DB4ML – An In-Memory Database Kernel with Machine Learning Support

Matthias Jasny* TU Darmstadt
Tobias Ziegler* TU Darmstadt
Tim Kraska MIT
Uwe Roehm The University of Sydney
Carsten Binnig TU Darmstadt

ABSTRACT

In this paper, we revisit the question of how ML algorithms can be best integrated into existing DBMSs to not only avoid expensive data copies to external ML tools but also to comply with regulatory reasons. The key observation is that database transactions already provide an execution model that allows DBMSs to efficiently mimic the execution model of modern parallel ML algorithms. As a main contribution, this paper presents *DB4ML*, an in-memory database kernel that allows applications to implement user-defined ML algorithms and efficiently run them inside a DBMS. Thereby, the ML algorithms are implemented using a programming model based on the idea of so called iterative transactions. Our experimental evaluation shows that *DB4ML* can support user-defined ML algorithms inside a DBMS with the efficiency of modern specialized ML engines. In contrast to *DB4ML*, these engines not only need to transfer data out of the DBMS but also hard-code the ML algorithms and thus are not extensible.

ACM Reference Format:
Matthias Jasny, Tobias Ziegler, Tim Kraska, Uwe Roehm, and Carsten Binnig. 2020. DB4ML – An In-Memory Database Kernel with Machine Learning Support. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3380575

1 INTRODUCTION

Based on a recent survey from Kaggle, relational data is the most commonly used type of data in data science teams and is typically used for building classical machine learning models such as regression models, decision trees, or unsupervised algorithms such as PageRank or clustering algorithms.[1] The standard approach for applying machine learning (ML) algorithms on relational data is to first select the relevant entries with an SQL query and export them from the database into an external ML tool. Then the ML algorithm is run over the extracted data — outside the DBMS — using statistical software packages or ML libraries such as R, SPSS, or Scikit-learn.

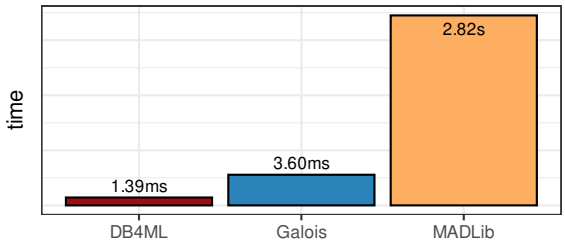


Figure 1: DB4ML vs. Galois vs. MADLib (PageRank) — *Numbers are averaged over 5 runs on the Wikivote dataset. In order to provide a fair comparison of* DB4ML *and Galois (which are main-memory engines) with MADLib (which is based on PostgreSQL) we used the maximal memory size for the buffer pool and warmed up the buffer by scanning the required tables.*

However, this approach can impose a high overhead due to expensive data transfers which can significantly slow down the overall learning procedure especially if the datasets are large. Integrating ML algorithms into DBMSs is thus an ongoing effort in both academia and industry. But performance is not the only reason why vendors such as IBM, Microsoft, Oracle, SAP, etc. integrate ML into a DBMS. Another major reason is compliance (e.g., requirements from HIPAA or the financial sector) that often discourage applications to export any data out of a DBMS since DBMSs already provide rich security frameworks to protect the data from unauthorized access.[2] Moreover, more recent regulations such as GDPR enforce that an enterprise must ensure that all information

*Both authors contributed equally to this research.

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3380575

[1] https://www.kaggle.com/surveys/2017
[2] https://download.oracle.com/database/oracle-database-security-primer.pdf

about a specific consumer to be deleted from enterprise storage, when requested. Allowing applications to copy data out of a centralized DBMS renders finding and deleting all copies of data on a particular consumer almost impossible [15].

One prevalent approach for integrating machine learning into a DBMS is to extend the SQL execution engine with iterative concepts, as proposed by [9, 11, 12], and translate the ML algorithm into a sequence of SQL queries. A main issue of this approach is that SQL engines only offer bulk synchronous parallelism for ML algorithms; i.e., each iteration of an ML algorithm is implemented by executing an SQL query — potentially in parallel — over the full data set before the next iteration can start. For example, in MADLib [12] PageRank is implemented by using a driver program that runs one SQL query, which computes the new PageRank values for the next iteration for the complete graph, and repeats this until the PageRank algorithm converges.

Unfortunately, this approach can be orders-of-magnitude slower than modern parallel ML algorithms [30, 33, 38], which benefit significantly from fine-grained concurrent execution schemes with relaxed consistency models. For example, in PageRank each node in the graph can progress independently, leading to overall faster convergence as shown by [33]. Similar observations have been made for other ML algorithms [30, 38].

Contributions. In this paper, we thus revisit the question of how ML algorithms for structured data should be best integrated into existing DBMSs. Our approach is based on the key observation that modern in-memory database systems already support fine-grained concurrency control in the form of transactions. Yet, traditional transaction execution schemes are often too heavyweight. Consequently, we show how it is possible to efficiently leverage transaction semantics also for ML algorithms. As a main contribution, we present a new in-memory database kernel called *DB4ML* that is based on transactions but adds extensions to enable ML algorithms on top of classical transaction processing.

As a first extension to enable ML algorithms inside a DBMS, we propose the concept of *iterative transactions*. Different from normal transactions, with iterative transactions the very same transaction can be re-executed multiple times until convergence without the need to be actively re-scheduled for every iteration by a driver program (i.e., a client) as done in existing approaches. Furthermore, as a second extension, we add new isolation levels for machine-learning into *DB4ML*, such as bounded-staleness [7], and show how they can be efficiently implemented based on top of an MVCC-based storage manager that was designed for OLTP.

In order to analyze the efficiency of our approach we implement the ideas of this paper in *DB4ML*. As Figure 1 shows, *DB4ML* can not only outperform the classical approach of integrating ML into DBMS (MADLib) but more importantly also provides a performance comparable to modern parallel ML algorithms implemented outside the DBMS (Galois). In our experimental evaluation in Section 7, we show a more detailed analysis of *DB4ML* for different ML algorithms and datasets.

While we envision that the techniques implemented in *DB4ML* will help to guide the design of modern in-memory database system to support ML inside the DBMS, the techniques presented in this paper are not limited to that scenario: First, if data exports are not a problem, *DB4ML* can also be used as a standalone ML engine since its abstractions (iterative transactions and ML isolation levels) provide applications with ways to implement their ML algorithm in a high-level manner and execute them in an efficient ML runtime system without the need to hand-tune each individual ML algorithm. Second, we further believe that the techniques proposed in this paper are not limited to single-node in-memory DBMSs and can be extended towards disk-based DBMSs or even for the distributed settings. However, showing this is beyond the scope of this paper and represents an interesting avenue of future work.

To summarize, the contributions of this paper are:

- We define a programming model for user-defined iterative transactions that supports a wide class of ML algorithms and allows developers to easily integrate new ML algorithms into a DBMS.
- We discuss the implementation of our transactional database kernel called *DB4ML* including a storage manager and execution engine that can efficiently run parallel ML algorithms.
- We showcase through two use cases (PageRank as well as Stochastic Gradient Descent) how ML algorithms could be implemented inside *DB4ML*.
- Our experimental evaluation shows for the aforementioned use cases that *DB4ML* can support ML algorithms with the efficiency of modern specialized ML engines without the need to transfer data out of the DBMS. In our evaluation, we compare *DB4ML* against state-of-the-art SGD and graph-engines [26, 30, 38].

Outline. The remainder of this paper is organized as follows: The next section gives an overview of the architecture of *DB4ML* as well as the proposed programming model for user-defined iterative transactions. Section 3 and Section 4 then present our storage manager and the execution engine of *DB4ML* while Section 5 discusses several optimizations that are crucial to achieve an efficient execution of ML algorithms. Afterwards, in Section 6 we demonstrate by implementing two use cases (PageRank and SGD) how these algorithms can be realized inside *DB4ML* with the help of user-defined transactions. Section 7 presents the evaluation

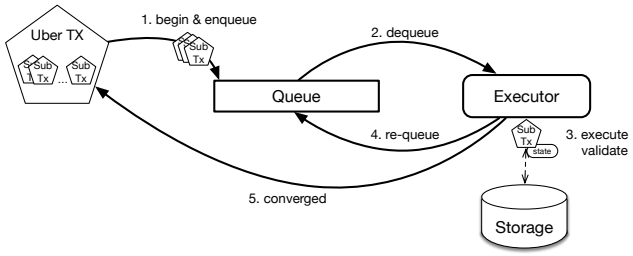


Figure 2: High-level Architecture of *DB4ML*.

```
1  enum T_Action {COMMIT,ROLLBACK,DONE};
2
3  interface IterativeTransaction {
4     void begin(T_State initial_state);
5     void execute();
6     T_Action validate();
7
8   private:
9     T_State tx_state;    // local state of tx
10 };
```

Listing 1: Interface for Iterative Sub-Transactions.

of *DB4ML* and shows that *DB4ML* is able to process ML algorithms inside the DBMS as efficient as specialized in-memory engines. Section 8 gives an overview of related work, and Section 9 concludes the paper.

2 OVERVIEW

In the following, we first present the overall design of our novel in-memory database kernel *DB4ML*. We then further discuss the programming model used to implement and execute ML algorithms using *DB4ML*.

2.1 High-level Architecture

Figure 2 shows the high-level architecture of our database kernel for ML called *DB4ML*, which consists of two main components: an in-memory *storage* manager and an *executor* for iterative transactions. While *DB4ML* is based on the design of a modern in-memory storage layer that supports MVCC, it is not a full-fledged DBMS.

The main idea is that *DB4ML* implements a storage and an execution engine that are optimized for ML while still supporting classical OLTP workloads effectively. That way, tables created inside *DB4ML*'s storage engine (called ML-tables) can be queried and updated through classical transactional workloads. This is a major difference to specialized ML engines such as Hogwild which do not integrate well into a DBMS architecture. Consequently, they require copying or exporting the data instead of using the data stored inside the DBMS directly.

In order to run an ML algorithm (e.g., PageRank) inside *DB4ML*, we extend the traditional transaction execution model of a modern in-memory DBMS to a nested transaction model [36] with a top-level *uber-transaction* that initiates a number of sub-transactions for running the ML algorithm in parallel on ML-tables (e.g., one sub-transaction per node in the graph). The sub-transactions are executed and rescheduled iteratively by the executor. This design allows *DB4ML* to easily parallelize the execution of the ML algorithm in a fine-grained manner.

When starting an ML algorithm, *DB4ML* instantiates the uber-transaction for the particular ML algorithm. The uber-transaction then first instantiates multiple sub-transactions of the same type on different parts of the data and adds those sub-transactions to a lock-free queue for scheduling (step 1 in Figure 2). For example, in order to start a PageRank algorithm, we start one sub-transaction for each node of the graph to iteratively refine its PageRank.

Iterative sub-transactions are executed by *DB4ML* using worker threads that are pinned to a CPU core. Whenever a worker thread in *DB4ML* is idle, it fetches a new sub-transaction from the work queue (step 2) that then updates the learned model stored in our MVCC storage (step 3). Based on the outcome of a sub-transaction's execution, the sub-transaction is either re-scheduled by the executor for the next iteration, or dequeued as soon as it has converged (step 4 and 5 in Figure 2).

During execution, the sub-transactions can access data in the storage layer to read/update the current state of the computation (e.g., the current PageRank values of nodes in the graph). As this state might be read/modified by multiple concurrent sub-transactions, *DB4ML* provides isolation levels that implement well-known ML synchronization schemes for parallel execution on top of its MVCC-based storage as discussed next.

2.2 Synchronization Schemes

Parallel ML algorithms typically follow a data-parallel pattern, where workers independently run the learning algorithms for subsets of the data. In order to guarantee convergence the most common scheme is that workers synchronize the results after every iteration. For example, to support a parallel PageRank we have to enforce that all nodes in a graph are processed and a new PageRank is computed before starting the next iteration.

Recent work in ML however relaxed the synchronization restriction, allowing workers to also read inconsistent state and even overwrite the results of other workers. A well-known instance of this scheme is the lock-free approach to parallelize SGD from Hogwild! [30]. Hogwild! allows workers to read parameters and update them completely asynchronously, potentially overwriting each others updates. Hogwild! has been proven to converge for sparse learning problems, where updates only modify small subsets of the learned parameters. However, asynchronous schemes do not give guarantees to converge in all cases. For example,

Hogwild! provides no guarantees for non-sparse learning problems. Hence, another synchronization scheme has been developed called bounded-staleness to allow only a limited degree of asynchronicity and thus guarantee convergence. The main idea of bounded-staleness is that in between a model read and update of one worker only S other workers can update the same model parameters. Moreover, another effect of bounded-staleness is that it also better mitigates the effects of data skew (or stragglers in general) compared to the asynchronous scheme which under skew often spends unnecessary computation and converges more slowly while the synchronous scheme blocks if stragglers occur [7].

The contribution of *DB4ML* is that it implements these well-known ML synchronization schemes [3] (synchronous, asynchronous, and bounded-staleness) as isolation levels on top of our MVCC-based storage for running sub-transactions in parallel and coordinating the visibility of model-updates. That way, an application can choose the most appropriate isolation level for running their ML algorithm without hard-coding this into the algorithm design. As a result, for example, *DB4ML* can run its parallel SGD fully asynchronously in case the learning problem is sparse or use bounded-staleness as isolation level for non-sparse learning problems by just implementing the algorithm once.

2.3 Programming Model

In order to add a new ML algorithm to *DB4ML*, users have to implement two types of transactions: an uber-transaction and an iterative sub-transaction. Uber-transactions are implemented using normal non-iterative database transactions. As mentioned before, the uber-transaction coordinates the overall execution and checks for the convergence of the ML algorithm.

To implement iterative sub-transactions, *DB4ML* provides a new transaction interface as shown in Listing 1. At its core, the interface defines an `execute()` function. The idea of this function is that it implements a single iteration of the ML algorithm. For example, the `execute()` function of an iterative sub-transaction for PageRank could compute the new rank for a single node or a subset of nodes in a graph that are stored using ML-tables in *DB4ML*.

Another important extension over normal transactions is that iterative sub-transactions can read/write data into/from a transaction-local storage (called `tx_state`), which is private to each instance of a sub-transaction. Parameters to initialize the local state, can be handed over to the `begin()` function which is called once before starting the execution of the sub-transaction by the uber-transaction. The transaction-local storage is kept between different iterations of the transaction and can be used for caching parameters or other values that otherwise would need to be fetched from ML-tables in every iteration. In PageRank, for instance, pointers to neighbors of a node or other parameters of the PageRank algorithm (e.g., the epsilon parameter to control the convergence) can be cached in the transaction-local storage.

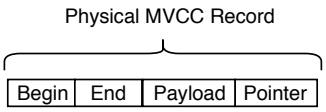


Figure 3: Physical Layout for Normal Records.

Finally, the interface of iterative sub-transactions defines a method called `validate()` to control the execution flow of a sub-transaction. The `validate()` method is called by *DB4ML* after the `execute()` function returns, in order to determine the next action for an iterative sub-transaction. The possible return values (`T_ACTION`) of `validate()` are:

- The `validate` returns `COMMIT` if the sub-transaction finished one iteration but did not converge yet. In this case, the updates of the last iteration of the sub-transaction are committed (e.g., new PageRank values are written) and made visible only to other sub-transactions initiated by the same uber-transaction (i.e., not globally). Moreover, the sub-transaction is enqueued again for the next iteration.
- The `validate` returns `ROLLBACK` if the last iteration was not successful. This could happen if the ML-isolation guarantees are violated as we discussed before (e.g., the data read by the sub-transaction is too stale w.r.t. the staleness-level defined for the ML algorithm). In this case, the updates of the last iteration of the sub-transaction are discarded. The sub-transaction is enqueued again to repeat the iteration.
- The `validate` returns `DONE` if the sub-transaction converged. In this case, the updates of the last iteration of the sub-transaction are committed but the transaction is not re-scheduled anymore (since it converged).

Once all sub-transactions of an uber-transaction converged, the results of the ML algorithm are made visible globally for other transactions in the DBMS. In the following, we discuss the detailed implementation of the storage manager and the execution engine of *DB4ML*.

3 STORAGE MANAGER

The storage manager of *DB4ML* provides so called *ML-tables* that enables *DB4ML* to support iterative ML algorithms inside a DBMS. As we will see later in Section 4, using our storage layout allows to elegantly realize different isolation levels especially geared towards ML algorithms.

3.1 Basic Storage Layout

Iterative ML algorithms typically show the following workload pattern: In each iteration multiple data items are read

from the current database state in order to compute a new global state (e.g., the new PageRank of a node in a graph). The new state is then updated in the database and made visible to other transactions so that subsequent iterations are based on this new state. Consequently, these workloads are characterized by many reads, few updates and multiple read-write dependencies between different iterative transactions. This type of workload is not well suited for locking-based concurrency control protocols, where readers and writers block each other when accessing the same data. Instead, Multi-Version Concurrency Control (MVCC) [24] is a well-known solution to tackle this problem which allows readers to read a version without blocking writers to install new versions.

Figure 3 illustrates the basic MVCC-based record format that is used in our storage engine for ML-tables which is based on the design of [16]. Each record consists of a header, a payload, and a pointer to the previous version. The header defines the valid lifetime for a particular version as specified by the `Begin` and `End` timestamps that define the visibility of different versions of the same record. For the most recent version of a record, `Begin` marks the commit time of the transaction which installed the latest version and the `End` timestamp is set to `INF`.

In *DB4ML*, a new version of a record can be installed either by a normal database transaction or by an uber-transaction once it commits; i.e., once the ML algorithm terminates and the uber-transaction makes the results visible to other DBMS transactions.

3.2 Extensions for Sub-Transactions

An important difference of our storage layout to the basic MVCC-based layout in [16] is how we support intermediate versions of iterative sub-transactions within the same uber-transactions to synchronize concurrent updates while running an ML algorithm (e.g., PageRank). The key concept is that sub-transactions spawned in the context of the same uber-transaction use the following nested MVCC scheme:

- After the start of an uber-transaction, all sub-transactions are initialized with the same begin timestamp T_{TB} as the uber-transaction; i.e., in the first iteration all sub-transactions started by the same uber-transaction read the same snapshot.
- Furthermore, sub-transactions that are started by the same uber-transaction are also able to see updates of the other sub-transactions that are committed after each iteration.
- Other (uber-)transactions and their sub-transactions (as well as normal database transactions), however, are not allowed to read in-flight iterative versions. Only after an uber-transaction commits once all sub-transactions converged, the result of the ML algorithm is made visible to other transactions.

In order to enable this nested MVCC scheme, sub-transactions do not create a new version of a record for each intermediate commit. Instead, for efficient processing, an uber-transaction creates a so called iterative record, which is used by all sub-transactions of the uber-transaction to store their intermediate commits as so called iterative snapshots in the payload of an iterative record. The physical layout of an iterative record is shown in Figure 4 (upper part).

Similar to the normal MVCC-based record layout shown in Figure 3, an iterative record also has a `Begin` and an `End` field to control the global visibility of the record; i.e., when other transactions are allowed to see the version. However, in addition to a normal record, the header of an iterative record contains an additional version counter (called `IterCounter`) which is incremented with each commit of a sub-transaction and is used to ensure consistency among sub-transactions of the same uber-transaction. This counter is initialized with 0.

Furthermore, the payload of an iterative record allows sub-transactions to store multiple intermediate versions into a so called intermediate version array I of size n denoted as $Version_1, \ldots, Version_n$ in Figure 4. An important aspect of an iterative record is that it actually only needs to store a fixed number of intermediate versions (also referred to as iterative snapshots) into I while I is used as a circular buffer. This allows efficient iterative processing without any additional allocations to grow I. For installing a new version into I, the $IterCounter$ is incremented and the new incremental version of the record is written into the array position $IterCounter\%size(I)$.

Figure 4 (lower part) shows an example of the result after running PageRank on a `Node` table implemented as ML-table storing a graph. In the example, we see the versions for the two nodes 1 and 2 (i.e., two tuples of the table). For each tuple, we see they were initially installed using a normal database transaction that resulted in a normal version that was valid from timestamp 0 to 10. Afterwards, PageRank was started which created a new iterative record that is the most recent version. The iterative record has a snapshot array I of size 3. During the execution of PageRank, iterative snapshots were installed into I whenever an iterative sub-transaction committed a new PageRank (PR). As we see in the example, the version counter for both nodes (NodeID 1 and 2) is set to 97 and 93 respectively, which means that iterative sub-transaction committed already 97 and 93 intermediate snapshots for these two nodes. Moreover, we can see that the uber-transaction that was running PageRank already terminated and committed its final output since the iterative record was made visible to other transactions by setting the `Begin` field of the iterative record to the commit timestamp

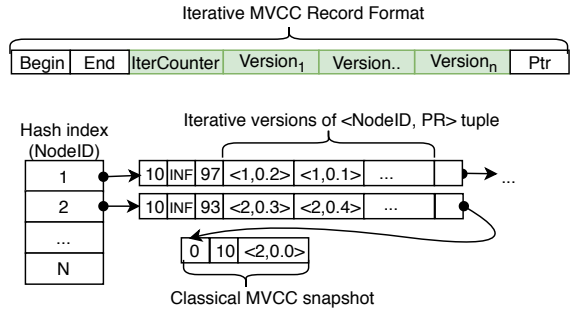


Figure 4: Physical Layout for Iterative Records.

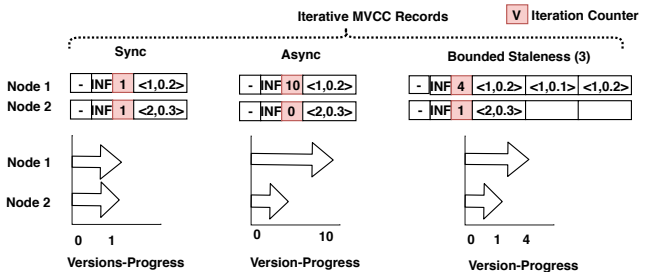


Figure 5: Different ML Isolation Levels.

T_{TE} of the uber-transaction (10 in our example) and the End field to INF.

4 EXECUTION ENGINE

In the following, we discuss how ML algorithms implemented using our programming model are being executed in *DB4ML*. We then show how ML-native isolation levels are implemented on top of our MVCC-based storage manager. At the end of this section, we also briefly outline some optimizations to enable an efficient execution on modern multi-socket machines.

4.1 Executing ML algorithms

Uber-transactions for ML are executed as normal database transactions in *DB4ML*; i.e., if a new uber-transaction is started, the executor assigns the current timestamp as begin timestamp T_{TB} to the uber-transaction, which determines the versions that an uber-transaction can read. Furthermore, all sub-transactions that are started by the same uber-transaction inherit this timestamp and therefore see the same snapshot. However, different from normal database transactions, iterative sub-transactions in *DB4ML* use iterative records (as discussed before in Section 3) to install intermediate versions. In order to determine which intermediate version to read, sub-transactions use the current iteration counter *IterCounter* of an iterative record.

Once initialized, each sub-transaction keeps re-executing until it converges. After every iteration, a sub-transaction increments the iteration counter of the records it updates to install a new intermediate version as discussed before. Based on the isolation level, concurrent writes of sub-transactions to the same record are handled differently as we describe below. In case the execution of one iteration of a sub-transaction is not successful (i.e., the execute-function returns ROLLBACK) the iteration counter is not increased and the write-set is not installed.

Once all sub-transactions of an uber-transaction converged (i.e., all sub-transactions return DONE as a result of the last validate call), the last intermediate snapshot that was committed by the sub-transactions is installed as a new global snapshot by the uber-transaction using the current timestamp as commit timestamp T_{TE}.

4.2 ML Isolation Levels

In the following, we explain how the different isolation levels are implemented based on our previously discussed MVCC-based storage engine:

Synchronous. In this level, a sub-transaction is allowed to commit a new iterative version if in between a read and a potential write to the same record, no other sub-transaction installed a new version. Furthermore, if a sub-transaction reads multiple records, they must all have the same version. That way, synchronous execution is in fact a parallelized bulk-synchronous execution of an ML algorithm. In Figure 5 (left), transaction T1 calculates the PageRank for Node 1 and T2 for Node 2 respectively. In the synchronous isolation level all nodes progress at the same speed.

Asynchronous. If this isolation level is set, no constraints exist on which intermediate version of a tuple a sub-transaction is allowed to read; i.e., the tuples can still read the most recent version but different from the synchronous isolation level, a transaction can read different versions for different records. Furthermore, a new version of a record can be installed without any further checks. This might lead to unequal progress of iterative sub-transactions as shown in Figure 5 (center).

Bounded Staleness. This isolation level combines the ideas of both asynchronous and synchronous; i.e., if a sub-transaction reads multiple records it is allowed to use any intermediate snapshot with a version in the range $[IterCounter - S, IterCounter]$ where s is a configurable staleness factor [7]. In our example in Figure 5, S is set to 3.

5 OPTIMIZATIONS

As discussed before, *DB4ML*'s design is a general engine for user-defined ML algorithms based on an MVCC-based storage manager and a transaction-based execution engine. This

comes with overhead regarding storage management (e.g., versioning) as well as transaction scheduling. In order to alleviate the overhead of being a general engine for user-defined ML algorithms, *DB4ML* implements the following important optimizations to achieve a performance comparable to more specialized hand-tuned engines that can run only a fixed set of algorithms.

5.1 Storage Optimizations

The main overhead of the MVCC-based storage engine is that it needs to keep track of multiple versions and their versioning information. In the following, we discuss optimizations for the different isolation levels to minimize the versioning overhead.

- In order to implement the synchronous isolation level, *DB4ML* currently uses a barrier after each iteration instead of using the multiple versions and executing the version-checking mechanism to guarantee a synchronous execution. This has not only the effect that version-checking can be completely avoided but also that sub-transactions are not unnecessarily executed and then directly aborted.
- The same observation holds also for the asynchronous isolation level. Same as before, in this case *DB4ML* also only needs to keep the most recent iterative snapshot. However, instead of using a barrier to synchronize iterations, we simply install a new version using an atomic store operation without any barrier.
- For providing bounded-staleness *DB4ML* has to execute ML algorithms using the most general storage implementation as discussed in Section 3 where multiple versions are kept in an iterative record. However, if we know that only one sub-transaction updates the same tuple as it is the case for our PageRank implementation where the PageRank of one node is only updated by the same transaction we can also use only one version to provide bounded-staleness. This optimization can be enabled by the ML algorithm using a hint when starting the uber-transaction.

5.2 Execution Optimizations

Another source of overhead in *DB4ML* stems from the fact that *DB4ML* needs to (re-)schedule transactions after every iteration. In order to reduce this overhead, sub-transactions are pre-grouped into batches to reduce the load on the scheduling queues; i.e., the scheduling queues hold pointers to batches instead of pointers to individual sub-transactions. The workers thus also keep track of the number of converged sub-transactions per batch; i.e., only when batch-wise convergence is reached, the batch is not rescheduled into the queue anymore.

Second, similar to other main-memory DBMSs, *DB4ML* also allows to partition the data to individual NUMA regions [29] to optimize for data locality. At the moment, *DB4ML* provides the classical data-partitioning schemes such as hash-partitioning, round-robin, and range-partitioning tables. Furthermore, for execution, worker threads in *DB4ML* are pinned to a CPU core and grouped per NUMA region. All worker threads in one NUMA region use a single queue per group to avoid remote NUMA traffic. Each worker thread in a NUMA region processes transactions by dequeuing a sub-transaction from the assigned queue.

6 USE CASES

In this section we show how *DB4ML* can be leveraged to run ML algorithms exemplified by two iterative ML algorithms: PageRank and Stochastic Gradient Descent (SGD).

6.1 Use Case 1: Parallel PageRank

6.1.1 Overview. The first use case is the well-known PageRank algorithm [28] which measures the relative importance of web pages. PageRank assumes a graph structure where individual web-pages are modeled as nodes and hyperlinks between web-pages as directed edges.

The goal of PageRank is to assign a numerical weight to each node, called PageRank ($PageRank(node)$), which indicates the importance of a node. This weight represents the possibility of a random visit and is affected by the weights of incoming edges [28].

In our implementation we calculate the PageRank using Equation 1, which was proposed by [28]. It computes the PageRank of a node u based on the incoming edges from the neighbors v. N represents the total number of nodes in the graph and d is a damping factor. In our implementation, we set d to 0.85, which is a damping factor commonly used [10].

$$PageRank(u) = \frac{1-d}{N} + d \sum_{(v,u) \in Edges} \frac{PageRank(v)}{outdegree(v)} \quad (1)$$

Each iteration of the PageRank algorithm evaluates the formula in Equation 1 for each node in the graph. PageRank converges if the weights remain stable according to a threshold or after a pre-set and fixed number of iterations, which prevents too long running executions. In the following, we describe the process of implementing the PageRank algorithm in our programming model using uber-transactions and iterative sub-transactions.

6.1.2 Mapping to DB4ML. We first explain the relational data model and then show the pseudo-code for the transactions.

NodeID	PR
1	0.1
2	0.3
...	...

NID_From	NID_To
1	2
2	1
...	...

Figure 6: Node and Edge table for PageRank

Data Model. Our PageRank implementation leverages the data model as shown in Figure 6. The Node table (shown left) contains a tuple of the form <NodeID, PR> representing the nodes in the graph with their NodeID and PageRank PR. The Edge table (shown right) describes the adjacency list in the form of <NID_From, NID_To>. In order to efficiently retrieve the PageRank of neighbor nodes, we created an index on the Node table on NodeID and on the Edge table on NID_To.

Uber-Transaction. The uber-transaction is responsible to schedule iterative sub-transactions as described in Algorithm 1. Further, it defines the isolation level for all sub-transactions as being either synchronous, asynchronous, or bounded staleness. The details of these isolation levels are discussed in Section 4.

To compute the PageRank, the uber-transaction schedules a sub-transaction for each node in the Node table while passing the nodeId and respective neighbor node identifiers neigbourIds as local state. The sub-transaction is created and enqueued in *DB4ML*'s execution system using its begin function. The uber-transaction waits until all sub-transactions are converged, and then persists the final PageRank values by issuing a COMMIT.

Algorithm 1: PageRank — Uber-Transaction

```
1  BEGIN TRANSACTION
2      SET SUB-TX ISOLATION LEVEL {SYNC|ASYNC|BOUNDED-STALENESS}
3      for each node in Node Table do
4          nodeId = node.NodeID
5          neighborIds = get_neighbors(nodeId)
6          sub_tx = new pr_sub_tx()
7          sub_tx.begin(nodeId, neigbourIds)
8      end
9      WAIT //until all sub_tx converged
10     COMMIT
11 END
```

Iterative-Transaction. Algorithm 2 describes how the iterative sub-transaction interface (see Section 2.3) is implemented for the PageRank algorithm.

The begin method (line 2–5) defines the state of a PageRank transaction as consisting of four elements: nodeId, neighborIds, and the intermediate PageRank values. Since, the first two elements never change over the course of the algorithm, they are cached in the transaction state. This allows to efficiently retrieve the neighbors PageRank in every iteration; i.e., in the execute function. The actual calculation of the PageRank value is defined in execute (line 7–11).

Once the PageRank for the new iteration is calculated according to Equation 1, validate is called. This method checks whether the convergence criteria is met and if so returns DONE or otherwise COMMIT. Both actions persist the new PageRank so that it is visible to other iterative sub-transactions within the uber-transaction. In case *DB4ML* detects a violation of the isolation level during commit, the changes are rolled back (i.e., the last iteration of the transaction is aborted). After a commit or abort, the sub-transaction is re-scheduled for further iterations.

Algorithm 2: PageRank — Iterative sub-transaction

```
1  function begin (T_State initial_state)
2      tx_state.nodeId = initial_state.nodeId
3      tx_state.nnIds = initial_state.neighbors_nodeIds
4      tx_state.pr = 0
5      tx_state.old_pr = 0
6
7  function execute ()
8      neighbors = read_neighbors(tx_state.nnIds)
9      tx_state.old_pr = tx_state.pr
10     tx_state.pr = calculate_pr(neighbors,0.85) //Eq. (1)
11     update Node table with tx_state.pr
12
13 function validate ()
14     if converged(tx_state.pr, tx_state.old_pr) then
15         return DONE
16     else
17         return COMMIT
18     end
```

6.2 Use Case 2: SGD

6.2.1 Overview. As a second example, we implement stochastic gradient descent (SGD), which is one of the key-techniques for solving large-scale ML problems. SGD randomly chooses a training sample in each iteration and updates the model vector accordingly by estimating the current gradient. We will use a parallel version of this usually sequential algorithm as also done in Hogwild! [30]. In the following, we describe the process of porting the Hogwild! algorithm into *DB4ML* and of applying NUMA optimizations from Hogwild++.

6.2.2 Mapping to DB4ML. To begin with, we explain the relational data model we used to store training samples and model the parameter vector. We then show the pseudo-code for the transactions inside *DB4ML*.

Data Model. The training data and the parameter vector for gradient descent are represented in two tables as shown in Figure 7. The GlobalParameter table (shown left) represents the parameter vector x_v and each element is updated using the Hogwild! [30] updating scheme:

$$x_v \leftarrow x_v - \gamma b_v^T G_e(x) \quad \text{for each } v \in e \tag{2}$$

The Sample table contains all data items from the training-set, where the features X and the labels Y are represented as

columns. Note, in order to enable random sampling from the table, which is required by Hogwild!, we shuffle the training data before starting the uber-transaction. This prevents the database from using sorted data. While in our actual implementation we shuffle the data at the beginning of the uber-transaction, for simplicity of this presentation, we assume that the table is already shuffled before the uber-transaction starts. We also inserted a primary-key column, *randID* and built an index for that column, in order to pick random samples efficiently.

ParameterID	Value
1	0.1
2	0.3
...	...

RandID	X_1	...	X_n	Y_1	...	Y_n
1	...					...
2	...					...
...	...					...

Figure 7: GlobalParameter and Sample Table

Uber-Transaction. In the uber-transactions as shown in Algorithm 3 all constants as well as the asynchronous isolation-level are set to comply with the updating scheme of Hogwild!.

In contrast to the PageRank example, the number of sub-transactions in our pseudo-code is fixed and each sub-transaction is responsible for executing SGD on a partition of the pre-shuffled training data (i.e., `Sample` table). Consequently, the uber-transaction schedules the defined number of sub-transactions and coordinates the assignment of samples. Additionally other required parameters e.g. the number of epochs, or the step-decay, are passed to the sub-transactions. We omit the additional parameters used by Hogwild! for readability. At the end, the uber-transaction again waits until every sub-transaction converged, which is the case when the specified number of epochs has been computed.

Algorithm 3: Hogwild! — Uber-Transaction

```
1  BEGIN TRANSACTION
2      #rows = SELECT COUNT(*) FROM GlobalParameter
3      #subtxs = #cpu_cores
4      numEpochs = 20
5      stepDecay = 0.8
6      stepSize = 5e-2
7      SET SUB-TX ISOLATION LEVEL ASYNC
8      for i = 0...#subtxs do
9          startKey = i * (#rows/#subtxs)
10         endKey = lowKey + (#rows/#subtxs) - 1
11         sub_tx = new sub_ tx()
12         sub_tx.begin(numEpochs, stepDecay, stepSize, startKey,
             endKey))
13     end
14     WAIT //until all sub_tx converged
15     COMMIT
16 END
```

Iterative-Transaction. Algorithm 4 describes how the Hogwild! algorithm is implemented in our iterative sub-transaction interface (see Section 2.3). We only adapted the part which interacts with the training data and model vector, the rest of the code could be seamlessly mapped to our programming model.

Algorithm 4: Hogwild! — Iterative sub-transaction

```
1  function begin (T_State initial_state)
2      tx_state.currentEpoch = 0
3      tx_state.numberEpochs = initial_state.numberEpochs
4      tx_state.stepSize = initial_state.stepSize
5      tx_state.stepDecay = initial_state.stepDecay
6      tx_state.lowKey = initial_state.lowKey
7      tx_state.highKey = initial_state.highKey
8
9  function execute ()
10     for tx_state.lowKey ... tx_state.highKey do
11         rid = randomSample(tx_state.lowKey, tx_state.highKey)
12         sample = SampleTable.getTuple(rid)
13         update = Hogwild_ModelUpdate(sample) // see Hogwild! for
             details
14         for u in update do
15             GlobalParameter.updateTuple(u)
16         end
17     end
18     tx_state.stepSize *= initial_state.stepDecay
19     tx_state.currentEpoch++
20
21 function validate ()
22     if tx_state.numberEpochs reached then
23         return DONE
24     else
25         return COMMIT
26     end
```

In the `begin` function, the following initial state of the algorithm is defined: the number of epochs until completion, the subset of training samples, the step size, the step decay and the number of epochs The range (lowKey, highKey) defines a subset of the Sample table from which a sub-transaction picks an entry to compute a Hogwild! update.

The function `execute` is called in every iteration (i.e., once per epoch for each sub-transaction). First a random *RID* is picked from the predefined range using the index on *RandID*. Then the corresponding sample is retrieved using `table.getTuple(rid)` (line 12). `Hogwild_Model-Update(sample)` returns only changed entries of the model vector using Equation 2. These updates are then written to the `GlobalParameter` table using `table.update-Tuple(tuple)`, which corresponds to updating a single tuple in the table. After that the step-size is increased and the epoch counter is incremented.

After each call of the `execute` function, the `validate` function checks if the algorithm converged. If the convergence criteria is met (i.e., the maximum number of epochs is reached), DONE is returned. If not, COMMIT is returned and the sub-transaction is re-scheduled.

NUMA Optimizations. In order to scale across NUMA-regions, we implemented the NUMA optimizations from Hogwild++ [38] in our iterative sub-transaction. The `Global-Parameter` table is split round-robin over each NUMA region. This ensures equal distributed write load on all memory controllers. Further details are omitted due to brevity. However, more details are explained in [38] and can be applied to *DB4ML*'s programming model.

Dataset	Nodes	Edges
gplus	107,614	18,112,696
patents	3,774,768	22,637,404
pld	39,497,204	704,376,276

Table 1: PageRank Datasets [18, 19, 34, 35]

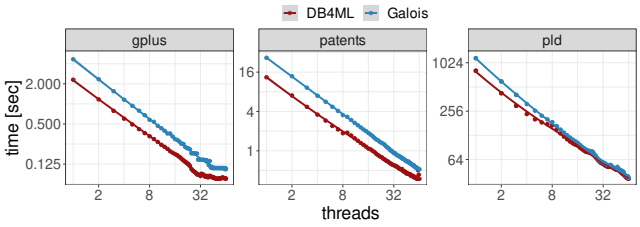


Figure 8: PageRank — *Runtime for* 1 *to* 64 *Cores.*

7 EXPERIMENTAL EVALUATION

We evaluated *DB4ML* with regard to its overall performance — measured in runtime — and its scalability across NUMA regions. We also studied the runtime-accuracy trade-off between varying isolation levels. In our evaluation, we compare *DB4ML* with three state-of-the-art implementations that are optimized for the particular workload. To that end, we use Galois [26], an optimized graph-engine, for PageRank. The SGD use case is evaluated against Hogwild!, respectively a NUMA-optimized version called Hogwild++ [30, 38]. All these systems implement parallel manually-tuned engines for the given workload and run outside a database system. Both baselines — Galois and Hogwild++ — are NUMA-aware and range partition the incoming data across NUMA nodes. In order to be comparable, we employ the same partitioning scheme as our baselines do. The goal of the evaluation is to analyze whether our database kernel for iterative transactions achieves a competitive performance despite running inside a database without any major overhead for iterative transaction handling.

7.1 Evaluation Environment

The experimental evaluation was conducted on a multi-socket server with 64 cores, organized into eight NUMA regions each with an $Intel^{(R)}Xeon^{(R)}E7-8830$ CPU at 2.13 GHz (with Hyper-Threading disabled). The server is running Ubuntu 16.04.03 LTS and has 512 GB of RAM. The prototype of *DB4ML* was implemented using C++14 and compiled using gcc-5.5.0.

7.2 Evaluation of PageRank

We evaluate *DB4ML*'s PageRank implementation as described in Section 6.1 against a PageRank implementation from Galois [26].

As baseline, we used the synchronous pull-based implementation, which uses the same PageRank implementation as *DB4ML* where workers pull PageRank from neighboring nodes to update the PageRank of the node they are responsible for. Unfortunately, Galois does not provide an asynchronous variant for the pull-based PageRank algorithm. Instead, another version (called push-based PageRank) is available that provides an asynchronous scheme. However, we do not compare to this version since it provides bad scalability (i.e., the NUMA-locality of this version is bad since many writes cross NUMA boundaries) and is not algorithmically comparable to our implementation.

For the evaluation we used multiple graph datasets with different sizes as shown in Table 1.

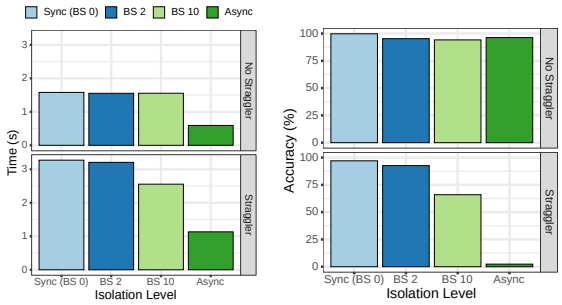
7.2.1 Exp. 1 - Scalability of PageRank. In the first experiment, we investigate the scalability of *DB4ML*'s synchronous isolation level compared to the synchronous version of Galois on our eight socket NUMA system with 64 cores. We designed our PageRank algorithm carefully to match Galois convergence criteria and thus results in the same ranking and PageRank values. For showing the scalability of *DB4ML*, we increase the number of cores from 1 to 64 and report the runtime.

Figure 8 compares the scalability on our three datasets *gplus*, *patents* and *pld*. The y-axis (log-scaled) shows the runtime in seconds and the x-axis (log-scaled) the number of cores being used. We found that both systems scale equally well on all datasets. There is a small advantage for Galois when using more cores (48+). The reason is that *DB4ML* has a minimal overhead due to our tuple format and hence the QPI-links get earlier saturated.

Moreover, although both systems scale similarly well there is a difference in runtime. *DB4ML*'s runtime is lower, which is due to its batched execution. In this experiment, we use a batch size of 256 which is optimal for all data sets. We discuss the effects of using different batch sizes on the runtime of *DB4ML* in Section 7.2.3 in more detail.

7.2.2 Exp. 2 - Effects of ML-Isolation Levels. Next, we examine the effect of using different isolation levels on the average runtime and accuracy of PageRank. Since PageRank is known to converge under asynchronous execution, we expect that it outperforms the other schemes (bounded-staleness and synchronous) since it has the least overhead if runtime across sub-transactions is the same. However, in case of stragglers that might result from skew in the connectivity of the graph, the bounded-staleness scheme might be beneficial for accuracy since it mitigates that sub-transactions read too stale values and thus might not make any progress (or even stop too early).

For this experiment we thus ran PageRank on the *gplus* data set using a fixed number of iterations. We executed the experiment with 4 cores in total and used all isolation



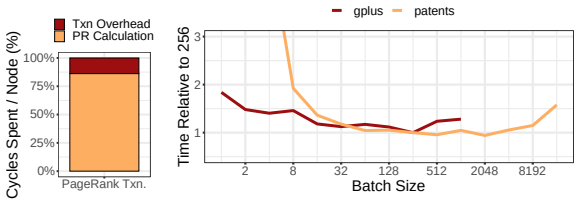
(a) Runtime (b) Accuracy

Figure 9: Effect of ML-Isolation Levels

schemes of *DB4ML*. For bounded-staleness we used the staleness factors 2 and 10. We executed the experiment with and without having a straggling worker. In order to simulate a straggler we let one worker randomly sleep between 0 and 100*ms* per iteration. We report the resulting average runtime per worker and the pair-wise accuracy. The pair-wise accuracy is based on the synchronous converged results, meaning we compare the ranking of each isolation level with our synchronous version (which is assumed to produce the correct ranking). The results of this experiment are shown in Figure 9.

Figure 9(a) shows the runtime without a straggler (upper plot) and with a straggler (lower plot). Unsurprisingly, the asynchronous isolation level has the lowest average runtime in both cases since it has no synchronization overhead. However, when looking at the accuracy in Figure 9(b) we clearly see that the asynchronous scheme has the lowest accuracy in the presence of a straggler (i.e., it only reaches 2% of the pair-wise accuracy compared to the synchronous scheme which on the other hand has the highest runtime). Further, we can observe that bounded-staleness represents a trade-off between these two schemes. For instance, with a staleness of 10, a worker only needs 2.5 seconds (compared to 3.2 seconds for synchronous) on average and the overall accuracy is already at 60%. As a result, bounded-staleness represents a robust trade-off compared to the asynchronous and synchronous execution. It provides a smaller runtime compared to the synchronous scheme and a higher accuracy than the asynchronous scheme for skewed and non-skewed scenarios.

7.2.3 Micro-architectural Analysis. In the following, we conduct a micro-architectural analysis of *DB4ML* when executing PageRank to understand how *DB4ML* is able to provide a performance similar to a specialized engine such as Galois. As mentioned before, *DB4ML*'s transactional execution engine is based on the standard transaction semantics in which transactions need to `validate` and `commit`. Therefore each iterative sub-transaction performs virtual function calls in every iteration. For iterative sub-transaction with a complex computation this overhead is negligible. However, in simple tasks, e.g., PageRank, it becomes significant as it does not contribute to the actual machine learning computation as shown in Figure 10.



(a) Overhead (b) Batch Sizes

Figure 10: Transaction Overhead and Effect of Batch Sizes — *(a) Batch Size=1, 1 Core (b) Runtime measured on 64 Cores*

We initially focus on the additional transaction overhead that comes with *DB4ML* and discuss how we minimized this overhead using our optimizations of the storage and execution scheme as discussed before. Figure 10(a) shows the percentage of cycles spent in one PageRank transaction responsible for one node using the *gplus* data set. As can be seen, 20% of CPU-cycles are spent in transaction related methods and 80% for the actual PageRank computation.

In order to alleviate the overhead of transactions, *DB4ML* makes use of batching as discussed in Section 5. We analyzed the effects of batching in Figure 10(b) which shows the performance for the *gplus* and *patents* datasets with 36 iterations using all 64 cores. We normalized the runtime relative to the runtime with batch size 256. For both data sets, we can see that a batch size of 256 – 512 is optimal and efficiently reduces the runtime even further compared to a batch size of only 1.

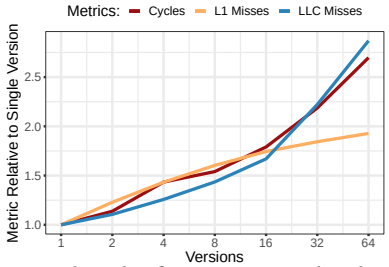


Figure 11: Overhead of Storing Multiple Versions — *CPU metrics relative to a single version*

A second aspect that causes overhead in *DB4ML* compared to a specialized engine such as Galois is that *DB4ML* uses an MVCC-based storage engine. To show the overhead of storing multiple iterative-versions we scaled the number of

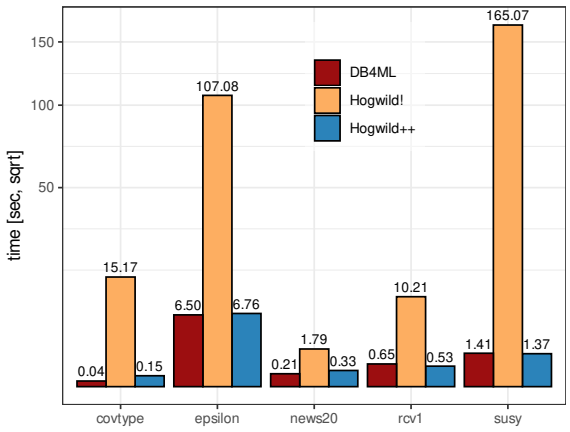
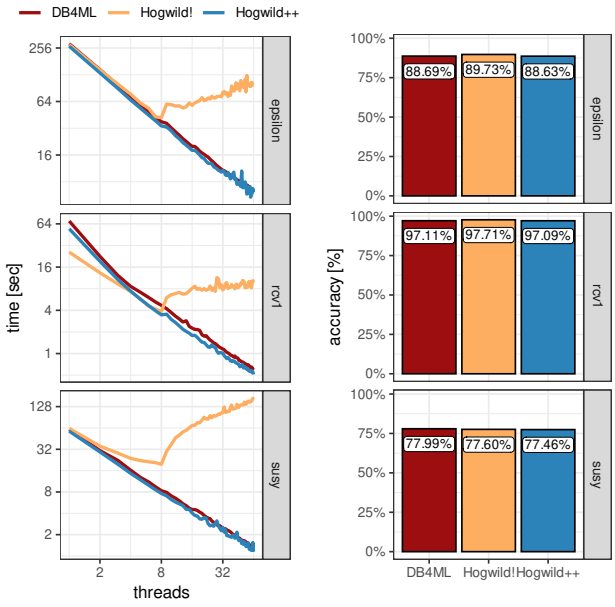


Figure 12: SGD — *Runtime Comparison on 64 Cores.*



(a) SGD Runtime (b) SGD Accuracy

Figure 13: SGD — *Comparison of Scalability Between* DB4ML *and Hogwild! / Hogwild++.*

versions from 1 to 64 PageRank versions. Figure 11 shows the impact of physically storing multiple versions on CPU-cycles, L1-misses and LLC-misses for one iteration of PageRank and one node using *gplus*. The metrics are shown relative to the overhead of storing only a single version. We can see that the computational overhead increases with more versions being stored and causes up to 2.75× more cycles that need to be executed – which is significant. However, as discussed in Section 3 in the best case our optimizations enable us to only store a single version for all ML-centric isolation levels and thus do not need to pay a versioning overhead.

To summarize, by looking at the transaction overhead and the effects of batch processing as well as the optimized storage layout, we found that *DB4ML*'s overhead can be reduced significantly on all data sets used for PageRank. As a result, *DB4ML*'s achieves a performance comparable to specialized engines with no transactional overhead, such as Galois.

7.3 Evaluation of SGD

We evaluate *DB4ML*'s implementation of the Hogwild++ algorithm against two baselines (Hogwild! [30] and Hogwild++ [38]) using the datasets from [38] (see Table 2). Both systems implement a lock-free parallelized Stochastic Gradient Descent, i.e., asynchronous SGD. However, Hogwild! is not NUMA-optimized and hence does not scale across sockets. To alleviate this scalability issue, Hogwild++ introduced a token-based synchronization, see [38] for more details. The following experiments are trained on the same linear SVM task as specified in [38] formula (2). For Hogwild++ and *DB4ML*, we used the same settings as reported in [38] for Hogwild++ and fixed the training duration to 20 epochs.

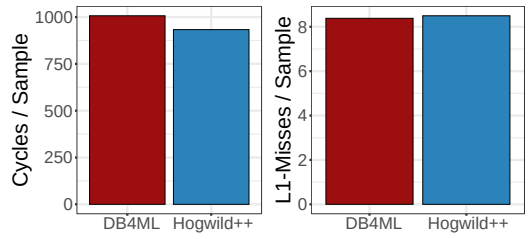
Dataset	Classes	Training set	Test set	Features
rcv1	2	677,399	20,242	47,236
susy	2	4.500,000	500,000	18
epsilon	2	400,000	100,000	2,000
news20	2	16,000	3,996	1,355,191
covtype	2	464,810	116,202	54

Table 2: SGD Datasets [21]

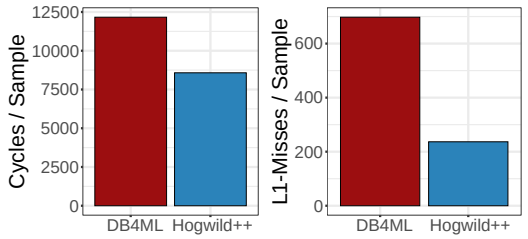
7.3.1 Exp. 1 - Runtime of SGD. In our first experiment, we compare the multi-threaded performance (64 cores) of Hogwild!, *DB4ML*, and Hogwild++ on all five datasets. As already shown by [38], Hogwild! has a significantly higher runtime due to missing NUMA optimizations. This is confirmed by our experiment for all datasets (cf. Figure 12). Further, we can observe that *DB4ML*'s runtime is comparable to that of Hogwild++ on all datasets. This shows that *DB4ML*'s MVCC and lightweight transaction scheme do not impose a high overhead and that our general execution model is competitive to a specialized engine.

7.3.2 Exp. 2 - Scalability of SGD. The main goal of this experiment is to demonstrate how our database kernel scales in comparison to Hogwild! and Hogwild++ when using $1 - 64$ cores. The results of this experiment are shown in Figure 13. The figure shows the runtime in seconds as well as the average accuracy for three datasets (rows) and the systems.

A first observation is that *DB4ML* is always comparable to Hogwild++. For instance, *DB4ML* and Hogwild++ achieve a speedup of 20 for the epsilon data set, while having a comparable overall runtime. Furthermore, we can see that Hogwild! does not scale across NUMA regions, while Hogwild++ and



(a) Few features, *covtype*



(b) Many features, *rcv1*

Figure 14: CPU Cycles and L1-Misses — *For rcv1 and covtype single threaded execution.*

DB4ML achieve a better speedup across NUMA regions. In terms of accuracy, *DB4ML* and Hogwild++ are also comparable since they share the same algorithm. In conclusion, our execution paradigm is able to keep up with the performance of hand-tuned engines, while still being accurate.

7.3.3 Micro-Architectural Analysis. In the previous experiments we solely focused on comparing runtime and accuracy. As we saw, *DB4ML* can achieve a performance similar to that of a specialized engine such as Hogwild++. In the following, we thus compare important differences of *DB4ML* and Hogwild++ on the micro-architectural level. Although both engines implement the exact same algorithm there are two noteworthy differences: (1) the underlying storage scheme and (2) how multiple threads synchronize their work.

Regarding the storage scheme, *DB4ML* uses a MVCC-based storage engine while Hogwild++ uses plain C++ arrays. As discussed in Section 3, for the asynchronous execution scheme that we use for SGD, we only need to store 1 version for each value of the parameter vector. However, even when storing one version only *DB4ML* still has a minor overhead compared to plain C++ arrays as used by Hogwild++ due to maintaining the version information. Moreover, for the coordination of workers Hogwild++ is using a token-ring synchronization which is implemented by passing a `std::atomic`. In *DB4ML*, we mimicked this scheme using an additional relation where each worker has a separate row. We set the flag in the corresponding entry when the worker can synchronize their trained parameters.

In the following, we will analyze the overhead resulting from these two architectural differences using two datasets which differ in the number of features and thus the state that needs to be updated. As already shown in Figure 12, the overhead for the *covtype* dataset is negligible, while the performance overhead for the *rcv1* dataset is much higher. To understand this effect better, Figure 14 shows CPU-cycles and L1-Misses statistics per trained sample for a single-thread execution for those datasets. For *covtype* which has only a few features, most data that needs to be updated fits into the cache and thus results in similar metrics regarding CPU-cycles as well as cache misses as Hogwild++. In contrast, the *rcv1* dataset which has many more features does not fit into the cache. As shown in Figure 14(b), compared to Hogwild++ *DB4ML* spends around 40% more CPU cycles which is strongly correlated to the increased L1-misses as shown in the right side of the Figure.

This observation may lead to the conclusion that *DB4ML* has a much higher overhead for data sets with many features, however this is only true for a single-threaded execution. As shown in Figure 13 for *rcv1*, the multi-threaded execution is comparable to Hogwild++. The reason for a comparable multi-threaded performance is two-folded. First, the number of training-samples are divided among cores, which in turn leads to less training samples per core resulting in a tighter (more cache efficient) training loop. Secondly, each core has its private L1-cache, consequently, the more cores we utilize the more L1-cache is available. Therefore, each worker (pinned on a core) accesses more often the same and less data and is able to utilize the L1-cache more efficiently. In contrast, in single-threaded execution one worker loops over all training-samples and various weight updates result in more L1-misses.

8 RELATED WORK

We first focus on related approaches for integrating ML into databases, then discuss hand-tuned parallel ML algorithms as well as other general data-parallel execution frameworks that can be used to run ML algorithms. Furthermore, there exists a huge body of work on NUMA-optimizations for databases (such as [14, 29]), as well as MVCC-based transaction processing approaches (such as [2, 17, 25, 27]). While all these results are relevant for this paper, the approaches are orthogonal to *DB4ML* and are not further discussed.

ML in the Database: Recent papers suggested to improve ML support inside the database [6, 9, 11–13, 20, 31, 32]. These papers are mainly based on the idea of extending the SQL query engine to better support ML (e.g. with iterative concepts, or to integrate better linear algebra with SQL).

For instance, the BISMARK framework proposed by Feng et al. [9] implements a unified architecture to define data

analytic tasks inside a relational DBMS via user-defined aggregation functions. BISMARK takes a high-level analytic task specification and runs it using user-defined aggregates.

GRFusion from Hassan et al. [11] extended a relational database engine with graph support. The idea of GRFusion is to add an efficient in-memory graph data structure to the DBMS. To process the graph efficiently with the relational database engine, they further propose specialized graph operators. Their concept allows users to run graph queries, relational queries, or queries that mix both workloads.

MADlib [12] is another approach to integrate ML algorithms into a DBMS. MADLib is a library of analytic methods that executes analytical and data-rich computations inside PostgreSQL or Greenplum. The library's core functionality is written as C++ user-defined functions that are invoked via SQL. However, tasks that require iterations are coded in specialized Python driver routines outside the DBMS which makes iterative ML algorithms rather inefficient.

While BISMARK [9] and GRFusion [11] only support a limited set of problems, *DB4ML* supports a general programming model for iterative ML algorithms. Another aspect, of all before-mentioned papers, is that they do not support fine-grained parallelism, which is shown to be a substantial part of effective machine learning [33, 38]. Finally, none of those approaches uses the ideas of transaction processing to integrate ML algorithms into DBMS as we do in *DB4ML*.

Furthermore, many modern data systems often support libraries for analytics and machine learning inside the database. Examples include SAP PAL [23] for SAP HANA, as well as the R integration for Oracle databases or SQL Server. However, these libraries are not tightly integrated but are typically executed in a separate process and thus suffer from the data transfer problem similar to ML solutions that are not integrated into a database.

ML outside the Database: There are many approaches and libraries that provide machine learning capabilities outside the database such as R, scikit-learn, etc. One category that is of interest for this paper is ML implementations that leverage a fine-grained parallelization model and relaxed consistency schemes such as asynchronous execution or bounded-staleness. In the following, we exemplary discuss some of these solutions. We believe that *DB4ML* provides a good starting point to integrate these algorithms into a database.

Niu et al. proposed asynchronous SGD (Hogwild!) [30], where each core updates a global model vector stored in a shared memory simultaneously, without using explicit locks. Hogwild achieves a nearly optimal convergence rate and the authors prove experimentally that it outperforms alternative parallelization schemes based on locking. However, the drawback of Hogwild is that it does not scale well on multi-socket CPUs. Therefore Zhang et al. proposed Hogwild++ [38], a decentralized asynchronous SGD algorithm that achieves nearly linear speedup on multi-socket NUMA systems.

Shang et al. designed and implemented the HSync scheduler through fine-grained parallel vertex transactions [33]. They used a vertex-centric data model where each vertex is stored with its edge-list. For computation, a user-defined-function (UDF), e.g. PageRank, is applied to a vertex and its edges. A transaction contains the computations for one vertex in each iteration.

All these papers implement hand-tuned algorithms outside a database system, and thus they would need to pay data transfer costs. Nonetheless, Hogwild++ proposed interesting ideas for asynchronous execution which is partially integrated into *DB4ML*. Furthermore, HSync, supports a fine-grained parallelism by using the abstraction of transactions similar to *DB4ML*. However, different from *DB4ML*, HSync does not aim to extend a transactional engine with a relational data model with ML support. Instead, they use the ideas of transactions to parallelize the computation in an engine which builds on a native graph data model.

Other Data-parallel ML Frameworks. The general idea to implement iterative machine learning algorithms with small user-defined functions applied via a standard interface to a large data set is also followed within parallel data processing platforms such as Apache Spark [37], Apache Flink [5], and many others [1, 4, 8, 22]. These systems are optimized for distributed processing over large clusters.

9 CONCLUSIONS

In this paper, we proposed a novel approach and presented *DB4ML*, an in-memory transactional database kernel to efficiently execute ML algorithms. *DB4ML* offers a new programming model and an execution engine with isolation levels that provide the concurrency schemes required by modern parallel ML algorithms. A central aspect of *DB4ML* is that the programmer can implement user-defined ML algorithms in *DB4ML* without having to worry about the low-level details of synchronization. That way, the implementation will automatically benefit from all the parallelization and architectural optimizations which *DB4ML* contains - as to be expected from a database system. The experimental evaluation using PageRank and SGD showed that *DB4ML* allows to implement parallel versions of these algorithms with minimal boiler-code and knowledge of optimization techniques, and that *DB4ML*'s transactional layer does not set it behind other existing solutions in regard to execution time.

ACKNOWLEDGMENTS

This work was partially funded by the German Research Foundation (DFG) under grants BI2011/1 and gifts from Mellanox and Huawei.

REFERENCES

[1] M. Abadi et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, OSDI'16, Berkeley, CA, USA, 2016. USENIX Association.

[2] M. Alomari et al. Performance of program modification techniques that ensure serializable executions with snapshot isolation DBMS. *Inf. Syst.*, 40:84–101, 2014.

[3] T. Ben-Nun et al. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Comput. Surv.*, 52(4):65:1–65:43, 2019.

[4] M. Boehm et al. Systemml: Declarative machine learning on spark. *PVLDB*, 9(13), Sept. 2016.

[5] P. Carbone et al. Apache Flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 38(4):28—-38, 2015.

[6] L. Chen et al. Towards linear algebra over normalized data. *PVLDB*, 10(11):1214–1225, 2017.

[7] J. Cipar et al. Solving the straggler problem with bounded staleness. In *HotOS*, volume 13, pages 22–22, 2013.

[8] J. Dean et al. Mapreduce: Simplified data processing on large clusters. In *OSDI*, OSDI'04, 2004.

[9] X. Feng et al. Towards a unified architecture for in-RDBMS analytics. In *ACM SIGMOD*, pages 325–336. ACM, 2012.

[10] H.-H. Fu et al. Damping factor in google page ranking: Research articles. *Appl. Stoch. Model. Bus. Ind.*, 22(5-6), Sept. 2006.

[11] M. S. Hassan et al. Extending in-memory relational database engines with native graph support. In *EDBT*, pages 25–36, 2018.

[12] J. M. Hellerstein et al. The madlib analytics library or MAD skills, the SQL. *PVLDB*, 5(12):1700–1711, 2012.

[13] M. A. Khamis et al. In-database learning with sparse tensors. In *ACM SIGMOD*, pages 325–340, 2018.

[14] T. Kiefer et al. Eris live: A numa-aware in-memory storage engine for tera-scale multiprocessor systems. In *ACM SIGMOD*, SIGMOD '14, 2014.

[15] T. Kraska et al. Datumdb: A data protection database proposal, 2019.

[16] P.-A. Larson et al. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, Dec. 2011.

[17] V. Leis et al. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *ACM SIGMOD*, pages 743–754, 2014.

[18] J. Leskovec. G-Plus dataset. https://snap.stanford.edu/data/egonets-Gplus.html, 2018.

[19] J. Leskovec. Patents dataset. https://snap.stanford.edu/data/cit-Patents.html, 2018.

[20] X. Li et al. Mlog: Towards declarative in-database machine learning. *PVLDB*, 10(12):1933–1936, 2017.

[21] C.-J. Lin. LIBSVM datasets. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html, 2018.

[22] Y. Low et al. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[23] J. MacGregor. *Predictive Analysis with SAP: The Comprehensive Guide*. SAP PRESS, 2013.

[24] C. Mohan, H. Pirahesh, and R. Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, SIGMOD '92, pages 124–133, New York, NY, USA, 1992. ACM.

[25] T. Neumann et al. Fast serializable multi-version concurrency control for main-memory database systems. In *ACM SIGMOD*, pages 677–689, 2015.

[26] D. Nguyen et al. A lightweight infrastructure for graph analytics. In *ACM SIGMOD*, pages 456–471, 2013.

[27] H. H. others. Scalable serializable snapshot isolation for multicore systems. In *ICDE*, pages 700–711, 2014.

[28] L. Page et al. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, November 1999.

[29] I. Psaroudakis et al. Adaptive numa-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *PVLDB*, 10(2):37–48, 2016.

[30] B. Recht et al. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701, 2011.

[31] M. Schleich et al. Learning linear regression models over factorized joins. In *ACM SIGMOD*, pages 3–18, 2016.

[32] M. E. Schüle et al. Mlearn: A declarative machine learning language for database systems. In *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning, DEEM@SIGMOD 2019, Amsterdam, The Netherlands, June 30, 2019*, pages 7:1–7:4, 2019.

[33] Z. Shang et al. Graph analytics through fine-grained parallelism. In *ACM SIGMOD*, pages 463–478. ACM, 2016.

[34] SNAP. Wikivote dataset. https://snap.stanford.edu/data/wiki-Vote.html, 2019.

[35] WDC. PLD dataset. http://webdatacommons.org/hyperlinkgraph/, 2018.

[36] G. Weikum and H.-J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In A. K. Almagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 515–553. Morgan Kaufmann Publishers, 1992.

[37] M. Zaharia et al. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing*, 2010.

[38] H. Zhang et al. Hogwild++: A new mechanism for decentralized asynchronous stochastic gradient descent. In *ICDM*, pages 629–638. IEEE, 2016.