# Integrating Deep Learning Frameworks into Main-Memory Databases

Maximilian Rieger
Technische Universität München
max.rieger@tum.de

Moritz Sichert
Technische Universität München
moritz.sichert@tum.de

Thomas Neumann
Technische Universität München
neumann@in.tum.de

## ABSTRACT

Data analytics has evolved to include complex and computationally expensive methods, especially deep learning. While in the past mostly analytical SQL queries were used, the current landscape is divided into database systems that manage the storage of data, and deep learning frameworks that can efficiently process complex algorithms. The communication between two systems creates an inherent performance bottleneck as it requires exporting data from a database system and importing it into a deep learning framework. The strict separation between both systems also prevents users from combining the use of existing SQL features with complex data analytics pipelines.

In this paper we present an approach to integrate deep learning frameworks directly in a database system to overcome these drawbacks. We take the widely used framework PyTorch and integrate it into our main-memory-first database system Umbra. We demonstrate how PyTorch modules can be used directly in SQL queries and show that this leads to efficient execution of deep learning workloads directly in the database system.

**AIDB Workshop Reference Format:**
Maximilian Rieger, Moritz Sichert, and Thomas Neumann. Integrating Deep Learning Frameworks into Main-Memory Databases. *AIDB 2022.*

## 1 INTRODUCTION

Modern data processing techniques are evolving quickly. In particular deep learning methods have become popular in recent years. For deep learning, specialized systems such as PyTorch [17] and TensorFlow [1] are often used. They offer an easy-to use API that provides several building blocks for deep learning analytics. These building blocks can be combined to formulate complex analytics pipelines for many different use-cases.

The data that is processed by deep-learning frameworks often comes from data-warehouses. Data-warehouses use traditional relational database systems to manage and analyze data which is consolidated from different sources. Database systems provide versatile data manipulation and analysis capabilities using SQL. However, database systems usually provide neither the accelerated computational capabilities required for deep learning nor are current SQL dialects suitable to express complex deep learning models.

Storing data in data-warehouses but processing it in specialized deep-learning systems leads to costly communication between both systems. Moving data between different processes on the same machine or even different machines inevitably increases query latencies. Especially for nested queries, where the output of machine learning models is further processed within the database, communicating between two systems creates a severe bottleneck.

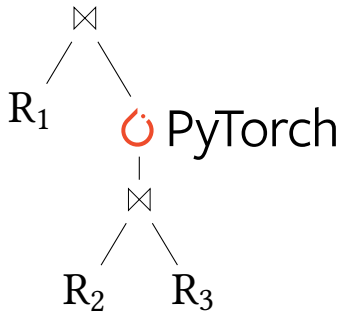



**Figure 1: PyTorch Operator in a Relational Algebra Tree. Our system integrates PyTorch as an operator into the relational algebra of an existing database system. This allows full interoperability of PyTorch and SQL.**

In this work we present our Integration of the deep learning framework PyTorch [17] into our database system Umbra [14]. We eliminate most communication overhead between PyTorch and the database by directly integrating it into the query and execution engine as a relational algebra operator.

We chose the PyTorch framework as it is widely used to design, train, and apply deep learning models. PyTorch's extensive capabilities enable many new use cases for the database system. Additionally, PyTorch allows to accelerate linear algebra operations using GPUs out of the box. The easy to use interface of PyTorch models make the use of these methods available to a broad range of users. Considering that many models might be created using PyTorch anyway makes the possibility to just import these more attractive. Not only does the integration of PyTorch enable these things, it is also vastly less implementation effort to add a framework than to create a custom solution within the database system.

Our relational database system Umbra leverages modern techniques to provide in-memory query performance as long as the working set fits into RAM. Umbra uses just-in-time query compilation to generate highly optimized machine code to achieve optimal throughput. Combining Umbra's excellent query execution speed with PyTorch's deep learning capabilities results in modern, efficient, yet easy-to use deep learning analytics pipelines.

## 2 RELATED WORK

Kraska et al. identified that not only are machine learning based methods an interesting topic for modern database systems, but

```
create function regression(table)
returns table language 'PyTorch' as $$(
    module = 'my_module.pt',
    input_shape = (2),
    inputs = (x, y),
    input_type = float32,
    output_type = float64,
    output_size = 2,
    gpu = true,
    batches = false,
)$$;

select id, name, x, z1, z2 from regression(
    table (
      select id, name, x, y from customers
    )
);
```

**Listing 1: Usage of the PyTorch operator directly from SQL. A PyTorch module can be defined by using `create function`. It can then be used as a table function in SQL queries.**

the tight integration of machine learning algorithms into database systems forms also an interesting topic [9]. We divide existing approaches into separated systems, which of course suffer from communication overhead, and integrated machine learning solutions, which are generally restricted in some ways.

*Connecting Databases to Regular Data Processing.* The deep learning framework TensorFlow provides a way of retrieving data from SQLite database engines [26]. Furthermore, it includes an experimental feature *IODataset*, which can connect to a Postgres endpoint [25].

Katsipoulakis et al. [7] propose a general way of integrating machine learning into SQL and streaming data between storage and processing systems.

Khan et al. present Fireworks [8], which includes data retrieval from databases using SQLAlchemy [24] as a part of their machine learning pipelines.

All of these methods require a transfer of all input data from the database to a processing environment. Especially nested queries, which require multiple environment switches will be negatively affected by this.

*Integrated Machine Learning in Databases Systems.* Microsoft SQL Server's stored procedures use Python and R scripts to execute virtually arbitrary processing, including deep learning [4]. Furthermore, Microsoft SQL Server provides access to many data mining methods including clustering, decision trees, and basic neural networks [2]. Microsoft's ML system Raven adds ONNX models to SQL Server and Spark and shows interesting optimizations for queries using tree-based models [6] [16]. Raven improves ONNX Runtime performance of deep learning models leveraging SQL Server's execution parallelization and model caching.

Google's cloud-based data warehouse BigQuery supports the use of various classical machine learning methods including deep learning models with extended SQL [3]. Furthermore, it allows to deploy trained TensorFlow models. Amazon's sagemaker can be accessed from the database systems Redshift and Athena [11]. In contrast to our work, these systems are divided into distinct services instead and inherently have larger communication costs.

Oracle provides a set of machine learning algorithms including simple neural networks to use directly within SQL queries [15]. Oracle's implementation performs all computations in the database kernel, which avoids the need to move the data for processing.

Sandha et al. present the integration of linear regression and basic neural networks directly integrated to the Teradata SQL Engine using a Python runtime [19]. Their distributed approach is especially interesting for expensive training algorithms.

With TensorDB Kim et al. present a database system which adds tensor data types to the relational model, supporting many tensor operations. Missing automatic differentiation and GPU support render it, however, impractical for neural networks.

Luo et al. [12] extend the parallel database system SimSQL to support linear algebra, which can be used for a number of machine learning tasks.

With MADlib, Hellerstein et al. create a comprehensive open-source machine learning framework based on SQL and user defined functions for Postgres and Greenplum [5]. While the concepts of MADlib are transferable between database systems, it currently only supports Postgres and Greenplum Database.

Schüle et al. introduce lambda expression based tensor processing with automatic differentiation to the main-memory database HyPer [20]. Due to its lack of syntactical abstractions and GPU support it is only of limited use for sophisticated deep learning models. In their follow-up work, Schüle et al. add the above lambda expression based tensor processing to the Umbra database system, adding just in time GPU code generation [21]. The use of just in time compilation of GPU code allows query specific optimizations.

## 3 INTEGRATING PYTORCH INTO UMBRA

To integrate modern machine learning capabilities directly into databases, we integrated the high performance deep learning framework PyTorch into our research database system Umbra [14]. Specifically we leverage PyTorch's implemented deployment method called *TorchScript modules* to apply machine learning models and also algorithms created with PyTorch directly into Umbra's execution engine.

To integrate PyTorch into Umbra, we extend the relational algebra by adding a new operator called *PyTorch operator*. Users can define input values to their modules using standard SQL, and use the outputs of the module just like results of nested queries. Hence, it is also possible to alternate between regular SQL data processing and TorchScript modules within a single query. Furthermore, the user can choose to execute any module on the GPU, leveraging the vast performance advantage of vectorized computation on GPUs.

Listing 1 shows how a TorchScript module can be used directly from SQL. An existing module can be added to the database system by using a `create function` statement. This new user defined function takes a table as an argument, as indicated by the `table ( ... )` syntax, and returns a new table as result. It contains the path to the module file and some additional parameters that specify its input

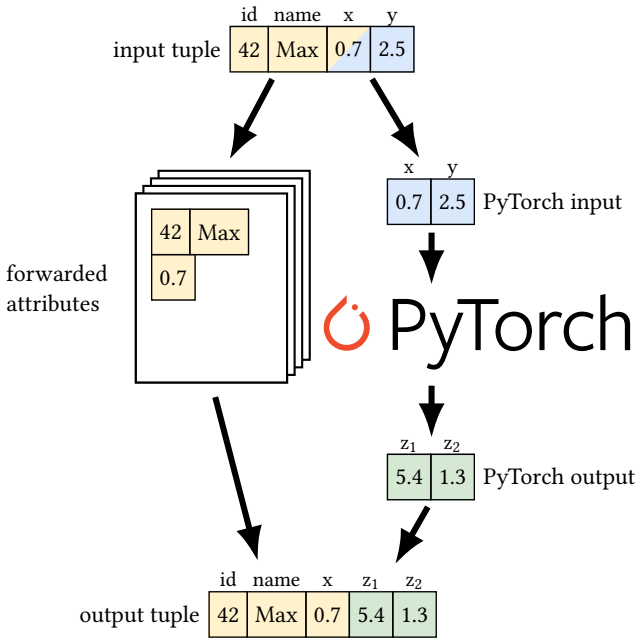



**Figure 2: Conceptual flow of tuples processed by the PyTorch relational algebra operator. The attributes of every input tuple are stored in a temporary buffer for all forwarded attributes and/or passed to the PyTorch module. The output tuple consists of the forwarded attributes and the output of the PyTorch module.**

and output types. SQL queries can use the module like a table function. In our example the TorchScript module takes some metrics of a customer from the attributes `x` and `y`, runs a regression on them, and finally outputs the two values `z1` and `z2` for every customer. This syntax comes with some limitations: Each scalar value for the input tensor must be stated explicitly as an expression, in this case `(x, y)`. This list of expressions is restructured as a tensor using the `input_shape` information. Each input tuple will be converted to this shape, variable shapes are not supported.

Umbra's query optimizer does optimize the input to the PyTorch operator as well as the query on its outputs separately. Although out of scope for this work, it would be possible to implement a joint optimization of the whole query to allow e.g. predicate pushdown or reordering the operator with regard to joins. This would require the user to ensure that the module does not necessarily require the input defined by the input query, and manually toggle whether optimizations are allowed.

## 3.1 Umbra's Execution Model

Umbra features main-memory performance while still being able to efficiently load data from disk.

Umbra's execution model is based on the tuple pushing model. In contrast to the iterator model, which is widely used in other database systems, operators in this model do not request new tuples from their input operators by calling a `next()` function. Instead, each operator pushes its current tuple to the respective output operator's `consume()` function. Scan operators scan the base relations and push all tuples to their output operator. This operator in turn pushes each tuple further to the next operator. This goes on until an operator in the hierarchy needs to materialize its input, e.g. a join or an aggregation. We call these operators *pipeline breakers*. The sequence of operators between two pipeline breakers forms a *pipeline*. One big advantage of this model is, that tuples are never materialized within a pipeline, which reduces the load on the memory.

A Pipeline conceptually forms a large for-loop, which iterates over all tuples and then performs all operations on it, until the resulting tuple gets materialized. To saturate the main-memory bandwidth, Umbra uses code-generation to translate these loop structures directly to efficient machine code [13].

This generated code can scale well even on systems with hundreds of cores. Umbra's query execution system leverages morsel-driven parallelism [10] to achieve this scalability. Several threads can execute the same pipeline concurrently. Each thread works on its own *morsel* of the input data and materializes outputs in its own data structures. After all threads executing a pipeline finished, the output data structures are merged to a single data structure.

## 3.2 TorchScript Modules

TorchScript modules are a way to export models created with PyTorch and Python and call them from C++ code. We consider this a good approach to enable deep learning engineers to deploy their models to our system without changing their development workflow.

A TorchScript module takes a PyTorch `Tensor` object as input. In the example depicted in Figure 2, the input tensor consists of the two float values `x` and `y`. To apply the module to several elements at once, the values of multiple elements can be concatenated to a single tensor. When a module is applied to an input tensor it outputs a single `Tensor` object, which holds the resulting values for each input element. In Figure 2 the resulting values for the input tuple are $z_1$ and $z_2$. All necessary computations and model parameters to execute the module are stored as a part of the module itself. By applying Torchscript modules, our operator supports inferencing. We consider updating or training new modules out of scope for this work.

The versatility of TorchScript modules allows us to use them not only for deep learning models. We can implement any algorithm as a TorchScript module. PyTorch executes modules using a light-weight interpreter, which supports a subset of Python. Hence, it is possible for users to implement and use algorithms through our operator, but we do not implement full support for user defined Python functions. E.g. it is not possible to use any Python libraries except PyTorch and Python's builtin math module. Of course, although this interpreter is optimized, it is slow compared to optimized and compiled C++ code. However, if tensor operations make up the majority of the workload, the module can benefit from PyTorch's efficient vectorized implementations. This includes a large set of common machine learning algorithms, which can be executed quickly. So while implementing merge-sort as a TorchScript

module may be extremely slow, we will show that e.g. the K-Means algorithm can be implemented well as a TorchScript module.

## 3.3 Dataflow Schema

Every tuple that is processed by the PyTorch operator is processed conceptually as shown in Figure 2. Some or all attributes of the operator's input may be used directly by a TorchScript module. So, these attributes are taken from the input and passed to the module. Some attributes will not be used by the TorchScript module directly but the output of the PyTorch operator should still contain them. These forwarded attributes need to be stored independently from the TorchScript module. Later, they are combined with the output of the TorchScript module to generate an output tuple. It is possible for attributes to be forwarded and passed to the TorchScript module at the same time, such as the attribute x in Figure 2.

This dataflow schema fits well into Umbra's tuple pushing model. The PyTorch operator forms a pipeline breaker, which materializes both, the input values for the TorchScript module and the forwarded attributes. Then, it applies the module in some way to all input tuples and stores the respective outputs. Finally, it iterates over the materialized forwarded attributes, finds the respective output values for each tuple and pushes the combined output tuple to the output operator. An implementation for this rough outline is depicted in Figure 3.

In the following we distinguish two fundamental approaches to applying PyTorch modules to input elements.

*3.3.1 Mini-Batches.* Large deep learning models usually use a lot of memory to process their input, scaling linearly with the number of input elements. To meet memory restrictions, especially on GPUs, we have to apply them only to very small sets of input elements, so called *mini-batches*. Our PyTorch operator can split large input sets into mini-batches and apply the TorchScript module to each batch individually.

*3.3.2 Single Batch.* There are also use cases where it is strictly necessary to feed all inputs within a single batch to the module. For example, our PyTorch implementation of K-Means requires all data points at once to apply proper clustering. Hence, the PyTorch operator also supports creating a single PyTorch `Tensor` from all input elements and then applying the TorchScript module to all elements at once. We call this the *single batch* method.

## 3.4 Dataflow Implementation

For both, the mini-batches and the single batch method, we use a uniform strategy to materialize the input values for the module as well as the forwarded attributes of each tuple. After the materialization, the PyTorch operator implements different strategies for each batching method.

*3.4.1 Uniform Materialization.* We use *chunked lists* which are tailored specifically to Umbra's execution model to materialize incoming tuples. Chunked lists consist of blocks of memory, called chunks. An example of a chunked list is depicted in Figure 4. A chunk can be instantiated with an arbitrary size of memory to store tuples of different, arbitrary sizes. The size of a chunk is store in its header. Each chunk has a header which stores the total size of the chunk and a pointer to the next chunk to form a linked list of chunks. Tuples are stored in the chunk's memory following the header.

During pipeline execution every thread uses a thread-local chunked list to materialize tuples. If a chunk has too little memory left to store the next tuple, we create a new chunk and link it to the previous one. Further tuples are then stored in the new chunk. After the pipeline is done, the thread-local chunked lists need to be merged. Merging two chunked lists is very easy, because we only need to change the next pointer of the last chunk in one list to point to the first chunk of the other list.

When the PyTorch operator consumes a new tuple from its incoming pipelines, it splits the tuple into input values for the TorchScript module and the forwarded attributes. The different parts of the tuple are stored in two separate chunked lists as shown in Figure 3. To keep track of the positions of each tuple, we keep track of the order of incoming tuples and store the index of the first contained tuple in each chunk. Note, that the order is the same in both lists. We need to pay special attention when merging the thread local lists to maintain the same order in both lists. Both lists need to be merged in the exact same order. After merging the tuple index in each chunk needs to be updated to account for previous tuples of other thread local lists.

*Collecting the Input Tuples.* The input values for the module are all materialized in a uniform data type which is set by the TorchScript module, most commonly `float32`. The module also specifies the number of value per input element. So, because the number of attributes that need to materialized for every tuple is fixed and known before query processing, they are arranged in memory like a two-dimensional array.

*Forwarding Tuple Attributes.* In contrast to the restricted format of the module inputs, forwarded attributes can have different types, including strings. Hence, the size of individual materialized tuples can vary as shown in Figure 4. Consequently, the number of tuples per chunk varies, too. Because the materialized tuples have variable sizes, it is not easily possible to directly fetch a tuple at a given index. For this reason we store the ordinal position of each chunks first tuple in each the chunk header for both lists.

*3.4.2 Processing Mini-Batches.* The first of our two approaches to apply TorchScript modules is using mini-batches. In this approach we repeatedly load a small subset of the tuples as a PyTorch `Tensor` object to which we apply the module. As our input values are already stored in an array-like manner within chunks, the chunks can be used directly as inputs to a TorchScript module.

*Applying Modules to Mini-Batches.* To meet the possible memory constraints of applying TorchScript modules, we apply the module sequentially to all mini-batches. Usually, mini-batches are much smaller than the chunks generated by the materialization of the inputs by the PyTorch operator. Thus, we conceptually split a chunk into multiple mini-batches. As the values in the chunks are already materialized like a two-dimensional array, entire chunks or parts of it can be referenced directly as PyTorch `Tensor` objects without creating copies of the data. We can then apply the TorchScript module to these `Tensor` objects. In turn, we receive the output of the module as a new `Tensor` object and store it in a list. Then, we load the next mini-batch and proceed in the same way. At the end

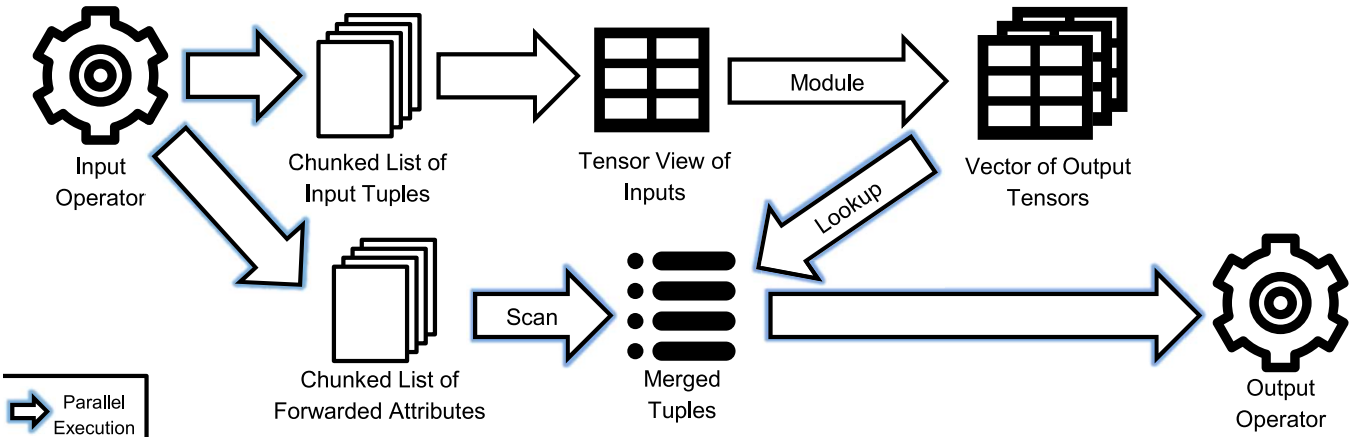



**Figure 3: Dataflow of the PyTorch operator using mini-batches. Incoming tuples are materialized in two chunked lists. One for forwarded tuples and one for inputs to the TorchScript module. The inputs are then loaded as tensors and the module is applied to them. The results are stored in a list. Finally, the operator iterates over the forwarded attributes of all tuples, looks up the respective output values for each tuple and pushes the merged tuple to its output operator.**

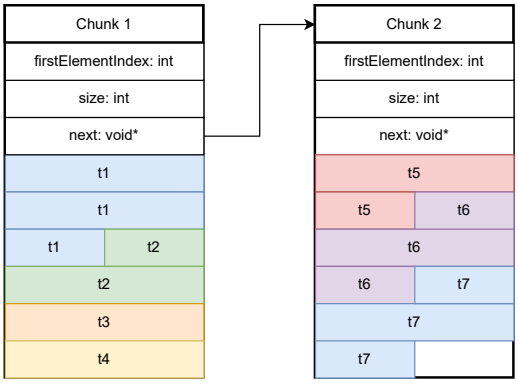



**Figure 4: Visualization of a chunked list of forwarded attributes**

of a chunk, we load a smaller mini-batch if necessary. This part of the dataflow is depicted at the top of Figure 3.

We keep track of the ordinal position of the first tuple in each mini-batch. This allows us to match the outputs of the module to the forwarded attributes.

*Merging Forwarded Attributes with Results.* After applying the TorchScript module to all mini-batches, the PyTorch operator holds two collections of data: the unchanged chunked list of forwarded attributes and the list of output tensors generated by executing the modules. To be able to generate output tuples, which contain the forwarded attributes as well as the output of the TorchScript module, the PyTorch operator needs to merge the two collections again.

To implement this in the morsel-driven parallelization framework used by Umbra, we iterate over the materialized list of forwarded attributes using several threads. As described above, we already keep track of the ordinal positions of the forwarded attributes and the outputs of the module. Thus, while iterating over the forwarded attributes, we know the absolute position $i$ of the current tuple. To efficiently find the module outputs for the tuple at position $i$, the worker threads exploit the fact that the order of elements in both lists is equal and strictly ascending. Also, the output tensors are collected in a vector of references to these tensors. Together, this means that a binary search over the vector of output tensors can be used to efficiently find the module output tuple with index $i$. At the end, the forwarded attributes from the current iteration are combined with the output attributes found by the binary search to form a new output tuple of the PyTorch operator which is passed to its parent operator.

To avoid performing a binary search for each tuple, we maintain the current position within the tensor so that we can access the immediately following element in constant time. Finding the output of the next tuple is done by simply accessing the next element in the current `Tensor` object. When the current tensor contains no more tuples, the next tensor in the vector of tensors is selected, still without requiring a binary search.

*3.4.3 Processing Single Batch.* As mentioned in Section 3.3.2, we want to support applying the TorchScript module once to all input values at the same time. This is required to implement and apply some algorithms, e.g. K-Means, as TorchScript modules.

*Moving to Single Batch.* To load the input values for all tuples as a PyTorch `Tensor` object, they need to be stored contiguously in memory. Like when using mini-batches and as depicted in Figure 3, initially they are materialized in a chunked list. Since our execution runs multi-threaded and the query execution does not know the number of tuples the PyTorch operator will receive in advance, it is not possible to materialize them in contiguous memory directly.

Only after the entire input is collected, the exact number of tuples is known. As the inputs are distributed over several different chunks, but the TorchScript module expects one contiguous are of memory,

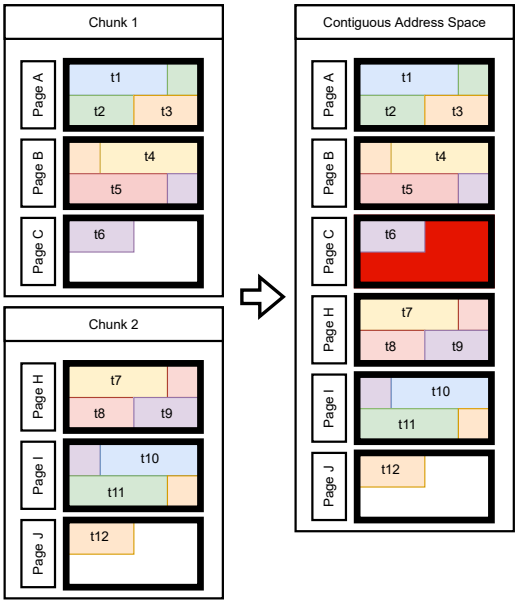



**Figure 5: Visualization of alignment problems that arise, when the memory pages of all chunks are simply mapped to a contiguous address space. As memory page C is only partly filled, there remains a gap after tuple t6 in the new address space.**

we need to combine the chunks. To avoid unnecessary, expensive copies of memory by copying the contents of all chunks into a newly allocated array, our implementation uses a sophisticated memory remapping strategy.

*Memory Mapping.* The `mmap` system call in POSIX systems provides a way to map the contents of a file to a virtual memory address region. It allows us to choose the virtual address we want to map the file to. It is also possible to use `mmap` multiple times on the same file resulting in distinct virtual memory address for the same file contents.

We can use this feature of `mmap` to efficiently create a contiguous area of virtual memory for a single `Tensor` object without using expensive copy operations: First, we store all collected module input data in a memory mapped file. Then, we map this file again to a new virtual memory address space in a suitable order. The resulting virtual address space can be viewed as any other contiguous memory region.

As we do not need actual files on disk, we create files which only live in main memory and do not interact with the file system with the `memfd_create` system call. This avoids performance impacts of the file system and disk access.

*Alignment Requirements.* Even though this memory mapping technique seems to perfectly fit our use case, there are some constraints, which make it non-trivial to use. Specifically we are faced with two distinct alignment requirements.

(1) The address at which `mmap` places a memory mapped file needs to be a multiple of the operating system's page size. In our case, this is 4096 bytes.
(2) Each element in a `Tensor` object must be placed at an offset which is a multiple of the size of a single element, starting from the first element in the tensor. An element consists of a fixed number of fixed-size values.

Additionally, we need to prevent gaps between elements in the `Tensor` object, as we would apply the module to these "gap values", too, which would alter the results for some algorithms such as K-Means. Figure 5 displays a case where plain mapping of all memory pages results in an alignment error. In this example, we map the pages of Chunk 1 at the beginning of our address space and the pages of Chunk 2 at the end of our address space. The last tuple `t6` in Chunk 1 does not end at the end of its memory page `C`. The next tuple `t7`, which is stored in Chunk 2 is located at the beginning of page `H`. Even after we mapped page H right after page `C`, there is still a gap between `t6` and `t7`.

*Hybrid Mapping and Copying.* To meet alignment requirements and avoid copying as much as possible, we use the following algorithm: We create a list of references to all chunks in the chunked list and sort it by their size. To avoid needing to copy the largest chunks, we start with the largest chunk and map it into our new, contiguous memory address space. Then, we iteratively take the next largest chunk and map it to the end of the contiguous address space. Due to the alignment requirements, this creates gaps between the individual chunks, as depicted in Figure 5. We use the smallest chunks to fill these gaps by copying their contents into the gaps. We call this strategy the *hybrid* method, because it uses both memory mapping and copying for some values.

Figure 6 shows how the hybrid method is applied to a set of three chunks. First, the largest chunk (Chunk 3) is selected and mapped in to the new address space. The tuples `t13` to `t24` are now at the beginning of our new address space. After `t24` there is still some space left in the memory page, so we cannot map the next chunk directly. We use tuples from Chunk 1, the smallest chunk, to fill the space. As the end of page `T` does not align with the end of the copied tuples, we insert a new memory page `X` to copy tuples to. After copying `t1` - `t4`, the end of a tuple aligns with the end of a memory page. Now we can map the contents of Chunk 2 after the current position without creating a gap. The remaining tuples of Chunk 1 `t5` and `t6` are copied to the end of the address space.

As the memory mapping system calls have noticeable cost, mapping the contents of a chunk to a new virtual address space does come at some cost too. Mapping the largest chunks and copying the smallest, ensures that we map chunks, for which the advantage over copying is the largest.

*Applying Module to Single Batch.* As we can load all values as a single PyTorch `Tensor` object now, we just need to apply the module once to that tensor and store the resulting tensor with all outputs.

*Merging Forwarded Attributes with Results.* As we change the order of tuples in the hybrid method, we need to maintain some position information to be able to find the output value for each tuple. We divide the new address space we mapped the inputs to

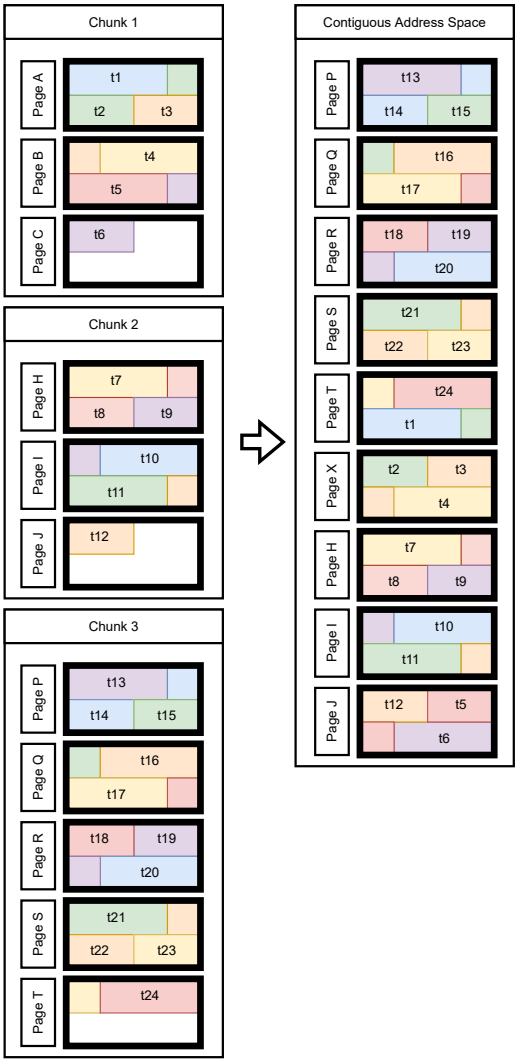



**Figure 6: Effect of the hybrid mapping method. The contents of chunk 3 are mapped first. Then, the tuples t1 - t4 are copied from chunk 1 so the end of a tuple coincides with the end of a memory page. Next, the contents of chunk 2 are mapped. Finally, the remaining elements (t5 - t6) of chunk 1 are copied to the end.**

into segments. Within a segment, all tuples are stored in the same order as before the reordering. For each segment, we store the new position in the new ordering as well as the original tuple position of the first element of the segment in a lookup table. Figure 7 shows the lookup table for the example in Figure 6. The first segment in the new address space includes the tuples `t13` - `t24`. It is stored at the first position, so we store the entry `[t13, 1]` in the lookup table. The next segment is located at position 13 and covers the tuples `t1` - `t4`. Its respective entry for this segment is `[t1, 13]`. Next, there is the segment covering `t7` - `t12` with the entry `[t7, 17]` and finally the last segment with the entry `[t5, 23]`. The entries in the table are sorted by the original tuple position for faster lookups.

When the data paths are merged again, each element can be found with this table. First, we find the largest element $i_{\text{table}}$ in the lookup table, which is smaller or equal than the value $i_{\text{search}}$ we are looking for. Then, we retrieve the new position $i_{\text{new}}$ by adding the respective new position $j_{\text{table}}$ to the distance between $i_{\text{search}}$ and $i_{\text{table}}$ as follows: $i_{\text{new}} = j_{\text{table}} + (i_{\text{search}} - i_{\text{table}})$. For example, we want to find the tuple with the original position `t9` in the lookup table in Figure 7. First, we find the largest element, which is still smaller than `t9`. This is `t7` and the respective new position is 17. Now we compute the new position of `t9` as follows: $i_{\text{new}} = 17 + (\texttt{t9} - \texttt{t7}) = 19$.

When a thread picks a chunk and starts a pipeline, it uses this strategy to look up the new position of the result for the first tuple in the chunk. Using this position it can merge the forwarded attributes with the output value and push the tuple to the next operator. For all tuples in the same segment, we can omit the lookup through the table and simply pick the next value. At the end of a segment we repeat the same lookup strategy to find the position of the next tuple.

## 4 EVALUATION

To evaluate our implementation, we perform several benchmarks using a simple linear regression TorchScript module, computationally more expensive neural network modules, and a K-Means implementation as a Torchscript module.

All benchmarks were performed on a machine with a 8-core AMD Ryzen 7 3700X, clocks locked at 3.6 GHz and 48 GB DDR4 RAM clocked at 2133 MHz. Directories containing relevant binaries and data were cached in RAM with `vmtouch`. Each benchmark was executed 11 times. We reported the median of these runs for all experiments.

### 4.1 Query Stages

We divide the execution of queries into 6 stages.

(1) Query compilation
(2) Loading the required data and the PyTorch / TorchScript module
(3) Computing the input features for the module
(4) Moving the input tuples to a `Tensor` object
(5) Applying the module
(6) Removing temporary data structures and ending the query

This division is applicable to our implementation, but also to other workflows which perform similar tasks. Figure 8 shows the execution of a query using a linear regression model and some input computation. There are three graphs, which depict the benchmarks for different batching methods. *batched* stands for the approach of iterating over the input data in mini-batches described in Section 3.4.2. The methods *copy* and *hybrid* move the input data to a single batch, to which the module is only applied once. While *copy* does this by simply coping all inputs to a contiguous memory region, *hybrid* uses the memory mapping method described in Section 3.4.3.

The plot in Figure 8 already gives us some insights into the cost of the individual stages. First, the compilation stage makes up a very little part of the total execution time. Since Umbra uses an adaptive compilation backend, which starts executing the generated IR immediately with a virtual machine, this is expected.

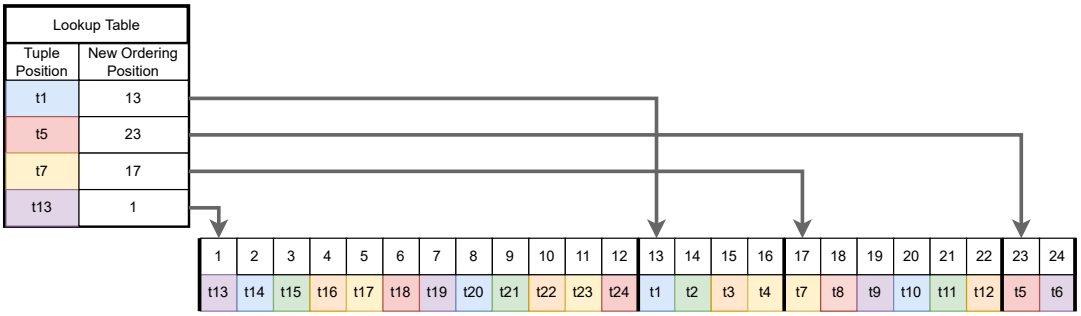



**Figure 7: Visualization of element reordering and maintained position information. This table maintains a mapping from the original tuple positions to the new positions after the order was changed. It suffices to store only the first index of sequential segments in the new ordering. Subsequent tuples can be found with their respective offset.**

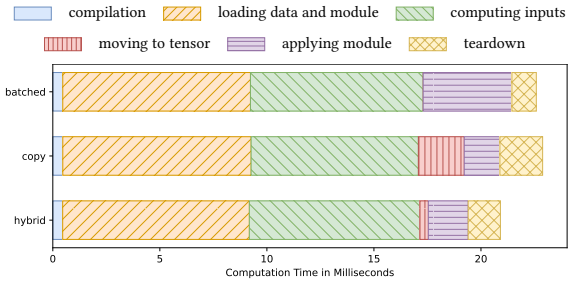



**Figure 8: Linear regression execution for 100 thousand tuples. Execution time of a query split into stages for different Umbra configurations.**

Loading the TorchScript module does, however, take about a third of the total time. This is a significant overhead. Furthermore, this overhead does not depend on the input size. In the following, we will see that it strongly affects performance for small queries.

Computing the inputs for the model in the third stage is done with plain Umbra operators. Nonetheless, we used custom materialization methods in our implementation. Note that the *hybrid* method's use of memory mapped files can lead to slower materialization, which will be part of this stage.

While the *batched* method does not need to move the input data to feed it to the module, *copy* and *hybrid* do spend some time in this stage. Naturally, copying the whole input to a new memory region does take measurable time. But also the *hybrid* method, which copies only very little data takes some time in this stage.

The module application stage is the most interesting for us. We can see that the methods take very different times here. In the *copy* method, we measure the time of a single call of the modules `forward` method. However, this is also the case for the *hybrid* method. Here we can see that mapping chunks of memory into new regions has performance implications when accessing these regions again. Profiling has shown that the number of page faults in the *hybrid* method higher compared to the *copy* method. Nonetheless, the time required to copy the input easily outweighs that overhead in this case. The *batched* method spends much more time in the application stage. This is the case because, in this method, we call the modules `forward` method many times. Even though TorchScript uses a lightweight interpreter to execute the modules, there is some significant overhead for each individual call. We verified that the increase in computation time is caused by the `forward` method. Profiling has shown that the time spent in other parts of this stage is negligible.

Finally, there is the teardown stage, which comprises everything happening after the application of the TorchScript module. This includes the deallocation of allocated memory during the query. In case of the *hybrid* method, it also entails some system calls, which release the memory mapped files. Furthermore, this stage also includes a simple aggregation, which counts all elements. This is needed as a final consumer in the last executed pipeline. Hence, also the position based lookup described in Section 3.4.3 is included in this stage. We did not observe noteworthy effects in this stage and it requires only little time in all experiments.

## 4.2 Linear Regression Performance

As a first benchmark we chose linear regression on the New York Taxi dataset from 2016 [23]. The usual objective for this dataset is to predict the duration of individual taxi trips. However, we do not build a good model for this prediction, as we are only interested in the performance of our data processing workflow. Nevertheless, we use the realistic data points to create a realistic computational task. In the following, we will see that input computation can pose considerable computational cost. As inputs to our models we use:

- Vendor id of the taxi
- Trip start time of the day normalized to a range of [0, 1]
- Count of passengers in the trip
- Coordinates of the start and end point of the trip normalized to a range of [0, 1]
- The air distance between start and end point of the trip normalized to a range of [0, 1]

The normalization and distance calculations make up a realistic input feature computation task.

*4.2.1 Choosing the Best Umbra Batching Mode.* Table 1 shows the execution times of our linear regression query for input set sizes from 10 to $10^8$ tuples. In the case of linear regression, the TorchScript module does not require much memory. Therefore,

| Batching Mode | $10^1$ | $10^2$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ | $10^7$ | $10^8$ |
|---|---|---|---|---|---|---|---|---|
| batched | 12 ms | 12 ms | 13 ms | 15 ms | 23 ms | 49 ms | 246 ms | 2,061 ms |
| copy | 11 ms | 11 ms | 11 ms | 12 ms | 23 ms | 64 ms | 426 ms | 3,870 ms |
| hybrid | 11 ms | 11 ms | 11 ms | 12 ms | 21 ms | 75 ms | 246 ms | 1,666 ms |

**Table 1: Linear regression execution times in milliseconds of Umbra batching methods over different input sizes**

we can choose the batching method freely here, while we need to choose the *batched* method for larger modules. Here we can see that the different computational characteristics of each method cause differing overhead for distinct input sizes. Copying to a single batch is naturally inefficient and listed here only for reference. This can be observed clearly for larger inputs.

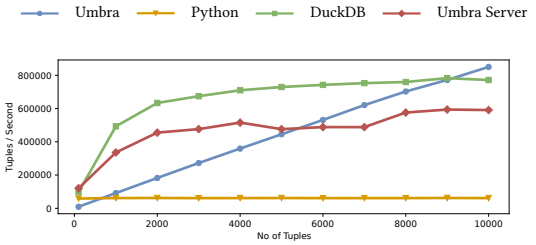
For small input sets, the multiple calls to the modules `forward` function when using multiple batches outweigh even the overhead of the *copy* method. Also for very large input sets the *batched* method performs significantly worse than the *hybrid* method. Nonetheless, for input sizes of $10^6$ and $10^7$, there is no advantage of using *hybrid* over *batched*. Because the performance difference between those methods is so small we default to the *hybrid* method whenever the memory constraints allow it.

*4.2.2 Comparison to Alternative Systems.* To evaluate our implementation, we compare it to other possible systems, which achieve the same results:

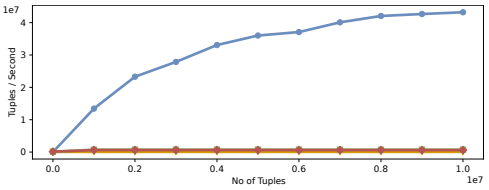
- Python: Load the data from a CSV file and perform all operations in Python.
- DuckDB: The data is stored in-memory in a DuckDB [18] instance. DuckDB is used to compute input features, which are further processed in a Python runtime, which applies the module.
- Umbra Server: The data is stored in Umbra. Umbra computes the input features and transfers them to a Python runtime analogously to DuckDB.

Figure 9 shows the results of executing linear regression queries on the New York Taxi dataset. We can see at Figure 9a that our implementation yields a low throughput for very small input sets. However, the throughput is strongly increasing with larger input sets. When looking at 10a, we can see that the reason for this is that our implementation needs to spend a lot of time loading the module. This constant overhead has a large effect on small queries, but it becomes more and more irrelevant, as the size of the input set and along with the computation time of other execution stages of the query grow larger.

When looking at 9b and 10b, we can see that our implementation yields clearly superior performance over the other systems. Figure 10b reveals furthermore, how most of the computation time for larger input sets is spent. The Python system is unsurprisingly slow and spends significant time on loading the data from CSV files, but furthermore spends most of its time with computing the input features for the module. More interesting insights can be gained from the decompositions of the DuckDB and Umbra Server systems computation time. First, the input computation stage of DuckDB is faster than using the Umbra Server. Umbra's query engine is much faster than DuckDB. So here we can observe, that the inter-process communication between two systems is expensive. The reason for

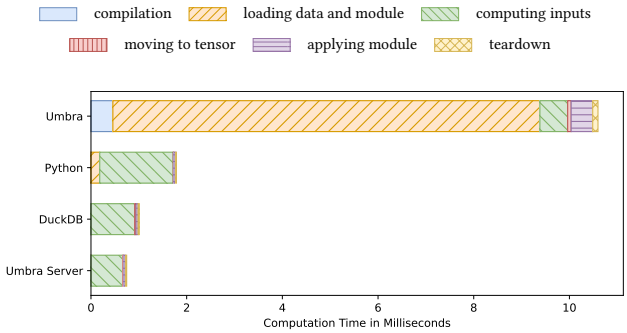



**(a) Input scaling up to 10 thousand tuples**

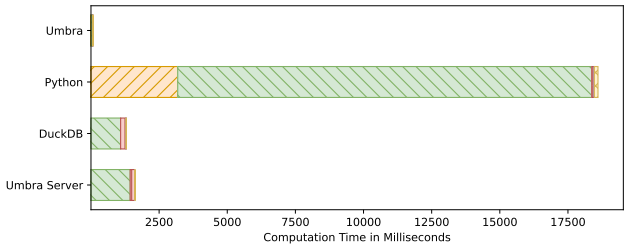



**(b) Input scaling up to 10 million tuples**

**Figure 9: Throughput benchmark of linear regression for different input sizes.**




**(a) Linear regression execution for 100 tuples**




**(b) Linear regression execution for 1 million tuples**

**Figure 10: Runtime decomposition of linear regression.**

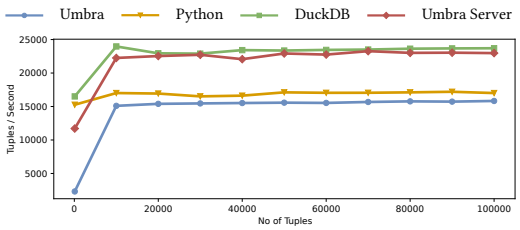



**Figure 11: Throughput scaling of 5 million parameter neural network benchmark in tuples per second.**

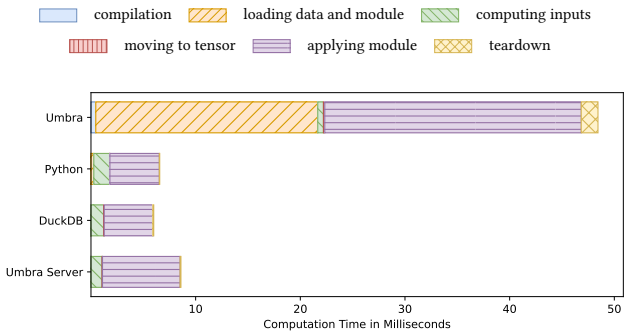
this, is that DuckDB runs within the Python process, which will use the data. This allows the use of shared memory between the database and the Python process. In contrast to that, the Umbra Server needs to send all of its data over a Unix socket.

Another key insight we gain from 10b is that our implementation can execute the whole query faster than Python can create a PyTorch `Tensor` object. Furthermore, the feature computation and moving them from the database to Python takes much longer than that. We conclude from this that integrating computation into the database yields irrefutable performance improvement potential compared to systems using a Python runtime.

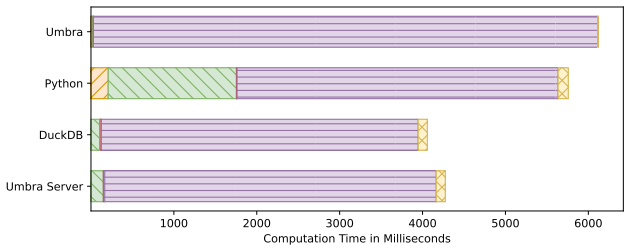
## 4.3 Neural Network Performance

Using a simple linear regression module results in very little computation time spent by applying the module, as linear regression is not very computationally expensive. To explore the behavior of our implementation with more computationally intensive TorchScript modules, we perform the same benchmark with a neural network instead of a linear regression module. This neural network consists of roughly 5 million parameters organized in five large layers. In comparison to modern neural networks, this network is still relatively small. To reduce the slow downs of control flow in Python for the competing systems we apply the module to a single batch of input tuples. As a result, we need to limit the number of tuples to keep the memory usage manageable.

Figure 11 shows the results of our benchmark with input set ranging from 100 to 100 thousand input tuples. It reveals that in this case our implementation has a significantly lower throughput than all other systems we tested. The difference of roughly 40% is not tremendous, but still unexpected. When looking at 12a, we see that the required time to load the module increases in comparison to the linear regression example from about 8 milliseconds to about 20 milliseconds. This is expected, as the neural network module is much larger than the linear regression module. Furthermore, we can see that already for 100 input tuples, the required time to apply the module to the data is several times larger, than the other systems. Figure 12b shows that this is also the case for larger input set. Here, all other parts of the query take almost no time. The module application section of the query consists of a single call to the `forward` method of the TorchScript module in this case, as we are using a single input batch. This leaves no other explanation than the implementation of TorchScript modules in C++ simply performs worse than regular PyTorch used from Python.




**(a) 5 million parameter neural network for 100 tuples**




**(b) 5 million parameter network for 100 thousand tuples**

**Figure 12: Running time decomposition of neural network.**

| | |
|---|---|
| TorchScript C++ | 4.6 s |
| PyTorch C++ | 4.0 s |
| TorchScript Python | 4.1 s |
| PyTorch Python | 4.0 s |

**Table 2: Comparison of execution times in different versions of PyTorch**

*4.3.1 TorchScript Performance Comparison.* It seems plausible that using a TorchScript module in a C++ environment should yield roughly the same performance as as using the same TorchScript module from Python or performing the same operations as regular PyTorch methods in Python. As we clearly observe otherwise in Figure 11, we create a new benchmarks comparing the different possible ways of using PyTorch to apply the neural network module. Table 2 shows that all configurations perform very similarly, except using a TorchScript module from C++, which is about 14% slower.

There is a reasonable explanation why TorchScript might be slower than PyTorch. As TorchScript modules are essentially just code blocks consisting of a subset of Python combined with the parameter data, they cannot be immediately executed. In C++ environments PyTorch invokes an interpreter to execute modules. It is plausible that this interpreter executes the module slower than the regular Python interpreter can. Still, this difference is large and we suspect that there might be potential for optimizations of TorchScript modules in PyTorch. Additionally, the difference we observe when applying the module in our integrated system is significantly larger than in this benchmark. Finding a reason for this is subject to further research.

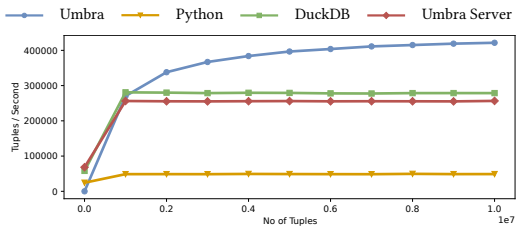



**Figure 13: Throughput of 5 million parameter neural network benchmark using the GPU applied to 10 million tuples**

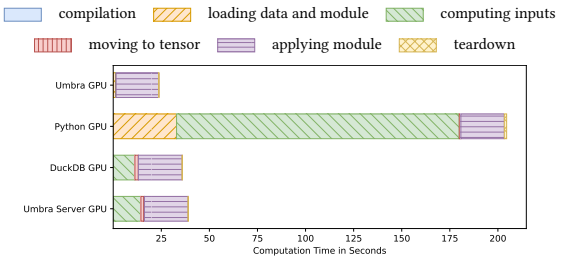



**Figure 14: Running time of 5 million parameter neural network benchmark using the GPU applied to 10 million tuples**

*4.3.2 GPU Performance.* The sub-par performance of our implementation in the neural network benchmark does, however, not necessarily mean that our implementation cannot perform well on queries involving neural networks. An essential method of accelerating neural network computations is using GPUs.

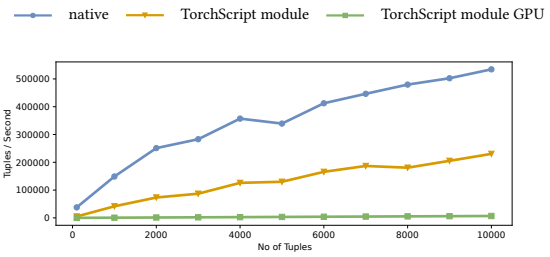
Figure 13 shows that the performance of our implementation can be better than the other tested systems for computationally expensive neural networks. There is still some constant overhead for loading the TorchScript module to the GPU which limits performance for small input sets. Again, the overhead is larger than in a Python runtime of PyTorch. Figure 13 shows that for large input sets our implementation is nevertheless faster than the other tested systems. As visible in Figure 14, the time required to apply the module on the GPU is roughly equal in all systems. Exactly as in the linear regression benchmark, our implementation has a performance advantage in the feature computation and tensor creation stages.

## 4.4 K-Means Performance

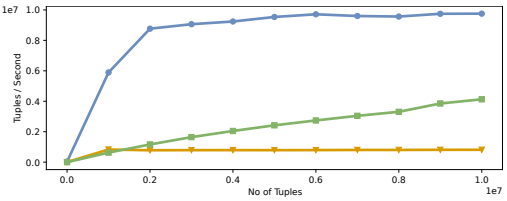
Additionally to linear regression and neural networks, we test our implementation with a module that implements the K-Means clustering algorithm. Umbra does already provide a K-Means operator to perform the algorithm.

The majority of the computation in the K-Means algorithm is spent for computing the distance between all data points and all cluster centers. These distances are then used to determine the nearest cluster for each data point.

There is a fundamental difference between the two implementations we are testing in this step. Umbra's native implementation iterates over the data points and computes the distance to each




**(a) Input scaling comparison up to 10 thousand tuples.**




**(b) Input scaling comparison up to 10 million tuples.**

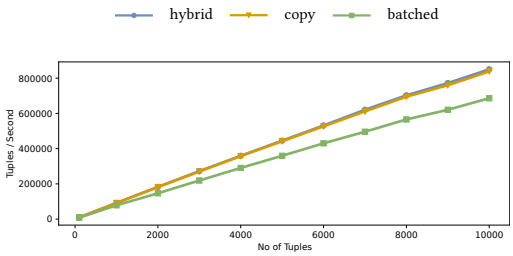
**Figure 15: K-Means throughput scaling for different input sizes.**

cluster center for each point. It then immediately determines the nearest cluster and does only store the new associated cluster id.

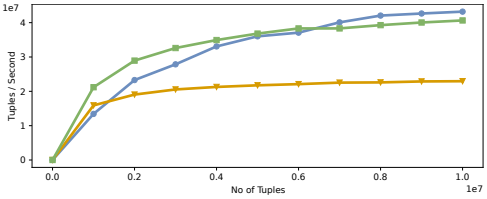
In contrast to that, our PyTorch implementation can only excel by using vectorized functions over large chunks of data. This means that the best way to implement K-Means using PyTorch is to compute a distance matrix between all data points and all cluster centers. The nearest cluster center for each data point is then computed by another PyTorch function using the distance matrix. Creating this distance matrix results in the PyTorch implementation using significantly more memory bandwidth. Therefore, it is expected that our implementation using a TorchScript module does not reach the same performance as the native operator does.

Figure 15 shows that the TorchScript module is roughly two times slower than the native implementation for smaller input sets. This gap widens to a factor of roughly 10 times for input sizes above 400 thousand tuples, after which it remains stable. Our TorchScript module is also capable of running on the GPU. In figures 15a and 15b we can see, that using the GPU yields very low performance for small inputs, but surpasses the CPU version for larger input sets. We also observe that most of the time is spent transferring the data to the GPU. The actual computation of our TorchScript module on the GPU is slightly faster than Umbra's native K-Means implementation. It is, however, important to note that the GPU version cannot be arbitrarily scaled to larger input sizes. As all data points need to be present in memory and available memory on GPUs is usually much smaller than the main memory, this method will stop working for much smaller input sets, than are possible with the native implementation.

Yet, one noteworthy aspect is that implementing a new Umbra operator is a substantial amount of work requiring expert knowledge. Umbra's native K-Means implementation consists of over 1000 lines of code distributed over several header and implementation files. In contrast to this, our PyTorch module for K-Means is only 25 lines long and comparably very simple. Additionally, there

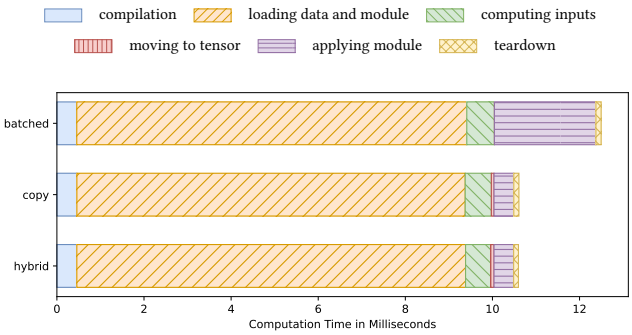



(a) Input scaling comparison up to 10 thousand tuples.

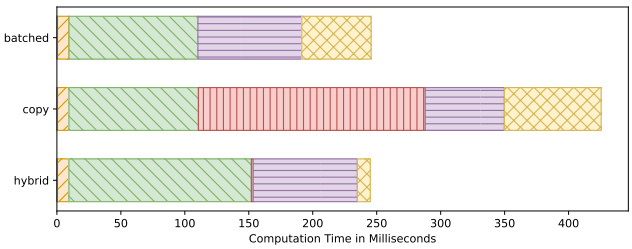



(b) Input scaling comparison up to 10 million tuples.

**Figure 16: Throughput comparison of hybrid method and copy for linear regression**




(a) Linear regression applied to 100 tuples.




(b) Linear regression applied to 10 million tuples.

**Figure 17: Running time decomposition of hybrid method and copy for linear regression**

are many open source implementations of K-Means using PyTorch, which we could just reuse. So while the native operator for K-Means is clearly superior, one would need to evaluate whether implementing a native Umbra operator is worth the additional effort for new algorithms. While user defined operators introduced by Sichert et al. also provide a way for users to integrate new algorithms into the database, the out-of-the box GPU support might still make our approach more attractive [22].

## 4.5 Performance Implication of Using a Single Batch

Section 3.4.3 shows a way of moving data to different virtual address spaces. Our hybrid method relies on mapping memory pages to new addresses and only copying small parts of the data for better speed. To verify that this method does really improve performance, we compare it to just copying all tuples. Figure 16a shows that using our hybrid method does not make a large difference for small input sets, compared to just copying all values. Using a larger input set, however, makes the copy method significantly slower, which can be seen in Figure 16b and 17b.

In Figure 17b we cannot only see that copying does take a lot of time. It also shows that applying the module using the hybrid method is slower than using the copy method. Furthermore, profiling has shown that the time spent handling page faults increased by over 30% when switching from the copy method to our hybrid method. This results in an increase of 30% of time spent in the module application stage for linear regression. The reason for this is that the MMU needs to translate the new virtual addresses, we use to access the input data. Our use of `mmap` which maps these new virtual addresses to existing physical memory changes the page table and puts additional load on the MMU translating virtual addresses to physical addresses.

The additional cost of accessing the new virtual memory addresses is outweighed by the avoidance of copying all data, as visible in Figure 16b. We also observe much smaller impact on the application of the larger neural network module compared to the linear regression module. This reduces the module application time increase to only 1%. Nonetheless, we consider it noteworthy that our hybrid method, albeit useful in our case, does not come completely for free.

# 5 CONCLUSION

In this work we presented how we implemented an integration of PyTorch into Umbra as a relational operator. We show that modern deep learning frameworks can be integrated within complex compiling database systems with relatively low implementation effort. Nonetheless, we added a large set of computational capabilities to the database system allowing not only the application of deep learning models but also providing a way to efficiently execute arbitrary vector operation based algorithms, while supporting the use of GPU acceleration.

We show that our integrated implementation outperforms currently popular alternative workflows. Finally, we conclude that a tight integration of computational operations into database systems is a promising way of enabling faster query processing than current solutions. Even though our implementation focuses on inferencing, we deem this approach also usable for training new models.

# REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard,

Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR* abs/1603.04467 (2016).

[2] Owen Duncan, John Parente, and Tim Sherer. 2021. *Data Mining Algorithms (Analysis Services - Data Mining).* https://docs.microsoft.com/en-us/analysis-services/data-mining/data-mining-algorithms-analysis-services-data-mining

[3] Google. 2021. *What is BigQuery ML?* https://cloud.google.com/bigquery-ml/docs/introduction

[4] Michael Hansen, David Coulter, Gary Ericson, and Mike Ray. 2021. *What is SQL Server Machine Learning Services with Python and R?* https://docs.microsoft.com/en-us/sql/machine-learning/sql-server-machine-learning-services

[5] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library or MAD Skills, the SQL. *CoRR* abs/1208.4165 (2012). arXiv:1208.4165 http://arxiv.org/abs/1208.4165

[6] Konstantinos Karanasos, Matteo Interlandi, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Doris Xin, Supun Nakandala, Subru Krishnan, Markus Weimer, Yuan Yu, Raghu Ramakrishnan, and Carlo Curino. 2020. Extending Relational Query Processing with ML Inference. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p24-karanasos-cidr20.pdf

[7] Nick R Katsipoulakis, Yuanyuan Tian, Fatma Ozcan, Hamid Pirahesh, and Berthold Reinwald. 2015. A Generic Solution to Integrate SQL and Analytics for Big Data.. In *EDBT*. 671–676.

[8] Saad M. Khan and Libusha Kelly. 2019. Fireworks: Reproducible Machine Learning and Preprocessing with PyTorch. *Journal of Open Source Software* 4, 39 (2019), 1478. https://doi.org/10.21105/joss.01478

[9] Tim Kraska, Umar Farooq Minhas, Thomas Neumann, Olga Papaemmanouil, Jignesh M Patel, Chris Ré, and Michael Stonebraker. 2021. ML-In-Databases: Assessment and Prognosis. *Data Engineering* 3 (2021).

[10] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD Conference*. ACM, 743–754.

[11] Edo Liberty, Zohar S. Karnin, Bing Xiang, Laurence Rouesnel, Baris Coskun, Ramesh Nallapati, Julio Delgado, Amir Sadoughi, Yury Astashonok, Piali Das, Can Balioglu, Saswata Chakravarty, Madhav Jha, Philip Gautier, David Arpin, Tim Januschowski, Valentin Flunkert, Yuyang Wang, Jan Gasthaus, Lorenzo Stella, Syama Sundar Rangapuram, David Salinas, Sebastian Schelter, and Alex Smola. 2020. Elastic Machine Learning Algorithms in Amazon SageMaker. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 731–737. https://doi.org/10.1145/3318464.3386126

[12] Shangyu Luo, Zekai J Gao, Michael Gubanov, Luis L Perez, and Christopher Jermaine. 2018. Scalable linear algebra on a relational database system. *IEEE Transactions on Knowledge and Data Engineering* 31, 7 (2018), 1224–1238.

[13] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.

[14] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. www.cidrdb.org.

[15] Oracle. 2021. *Neural Networks.* https://docs.oracle.com/en/database/oracle/oracle-database/18/dmcon/neural-network.html

[16] Kwanghyun Park, Karla Saur, Dalitso Banda, Rathijit Sen, Matteo Interlandi, and Konstantinos Karanasos. 2022. End-to-End Optimization of Machine Learning Prediction Queries. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 587601. https://doi.org/10.1145/3514221.3526141

[17] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[18] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 19811984. https://doi.org/10.1145/3299869.3320212

[19] Sandeep Singh Sandha, Wellington Cabrera, Mohammed Al-Kateb, Sanjay Nair, and Mani Srivastava. 2019. In-database distributed machine learning: demonstration using Teradata SQL engine. *Proceedings of the VLDB Endowment* 12, 12 (2019).

[20] Maximilian Schüle, Frédéric Simonis, Thomas Heyenbrock, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. 2019. In-database machine learning: Gradient descent and tensor algebra for main memory database systems. *BTW 2019* (2019).

[21] Maximilian E Schüle, Harald Lang, Maximilian Springer, Alfons Kemper, Thomas Neumann, and Stephan Günnemann. 2021. In-Database Machine Learning with SQL on GPUs. (2021).

[22] Moritz Sichert and Thomas Neumann. 2022. User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases. *Proc. VLDB Endow.* 15, 5 (2022), 1119–1131. https://www.vldb.org/pvldb/vol15/p1119-sichert.pdf

[23] City of New York. 2016. Taxi and Limousine Commission Trip Record Data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[24] SQLAlchemy authors and contributors. 2021. *The Python SQL Toolkit and Object Relational Mapper.* https://www.sqlalchemy.org/

[25] The TensorFlow Team. 2021. *Reading PostgreSQL database from TensorFlow IO.* https://www.tensorflow.org/io/tutorials/postgresql

[26] The TensorFlow Team. 2021. *tf.data.experimental.SqlDataset.* https://www.tensorflow.org/api_docs/python/tf/data/experimental/SqlDataset