Sharing Opportunities for OLTP Workloads in Different Isolation Levels

Robin Rehrmann[*]
TU Dresden
Dresden, Germany
robin.rehrmann@mailbox.tu-dresden.de

Carsten Binnig
TU Darmstadt
Darmstadt, Germany
carsten.binnig@cs.tu-darmstadt.de

Alexander Böhm
SAP SE
Walldorf, Germany
alexander.boehm@sap.com

Kihong Kim
SAP Labs
Seoul, Korea
ki.kim@sap.com

Wolfgang Lehner
TU Dresden
Dresden, Germany
wolfgang.lehner@tu-dresden.de

ABSTRACT

OLTP applications are usually executed by a high number of clients in parallel and are typically faced with high throughput demand as well as a constraint latency requirement for individual statements. Interestingly, OLTP workloads are often read-heavy and comprise similar query patterns, which provides a potential to share work of statements belonging to different transactions. Consequently, OLAP techniques for sharing work have started to be applied also to OLTP workloads, lately.

In this paper, we present an approach for merging read statements within interactively submitted multi-statement transactions consisting of reads and writes. We first define a formal framework for merging transactions running under a given isolation level and provide insights into a prototypical implementation of merging within a commercial database system. In our experimental evaluation, we show that, depending on the isolation level, the load in the system and the read-share of the workload, an improvement of the transaction throughput by up to a factor of 2.5× is possible without compromising the transactional semantics.

PVLDB Reference Format:
R. Rehrmann et al.. Sharing Opportunities for OLTP Workloads in Different Isolation Levels. *PVLDB*, 13(10): 1696-1708, 2020.
DOI: https://doi.org/10.14778/3401960.3401967

1. INTRODUCTION

Motivation. OLTP applications often need to serve 1'000s of clients concurrently demanding high throughput of statements or transactions. A well known technique to avoid overload situations of the DBMSs, is to reduce the overall load by using techniques such as shared execution. For shared execution of read-only OLAP-style workloads, many techniques have been proposed in the past ranging from multi-query optimization [33] over shared operators [20, 27] to materialized views [31]. In OLAP workloads, the sharing potential arises from the fact that long-running and complex queries with multiple joins need to be executed and the data read usually remains static during the execution.

Different from OLAP workloads, however, OLTP workloads are mainly characterized by short-running statements, rendering the sharing potential not obvious. Nevertheless, its distinctive characteristics make it extremely interesting to investigate merging for OLTP scenarios. First of all, OLTP workloads in general consist of a few distinct statements, only differing in their concrete parameter values. For example, 89% of 7.383 open-source projects listed within the CMU Database Application Catalog (CMDBAC [38]) have only 10 or even less distinct statement strings [29]. Similar observations also hold for synthetic OLTP benchmarks such as the TPC-C [36] or TATP [35] benchmark, which consist only of 11 (or 5) distinct statement types.

Beyond benchmarks, Alibaba [14] or IBM [25] report workloads of the same characteristics. Furthermore, two third of a common SAP ERP workload consist of ten distinct prepared statements [24]. An analysis of OLTP workloads on Oracle [32] shows that read statements usually make up $70\% - 90\%$ within OLTP workloads, whereas 80% are reported for Microsoft SQL Server [8] and SAP [15, 30].

These characteristics open up the opportunity of merging statements within OLTP applications. Recently, efforts were thus made to apply such sharing techniques also to OLTP workloads. For example, SharedDB [12] as well as BatchDB [21] compile incoming read statements into a large plan, [6] batches full transactions to decrease aborts. All these approaches, however, assume that transactions are fully known to the database.

Contribution. In a previous paper [29], we have already shown that merging single-statement read-only transactions that are submitted interactively already results in significant throughput increase especially under high system load. In this paper, we significantly extend this idea and tackle the

[*]This work was done while at SAP SE, Germany

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 13, No. 10
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3401960.3401967

challenge to merge multi-statement read/write transactions where operations are submitted by clients interactively under different isolation levels. The main idea in this paper is to isolate read operations from read/write transactions and merge them into so-called *Merged Read*s if this satisfies the desired isolation level.

One could now ask, "why not also merge writes as well?". Firstly, merging just reads allows us to use the existing mechanisms of a DBMS to handle not only write-write conflicts but also the functionality for rollback and recovery without complicated extensions of the affected core database components. Secondly, as mentioned before, many OLTP applications of our customers are read-heavy at the beginning to retrieve information from the database before writes are executed, which supports our sharing strategy.

For example, in a webshop scenario, customers may first look at the details of a product, place an order for that product and then show the order status. In such a pattern, the read retrieving the product info could be executed in a shared manner, while the write placing the order and the read showing the order status would be executed in isolation. Such a "look-to-book" ratio is usually 1000 : 1 [23]. In consequence, it makes sense to concentrate on merging reads rather than writes.

Finally, as mentioned before, the target of our approach is to improve performance under peak workloads. For those scenarios, customers often implement their workload as stored procedures to optimize performance by reducing sending requests over the network from clients for multi-statement transactions. Hence, in addition to support merging for multi-statement read/write transactions where operations are submitted by clients interactively, we added support for merging when multi-statement read/write transactions are executed as stored procedures.

In summary, we provide the following contributions:

- First, we provide a formal framework to analyze the correctness of shared execution strategies under different isolation levels implemented in *multi-version concurrency control* (MVCC), which is a common concurrency control method in many database engines today.
- Second, we outline the implementation of our proposed sharing strategy within a research prototype based on SAP HANA [9], a relational DBMS which provides an MVCC storage scheme and two different isolation levels (*Read Committed* and *Snapshot Isolation*).
- Third, we share our insights of the results derived from our experiments for the TATP as well as a SAP Hybris benchmark. We show that depending on the isolation level and the overall system load, our approach may improve throughput by up to a factor of 2.5.

Outline. The remainder of this paper is structured as follows: Section 2 provides an overview of the core problems and conceptual ideas of the proposed sharing strategy. This will be followed by an in-depth discussion of the theory of merging general OLTP workloads (Section 3). Thereafter, we discuss restrictions of different isolation levels on the merging strategies (Section 4). Section 5 introduces the core components of our implementation as well as some design decisions. Section 6 presents the results of our evaluation. Section 7 discusses related work and Section 8 concludes the paper.

2. OVERVIEW

In this section, we first provide an example where a naïve merging approach fails to provide *Snapshot Isolation*. Afterwards, we give a high-level introduction to our idea of statement merging and show how it would resolve the problems of naïve merging on the same example.

2.1 Why Naïve Merging Fails!

Snapshot Isolation protects reads from other users' writes. For example, consider Alice and Bob with separated bank accounts who agreed on always having more than 300€ together on both bank accounts. Currently, both bank accounts are filled with 400€, each. In our example, schematically given in Figure 1, Alice and Bob at the same time decide to withdraw 100€ and 200€, respectively, from their own bank account, without telling each other.

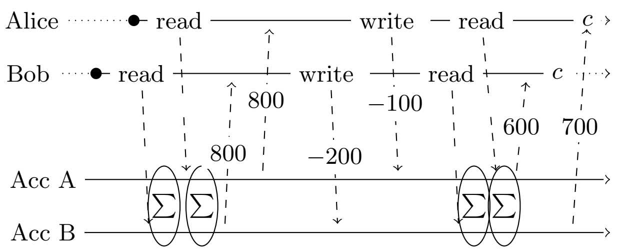


Figure 1: Example without merging.

As illustrated in Figure 1, both first check the sum of their bank account, which is 800€, withdraw the amount of money and check again, verifying their withdrawal did not violate their agreement. Since their read is protected from the other writes, Alice sees a sum of 700€ and Bob of 600€. As both receive the expected result, they commit. The final sum of their bank accounts is now 500€, which is correct with regard to (1) the chosen isolation level *Snapshot Isolation* as well as (2) their agreement to always have more than a sum of 300€ on their accounts.

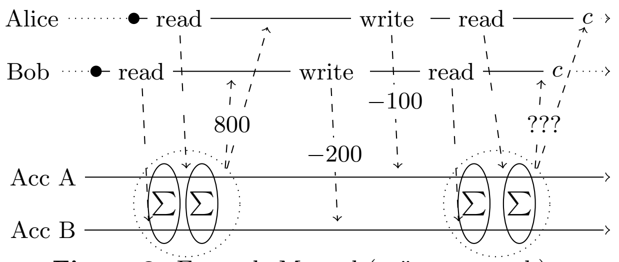


Figure 2: Example Merged (naïve approach).

Figure 2 shows how naïve merging applies to the example. We note that the calculation of the first sum may certainly be merged, since both read the same data and receive the same result. In this case it is also obvious, why it is intuitive to merge, since it is unnecessary to calculate the same result twice. After both updated their bank account within the database, they calculate the sum again. If we merge those read operations as well, i.e., calculating the sum once, and distributing the result, we have three options: (1) returning 500€, meaning that both see each other's write, thus violating the isolation properties of the database (2) returning 600€ or 700€, thus making either Alice or Bob believe, her or his write failed (and maybe trigger another withdrawal) or (3) returning 800€, so that none of the two can read their own write. Certainly, none of the above outcomes of merging are in line with the users' expectations.

In the next section, we present our approach to provide both (1) preserving the notion of the isolation level and (2) provide *read-your-own-writes* guarantees.

2.2 Our Approach

As several read statements enter the system, we merge them into a new read statement, which we refer to as a *Merged Read*. For merging transactions in this paper, we assume that the underlying DBMS supports MVCC, which allows us on a per-transaction level to read a certain snapshot. This is true for most commercial databases today such as Oracle but also SAP HANA where we built our prototype on.

For merging, we execute a *Merged Read* in the context of another transaction, which is transparent to the user and has a snapshot that is compatible with the individual reads. The main problem we address is handling the isolation properties between all operations of a transaction (also those executed as a *Merged Read*) by the underlying DBMS. For achieving this, we only need to provide the correct snapshot for all reads merged into a *Merged Read* and the other operation the client's transaction submits to the DBMS that are not merged.

To guarantee the isolation properties among all statements merged within the same *Merged Read*, we only merge those read statements that have the same view on the data (i.e., those which see the same snapshot). To additionally provide *read-your-own-writes*, we track the write set of a transaction and bypass merging for reads accessing those not yet visible writes within the transaction's context, i.e., we do not merge them into a *Merged Read*.

We show the effect of our approach in the example in Figure 3. Referring to our example from the beginning, Alice and Bob open a transaction and submit their read statement, calculating the sum over their bank accounts to the system. Since both have the same view on the data (i.e., there are no uncommitted changes in the system), we merge their statement into a *Merged Read*, which calculates the sum over both bank accounts. We execute that *Merged Read* in the context of an internal transaction, receive the result and return it to both transactions. Next, we track the writes to different accounts in the write-set of their transactions. As both now again submit their sum-calculation to the system, we note that these operations access data that has been altered by their own transactions. To provide *read-your-own-writes*, we execute these reads as originally submitted, without merging. Hence, Alice and Bob can see their own withdrawal without seeing the other withdrawals and are able to commit.

In an equivalent lock-based implementation, the *Merged Read* needs to read-lock both accounts and transfer these locks to Alice and Bob, once they submit their write requests. Next, these locks are promoted to withdraw money from the respective bank account. As our example applies to *Snapshot Isolation*, which is not supported in lock-based, the subsequent read-accesses by both parties will deadlock and fail. Nevertheless, our approach can also be implemented lock-based if transferring locks from one transaction to another is supported by the underlying DBMS. As lock-based is not supported by HANA, where we built our prototype, we focus on the MVCC implementation in the rest of our paper.

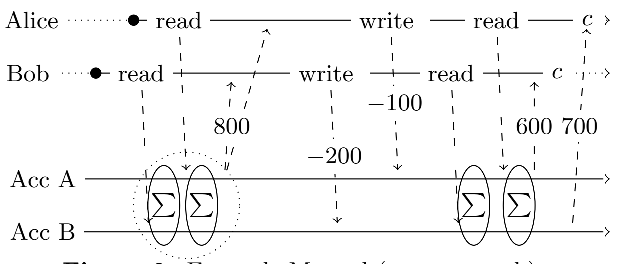


Figure 3: Example Merged (our approach).

In this paper, the main focus is on a setting where clients (i.e., Alice and Bob) submit operations interactively one-by-one since this setting is used in many real-world applications as we see in our customer workloads. However, in general our approach works, regardless whether Alice and Bob submit their requests in an interactive manner (i.e., operations are submitted one by one) or as stored procedure (i.e., the whole sequence of operations is submitted at once). In our evaluation, we will evaluate the benefits of merging under both settings.

3. FORMALIZING OUR APPROACH

To show our understanding of merging and how we decide which statements to merge, we formally introduce our definition of a *Merged Read* as well as our merge-decision algorithm in this section. For a better understanding of our definition and algorithm, we first provide a brief recapitulation of transactional execution under MVCC.

3.1 Transaction Theory

Transaction management has a long history in the database area. In this work, we build mainly on the transaction definition, provided by Bernstein et al. already published in 1983 [2].

In databases, clients submit two types of operations on data items: read and write operations. We denote a read on data item x submitted by transaction T_i as $r_i(x)$ and a write to data item y as $w_i(y)$. For simplicity but without loosing generality, we assume that a read operation always refers to an existing item. A write operation may (1) *insert* a non-existing data item into the database, (2) *update* an already existing data item, or (3) *delete* a data item from the database.

A transaction is an ordered sequence of those operations that are all executed using ACID guarantees and terminated by either a commit or an abort. We denote a commit of T_i as c_i and its abort as a_i. For the order of the sequence, we use the *happens-before* notation by [2], denoted as $<$: If T_i submits a (read or write) operation o_i before it submits another (read or write) operation p_i, then $o_i < p_i$. A data item, which is written by T_i, is in a state that [2] refer to as "uncertified". A c_i command certifies all uncertified data items created by T_i. An a_i resets all uncertified data items to their previous state.

Finally, a history H determines the order in which the DBMS executes the operations of multiple concurrent transactions. More precisely, according to [2], a history is defined as partially ordered set $H = (\sum, <)$ with the following properties.

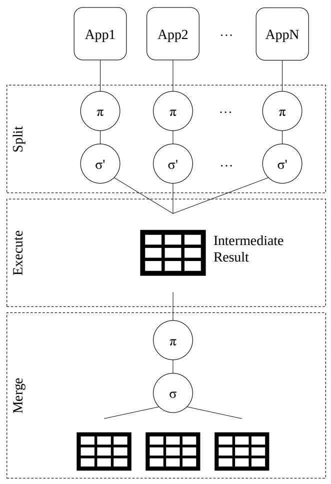


Figure 4: Execution of a *Merged Read*.

1. $\sum = \cup_{i=0}^{n} \sum_i$, i.e., all operations of H's transactions are executed and no further operations are added.
2. $< \supseteq \cup_{i=0}^{n} <_i$, i.e., the ordering of each transaction's operations remains stable in the global history.
3. Operation pairs that are executed on the same data item, with at least one operation being a write, are not executed in parallel, i.e., they are $<$ related.

MVCC Background. Most of modern relational database engines implement MVCC. In MVCC, every data item is enhanced with a *from-to*-timestamp that defines the visibility of its version. Timestamps are typically implemented by incrementing a global atomic counter (global-TID) [18]. A submitted operation determines the version it accesses by its own transaction ID (TID). The TID is typically assigned to a transaction, when it enters the system by incrementing global-TID and uses that as its own TID.

An operation with an attached TID is allowed to read a version of a data item, if the *from*-timestamp of the data item is smaller than the TID of the operation and the *to*-timestamp is larger. The latter may be the case, if (1) a newer version was written but not yet *certified* to the operation or (2) this is the newest version and the *to*-timestamp is ∞.

On commit, a transaction again increments the global-TID and uses this counter as commit ID (CID) to create a new version for all data items it updated / inserted: Thus, a write-operation overwrites the *to*-timestamp (currently ∞) of the most recent version of the data item with its CID and creates a new version of the data item with the *from*-timestamp set to the CID and the *to*-timestamp set to ∞, i.e., the transaction *installs* a new snapshot. During commit, additionally read- and write-conflicts must be checked, depending on the chosen isolation level.

To reason about multi-versioned histories, [2] introduce an h-function, which maps every read or write operation of a transaction T_i to the version being read or installed as follows:

1. Every submitted operation of a transaction T_i is transformed into a multi-versioned operation; i.e., $h(r_i(x)) = v_r$ and $h(w_i(x)) = v_w$ whereas v_r and v_w can be seen as the *from*-timestamp of the version read or being installed.
2. If a write is executed before a read in T_i, i.e. $w_i(x) < r_i(x)$, then the read returns the version of the write, i.e., $h(r_i(x)) = h(w_i(x))$.
3. Otherwise, $h(r_i(x)) = h(w_j(x))$, i.e. T_i returns the version of a transaction T_j that committed before T_i started[1].

3.2 Merging under MVCC

In this section, we provide our formal framework to analyze the correctness of shared execution strategies under different isolation levels: We first notate our definition of a *Merged Read* in the context of MVCC databases. Next, we present our algorithm that decides whether two read statements can be merged.

A *Merged Read* is a composition of several read statements. It is executed in the context of an internal transaction and runs with respect to a fixed and valid snapshot. That snapshot must contain the data-version requested by the individual read statements.

DEFINITION 3.1. *Merged Read*
A Merged Read $r_M(x, y, \ldots, z)$ *is a composition of read operations* $r_i(x), r_j(y), \ldots, r_n(z)$ *of transactions* T_i, T_j, $\ldots$, T_n, *such that* $h(r_M(x)) = h(r_i(x))$, $h(r_M(y)) = h(r_j(y))$, $\ldots$, $h(r_M(z)) = h(r_n(z))$, *running under a single arbitrary but fixed, existing and explicitly known TID.*

Figure 4 conceptually shows the merging procedure: The DBMS compiles the application's read statement into a plan with operators for table-access, filters (σ), projections (π), etc. We merge the filters and projection lists of statements against the same table access and write the intermediate result into an internal temporary table. The original plans may then fetch directly from that much smaller temporary table and send the results back to the client. Therefore, we do not only share the table access, such as ranged index lookup or a table scan, but also occupy only a single thread with the execution of n plans at once, leaving other threads for the execution of further incoming requests. Thus, the system may answer more requests, which is especially relevant in overload scenarios, where free threads become rare.

Algorithm 1 Check if read r_i of T_i to x and read r_j of T_j to y can be merged. $x = y$ maybe possible.

```
1: function IsMergeable($r_i(x)$, $r_j(y)$)
2:     if $h(r_i(x)) = h(r_j(x))$ then
3:         return true
4:     else if $h(r_i(y)) = h(r_j(y))$ then
5:         return true
6:     end if
7:     return false
8: end function
```

Merging reads requires the same view on the data for different statements. In Algorithm 1, we show how we decide if two read statements are mergeable. The function receives two read statements, submitted by two transactions. The read statements may or may not access the same data item.

According to Definition 3.1, a *Merged Read* is executed under a single TID. Hence, we need to check if there exists a single TID that returns the correct version of x and y to T_i

[1]In *Serializable*, other isolation levels may broaden the point of time when T_j committed

and T_j, respectively. Line 2 checks if an access to x with the TID of T_j returns the same version that T_i expects. If this is the case, we merge $r_i(x)$ and $r_j(y)$ into $r_M(x, y)$ which will then be executed within an internal transaction having the snapshot of T_j. Otherwise, we check if an access to y with the TID of T_i returns the same version that T_j expects (line 4) and if so, we can execute the resulting *Merged Read* with the snapshot of T_i. If neither is the case, $r_i(x)$ and $r_j(y)$ are not mergeable. In consequence, we return `false` in line 7.

Algorithm 2 Check if read r_i of T_i to x can be merged into r_M.

```
1: function IsMergeable($r_M(\dots)$, $r_i(x)$)
2:     return $(h(r_M(x)) = h(r_i(x)))$
3: end function
```

Once we composed two reads into a *Merged Read*, we use Algorithm 2 to check for all other reads in the system whether they can be compiled into this *Merged Read*, as well. All we have to do, is to check if the *Merged Read* with its given TID would return the same version of a data item as the original read. If this is the case, we compose that read into the *Merged Read*. For the remaining reads, we continue with Algorithm 1 followed by Algorithm 2, once we found a match, until no further match is found or all read statements were merged. Finally, the resulting *Merged Read*s are executed.

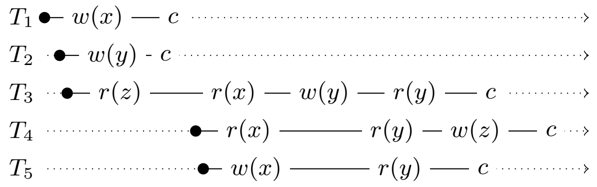
In the next section, we show when the expression of line 2 and line 4 in Algorithm 1 is evaluated to *true* for different isolation levels and how this affects the mergeability of workloads running under these isolation levels.

4. DIFFERENT ISOLATION LEVELS

As we stated previously, two read statements $r_i(x)$ and $r_j(y)$ can be merged, if $h(r_i(x)) = h(r_j(x))$ or $h(r_i(y)) = h(r_j(y))$. Because the outcome of these expressions for the same i, j, x and y depends on the isolation level, this section discusses the conditions under which these expressions evaluate to `true`.

Running Example. In order to discuss merging in the presence of different isolation levels, we use the running example in Figure 5 to intuitively explain the consequences of isolation levels on the merging potential. In the example there are five transactions, where each transaction operates in an interactive mode, i.e., each transaction submits only one read or write operation at a time and waits for the result, before submitting the next. Without loss of generality, we assume all transactions run in the same MVCC isolation level. We mark the start of the transaction with a dot. T_1, T_2 and T_3 start first and submit a write to x and y and a read to z, respectively. Next, T_2 and T_1 commit their writes, installing a new snapshot. Afterwards, T_4 and T_5 start. T_3 and T_4 submit a read to x at the same time, while T_5 overwrites that value. The subsequent statement is submitted by T_3 and overwrites y, followed by a read to y of all three remaining transactions. Finally, T_3 and T_5 commit, while T_4 submits a last write to z and commits as well.

As all transactions run in interactive mode, i.e., are blocking on every operation until they receive a result from the database, the DBMS is not able to reorder or antedate statements. For example, the DBMS cannot execute $r_3(x)$ before $r_3(z)$, since $r_3(x)$ is only submitted by T_3 once the result of $r_3(z)$ returns. Furthermore, the DBMS cannot simply execute $r_4(x)$ before T_2 commits, as T_4 starts after T_2 commits. In consequence, not knowing the overall history and having limited control over when an operation is going to be executed (we may always postpone the execution of a statement, though) further reduces our ability to merge.



T_1 ● $w(x)$ — c
T_2 ● $w(y)$ - c
T_3 ● $r(z)$ — $r(x)$ — $w(y)$ — $r(y)$ — c
T_4 ● $r(x)$ — $r(y)$ — $w(z)$ — c
T_5 ● $w(x)$ — $r(y)$ — c

Figure 5: Example of five interactively submitted transactions.

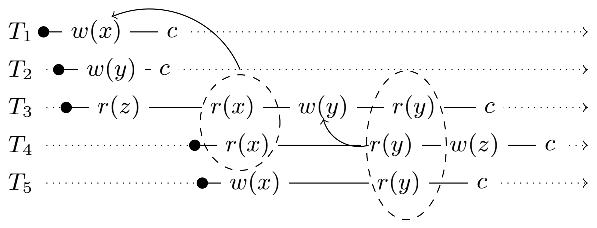
In the following, we describe how different isolation levels affect merging in this example.

4.1 Read Uncommitted

In *Read Uncommitted*, every transaction is allowed to read the newest version of every data item, even if it is not yet committed. Simply speaking, because

$$h(r_i(x)) = h(r_j(x)), \forall T_i, T_j \in H, \qquad (1)$$

Algorithm 1 always evaluates to `true`, thus we can merge all read statements in *Read Uncommitted*.



T_1 ● $w(x)$ — c
T_2 ● $w(y)$ - c
T_3 ● $r(z)$ — $r(x)$ — $w(y)$ — $r(y)$ — c
T_4 ● $r(x)$ — $r(y)$ — $w(z)$ — c
T_5 ● $w(x)$ — $r(y)$ — c

We circle the statements that we could merge from our example, above. Obviously, we can merge all read statements that arrive at a time. However, we cannot merge $r_3(z)$ with any other read statement, because there is no other read statement in the system at that time.

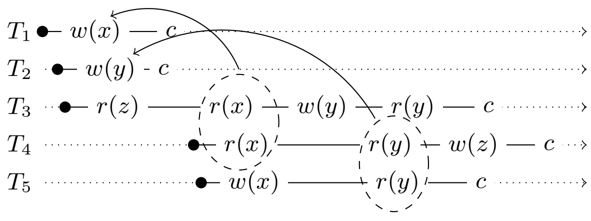
Conclusion. We conclude that *Read Uncommitted* does not restrict our mergeabilites at all, since we can merge all reads.

4.2 Read Committed

As our approach relies on the MVCC method, we assume the Oracle implementation of *Read Committed* [16], instead of the ANSI-*Read Committed* [1], which relies on locking. In *Read Committed*, all transactions have the same view on committed data items. Hence, we can merge all reads that do not refer to data currently residing locally within a transaction's write-set. More formally:

$$\begin{gathered} \forall\, T_i, T_j \in H, i \neq j \\ h(r_i(x)) = h(r_j(x)) \\ \Longleftrightarrow \\ h(r_i(x)) \neq h(w_i(x)) \wedge h(r_j(x)) \neq h(w_j(x)) \end{gathered} \qquad (2)$$

We circle the read statements of our example that can be merged when running under *Read Committed*, below. In comparison to *Read Uncommitted*, we merge one read less in this example, namely $r_3(y)$, which refers to a previous $w_3(y)$.



T_1 • $w(x)$ — c
T_2 • $w(y)$ - c
T_3 • $r(z)$ — $r(x)$ — $w(y)$ — $r(y)$ — c
T_4 • $r(x)$ — $r(y)$ — $w(z)$ — c
T_5 • $w(x)$ — $r(y)$ — c

Conclusion. In consequence, when incoming transactions are requesting *Read Committed* isolation, we can only merge reads that access committed data. For *Read Committed*, we cannot observe further restrictions.

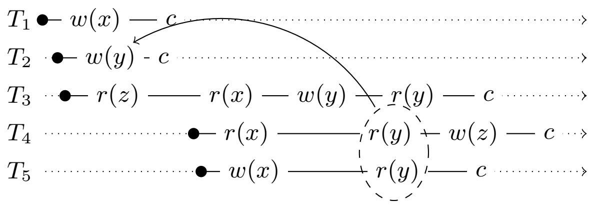
4.3 Snapshot Isolation

Snapshot Isolation forbids the anomalies *Inconsistent Read* in addition to *Lost Update* by providing each transaction a fixed snapshot, valid at transaction start [1]. That snapshot does not change, except for the transaction's own writes.

This implies the opportunity of merging two read operations if there exists a snapshot that holds both accessed data items in the requested version. More formally, w.l.o.g.:

$$\begin{gathered}\forall\, T_i, T_j, T_k, T_l \in H, i \neq j \neq k \neq l \\ h(r_i(x)) \neq h(r_j(x)) \\ \Longleftrightarrow \\ h(r_i(x)) = h(w_k(x)) < c_k < h(r_j(x)) = h(w_l(x)) < c_l\end{gathered} \tag{3}$$

We circle the sharing potential in our example under *Snapshot Isolation* below:



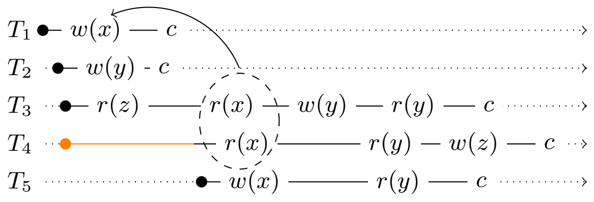
T_1 • $w(x)$ — c
T_2 • $w(y)$ - c
T_3 • $r(z)$ — $r(x)$ — $w(y)$ — $r(y)$ — c
T_4 • $r(x)$ — $r(y)$ — $w(z)$ — c
T_5 • $w(x)$ — $r(y)$ — c

As $r_3(x)$ and $r_4(x)$ now refer to different versions of x (namely $h(r_3(x)) = h(w_0(x))$ and $h(r_4(x)) = h(w_1(x))$, respectively), we cannot merge these two read operations, contrary to *Read Committed*. However, as $r_4(y)$ and $r_5(y)$ refer to the same version of y, created by T_2, we can still merge those. In consequence, as *Snapshot Isolation* increases the number of snapshots alive in the system at the same time in comparison to *Read Committed*, the merging abilities of reads operating on these snapshots further decreases.

Note, if the transactions were submitted as stored procedures and $r_3(z)$ and $r_3(x)$ were independent, we could reorder the execution of $r_3(z)$ and $r_3(x)$, thus merge $r_3(z)$ and $r_4(x)$ into $r_M(x, z)$, which we could execute with TID 4 (because $h(r_4(z)) = h(r_3(z)) = h(w_0(z))$).

Generalized Snapshot Isolation. As the number of different snapshots limits our merging abilities for *Snapshot Isolation*, we may reduce the number of snapshots by using *Generalized Snapshot Isolation* (GSI). With GSI, the database provides a view to the client that is consistent but may be slightly outdated as discussed in [7].

Assuming our example running under GSI, we give a possible outcome of the merging potential below. Alternatively, the one given for *Snapshot Isolation* also applies to GSI.



T_1 • $w(x)$ — c
T_2 • $w(y)$ - c
T_3 • $r(z)$ — $r(x)$ — $w(y)$ — $r(y)$ — c
T_4 • $r(x)$ — $r(y)$ — $w(z)$ — c
T_5 • $w(x)$ — $r(y)$ — c

As T_4 starts its transaction and submits $r_4(x)$, we find a possible merging potential with $r_3(x)$, submitted at the same time. However, as discussed earlier, both reads access two different snapshots. Since we run under GSI, we may choose our snapshot at transaction start, though. To share both read operations, we reset the snapshot of T_4 to the one of T_3 logically predating the start of T_4. In consequence, we can now merge $r_3(x)$ and $r_4(x)$ into $r_M(x)$, which is executed with the snapshot of T_3.

Later on, as $r_3(y)$, $r_4(y)$ and $r_5(y)$ are submitted, we cannot merge any of the operations, because $w_1(x) < c_2 < c_1 < w_5(x) < r_5(y)$, hence T_5 depends on the write of T_1 and at commit time of T_1 the snapshot of T_2 is already installed. Logically moving the transaction start of T_5 prior to the commit of T_2 to fulfill $h(r_4(y)) = h(r_5(y))$ would therefore result in an abort of T_5. However, if we delayed the execution of c_2 so that $c_1 < c_2$, we could logically move the start of T_5 between T_1's and T_2's commits so that neither T_4 nor T_5 see the result of $w_2(y)$ and in return get the same merging potential as with *Read Committed*.

Conclusion. We conclude that *Snapshot Isolation* limits our merging abilities further, as the number of snapshots, reads operate on, increases. An optimization regarding the merge options is to fall back to a slightly weaker isolation level, namely *Generalized Snapshot Isolation*, which lets the DBMS choose on which fixed snapshot an incoming transaction operates on. This has the potential to decrease the number of snapshots alive in the system and thus to increase our merging potential.

4.4 Serializable

ACID properties guarantee that a transaction runs isolated, i.e., as if it was alone in the system. This forbids any kind of anomaly, such as lost writes or write skews. *Serializable* is the only isolation level free of anomalies, providing true isolation. To prevent such anomalies, the DBMS needs to track all dependencies between transactions, such as *write-write*, *read-write*, *write-read*, and *read-read* and build a dependency graph, where a cyclic dependency between transactions marks a non-serialized execution.

Isolation levels considered so far, only require the write set of a transaction to check for a valid history, if the read was executed against the correct snapshot. Thus, we only used the h function to decide, whether a transaction's read operation can be merged (cf. Algorithm 1). However, if we merge two submitted reads into a new read operation, we basically hide the original reads from the read sets of their transactions. Thus, the DBMS is not able to decide whether the resulting history was serialized or not, introducing anomalies listed by Fekete et al. in [10]. As our running example from Figure 5 is a serialized history, we make our point with a different, smaller example, given below.

$$\begin{tikzcd} T_1 \bullet \arrow[r, dash] & r(x) \arrow[r, dash] & w(y) \arrow[r, dash] & c \arrow[r, dotted] & {} \\ T_2 \bullet \arrow[r, dash] & r(y) \arrow[r, dash] & w(x) \arrow[r, dash] & c \arrow[r, dotted] & {} \end{tikzcd}$$

The dependency graph of these two transactions is as follows:

$$\begin{tikzcd} T_1 \arrow[r, bend left, "rw"] & T_2 \arrow[l, bend left, "rw"] \end{tikzcd}$$

The execution of T_1 and T_2 is not serialized, as we note from the circle within the dependency graph. According to Algorithm 1 however, we can merge the two read statements into a new read operation and execute this within the context of an internal transaction T_M operating on the snapshot of any of the two transactions. Thus, we create a new transaction within the following dependency graph.

$$\begin{tikzcd} T_1 & & T_2 \\ & T_M \arrow[ul, bend left, "rw"] \arrow[ur, bend right, "rw"'] & \end{tikzcd}$$

As T_1 and T_2 now commit, the DBMS is not able to detect the original circular dependency among these transactions, since it is hidden within the *Merged Read*. Consequently, we need to alter Algorithm 1 for isolation level *Serializable*:

Algorithm 3 Check if read r_i of T_i to x and read r_j of T_j to y can be merged. $x = y$ is possible. Transfer read-dependency to DBMS is isolation level is *Serializable*.

```
1: function IsMergeable_Serializable($r_i(x)$, $r_j(y)$)
2:     if $h(r_i(x)) \neq h(r_j(x))$ & $h(r_i(y)) \neq h(r_j(y))$ then
3:         return false
4:     end if
5:     $DBMS \leftarrow$ add $x$ to read-set of $T_i$
6:     $DBMS \leftarrow$ add $y$ to read-set of $T_j$
7:     return true
8: end function
```

Algorithm 3 transfers the reads of T_i and T_j to the database (lines 5 and 6). Thus, the database can internally build a correct dependency-graph and detect occurring anomalies. As T_1 or T_2 finish in the example above, the database can abort the transaction due to the detected write-skew.

Conclusion. To fully support isolation level *Serializable*, we merge read statements the same way as under *Snapshot Isolation*, but have to propagate reads of statements merged to the collision detector of the DBMS.

4.5 Discussion

In summary, with increasing isolation level the mergeability of an arbitrary workload decreases.While we can merge all read operations in *Read Uncommitted*, *Read Committed* limits the read operations it can merge to already committed data accesses. As *Snapshot Isolation* introduces more snapshots alive in the system, the mergeability is even more decreased; we may adjust this using *Generalized Snapshot Isolation*. In *Serializable*, we merge as in *Snapshot Isolation*, but in addition we propagate the reads that were merged to the DBMS, not further limiting mergeability.

5. IMPLEMENTATION

This section outlines the implementation of our merging approach for isolation levels *Read Committed* and *Snapshot Isolation*. We start with an overview of our system with focus on its integration into and interaction with the underlying DBMS' core components and continue with our design decisions regarding the implementation of the two isolation levels.

5.1 System Design

We built our "Merger Component" as a research prototype based on SAP HANA. Our aim was to leverage the database's functionality where possible to decrease implementation overhead in both, Merger and database.

Figure 6 presents our system layout. As a new statement enters the system, we first check, whether the statement can be merged. This decision is supported by the transaction-local *write set*, we track. If we cannot merge the incoming statement, we forward it to the database execution engine. For write statements, we update the transaction's *write set*. Otherwise, we push the statement into an internal queue, which we refer to as *Merge Queue*. The *Merge Queue* contains several buckets, one bucket for each mergeable statement (details in Section 5.2). Inside the *Merger*, we keep a pool of threads, so-called *Merger Threads*, that check the buckets of the *Merge Queue* in regular intervals. If it finds statements inside of a bucket, it pops and merges these statements into a new *Merged Read* and forwards it to the database's execution engine. The execution engine compiles the *Merged Read* into a plan similar to Figure 4 and executes that plan retrieving an intermediate result. Within a post-process, the *Merger Thread* splits the intermediate result produced by the *Merged Read* to return the appropriate result to each client. We find that the implementation overhead of this architecture is rather low, as all the ACID properties, data management and execution of all queries are still handled by the database's execution engine. All we need to do is implementing the functionality of our *Merger Threads*, the *Merge Queue*, the *write set*-management as well as the merge decision.

Interactive vs. Stored-Procedures:. As mentioned before, the design presented in this paper enables that clients submit their operations of multi-statement transactions one-by-one and the merger threads analyze which of the submitted operations can be merged. In addition to the interactive mode, multi-statement transactions can also be implemented as stored procedures to avoid the high overhead of the network protocol between the clients and the DBMS. For supporting an execution of transactions as stored, we use the same architecture as shown in Figure 6. The only difference is that clients call a stored procedure inside the DBMS; i.e., all operations of a transaction can be submitted to the merger without expensive network roundtrips.

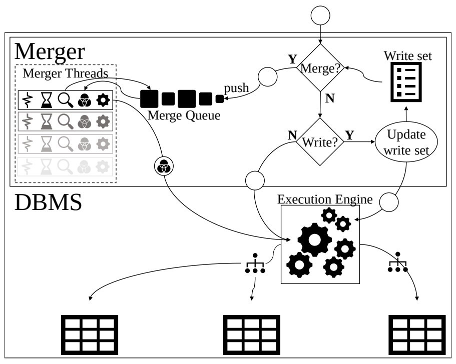


Figure 6: Our system layout.

5.2 Implementing Isolation Levels

As already explained in Section 3.2, *Merged Read*s are executed in the context of an internal transaction. We refer to such an internal transaction as *Merged Transaction* (MTx). On a high-level description, we have an MTx for every snapshot available in the system. The following sections describe this in more detail.

Read Committed. As discussed in Section 4.2, in *Read Committed* all read operations that refer to already committed data can be merged. In other words, in *Read Committed* only one snapshot exists. Hence, we provide only a single MTx, when running under *Read Committed*.

Equation 2 shows that comparing a read statement's key to the write keys of its submitting transaction is sufficient for this isolation level, therefore, we adopted Algorithm 1 for merge-decision in case of *Read Committed*:

Algorithm 4 *Read Committed* implementation: check if read r_i of T_i to x and read r_j of T_j to y can be merged. $x = y$ maybe possible.

```
1: function IsMergeable_RC($r_i(x)$, $r_j(y)$)
2:     if $h(r_i(x)) = h(w_i(x))$ then
3:         return false
4:     else if $h(r_j(y)) = h(w_j(y))$ then
5:         return false
6:     end if
7:     return true
8: end function
```

In line 2 and 4 we implement a lightweight check, if one of the submitting transactions tries to read their own writes.

We therefore track the write set by retrieving from each write (1) the accessed table id and (2) the given parameter and put these into a hash table, which we maintain for each transaction. For each incoming read request, we lookup the accessed table id and the parameter in the hash table of the specific transaction, which is more efficient than comparing to all writes that happened in the system. As we state in Section 1, transactions write far less than they read. So, in practice, that hash table often is empty.

All *Merged Read*s produced by the *Merger Threads* can thus be executed within the context of the same MTx, which is also executed in *Read Committed*. Hence, that MTx may start with TID 0 and never needs to commit, as it always provides the correct view for all reads not reading their own writes.

Because of the required properties outlined in Equation 1 in Section 4.1, we might use the same approach to also support *Read Uncommitted*. We would just have to omit the write set lookup, since all read statements can be merged under *Read Uncommitted*. However, as we implement our prototype inside SAP HANA, which does not support this isolation level, we do not discuss this implementation detail any further.

Snapshot Isolation. From Equation 3 in Section 4.3 we know that we can merge two read statements $r_i(x)$ and $r_j(y)$ running under *Snapshot Isolation*, iff either x or y lies in the intersection of T_i's and T_j's snapshot. As this means that the *Merger* has to have full access to the full snapshot of all transactions, leading to the *Merger* being a database itself, we restrict merging of read operations in *Snapshot Isolation* further.

We only merge two read statements $r_i(x)$ and $r_j(y)$ if T_i and T_j have the same snapshot on all data. Thus, Algorithm 5 shows our merge decision for *Snapshot Isolation*. To be able to check the commit time of a transaction, we extend the definition of h, introduced in Section 3.1, for commits: $h(c_f) = v_c$, where v_c can be seen as the commit timestamp of T_f. We define, if T_j started after T_f committed, then $h(c_f) < j$.

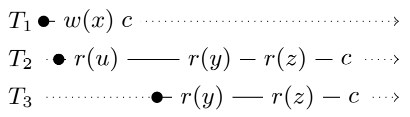
Algorithm 5 *Snapshot Isolation* implementation: check if read r_i of T_i to x and read r_j of T_j to y can be merged. $x = y$ maybe possible.

```
1: function IsMergeable_SI($r_i(x)$, $r_j(y)$)
2:     $C \leftarrow$ GetAllCommits($DBMS$)
3:     for all $c \in \mathcal{C}$ do
4:         if $min(i,j) < h(c) < max(i,j)$ then
5:             return false
6:         end if
7:     end for
8:     return IsMergeable_RC($r_i(x)$, $r_j(y)$)
9: end function
```

In line 4, we check if any commit has been submitted between the start of transactions T_i and T_j. If so, we do not merge. Otherwise, T_i and T_j operate on an identical snapshot and can thus be merged according to the rules of *Read Committed* (line 8).

Because comparison with all commits in the system is expensive, we propose a more efficient implementation than Algorithm 5. We extend our MTxs with three states: *open*, *depart*, and *committed*. A client's transaction is always attached to exactly a single MTx, when it requests *Snapshot Isolation*. If there is no MTx in the system, a new MTx is created with state *open* and the transaction is attached to it, meaning the transaction keeps the TID of the MTx, internally. Further incoming transactions will also attach to that MTx, which runs in isolation level *Snapshot Isolation* as well. Read statements of transactions that are attached to the same MTx are pushed into the same bucket within our *Merge Queue*. Thus, all read statements within the same bucket remain mergeable by the *Merge Threads* with low effort. However, as any client transaction decides to commit, a new snapshot is installed in the system. In consequence, all MTxs in state *open* switch to another state, *depart*. From now on, starting transactions cannot attach to these MTxs anymore and will need to open a new one. In *depart*, MTxs still allow merging of statements submitted by transactions already attached to this MTx, but no attachement of new transactions. Finally, when all transactions of an MTx have committed or aborted, the MTx switches its state again – from *depart* to *committed* – and finally terminates.

We are aware that our approach limits the merge potential with OLTP workloads, as illustrated exemplarily in the following scenario:



In this example, T_1 submits a write to x, while T_2 reads u. Next, T_1 commits and thereby installs a new snapshot that the subsequently starting T_3 operates on. T_2 and T_3 simultaneously submit a read to y and z, respectively, before they commit. Obviously $h(r_2(y)) = h(r_3(y))$ and $h(r_2(z)) = h(r_3(z))$. Thus, according to Algorithm 1, we can merge these read statements. However, as $2 < h(c_1) < 3$, (i.e., T_2 and T_3 are separated by the snapshot installed by T_1), Algorithm 5 would not allow to merge any of the reads in this example.

To overcome this drawback, we propose an optimization of Algorithm 5: Instead of running queries to multiple tables within the same MTx, we keep an open MTx for each table in the database. Thus, if x is related to a different table than u, y and z in the example above, we can still merge the submitted read statements. On commit, a transaction has to depart all MTxs related to tables in its write-set.

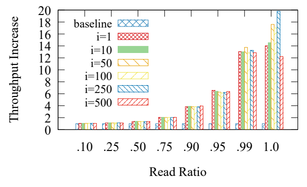
6. EXPERIMENTAL EVALUATION

This section evaluates our merging approach that we implement as prototype inside of the SAP HANA core database engine. We execute our benchmarks on a server with SUSE Linux Enterprise Server 12 SP1 (kernel: 4.1.36-44-default). Our machine has 512 GB of main memory in addition to four Intel(R) Xeon(R) CPU E7-4870 sockets with 10 cores. All run at a speed of 2.4 GHz and have a cache size of 30 720 kB. We have hyperthreading disabled.

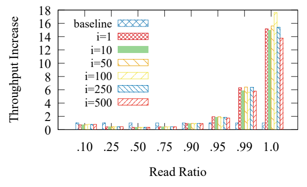
In the following, we report the results of different experiments: (1) We evaluate the effect of various parameters on the overall performance (throughput, latency) of our approach and show a cost breakdown of merging. For this experiment, we use the YCSB benchmark that is composed of single-table read/write statements as well as individual TATP transactions that are composed of more complex statements including read statements that join multiple tables. (2) As a second experiment, we use the full TATP as a standard OLTP benchmark to see the benefit of merging for different isolation levels under interactive execution. (3) Afterwards, we then run TATP as stored procedures to show the additional benefits of merging when using stored procedures instead of interactive execution. (4) Finally, we present our results when using SAP Hybris to evaluate the improvement of our approach for a real-world customer workload.

6.1 Experiment 1: Parameter Evaluation

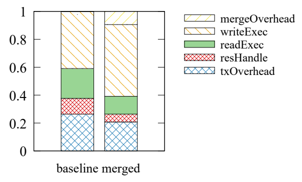
In this experiment, we first evaluate various parameters of the merging approach, namely the impact of the de-queuing interval, breakdown of the execution cost, latency, and the effect of the statement and transaction type on the merged result. For the first experiments, we use the YCSB benchmark since it allows us in addition to those parameters to vary the read/write ratio. Moreover, for the last experiment we use the TATP transactions as these consist of more complex statement types that involve joins across multiple tables. For running these benchmarks and simulating a high overload scenario, we use 700 clients and limit the DBMS to 3 worker threads only.



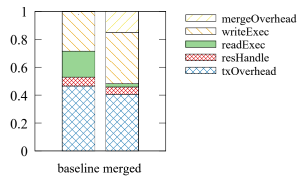
(a) *Read Committed*



(b) *Snapshot Isolation*



(c) Breakdown of 0.75 (RC)



(d) Breakdown of 0.75 (SI)

Figure 7: Micro-benchmark for YCSB

6.1.1 *YCSB: Read/Write Ratio and Dequeue Interval*

To show the impact of the workload's read/write ratio on our merging approach, we implement a multi-statement YCSB where each transaction is read- or write-only and consists of ten such statements. Clients execute the workload in an interactive setting (i.e., they submit statements one-by-one).

For running the workload, we use different read/write ratios and plot our results in Figure 7a for *Read Committed* and Figure 7b for *Snapshot Isolation*. More precisely, we have chosen the read ratios of 0.1 (i.e. 10% of all transactions are read-only), 0.25, 0.5, 0.75, 0.9, 0.95, 0.99, as well as 1.0 reflecting a read-only workload. We also varied the dequeuing intervals and used 1 µs, 10 µs, 50 µs, 100 µs, 250 µs, and 500 µs. We plot all results relative to the no-merging baseline indicated at 1.

As Figure 7a reveals, our approach can achieve a throughput increase of factor 20 for a read-only workload and an interval of 250 µs under *Read Committed* and 18× for an interval of 100 µs under *Snapshot Isolation* (cf. Figure 7b). However, with more writes within the workload, the throughput declines drastically to a throughput improvement of factor 12 respectively 6 for a workload with 99% reads and factor 8 respectively 2 for a workload with 95% reads. The case of 75% reveals a throughput improvement of about 50% under *Read Committed* and no improvement under *Snapshot Isolation*.

6.1.1.1 *YCSB: Breakdown of Execution Cost.*

To have a better overview where time is spent during execution of the workload of Figures 7a and 7b, we break down the execution for the workload with a read-proportion of 0.75 since this is the typical read ratio of OLTP as discussed before. Figure 7c and 7d compare the baseline to the merged

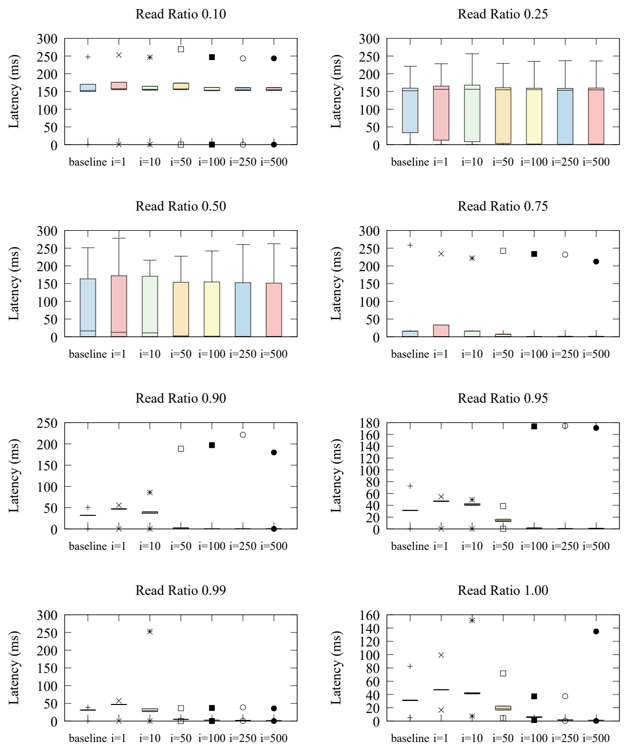
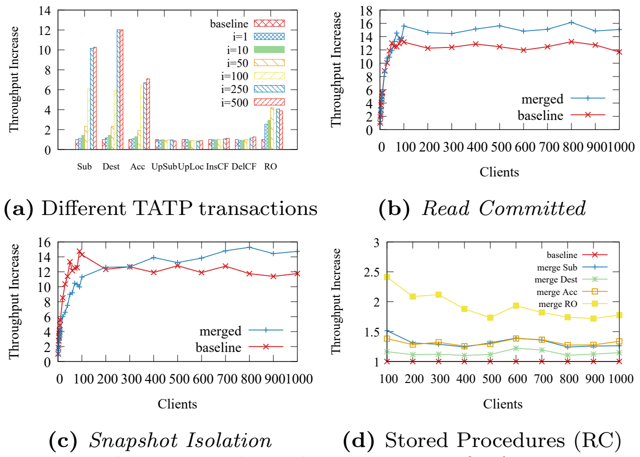


Figure 8: System latencies for different read/write ratios

execution with an interval of 100 µs for both, *Read Committed* and *Snapshot Isolation*. Naturally, the proportion of executing reads, compared to writes, shrinks, when merging is applied, as a *Merged Read* takes less execution time than executing all its reads one by one. In consequence, executing writes makes up about 40% of the baseline execution, but 50% when merging is applied for *Read Committed*. In *Snapshot Isolation*, the transaction overhead requires more execution time, than in *Read Committed*. Still, as Figure 7b reveals, the percentage of writes increases from 28.5% to 36.6%, when merging is applied. This ratio of non-mergeable queries, i.e., writes, will further increase, as more reads are merged. In consequence, to further optimize performance, writes also need to be merged, which is an improvement for future work. Most importantly, we can see that the overhead of our additional merging logic is relatively low and attributes to approx. 10% of the overall execution cost in *Read Committed* and 15% under *Snapshot Isolation*.

6.1.1.2 YCSB: Latency.

To evaluate the increase of latency through merging, we run the benchmark of Figure 7a and measure latency of read/write statements individually with and without merging under *Read Committed*. As we see in Figure 8, latency does not increase much for low read ratios of 0.10-0.75 while the tail latency increases, as expected. However, for the 0.90 up to 1.0 read rations, the median latency even decreases for larger dequeue intervals. Interestingly, these are also the cases where we see major throughput gains. The reason is that under high read ratios the latencies of transactions become more predictable by executing all read operations in fixed dequeue intervals.



(a) Different TATP transactions **(b)** *Read Committed* **(c)** *Snapshot Isolation* **(d)** Stored Procedures (RC)

Figure 9: Throughput increase of TATP

6.1.2 TATP: Different Transactions and Interval

In order to analyze the impact of different transaction types for merging we refer to the TATP benchmark [35] and run each TATP transaction individually. For space reasons, we abbreviate them in the following manner: `Get Subscriber` (*Sub*), `Get New Destination` (*Dest*), `Get Acces Data` (*Acc*), `Update Subscriber` (*UpSub*), `Update Location` (*UpLoc*), `Insert Call Forwarding` (*InsCF*), as well as the `Delete Call Forwarding` (*DelCF*) transaction. In addition, we also run TATP's read-only transactions (*RO*), consisting of *Sub*, *Dest*, and *Acc*. We use the same system-settings and intervals, as in our previous experiment and execute all statements in *Read Committed*. Figure 9a plots our results.
Our first observation is that merging improves the throughput of *Sub* by a factor of 10 and *Dest* by a factor of 12. While merging increases the throughput of *Acc* by a factor of 6, we see almost no improvement for transactions that contain DML statements. More interestingly, the throughput of running a read only mix is increased by a factor of 3 with an interval of 250 µs. This supports our hypothesis that merging is extremely beneficial for read heavy workloads with a limited set of hot-spot queries, which in fact is a very typical pattern in practical settings.

6.2 Experiment 2: Interactive Execution

In the second experiment, we investigate the gains of merging when using TATP as an interactive workload and analyze the effects of stored procedures in the next experiment. For this experiment, we now limit our system's resources to 10 worker threads pinned to 10 cores. In the merging case, the 10 threads are shared for executing merged statements and non-merged statements. Otherwise, all 10 threads are executing non-merged transactions. In both cases, clients connect to the database from an external *C++* driver program via `SQLDBC` and therefore measure the *end-to-end* throughput.

6.2.1 TATP: Read Committed

For *Read Committed*, Figure 9b reports a performance increase of 25% to 33% for 100 clients and more, once the system is fully utilized. In comparison to the 20% reported in [29] for *Read Uncommitted*, we find that our results are slightly better, which is probably due to an improved implementation. In more detail, we see two sides of the coin:

on the one side, we observe a decrease of the sharing potential, compared to YCSB or isolated execution of TATP's transactions, with the existence of more complex queries such as the join query in the *Dest* transaction. On the other side however, we see how merging improves the performance of merge-able TATP queries under high load resulting in a positive net effect of merging for complex interactive OLTP workloads under *Read Committed*.

The next section analyses to what extent this also holds, when TATP is executed under *Snapshot Isolation*.

6.2.2 *TATP: Snapshot Isolation*

As discussed in Section 4, *Snapshot Isolation* offers less sharing potential than *Read Committed*. Figure 9c depicts the benefit of merging for TATP executed interactively under *Snapshot Isolation*. Our first observation is that merging also provides a benefit for OLTP workloads executed under *Snapshot Isolation*. The throughput increase is in average 25%.

In comparison to the results presented in Figure 9b for *Read Committed*, we note two things:

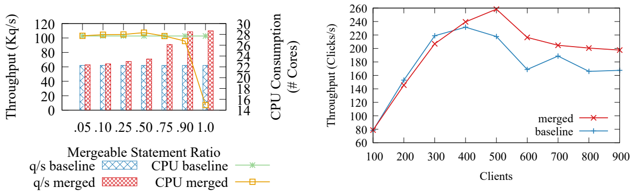
1. Workloads executed under *Snapshot Isolation* start to benefit from merging in more extreme overload situations. While TATP under *Read Committed* showed a performance improvement already for 100 clients, such an improvement is only visible from 300 clients onwards when executed under *Snapshot Isolation*.
2. The overall benefit shrinks, compared to *Read Committed*. As can be seen in Figure 9b merging leads to a throughput increase of 33%. For *Snapshot Isolation*, we improve throughput by 25%.

As Section 4 already stated that *Snapshot Isolation* decreases the merging potential of an interactive OLTP workload in comparison to *Read Committed*, we conclude that the above mentioned observations can be inferred from the stronger isolation level guarantees that *Snapshot Isolation* provides over *Read Committed*.

6.3 Experiment 3: Stored Procedures

Real-world OLTP workloads are often implemented as stored procedures to avoid sending every request over the network. As stored procedures are transactions under supervision of the DBMS, we consider them a subset of interactive transactions in this context. In order to outline the practicality of the merging approach also for such common settings, we implemented the TATP transactions used in the experiment before as stored procedures and executed TATP with an increasing number of clients, where each client in sequence submits calls to a TATP transaction. We run our experiment in *Read Committed* which allow us to compare the result to the results of the previous experiment.

Figure 9d presents our results relative to the baseline (always 1), when we execute TATP. First, merging is applied for all read-only operations (merge *RO*) which leads to an increase in throughput by a factor of 2.5×, as delaying the execution of read-statements leads to an earlier execution of non-mergeable statements. In addition, we observe that merging only *Dest* improves performance by 20%, while merging *Sub* or *Acc* improves performance by 40%. These numbers are easily explained, as *Sub* and *Acc* make 35% of the workload, each, while *Dest* makes only 10%.



(a) Analysis of single query type **(b)** Merging applied to workload

Figure 10: Application to SAP Hybris workload

6.4 Experiment 4: SAP Hybris

In our final experiment, we focus on SAP Hybris, a real-world enterprise OLTP workload traced from running a web-store on an app server with a SOLR server as web-page cache and a SAP HANA database on the same machine. The workload is designed as interactive workload running in *Read Committed*. The application server submits queries generated for web-page elements, such as product images. This setting implies few hot-spot queries, e.g., a single-select query for looking up product images, which we refer to as *media query*.

6.4.1 *SAP Hybris: Merging Potentials*

In a first step, we study the merging potentials of the hot-spot read-only queries (i.e., the *media queries*) before we run the full workload in the next experiment. For this experiment we run the *media queries* via ODBC against our prototype based on SAP HANA using 32 cores and merge 5%, 10%, 25%, 50%, 75%, 90%, or all of these incoming queries. This allows us to systematically observe the throughput increase and CPU consumption when merging is applied.

Figure 10a presents the throughput of thousand queries per second and the CPU consumption in active cores for both, baseline and merged approach. As expected, merging more of the incoming queries improves the throughput within this benchmark up to the point, where we increase the throughout by about a factor 2×, when 90%+ of all incoming queries are merged. Furthermore, we note that CPU consumption does not increase during merging; instead CPU consumption is decreased by almost a factor 2×, when all queries are merged. Compared to the baseline, half of the CPUs execute twice as many statements per second!

6.4.2 *SAP Hybris: Full Setting*

In the next step, we now run a SAP Hybris workload in a full Hybris landscape with an application server (12 cores), a SOLR server (4 cores), and our research prototype based on SAP HANA with 4 cores.
Figure 10b presents the throughput in clicks performed by all clients per second. We observe that by merging we increase performance by a constant factor of 20% for situations of more than 400 clients. We think that this gain is is significant, considering that SAP HANA is already highly optimized for this workload.

6.5 Discussion

Our experiments show that our approach improves the throughput of interactive multi-statement transactions such as TATP by 33% for *Read Committed* and 25% for *Snapshot Isolation*. However, using stored procedure increases throughput by 2.5× for *Read Committed*. Furthermore, the application of our approach on a business workload revealed a similar performance improvement as for TATP.

As we depict in Figure 6, such throughput increase may already be achieved with a non-intrusive extension of common in-memory databases, such as SAP HANA – the basis of our prototype implementation. To push the performance improvements even further, the database engine needs to integrate the merging support natively into its execution engine. E.g., with our approach, several statements may be executed in the context of the same MTx at the same point in time. In a common execution engine designed for executing one statement per transaction at a time, this may lead to undesired side-effects, which can only be avoided through locking, implicitly serializing statement execution. A database designed for a merged approach therefore needs to allow the flawless execution of statements within the same transaction at the same time.

Limitations. The benefit of merging is limited by three factors, as our experiments show: (1) The isolation level, (2) read/write ratio in the workload, and (3) number of tables read from. All of these factors limit the merging potential of a workload: isolation levels do so by increasing the number of snapshots to read from; writes do so by blocking execution threads and decreasing the amount of reads in the workload – and the more tables we have in our system, the fewer our abilities to merge incoming reads become. Furthermore, as discussed before, our inability of merging writes may decrease the potential of merging for some workloads.

7. RELATED WORK

Sharing techniques have been applied to different granularities of execution plans. Different shared scan techniques were described in [17, 39, 37, 28], techniques for merging other operators have been proposed in [20, 13, 26]. As these works focus mainly on scans and complex operators such as joins, they are hardly adaptable to OLTP workloads, which mainly consist of simple reads. Group Commit [5] merges the page-flush of commit operations, but does not further investigate into merging other OLTP operations.

An important work about merging subplans with similar expressions, commonly referred to as *Multi Query Optimization* (MQO) was submitted by [33]. Similar works are Cjoin [27, 3] that merge operations of star-queries, which often occur in OLAP scenarios, and Super-Operators [19] that aim for merging complex subplans into a single operator. While the latter focus on typical OLAP operations, MQOs could be adapted to OLTP workloads, as well, but finding common subexrepssions among OLTP queries might take more time than executing these queries.

In contrast to MQO, *Materialized Views* (MV) [31] can be used to store results of subplans for future submitted queries, as well. Keeping such MV updated in an OLTP scenario, is however very expensive and subject of ongoing research.

Finally, merging full statement plans, as proposed in this work, has been discussed in [22, 4, 11]. These works did neither consider the interval of waiting before batching nor any kind of transaction isolation. Another proposal how to design a database that supports merging throughout the full database stack has been discussed in [34]. SharedDB [12] as well as its successor BatchDB [21] propose a system that is able to merge statements of OLTP workloads and also support writes. However, a study or discussion on the mergeability under different isolation levels is missing in both works. Recently, Ding et al. [6] have described an optimistic system that allows batching of operations originating from fully submitted transactions. These batches are not implemented as merged accesses, but executed sequentially and are limited to fully submitted transactions (i.e., stored procedures) that do not require *read-your-own-writes*. OLTPShare [29] were the first to apply merging techniques within a state-of-the-art DBMS. However, they do not consider isolation levels within their workloads and in consequence can provide merging for *Read Uncommitted*, only. As most OLTP workloads demand higher isolation levels, OLTPShare therefore fails applicability in common customer scenarios.

In brief, most merging techniques focus on merging read-only scenarios with expensive calculations, commonly found in OLAP workloads, or fail to consider visibility of data items and the impact of isolation levels on the mergeability within the workload – a gap filled with our work.

8. CONCLUSIONS AND FUTURE WORK

Currently, merging of statements is only considered for read only scenarios, such as OLAP statements or OLTP workloads, where updates are processed in batches or the workload is considered to run in *Read Uncommitted*. As none of the previous approaches could deliver a solution how to deal with transactional isolation when merging, their application in real world systems has been difficult.

In this work, we describe how to implement merging of read statements in MVCC systems to provide both: transactional isolation as well as *read-your-own-writes*. Our idea is to find a snapshot that contains the requested versions of the data items accessed by the read statements to be merged and run the *Merged Read* with that snapshot. If such a snapshot cannot be found, we do not merge these statements and in consequence execute them in the context of the submitting transaction. Merging write statements on the other hand needs to provide error-handling, when a merged write fails, as well as complicated rollback and recovery extensions, in case a transaction aborts. Furthermore, different write statements need to be classified according to the isolation level under which they can be merged. Therefore, merging write statements was out of the scope of this paper and left for future work.

We implemented our approach as a prototype within SAP HANA and evaluated with two multi-statement transaction benchmarks: (1) the TATP benchmark executed interactively under different isolation levels and as stored procedure (2) a real-world SAP Hybris workload. Our results show that interactive OLTP workloads can benefit up to 33% from merging in overload scenarios, while keeping the isolation properties requested by the application (*Read Committed* and *Snapshot Isolation*). Throughput of stored procedures may even be improved by a factor of 2.5×. In conclusion, merging can now finally be used by real world industry systems.

9. REFERENCES

[1] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. ACM.

[2] P. A. Bernstein and N. Goodman. Multiversion concurrency control–theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, 1983.

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

[4] L. Chen, Y. Lin, J. Wang, H. Huang, D. Chen, and Y. Wu. Query grouping-based multi-query optimization framework for interactive sql query engines on hadoop. *Concurrency and Computation: Practice and Experience*, 30:e4676, 08 2018.

[5] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *Proc. ACM SIGMOD Int. Conf. Manag. Dat.*, SIGMOD '84, pages 1–8, New York, NY, USA, 1984. ACM.

[6] B. Ding, L. Kot, and J. Gehrke. Improving optimistic concurrency control through transaction batching and operation reordering. *PVLDB*, 12(2):169–182, 2018.

[7] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database replication using generalized snapshot isolation. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 73–84, Oct 2005.

[8] M. R. Emily Wilson and T. Kejser. Analyzing I/O Characteristics and Sizing Storage Systems for SQL Server Database Applications. Technical report, Microsoft, 2010.

[9] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The sap hana database - an architecture overview. *Bulletin of the Technical Committee on Data Engineering / IEEE Computer Society*, 35(1):28–33, 2012.

[10] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[11] Q. Ge, P. Peng, Z. Xu, L. Zou, and Z. Qin. FMQO: A federated RDF system supporting multi-query optimization. In *Web and Big Data - Third International Joint Conference, APWeb-WAIM 2019, Chengdu, China, August 1-3, 2019, Proceedings, Part II*, pages 397–401, 2019.

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

[13] 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.

[14] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li. X-engine: An optimized storage engine for large-scale e-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 651–665, New York, NY, USA, 2019. ACM.

[15] 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.

[16] T. K. Lance Ashdown and J. McCormack. Oracle® Database Database Concepts. Technical report, Oracle, August 2018.

[17] C. Lang, B. Bhattacharjee, T. Malkemus, S. Padmanabhan, and K. Wong. Increasing buffer-locality for multiple relational table scans through grouping and throttling. In *PVLDB*, pages 1136–1145, 05 2007.

[18] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4):298–309, 2011.

[19] J. Leeka and K. Rajan. Incorporating super-operators in big-data query optimizers. *PVLDB*, 13(3):348–361, 2019.

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

[21] 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.

[22] R. Marroquin, I. Müller, D. Makreshanski, and G. Alonso. Pay one, get hundreds for free: Reducing cloud costs through shared query execution. In *SoCC*, 2018.

[23] K. May. Airline system look-to-book ratios soar, expected to go 10x higher. https://www.phocuswire.com/Airline-system-look-to-book-ratios-soar-expected-to-go-10x-higher, December 2015. [Online; accessed 15-March-2019].

[24] N. May, A. Böhm, and W. Lehner. SAP HANA – The Evolution of an In-Memory DBMS from Pure OLAP Processing Towards Mixed Workloads. In B. Mitschang, D. Nicklas, F. Leymann, H. Schöning, M. Herschel, J. Teubner, T. Härder, O. Kopp, and M. Wieland, editors, *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*, pages 545–546. Gesellschaft für Informatik, Bonn, 2017.

[25] H. H. Ohad Rodeh and D. Chambliss. Visualizing Block IO Workloads. Technical report, IBM Research Division, October 2013.

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

[27] I. Psaroudakis, M. Athanassoulis, M. Olma, and A. Ailamaki. Reactive and proactive sharing across concurrent analytical queries. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 889–892, New York, NY, USA, 2014. ACM.

[28] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-memory scan sharing for multi-core cpus, 2008.

[29] R. Rehrmann, C. Binnig, A. Böhm, K. Kim, W. Lehner, and A. Rizk. OLTPshare: The Case for Sharing in OLTP Workloads. *PVLDB*, 11(12):1769–1780, 2018.

[30] R. Rehrmann, M. Keppner, W. Lehner, C. Binnig, and A. Schwarz. Workload merging potential in sap hybris. In *Proceedings of the Workshop on Testing Database Systems*, DBTest '20, New York, NY, USA, 2020. Association for Computing Machinery.

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

[32] M. Samet. Consolidating Oracle® OLTP Workloads with XtremIO. Technical Report Part Number H13828-1 (Rev. 02), EMC Corporation, December 2014.

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

[34] Z. Shang, X. Liang, D. Tang, C. Ding, A. J. Elmore, S. Krishnan, and M. J. Franklin. Crocodiledb: Efficient database execution through intelligent deferment. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[35] 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.

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

[37] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *PVLDB*, 2(1):706–717, 2009.

[38] 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].

[39] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: Dynamic bandwidth sharing in a dbms. In *In Proc. of the 33 rd Intl. Conf. on Very Large Databases (VLDB*, pages 723–734, 2007.