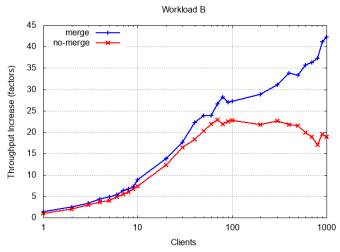
6.1 Workloads and Setup

As workloads, we use different OLTP benchmarks: first, we use the YCSB Benchmark [3] because it provides simple, yet representative queries simulating data access for typical web-based applications. Second, we use the TATP benchmark [17], a widely accepted OLTP benchmark.

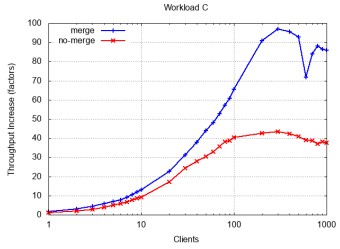
We execute those benchmarks on a server running SUSE Linux Enterprise Server 12 SP1 (kernel: 4.1.36-44-default) using 512 GB of main memory and four sockets with 10 cores, each. Our machine is equipped with Intel(R) Xeon(R) CPU E7-4870, which runs at a speed of 2.4 GHz and have a cache size of 30 720 kB. Since we aim for increasing the throughput in cases of over-utilization, we limit SAP HANA's core usage to 10 cores, i.e. one socket. In addition, we disable hyperthreading. For the experiment, we implemented the OLTPShare strategy in SAP HANA as discussed before. Using compile flags, we are able to turn on/off the sharing feature allowing us to compare OLTPShare against the commercially available version of SAP HANA without query sharing. All workloads run on tables, stored in the row store format; we



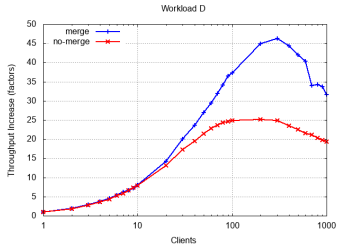
(a) Workload A



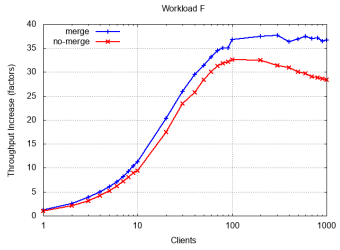
(b) Workload B



(c) Workload C



(d) Workload D



(e) Workload F

Figure 8: Throughput increase of YCSB.

Table 1: Workloads of the YCSB Benchmark.

Workload	READ	WRITE	INSERT	SCAN	RMW
A	50%	50%			
B	95%	5%			
C	100%				
D	95%		5%		
E			5%	95%	
F	50%				50%

show that our approach is also suitable for column store tables in Section 6.3.

6.2 Experiment 1: Throughput increase

In the following, we discuss the throughput increase of running the different workloads with an increasing number of clients. Our baseline is the throughput of running the workload with a single client without enabling our merging implementation (similar to the benchmark used in Section 1).

6.2.1 The YCSB Benchmark

The YCSB benchmark defines different workload mixes A-F as shown in Table 1. Examining these workloads in more detail, we noticed that running Workload E does not make sense in our case, because it consists of 5% insert queries and 95% scan queries, which require a top-k operator for which merging is not yet efficiently supported.

We thus executed YCSB workloads A, B, C, D, and F as defined in [3]. Each workload uses two distinct statement types, a read and an insert/update statement. For each of these workloads, we used only a single merge queue that collects the read statements of the respective workload as well as 5 MSE-threads and 10 SE-treads. The poll interval of the MSE-threads was set to 110 µs.

We plot our results of running the YCSB benchmark in Figure 8. The x-axis shows the increasing number of clients while the y-axis depicts the throughput increase in factors, where we divide the throughput in queries per second by the throughput with only one thread without using our sharing approach.

As we show in Figure 8a, workloads with a higher update rate benefit the most from our approach: since read queries are merged, the read-load on the table drops and more resources are left for executing updating statements.

As can be seen in Figure 8b, our approach scales well until the point where 1000 clients are connected to SAP HANA, which means, the system is 100× over-utilized. In contrast, the conventional SAP HANA scales until 100 clients and then slightly drops with respect to the overall throughput. Having the applications scenario in mind, this result is extremely relevant, because Workload B actually behaves like a real-world OLTP workload with a high ratio of read statements (Section 2).

With respect to sharing, we depict the results for the optimal workloads in Figure 8c. Notably, our approach has the sweet-spot with 200 clients, instead of the convenient SAP HANA implementation with a sweet-spot with 100 clients in all workloads. Workload C also has the best throughput in absolute numbers, because it consists only of read-queries thus avoiding any lock contention on the table.

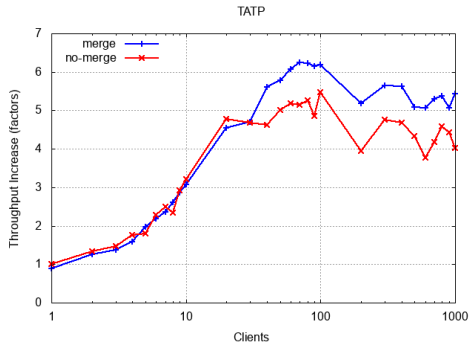


Figure 9: Results of TATP workload.

The results in Figure 8d show that inserting values in a workload does not have a negative impact on our merging abilities. It should be noted that the read-distribution for this workload is different, though: while Workloads A, B, C and F access keys with a zipfian distribution, Workload D is supposed to access the latest inserted keys. Due to beneficial cache effects, we see a higher throughput increase here, compared to Figure 8b, which also consists of 95% read queries.

The implication of Figure 8e, which plots the results for Workload F, is that it scales the least compared to the other workloads. However, since the ratio of this workload is 50% read and 50% read-modify-write, we actually have to deal with a workload of 75% read queries. In this regard, Workload F can be seen between Workload A and Workload B. The absolute throughput numbers underline this argument, providing throughput numbers between those of Figure 8a and Figure 8b. Hence, our approach does not scale worse for Workload F, but the conventional SAP HANA is keeping up due to the small number of writes.

6.2.2 The TATP Benchmark

For our last experiment, we run the TATP benchmark [17]. TATP is composed of seven distinct transactions with up to three queries, each. Those transactions access four different tables, stored in row-store format. We run the benchmark with 10.000 subscribers. The workload itself consists of 80% read statements and 20% update statements. Overall, there are nine distinct query strings within the workload, four of these are mergeable read-only statements. The other five statements are composed of three update, one insert and one delete statement. Due to the workload configuration, 70% of the workload is produced by two single-key lookup queries.

We configure OLTPShare with a Scheduler responsible for a pool of seven MSE-threads, checking the Merge Queue in an interval of 110 µs. In this experiment we use only one shared Merge Queue for all single-key lookup queries.

We show our results in Figure 9. As we see, our approach does not cause a throughput drop for an underload-scenario of less than 40 clients. From 40 clients onward, we observe an increase of the throughput of about 20% compared to conventional SAP HANA. The results are comparable to those of the YCSB Workload B in Figure 8b, in terms of having a large read-proportion. In contrast, this workload consists of multi-

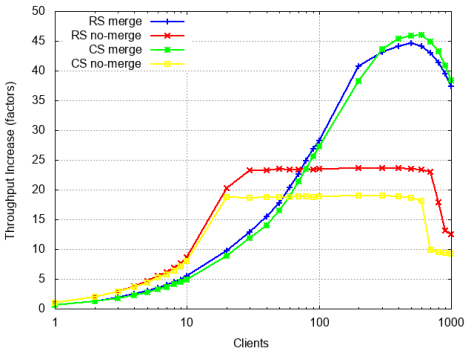


Figure 10: Throughput increase of row store and column store

ple distinct read-statements. Due to the fact that we are using only one queue in our current prototype implementation, the throughput increase is not as high as expected, since we only merge statements with the same string, i.e., statements with different strings are executed sequentially. Still, it should be noted that even with one queue for four different statement strings our approach produces a remarkable increase of the system's throughput, as Figure 9 shows.

6.3 Exp. 2: Row-Store vs. Column-Store

Previously discussed results use queries based on tables in the row store layout. In this section, we investigate, whether OLTPShare is applicable to other table types as well or not. To evaluate this, we run a single-query-workload. That workload selects two values from a 10 M table with two integer columns and an index on the primary key. We run this workload twice, once using a column store table and once using a row store table. Our configuration uses one Merge Queue[2] and one Scheduler thread for a pool of 5 MSE-threads that polls the Merge Queue in an interval of 110 µs. Each workload is executed twice, once with our approach and once without. We show the throughput increase of all four runs in Figure 10; the baseline is always the respective workload type (i.e. column store/row store) without sharing, executed with only one client.

As Figure 10 shows, the workloads for both table layouts do not only achieve a higher throughput when running with more than 80 clients, but the characteristics of their curves are also similar. On the other hand, we note that queries on column store tables benefit more from our approach than row store tables. We explain this by the advantage of SIMD-usage when using IN-lists on the column store, which is not available in the row store [21]. With this small synthetic experiment, we conclude that resource sharing in OLTP statements is not bound to a special table layout in SAP HANA, but a concept that can be generalized to also different storage layouts.

6.4 Exp. 3: Accuracy of Queuing Model

In Section 4, we discussed the queuing model of our approach. In this section, we analyze its accuracy. As we will show next, our theoretical model closely matches findings of our experimental evaluation. In order to show the accuracy of the queuing model, we consider a workload of read-only

[2]since we only have one distinct statement.

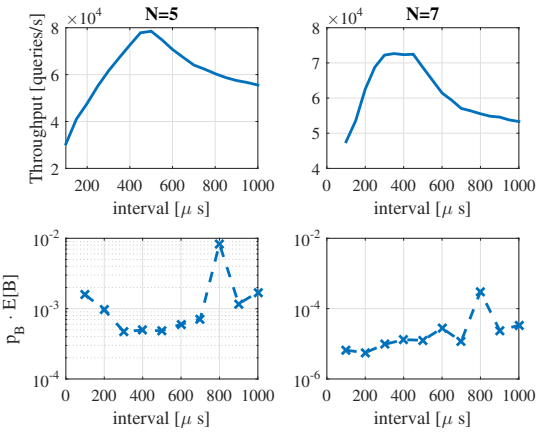


Figure 11: Tuning the interval I using the queuing model.

single-key lookup statements which is the same workload as depicted in Figure 1 arriving at one merge queue. The merge queue is serviced by $N \in \{5, 7\}$ MSE-threads. Furthermore, we vary the poll interval I in the range of 100 µs to 1 ms.

In order to see if our model predicts an interval which results in an approximately maximized throughput, we compare the measured throughput, i.e., given n queries per second, for varying I to the delayed batch fraction computed by our model. We want to see that the throughput is maximized at those poll intervals I which also minimize the delayed batch fraction as predicted by our queuing model in Section 4.

Figure 11 shows the result for a range of intervals I and two setups with $N \in \{5, 7\}$. We can see that where the delayed batch fraction $p_B(I)\mathsf{E}[B]$ is minimized the throughput is actually maximized. Given empirically measured average batch sizes $\mathsf{E}[B]$ and batch service times per thread s_I, Equation 1 can thus be used to determine the combination of the number of threads N and the interval length I which maximizes the system throughput. Note, that we observed an outlier in the calibration for $I = 800\mu s$. However, these outliers can be filtered out by smoothing, as described earlier.

7. RELATED WORK

Sharing resources in query execution has a long research tradition. In this chapter, we mention related work, categorized by work dealing with sharing resources in OLAP workloads, mixed workloads (i.e., OLAP and OLTP), pure OLTP workloads only and streaming workloads.

7.1 OLAP Workloads

To our knowledge, Multi Query Optimization (MQO) [16] has been the first work to deal with using shared resources in queries. The idea of MQO is to execute a subquery of an OLAP query only once, even if that subquery occurs more than once in its parent. Thereby, the result of that subquery is reused. The use case for this approach, however, is limited to sharing resources in a single query and only for queries which have multiple similar nested queries - a precondition we usually do not find in OLTP queries. Another well-known resource sharing work are Materialized Views [15]. Those

store frequently used intermediate results for reuse in different queries. Materialized Views make sense when storing read-only intermediate results. In OLTP scenarios, we also deal with write-queries; this requires to keep the Materialized Views up-to-date, which is very expensive. fAST [10] combines these approaches by merging similar update requests of Materialized Views, using the MQO strategy to keep the cost of up-to-date Materialzed Views low.

Other related work, which includes QPipe [8], CJOIN [1] and MQJoin [11] use pipelines for sharing resources. A comparison of QPipe and CJoin is given in [14]. QPipe is evaluating an operator-centric paradigm, where each operator is evaluating several queries. This paradigm is diametrically opposed to the query-centric paradigm, used in SAP HANA and was therefore not applicable in our approach. CJoin and MQJoin use pipelining first and foremost for sharing work in joins – a problem we are not targeting in our approach, since our evaluation of the queries showed that we do not have many joins in OLTP queries.

In summary, many of the proposed approaches offer good solutions for OLAP workloads but are not directly applicable for OLTP workloads.

7.2 Mixed Workloads

While the work presented above investigates the sharing of resources within OLAP queries, we will present related work that additionally considers OLTP queries, in this subsection. BatchDB [12] is such a system providing heavy throughput on mixed workloads. OLTP queries have to be executed in stored procedures and resource sharing is applied to OLAP queries, only. In addition, OLTP and OLAP queries are not executed concurrently but rotatorily.

The work of SharedDB [7] is to our knowledge the closest to ours. In SharedDB, incoming queries are compiled into one big plan. During execution of that plan, further incoming queries are queued, i.e., there is always just one query executed in that system. Like in BatchDB, this causes that the response time is bound to the execution of the slowest query. The presented work takes OLTP queries into consideration, but it still focusses on throughput of OLAP queries.

7.3 OLTP Workloads

Besides the many contributions of resource sharing within OLAP workloads with or without considering the concurrent execution of OLTP queries, some works investigate sharing resources in OLTP queries. Quro [22] is a C/C++ compiler extension that reorders queries on the client side to avoid lock contention.

In contrast, BOHM [4] and Calvin [18] propose to execute OLTP queries in batches and reorder the queries of a transaction on the server side. This requires the whole transaction to be known to the server - a precondition which is uncommon in most real-world systems. All these approaches are orthogonal and could additionally be applied in OLTPShare as well.

DORA [13] follows a data-centric paradigm; a thread is assigned to data, instead of a task. During task execution, a task enters several threads, thus avoiding lock contention. That approach offers the opportunity to share work between tasks, when they use data, handled by the same thread even with a mix of read and write tasks. However, this paradigm is diametrically opposed to the query-centric paradigm used in OLTPShare for SAP HANA and is therefore not applicable in our approach.

Zhou et al [24] investigate finding common subexpressions in queries (e.g. union statements) or sequences of queries (e.g. stored procedures). In addition to their work, we try to find common subexpressions of different queries of different transactions.

7.4 Streaming Workloads

Resource sharing is also evaluated in streaming queries such as State Slice [20]. In that work, the authors present an approach to efficiently split results when using joins on streams and thereby reuse parts of intermediate results. The motivation, here, is to reduce the memory and CPU consumption. In contrast, our main goal is to increase throughput for non-streaming queries. CPU utilization and memory consumption are currently not in our focus.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented an execution strategy called OLTPShare that implements a novel batching scheme for OLTP workloads. In order to analyze the sharing benefits, we integrated OLTPShare into the commercial database system SAP HANA and experimentally showed that it is capable of providing a significant throughput increase of up to $3\times$ under high-load scenarios compared to the conventional execution strategy without sharing.

In future, we will extend OLTPShare in different directions: First, we want to look into merging of update statements as well as merging of multi-statement transactions. For update statements, merging for transactions over hot items (e.g., a central counter) would be extremely beneficial since we then only require one write-lock for a batch of write-transaction which lowers contention significantly. Another future avenue is that we also want to add more physical execution operators that can efficiently support the execution of batched transactions instead of relying on the existing physical operators.

9. REFERENCES

[1] G. Candea, N. Polyzotis, and R. Vingralek. Predictable performance and high query concurrency for data analytics. *PVLDB*, 20(2):227–248, 2011.

[2] J. Cohen. *On Regenerative Processes in Queueing Theory*. Lecture notes in economics and mathematical systems. Springer-Verlag, 1976.

[3] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[4] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *PVLDB*, 8(11):1190–1201, 2015.

[5] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. Sap hana database: Data management

for modern business applications. *SIGMOD Rec.*, 40(4):45–51, Jan. 2012.

[6] P. M. Fernandez. Red brick warehouse: A read-mostly RDBMS for open SMP platforms. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994.*, page 492, 1994.

[7] G. Giannikis, G. Alonso, and D. Kossmann. Shareddb: Killing one thousand queries with one stone. *PVLDB*, 5(6):526–537, 2012.

[8] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 383–394, New York, NY, USA, 2005. ACM.

[9] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on read-optimized databases using multi-core cpus. *PVLDB*, 5(1):61–72, 2011.

[10] W. Lehner, R. Cochrane, H. Pirahesh, and M. Zaharioudakis. fast refresh using mass query optimization. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 391–398, 2001.

[11] D. Makreshanski, G. Giannikis, G. Alonso, and D. Kossmann. Mqjoin: Efficient shared execution of main-memory joins. *PVLDB*, 9(6):480–491, 2016.

[12] D. Makreshanski, J. Giceva, C. Barthels, and G. Alonso. Batchdb: Efficient isolated execution of hybrid oltp+olap workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 37–50, New York, NY, USA, 2017. ACM.

[13] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1-2):928–939, 2010.

[14] I. Psaroudakis, M. Athanassoulis, and A. Ailamaki. Sharing data and work across concurrent analytical queries. *PVLDB*, 6(9):637–648, 2013.

[15] N. Roussopoulos. View indexing in relational databases. *ACM Trans. Database Syst.*, 7(2):258–290, June 1982.

[16] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, Mar. 1988.

[17] M. M. Simo Neuvonen, Antoni Wolski and V. Raatikka. Telecommunication application transaction processing (TATP) benchmark description. Technical report, IBM Software Group Information Management, March 2009.

[18] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

[19] TPC-H. TPC BENCHMARK^{TM}C Standard Specification Revision 5.11. Technical report, Transaction Processing Performance Council (TPC), February 2010.

[20] S. Wang, E. A. Rundensteiner, S. Ganguly, and S. Bhatnagar. State-slice: New paradigm of multi-query optimization of window-based stream queries. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 619–630, 2006.

[21] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, 2009.

[22] C. Yan and A. Cheung. Leveraging lock contention to improve oltp application performance. *PVLDB*, 9(5):444–455, 2016.

[23] D. V. A. Zeyuan Shang and A. Pavlo. Carnegie Mellon Database Application Catalog (CMDBAC). http://cmdbac.cs.cmu.edu, 2018. [Online; accessed 01-March-2018].

[24] J. Zhou, P. Larson, J. C. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 533–544, 2007.