BenchCouncil Transactions on Benchmarks, Standards and Evaluations 4 (2024) 100161

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/

Full length article

TensorTable: Extending PyTorch for mixed relational and linear algebra pipelines

Xu Wen *

Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
University of Chinese Academy of Sciences, Beijing, China

ARTICLE INFO

Keywords:
Linear algebra
Relational algebra
Runtime system

ABSTRACT

The mixed relational algebra (RA) and linear algebra (LA) pipelines have become increasingly common in recent years. However, contemporary widely used frameworks struggle to support both RA and LA operators effectively, failing to ensure optimal end-to-end performance due to the cost of LA operators and data conversion. This underscores the demand for a system capable of seamlessly integrating RA and LA while delivering robust end-to-end performance. This paper proposes TensorTable, a tensor system that extends PyTorch to enable mixed RA and LA pipelines. We propose TensorTable as the unified data representation, storing data in a tensor format to prioritize the performance of LA operators and reduce data conversion costs. Relational tables from RA, as well as vectors, matrices, and tensors from LA, can be seamlessly converted into TensorTables. Additionally, we provide TensorTable-based implementations for RA operators and build a system that supports mixed LA and RA pipelines. We implement TensorTable on top of PyTorch, achieving comparable performance for both RA and LA operators, particularly on small datasets. TensorTable achieves a 1.15x-5.63x speedup for mixed pipelines, compared with state-of-the-art frameworks—AIDA and RMA.

1. Introduction

In recent years, mixed pipelines that integrate both relational algebra (RA) and linear algebra (LA) operators have become increasingly prevalent in fields like data science, artificial intelligence, and real-time analysis. For instance, analytical queries [1–4], which heavily rely on RA operators, leverage LA operators such as matrix multiplication for statistical computations. Meanwhile, machine learning pipelines [5–7], primarily built upon LA operators, utilize RA operators such as join for preprocessing. Real-time tasks [8–10] frequently switch between RA and LA operators to facilitate swift data processing and analysis. However, as shown in Fig. 2, currently widely-used frameworks are unable to support both RA and LA operators while ensuring optimal performance. RA systems [11–13] lack the optimizations for LA operators while LA systems [14–16] do not offer adequate support for RA operators. Cross-framework implementations [17–20] bring extra costs due to data copying and transformations between frameworks and limit optimizations.

The need arises for a system capable of seamlessly integrating RA and LA while delivering robust performance. Many previous works [17,18,21–30] attempt to address this issue. However, most of them prioritize the performance of RA operators but have high execution costs on LA operators and data conversion, thus leading to poor end-to-end performance. Our experiments shown in Fig. 3 corroborate this point. To tackle this problem, we aim to develop a system that seamlessly integrates mixed pipelines and provides a good end-to-end performance. We prioritize ensuring the performance of LA operators while supporting RA operators with comparable performance and reducing frequent data conversion.

Unfortunately, it is not a trivial task due to the inherent disparities between RA and LA. Specifically, RA and LA reflect distinct characteristics from the perspectives of data abstraction, data types, and implementation. In terms of data abstraction, RA operates on relational tables, whereas LA deals with vectors, matrices, and tensors. Regarding data types, RA accommodates both numerical and non-numerical data, whereas LA exclusively handles numerical data. From the perspective of implementation, to ensure the optimal performance of LA operators, frameworks should make full use of parallelism, keep good temporal and spatial locality, and utilize extended instructions properly. However, whether relational tables used in RA systems or DataFrames and RDDs used in general-purpose systems often fall short in guaranteeing optimal locality and instruction utility. Consequently, these frameworks tend to exhibit suboptimal performance when dealing with LA operators.

* Corresponding author at: Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China.
E-mail address: wenxu@ict.ac.cn.

https://doi.org/10.1016/j.tbench.2024.100161
Received 18 December 2023; Received in revised form 17 January 2024; Accepted 7 February 2024
Available online 7 March 2024
2772-4859/© 2024 The Authors. Publishing services by Elsevier B.V. on behalf of KeAi Communications Co. Ltd. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

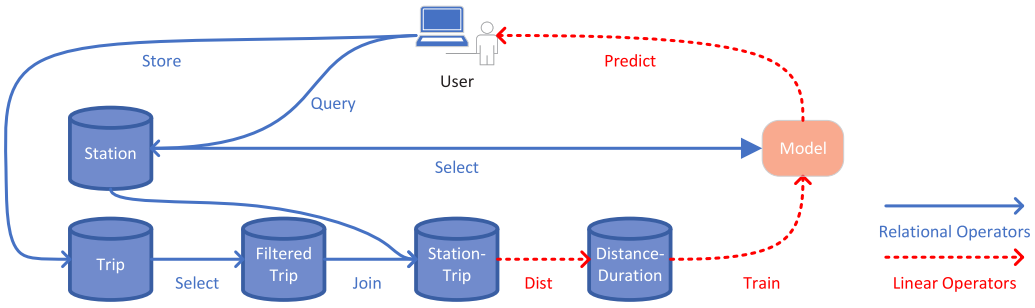


Fig. 1. A public bicycle sharing system leveraging mixed relational algebra (RA) and linear algebra (LA) pipelines to forecast user ride duration.

To prioritize ensuring the performance of LA operators, support RA operators with comparable performance, and reduce data conversion costs, we propose TensorTable as the unified abstraction for both RA and LA, which stores data in a tensor format. For LA, we directly encapsulate vectors, matrices, and tensors within TensorTable. For relational tables used in RA, we convert those non-numerical data into numerical data and use auxiliary dictionaries to preserve the mapping relations, subsequently storing the data in a tensor format. We provide TensorTable-based implementations for RA operators, covering selection, projection, join, group by, and aggregation. We establish a system capable of supporting the mixed pipelines of RA and LA, built on top of a typical LA system—PyTorch. Our work supports both RA and LA operators and achieves comparable performance.

This paper makes the following three contributions:

- We propose an abstraction—TensorTable, to represent both relational and linear algebra.
- We present TensorTable-based implementations for relational algebra operators and achieve comparable performance, especially for small datasets.
- We build a system for combining relational and linear algebra and achieve a 1.15x-5.63x speedup on mixed pipelines, compared with state-of-the-art frameworks—AIDA and RMA.

The remainder of the paper is organized as follows. Section 2 shows the background, motivation, and challenges. Section 3 shows the system overview. Section 4 introduces the design and implementation. Section 5 presents our evaluation. Section 6 illustrates the related work. Finally, we draw a conclusion in Section 7.

2. Background, motivation, and challenges

2.1. Background

Mixed relational algebra (RA) and linear algebra (LA) pipelines are becoming more and more common in recent years. Analytical queries [1–4], which heavily rely on RA operators, leverage LA operators such as matrix multiplication for statistical computations. Meanwhile, machine learning pipelines [5–7], primarily built upon LA operators, utilize RA operators such as join for preprocessing. Real-time tasks [8–10] frequently switch between RA and LA operators to facilitate swift data processing and analysis. Fig. 1 illustrates a typical real-time system—public bicycle sharing system [31] as an example. It leverages mixed LA and RA pipelines to forecast user ride duration and is a latency-sensitive task that requires optimal end-to-end performance. $Station$ stores station information and $Trip$ stores user trip history. The system uses RA operators such as select and join to process $Station$ and $Trip$, then uses LA operators such as calculate Euclidean distance and linear regression to train the predicting model. When users query the system, it uses RA operators such as select and LA operators such as linear regression to forest ride duration. Finally, new trip records are stored in $Trip$. This system integrates both RA and LA operators and requires optimal end-to-end performance.

2.2. Motivation

Despite the popularity of mixed pipelines, contemporary widely-used frameworks cannot support those pipelines and guarantee performance. As demonstrated in Fig. 2, our experiments encompass four typical relational operators – selection, projection, inner join, and group by, and two typical linear operators – matrix covariance and linear regression. Our findings reveal that LA systems like PyTorch cannot support many RA operators such as join and group by. Meanwhile, RA systems like MonetDB exhibit limited support for LA operators with subpar performance. General-purpose systems like Spark and pandas can handle both RA and LA operators, but their performance falls short of the ideal. Cross-framework implementations bring extra costs due to data copying and transformations between frameworks and limit optimizations. The last two columns in Fig. 2 display the conversion time from Spark DataFrame to Matrix and from pandas DataFrame to Tensor, respectively. This process incurs a substantial cost, even 6.7x-18.9x larger than that of RA operators over the same data sizes.

Additionally, we analyze the mixed pipeline introduced in Section 2.1, Fig. 3 shows the performance breakdown. We find that many previous works prioritize the performance of RA operators but have high execution costs on LA operators and data conversion, thus leading to poor end-to-end performance. Hence, there is a pressing need for a system capable of supporting mixed RA and LA pipelines, ensuring the performance of LA and RA operators and reducing data conversion costs, finally delivering optimal end-to-end performance.

2.3. Challenges

There are several challenges to overcome.

(1) Data abstraction. Data abstraction should fit the memory access patterns of its computation. RA computations are primarily based on column-based access, and some computations also rely on row-based access. LA computations are more complex, involving row-based access, column-based access, block-based access, and stride-based access. New data abstraction needs to be compatible with the different memory access patterns mentioned above. RA operators handle both numerical and non-numerical data, whereas LA operators exclusively consist of numerical data. New data abstraction should be able to handle both numerical and non-numerical data. The question that arises is: What constitutes the appropriate data abstraction for such mixed pipelines?

(2) Expressiveness. Mixed pipelines contain RA operators including selection, projection, join, group by, and aggregation as well as LA operators such as matrix multiplication. Existing algorithms may not be suitable for the new data abstraction. In this scenario, it is essential to propose new algorithms tailored to effectively express RA and LA operators.

(3) Performance. Many mixed pipelines are latency-sensitive tasks, which have high-performance requirements. A proper data abstraction that can fit the memory access patterns of its computation is necessary but not sufficient for achieving optimal performance. We should also consider hardware-related and dataflow graph optimizations, as well as data conversion costs.

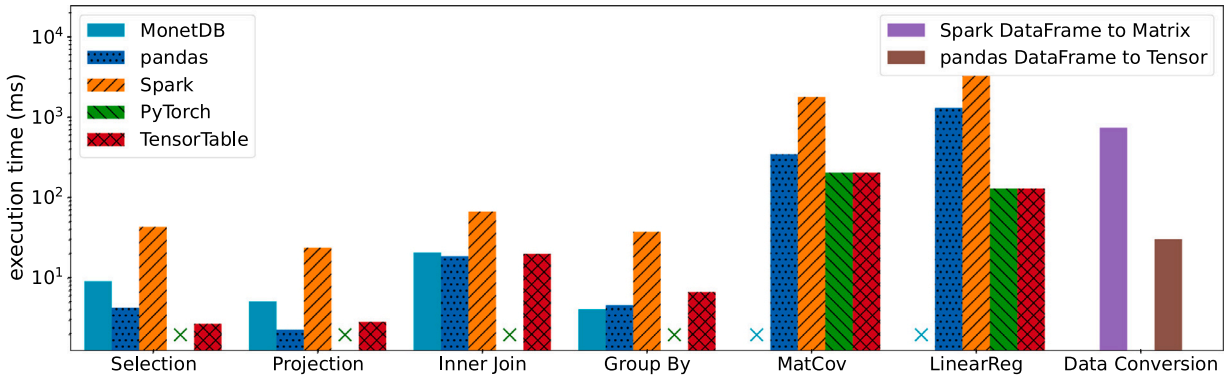


Fig. 2. The performance of popular frameworks on LA operators, RA operators, and data conversion. MatCov and LinearReg are short for matrix covariance and linear regression, respectively. "x" means unsupported. Currently widely-used frameworks are unable to support both RA and LA operators while ensuring optimal performance. Cross-framework implementations face large data conversion costs.

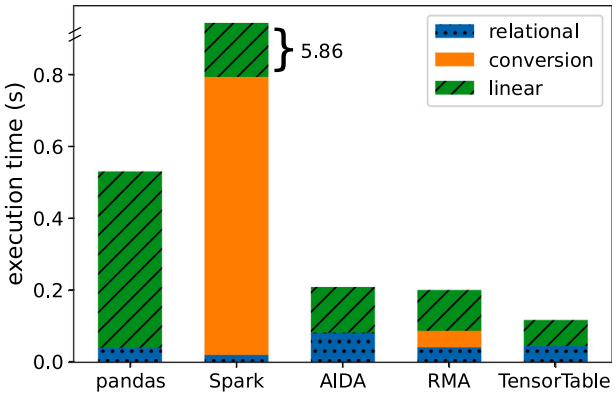


Fig. 3. The performance breakdown of the mixed pipeline in the bicycle sharing system. Many previous works have high execution costs on LA operators and data conversion, thus leading to poor end-to-end performance.

3. The system overview

This section shows the overview of our system.

3.1. Design choices

First, we analyze our design choices.

(1) The core objective is to ensure end-to-end performance for mixed pipelines. As analyzed in Section 2.2, many previous works have high execution costs on LA operators and data conversion, thus leading to poor end-to-end performance. Thus we prioritize ensuring the performance of LA operators, then support RA operators with comparable performance and reduce data conversion costs.

(2) Store all data in tensor formats. Tensors, with their data stored contiguously in the same memory block, enable access to data by computing offsets within the continuous memory block. This characteristic makes tensors suitable for accommodating all memory access patterns used in RA and LA computations, including row-based, column-based, block-based, and stride-based access. Besides, LA operators benefit significantly from key factors such as good parallelism, good temporal and spatial locality, and proper utilization of extended instructions—all of which are inherently tied to tensor abstractions. As a solution, we introduce a tensor-based abstraction called TensorTable, designed to represent both tensors and relational tables, while encoding non-numeric data into numeric representations.

(3) Utilize existing implementations and optimizations for LA operators and propose compatible algorithms and implementations for RA operators. We encapsulate LA operators without modifying their core functionality and supplement them with auxiliary functions to enable their seamless integration within mixed pipelines, not influencing existing hardware-related and dataflow graph optimizations. To support mixed pipelines, we offer TensorTable-based algorithms and implementations for RA operators, successfully achieving comparable performance without compromising the performance of LA operators.

(4) Perform data conversion during initialization instead of runtime to reduce data conversion costs. Frequent data conversion during runtime markedly impacts performance and should be minimized. The only necessary data conversion occurs during initialization, where we transform the original data into TensorTables. No additional data conversions are needed throughout the execution phase. To ensure optimal performance, each operator within mixed pipelines takes TensorTables as input and produces them as output without the need for extra data conversion.

3.2. System architecture

Fig. 4 illustrates the architecture of our system. We utilize TensorTable as the unified abstraction, as defined in Section 4.1. We convert the origin data to TensorTable during initialization and store them in memory. We propose Directed Acyclic Graph (DAG) Intermediate Representation(IR) to represent mixed pipelines, as detailed in Section 4.3.1. The Parser is responsible for translating mixed pipelines into DAG IRs, as elaborated in Section 4.3.2. The Optimizer makes graph-level optimizations, as discussed in Section 4.3.3. The Code Generator generates TensorTable-based operators, as shown in Section 4.3.4. For LA operators, we call PyTorch operators, while for RA operators, we provide TensorTable-based implementations, as defined in Section 4.2. Finally, those operators are executed on top of PyTorch Runtime, as described in Section 4.3.5.

4. The system design and implementation

4.1. The data abstraction: TensorTable

This section introduces our proposed abstraction—TensorTable. TensorTable represents relational tables in a tensor format, allowing for seamless processing. By taking TensorTables as both input and output, all RA and LA operators can be efficiently implemented on tensors. There are two notable benefits to storing data in a tensor format: Firstly, tensors exhibit better data locality compared to row-oriented and column-oriented tables, particularly benefiting LA operators. Secondly, tensors can leverage hardware features and compilation optimizations more effectively. TensorTable seamlessly encapsulates vectors, matrices, and tensors for LA operations, eliminating the need for

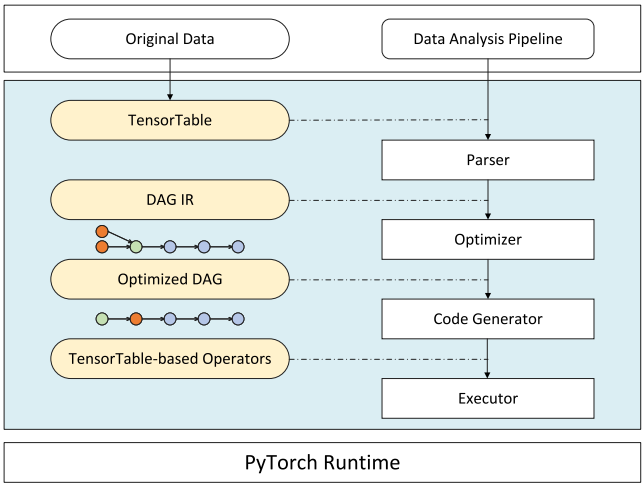


Fig. 4. The TensorTable Framework.

additional data conversions. As for RA operators, we convert relational tables to TensorTables and furnish them with TensorTable-based implementations.

TensorTable comprises four essential elements: *column_names*, *column_types*, *data_tensor*, and *str_dict_list*. Among these, *column_names* and *column_types* are string lists used for storing column names and data types, respectively. All data, whether numeric or non-numeric, is stored within the tensor *data_tensor*. Numeric data from relational tables can be directly stored in *data_tensor*. For non-numeric data, TensorTable encodes them into numeric representations and then stores them in the same tensor. Non-numeric data in relational tables fall into four categories: *date*, *boolean*, *string*, and *text*. For *date* data, we convert them into timestamps, represented as integers. *Boolean* data is converted into 0 or 1. For *string* and *text* data, we encode them as integers and maintain mapping relationships in dictionaries referred to as *str_dicts*.

String is usually used to represent categories, and RA operators such as join can operate on them. For *string* data, we sort them, get their unique elements, and then convert them to integers. As for *text* data, which typically has larger lengths, we map them directly to integers based on their order. Note that *str_dicts* are utilized solely for storing the mapping relations and preserving data order after complex relational transformation, without sophisticated processing such as embedding. Each string column maintains its own *str_dict*, while *str_dict_list* is a list containing these *str_dicts*, with its length corresponding to the number of string columns.

Fig. 5 showcases a TensorTable structure. The left in Fig. 5 shows the original relational table, with m rows and n columns. The right in Fig. 5 presents the corresponding TensorTable. Both *column_names* and *column_types* have a length of n, while *data_tensor* has a shape of m×n. Numeric data, such as that in the *code*, *latitude*, and *longitude* columns, is directly stored in *data_tensor*. The *name* column contains string data, which we map to integers. The mapping relation is retained in *str_dict*1. As this relational table has only one string column, the corresponding TensorTable has just one *str_dict*, namely *str_dict*1, and the *str_dict_list* consists of a single element, represented as [*str_dict*1].

4.2. TensorTable-based RA operators

This section introduces the implementations of RA operators based on the TensorTable abstraction. The supported RA operators include selection, projection, inner-join, outer-join, left-join, right-join, cross-join, group by, and aggregation.

4.2.1. Selection

The selection operator takes rows from a TensorTable that satisfy specified selection conditions.

There are three types of selection conditions:

(1) Comparison between two columns, as shown in Algorithm 1. The input is one TensorTable: $InTable$, two columns $col1$ and $col2$ to compare, and one comparison function $CompFunc$. Comparison functions include *equal*, *greater*, *less*, *greater_equal*, *less_equal*, and *not_equal*. Initially, we extract the corresponding tensors, *tensor*1 and *tensor*2, from InTable based on the specified columns. Both tensors have a shape of [m, 1], where m represents the number of rows. Subsequently, we perform element-wise comparisons between these two tensors, resulting in a mask tensor that utilizes 1 and 0 to indicate whether rows meet the selection condition. Finally, we use *nonzero* operators to get the indices that satisfy the selection condition based on the mask tensor and use the *index_select* function to extract rows from $InTable.data\_tensor$ and assign them to the *data_tensor* of the output TensorTable: $OutTable$. The *column_names*, *column_types*, and *str_dict_list* of $OutTable$ remain the same as $InTable$.

(2) Comparison between one column and a threshold. This type entails a broadcast comparison between the corresponding tensor and the specified threshold. The subsequent steps mirror those in Algorithm 1.

(3) Combination of multiple conditions. These conditions are organized using logical operators such as *and*, *or*, and *not*. We compute each condition's mask tensor and use their *intersection*, *union*, and *complement* to get the combined mask. This combined mask is then used to build the index and extract rows.

Algorithm 1 Selection

Input: InTable: Input TensorTable; col1, col2: Two columns to compare; CompFunc: Comparison function.
Output: OutTable: Output TensorTable.
1: tensor1 ← get_tensor(InTable, col1)
2: tensor2 ← get_tensor(InTable, col2)
3: mask ← CompFunc(tensor1, tensor2)
4: index ← nonzero(mask)
5: OutTable.data_tensor ← index_select(InTable.data_tensor, dim=0, index)
6: OutTable.column_names ← InTable.column_names
7: OutTable.column_types ← InTable.column_types
8: OutTable.str_dict_list ← InTable.str_dict_list

4.2.2. Projection

The projection operator takes columns from the input TensorTable: $InTable$ based on the specified *name_list*, which contains the selected column names, as shown in Algorithm 2. First, we parse the *name_list* to obtain the list of corresponding indices. Then we use the *index_select* function to select columns according to these indices and assign them to the *data_tensor* of the output TensorTable: $OutTable$. The *column_names* of $OutTable$ is set as *name_list*, while the *column_types* and *str_dict_list* of $OutTable$ are the subsets of $InTable$ based on *name_list*.

Algorithm 2 Projection

Input: InTable: Input TensorTable; name_list: Column names.
Output: OutTable: Output TensorTable.
1: index ← parse(name_list)
2: OutTable.data_tensor ← index_select(InTable.data_tensor, dim=1, index)
3: OutTable.column_names ← name_list
4: OutTable.column_types ← subset(InTable.column_types, name_list)
5: OutTable.str_dict_list ← subset(InTable.str_dict_list, name_list)

4.2.3. Join

The join operator combines columns from two or more input TensorTables, producing a new output TensorTable. We take inner-join as an example, as shown in Algorithm 3. The input is two TensorTables:

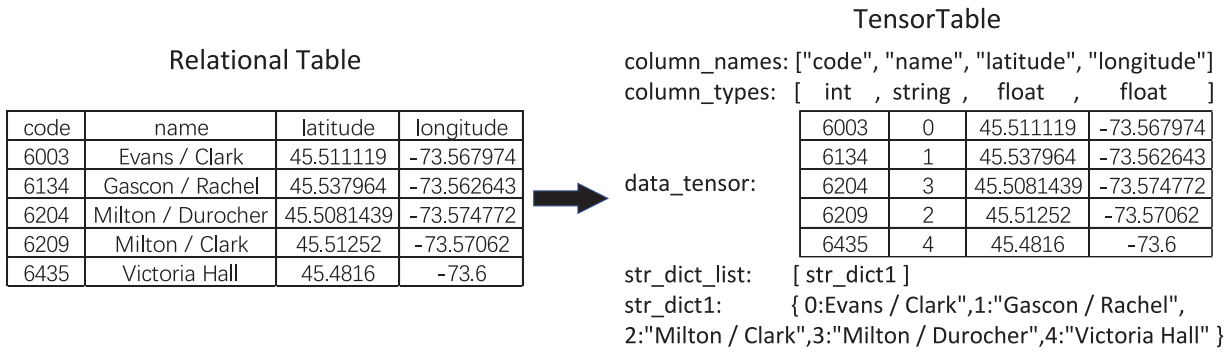


Fig. 5. TensorTable stores a relational table in a tensor format, mapping non-numeric values to numeric ones. For a relational table with m rows and n columns, the corresponding TensorTable consists of two string lists *column_names* and *column_types*, used to store the column names and data types, a m×n tensor *data_tensor* to store data, and a list of auxiliary dictionaries, *str_dict_list*, used to preserve the mapping relationships.

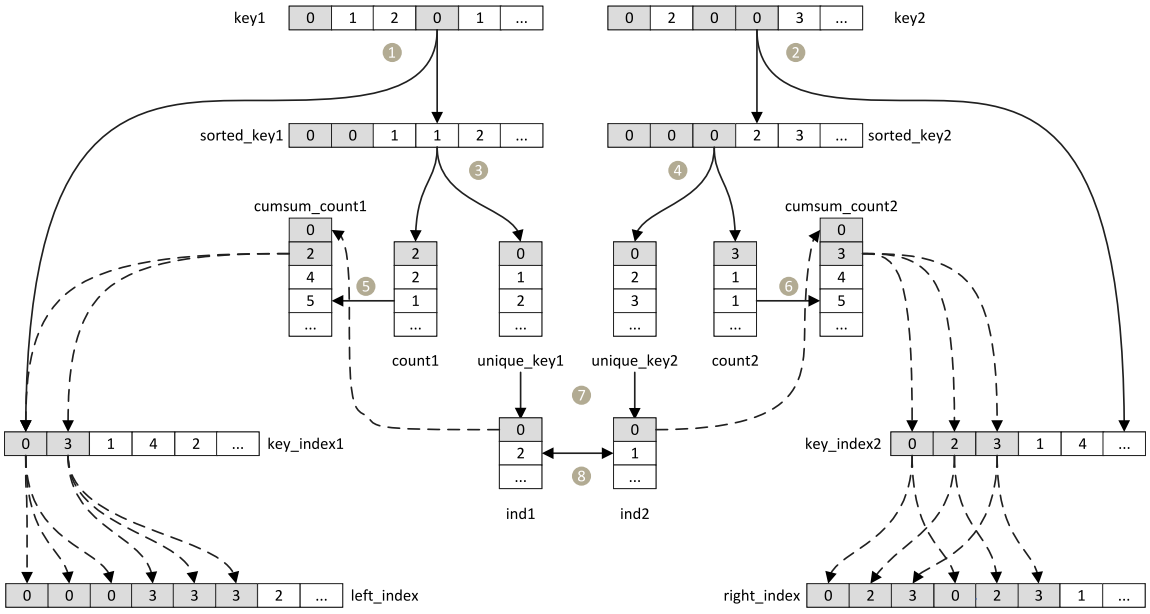


Fig. 6. The illustration of inner_join_index algorithm. Full lines represent operators, dotted lines represent indices.

Algorithm 3 Inner Join

Input: T1, T2: Input TensorTables; key1, key2: Join keys.
Output: OutTable: Output TensorTable.
1: key_tensor1 ← get_tensor(T1, key1)
2: key_tensor2 ← get_tensor(T2, key2)
3: left_index, right_index ← inner_join_index(key_tensor1, key_tensor2)
4: left_tensor ← index_select(T1.data_tensor, dim=0, left_index)
5: right_tensor ← index_select(T2.data_tensor, dim=0, right_index)
6: OutTable.data_tensor ← concatenate(left_tensor, right_tensor)
7: OutTable.column_names ← T1.column_names ∪ T2.column_names
8: OutTable.column_types ← T1.column_types ∪ T2.column_types
9: OutTable.str_dict_list ← T1.str_dict_list ∪ T2.str_dict_list

$T1$ and $T2$, and two keys to join: $key1$ from $T1$ and $key2$ from $T2$. The output TensorTable is $OutTable$. First, we extract key tensors from two TensorTables. Then we use the *inner_join_index* function to parse key tensors and return left and right indexes, which are used to select rows from two input TensorTables, as defined in Algorithm 4.

Next, we provide a detailed explanation of the *inner_join_index* algorithm in detail, as depicted in Fig. 6. This algorithm takes two key tensors as inputs and produces two indices used for selecting rows from two TensorTables. The process begins with the utilization of the *stable_sort* function to sort the two keys and get the sorted keys *sorted_key*1 and *sorted_key*2, and their respective indices *key_index*1 and *key_index*2, which represent the positions of elements in the original keys (lines 1–2 in Algorithm 4 and ❶❷ in Fig. 6). Subsequently, the *unique* function is applied to obtain the unique keys *unique_key*1 and *unique_key*2, along with their corresponding counts *count*1 and *count*2 (lines 3–4 and ❸❹). Following this, we employ the *cumsum* function to calculate the cumulative sum of counts, marked as *cumsum_count*1 and *cumsum_count*2 (lines 5–6 and ❺❻). We add 0 as the first element to both *cumsum_count*1 and *cumsum_count*2. Moving forward, we compute the intersection, denoted as *inter*, along with corresponding indices, *ind*1 and *ind*2, of the unique keys(lines 7 and ❼).

Finally, we iterate through *ind*1 and *ind*2 to construct *left_index* and *right_index* (lines 8–17 and ❽). We utilize *cumsum_count*1 and *cumsum_count*2 as indices to get elements from *key_index*1 and *key_index*2 and assign their values to *left_index* and *right_index* appropriately. Specifically, We take *cumsum_count*1[*ind*1[*i*]] and *cumsum_count*1[*ind*1[*i*] + 1] as the index to get elements from *key_index*1, that is *key_index*1[*cumsum_count*1[*ind*-1[*i*]], *cumsum_count*1[*ind*1[*i*] + 1]], marked as *left_tmp*. We then repeat each element in *left_tmp* for *count*2[*ind*1[*i*]] times and append them to *left_index*. Similarly, we take *cumsum_count*2[*ind*2[*i*]] and *cumsum_count*2[*ind*2[*i*] + 1] as the index to get elements from *key_index*2, that is *key_index*2[*cumsum_count*2[*ind*2[*i*]], *cumsum_count*2[*ind*-2[*i*] + 1]], marked as *right_tmp*. We repeat *right_tmp* for *count*1[*ind*2[*i*]] times, and append them to *right_index*. An example

Algorithm 4 inner_join_index

Input: key_tensor1, key_tensor2: Two keys in tensor formats.
Output: left_index, right_index: The indices to select rows from two TensorTables.

```
1: sorted_key1, key_index1 ← stable_sort(key_tensor1)
2: sorted_key2, key_index2 ← stable_sort(key_tensor2)
3: unique_key1, count1 ← unique(sorted_key1, return_counts=True)
4: unique_key2, count2 ← unique(sorted_key2, return_counts=True)
5: cumsum_count1 ← cumsum(count1)
6: cumsum_count2 ← cumsum(count2)
7: inter, ind1, ind2 ← intersection(unique_key1, unique_key2)
8: left_index ← Empty list
9: right_index ← Empty list
10: for i ← 0 to len(ind1) do
11:     left_tmp ← key_index1[cumsum_count1[ind1[i]], cumsum_count1[ind1[i]+1]]
12:     Repeat each element in left_tmp for count2[ind1[i]] times
13:     Append left_tmp to left_index
14:     right_tmp ← key_index2[cumsum_count2[ind2[i]], cumsum_count2[ind2[i]+1]]
15:     Repeat right_tmp for count1[ind2[i]] times
16:     Append right_tmp to right_index
17: end for
```

is illustrated in Fig. 6; the gray part shows where $key1$ and $key2$ are all 0, which means the variable i used in line 10 and step ❽ is 0. $Cumsum\_count1[0]$ and $cumsum\_count1[1]$ are 0 and 2, so we take the first two elements from key_index1, that is [0, 3]. Count2[0] is 3, so each element in [0, 3] is repeated three times, yielding [0, 0, 0, 3, 3, 3], which is appended to $left\_index$. $Cumsum\_count2[0]$ and $cumsum\_count2[1]$ are 0 and 3, so we take the first three elements from key_index2, that is [0, 2, 3]. Count1[0] is 2, so [0, 2, 3] is repeated twice, yielding [0, 2, 3, 0, 2, 3], which is appended to $right\_index$.

After the $inner\_join\_index$ function, we select rows from two TensorTables based on $left\_index$ and $right\_index$, concatenate them, and assign them to $OutTable.data\_tensor$. The $column\_names$, $column\_types$, and str_dict_list of $OutTable$ are the union of that of two input TensorTables. Outer-join, left-join, right-join, and cross-join have similar implementations, using their specific $join\_index$ algorithms to replace $inner\_join\_index$.

4.2.4. Groupby

The groupby operator groups rows based on one or more columns. The input is one TensorTable $InTable$ and a $GroupbyCol$ which marks the column names to make groups. The output is one TensorTable $OutTable$. First, we parse the $GroupbyCol$ and get the list of indices. Then we utilize the $index\_select$ function to select the column based on indices and assign the values to $GroupbyTensor$. Subsequently, we utilize the $unique$ function to get the unique elements and generate $GroupbyIdx$, which marks the group to which each row belongs. Finally, we append $GroupbyIdx$ to $data\_tensor$, append "group" to $column\_names$, and append int32 to $column\_types$ of $OutTable$. The str_dict_list of $OutTable$ remains the same as that of $InTable$.

Algorithm 5 Groupby

Input: InTable: Input TensorTable; GroupbyCol: Groupby column name.
Output: OutTable: Output TensorTable.

```
1: index ← parse(GroupbyCol)
2: GroupbyTensor ← index_select(InTable, dim=1, index)
3: UniqueElement, GroupbyIdx ← unique(GroupbyTensor, return_inverse=True)
4: OutTable.data_tensor ← concatenate(GroupbyIdx, InTable.data_tensor)
5: OutTable.column_names ← InTable.column_names
6: OutTable.column_names ← ["group"] ∪ InTable.column_names
7: OutTable.column_types ← [int32] ∪ InTable.column_types
8: OutTable.str_dict_list ← InTable.str_dict_list
```

4.2.5. Aggregation

The aggregation operator collects one or more columns and returns their aggregated values. The input is one TensorTable $InTable$, one aggregation function $AggFunc$, and the $name\_list$ which marks the column names. The output is one TensorTable $OutTable$. Aggregation functions include $count$, sum, avg, min, max, etc. First, we parse the $name\_list$ and get the list of indices. Then we use the $index\_select$ function to select the columns based on these indices and assign them to $AggTensor$. Finally, we make aggregation on $AggTensor$ and assign the results to $OutTable.data\_tensor$. The $column\_names$ of $OutTable$ is set as $name\_list$. The $column\_types$ and str_dict_list of $OutTable$ are the subsets of $InTable$ according to $name\_list$.

Algorithm 6 Aggregation

Input: InTable: Input TensorTable; AggFunc: Aggregation function; name_list: Column names.
Output: OutTable: Output TensorTable.

```
1: index ← parse(name_list)
2: AggTensor ← index_select(InTable.data_tensor, dim=1, index)
3: OutTable.data_tensor ← AggFunc(AggTensor)
4: OutTable.column_names ← name_list
5: OutTable.column_types ← subset(InTable.column_types, name_list)
6: OutTable.str_dict_list ← subset(InTable.str_dict_list, name_list)
```

4.3. The mixed RA and LA pipeline implementation

This section presents the implementation of mixed RA and LA pipelines.

4.3.1. DAG IR

First, we introduce the Directed Acyclic Graph (DAG) Intermediate Representation (IR), which serves as the representation for mixed pipelines. The DAG IR comprises a list of operators, variables, and utility functions used to build pipelines and execute them. The variables are all TensorTables, as defined in Section 4.1. These TensorTables form the fundamental data units within the representation. The operators take TensorTables as both input and output, including both linear and relational operators. Within the DAG IR, nodes represent operators, while edges represent variables and mark the data dependencies between operators. In this way, this unified abstraction allows for the seamless representation of mixed RA and LA pipelines.

4.3.2. Parser

Parser traverses the source codes and transforms them into DAG IRs. Operators are initialized as nodes within the DAG IR, marking their input and output variables, along with an implementation instance used to lower and execute operators. Variables are established as edges within the DAG IR, facilitating connections between nodes. This comprehensive process results in the construction of a corresponding DAG IR, which encapsulates the entire representation once all source code statements have been traversed.

4.3.3. Optimizer

Optimizer performs a series of functionally equivalent transformations for DAG IRs aimed at achieving optimizations. These optimizations primarily fall into two categories: $operator\ swap$ and $operator\ fusion$.

$Operator\ swap$ involves altering the order of operators within the DAG, which can reduce computation and create more opportunities for $operator\ fusion$. RA operators such as $selection$ and $projection$ can be swapped with other RA and LA operators without affecting the result. This optimization is similar to $pushdown$ used in the RA systems [32, 33], but is expended to LA. For instance, consider a scenario where users calculate the reciprocal of one column and then make a selection. By rearranging the sequence to perform selection first and then the reciprocal operation, we can reduce the computation.

Operator fusion fuses two operators to reduce memory footprint and computation. This optimization is similar to that in the LA systems [34,35], but is expended to RA operators. We can effectively harness PyTorch to fuse linear operators and, simultaneously, apply fusion techniques to RA operators based on our TensorTable-based implementations. For instance, multiple selections or projections on a single TensorTable can be fused into a single operation. Similarly, multiple join operations can be consolidated into a single operation. For example, if users initially join A and B, and then join the result with C, we can fuse these operations into a single join. This involves calculating the indices for A, B, and C, as detailed in Section 4.2.3. Subsequently, we select rows from A, B, and C and concatenate them without generating intermediate tables.

4.3.4. Code generator

Code Generator converts those optimized DAGs to TensorTable-based operators based on the implementation instances of operators in DAG IRs. When dealing with RA operators, we adhere to the implementations outlined in Section 4.2 and translate them into a list of tensor operations as well as some simple auxiliary functions to handle TensorTables. In the case of LA operators, we translate them into PyTorch LA operators to handle the *data_tensor* in TensorTables and do not deal with other proprieties.

4.3.5. Executor

We construct the Executor on top of PyTorch, orchestrating the execution flow through a series of carefully sequenced program calls. In the case of RA operators, we employ PyTorch to execute the TensorTable-based operators. Meanwhile, auxiliary functions are executed using native Python. For LA operators, we call PyTorch to handle TensorTables and keep other proprieties just the same. We use an interpreted mode in PyTorch to reduce compilation time.

5. Evaluation

5.1. Experimental setup

We deploy a server node equipped with two Xeon E5-2620 V3 (Haswell) CPUs and 64 GB memory to conduct the experiments. Each CPU contains six physical cores. The operating system is Ubuntu 16.04, and the other software includes PyTorch 1.13, pandas 1.3.5, Spark 3.3.0, MonetDB 11.39.17, AIDA [18], and RMA [29].

5.2. The RA operator performance

This section evaluates RA operators. We use the BIXI dataset [31] and test the results on 1k, 10k, 100k, 1 m, and 10 m rows. We compare our work against MonetDB, Spark, and pandas.

Fig. 7(a) shows the performance of the selection operator. TensorTable outperforms the other frameworks on both small and large data sizes for two reasons. First, TensorTable stores data in a tensor format, allowing for faster data access compared to row-oriented or column-oriented data structures. Additionally, TensorTable utilizes element-wise comparison based on vector instructions to accelerate.

Fig. 7(b) presents the performance of the projection operator. TensorTable achieves noticeable speedup on small datasets due to better data locality and faster data access. However, TensorTable's performance diminishes on larger datasets due to the use of the *index_select* function, which reassigns the tensor data structure to ensure data contiguity and causes redundant computation. Utilizing the *index_select* function is essential, as it can maintain a good data locality and guarantee the performance of other operators, especially linear operators.

Fig. 7(c) displays the performance of the inner join operator. MonetDB and Spark use hash-based join, while pandas and TensorTable use sort-based join. TensorTable continues to outperform the others on small datasets. However, hash-based joins have lower complexity than sort-based joins, giving Spark and MonetDB an advantage as the dataset size increases. We are actively exploring the integration of hash-based joins into TensorTable without compromising the performance of other tensor computations.

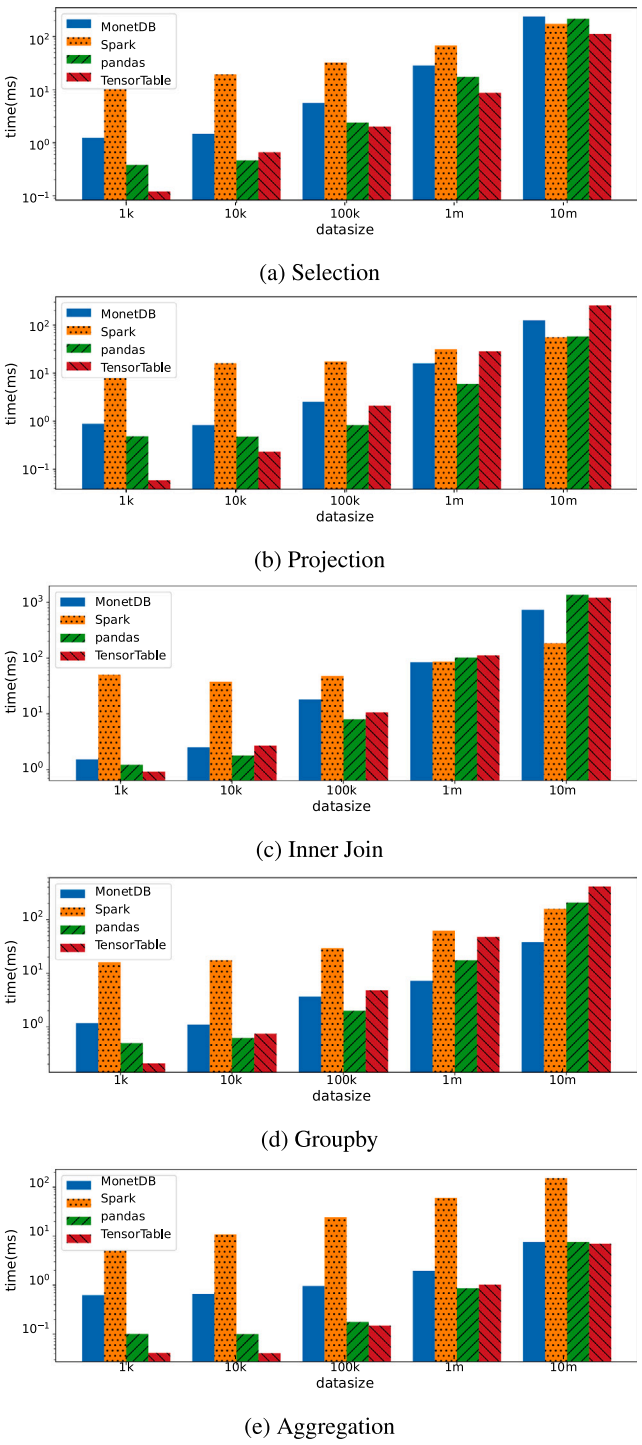


Fig. 7. The execution time of RA operators on TensorTable, MonetDB, Spark, and pandas over different data sizes.

Fig. 7(d) demonstrates the performance of the groupby operator, which exhibits similarities to the inner join operator. TensorTable excels on small datasets but lags behind on larger datasets due to limitations of the sort-based implementation. We are actively investigating the implementation of hash-based groupby operator in TensorTable.

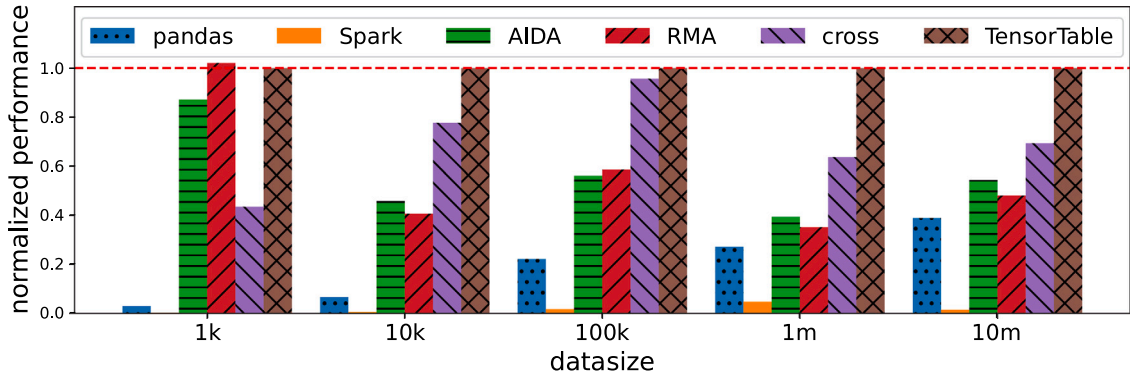
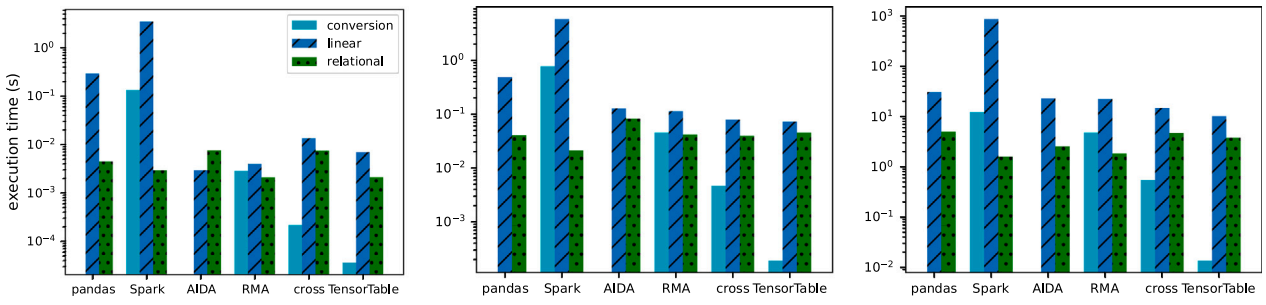


Fig. 8. The normalized performance of the distance-duration linear regression pipeline on different data sizes. Using the performance of TensorTable to normalize other frameworks, a smaller number has worse performance. "Cross" represents the cross-framework implementation using pandas for RA operators and PyTorch for LA operators.



(a) 1k (b) 100k (c) 10m

Fig. 9. The execution time breakdown of the distance-duration linear regression pipeline on different data sizes. It consists of three primary parts: data conversion, LA operators, and RA operators.

Fig. 7(e) illustrates the performance of the aggregation operator. This operator relies more on linear algebra computations than other relational operators, which benefits TensorTable on both small and large datasets.

In summary, TensorTable achieves superior performance on selection and aggregation operators across all data sizes. For projection, inner join, and group by operators, TensorTable performs better on small datasets while worse on large datasets. TensorTable achieves competitive performance on RA operators compared to RA and general-purpose systems.

5.3. Mixed pipeline performance

This section evaluates two mixed pipelines consisting of RA and LA operators.

5.3.1. Distance-duration linear regression

The first case derives from a public bicycle sharing system [18] that involves linear regression between distance and duration, utilizing the BIXI dataset [31]. There are two tables: *trip* contains start stations, end stations, and duration; *station* contains station codes, names, and coordinates. This pipeline comprises five steps: (1) Select rows from *trip* where the start station does not equal the end station. (2) Join *trip* with *station* to retrieve the coordinates of start and end stations. (3) Calculate the distance. (4) Train a linear regression model between distance and duration. (5) Test the model on the test dataset. Steps (1)(2) involves RA operators, and steps (3)-(5) involves LA operators. We do not use any built-in linear regression algorithms to avoid bias from these algorithms' implementation differences. We use those basic LA operators, such as matrix multiplication, to implement linear regression, guaranteeing consistency in computation for different frameworks.

We compare TensorTable with pandas, Spark, AIDA, RMA, and cross-framework implementation, which uses pandas for RA operators and PyTorch for LA operators. We use the performance of TensorTable to normalize the other five approaches, as illustrated in Fig. 8. Note that a value smaller than 1 indicates a worse performance compared to TensorTable and the smaller the value, the worse the performance. TensorTable achieves a 2.57x-32.33x speedup compared with pandas, a 20.77x-390.55x speedup compared with Spark, a 1.15x-2.53x speedup compared with AIDA, and a 1.04x-2.29x speedup compared with cross-framework implementation. Although TensorTable lags behind RMA by 2% when dealing with small datasets, it outperforms RMA as the dataset size grows, with a speedup ranging from 1.71x to 2.84x.

Fig. 9 provides a detailed breakdown of the results. The execution time of this mixed pipeline is divided into three primary components: data conversion, LA operators, and RA operators. Our work converts relational tables into TensorTables during initialization and does not need other data conversion in later computation, whose data conversion time is not apparent. Pandas handles DataFrames for both LA and RA operators without explicit data conversion, resulting in nearly zero conversion time. TensorTable achieves a 3.04x-41.71x speedup for LA operators and a 1.11x-2.09x speedup for RA operators compared with pandas. Spark implements RA operators using DataFrames and LA operators using matrices, necessitating frequent data conversion. While Spark excels in RA operators for sizable datasets, it lags in LA operators and involves more data conversion, resulting in the worst performance among the six implementations. TensorTable achieves a 41.44x-491.96x speedup for LA operators and an 868.25x-4546.49x speedup for data conversion compared with Spark. AIDA handles TabularData for both LA and RA operators without explicit data conversion, resulting in almost negligible conversion time. RMA handles RA operators and simple LA operators like addition based on binary association tables and executes complex LA operators such as matrix multiplication by calling external libraries like MKL [36]. This incurs notable data conversion costs, particularly with complex LA operators. AIDA

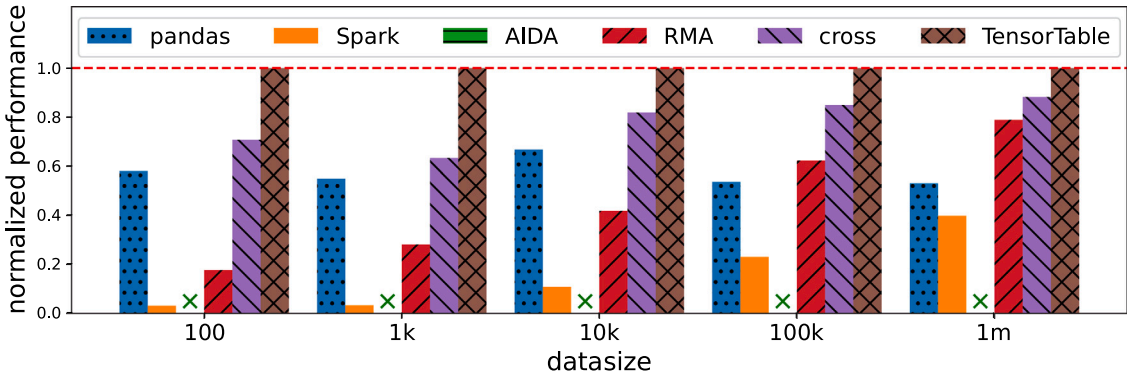
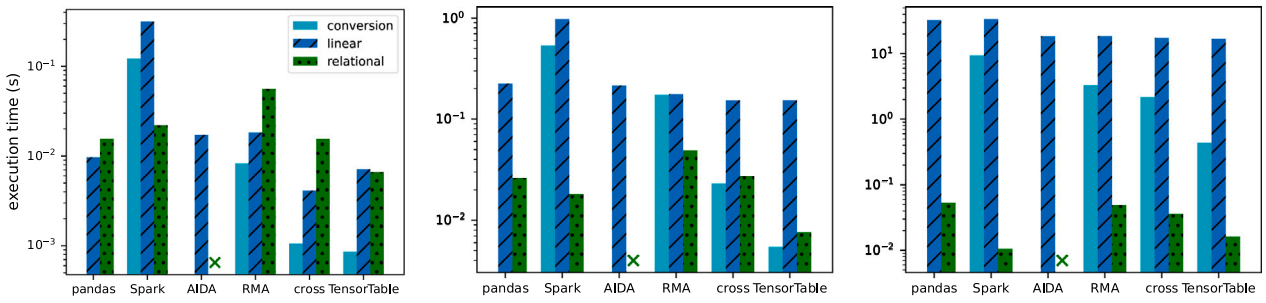


Fig. 10. The normalized performance of the conference covariance pipeline on different data sizes. Using the performance of TensorTable to normalize other frameworks, a smaller number has worse performance. "Cross" represents the cross-framework implementation using pandas for RA operators and PyTorch for LA operators. "x" means unsupported.



(a) 100 (b) 10k (c) 1m

Fig. 11. The execution time breakdown of the conference covariance pipeline on different data sizes. It consists of three primary parts: data conversion, LA operators, and RA operators. "x" means unsupported.

uses numpy for LA operators and RMA uses MKL for LA operators. These two libraries provide better optimizations for small-scale LA computations over PyTorch. Therefore, AIDA and RMA outperform TensorTable and cross-framework implementation for LA operators over smaller datasets. However, as the dataset grows, PyTorch demonstrates superior operator-level and graph-level optimizations, ensuring TensorTable and cross-framework implementation's superiority in LA operators for larger datasets. TensorTable achieves a 1.76x-4.12x speedup for LA operators and a 1.82x-3.54x speedup for RA operators compared with AIDA. TensorTable achieves a 1.58x-3.72x speedup for LA operators and a 78.22x-344.62x speedup for data conversion compared with RMA. While RMA exhibits slightly better performance on RA operators compared with TensorTable, its data conversion and LA operators compromise its end-to-end performance. The LA performance of TensorTable and cross-framework implementation have no significant differences. However, cross-framework implementation requires frequent data conversion between DataFrames and tensors during runtime. TensorTable achieves a 1.03x-2.61x speedup for RA operators and a 5.14x-39.43x speedup for data conversion compared with cross-framework implementation.

5.3.2. Conferences–covariance computation

The second case derives from an academic conference management system [29], it computes the covariance between A++ conferences and other conferences based on the number of publications per author and conference, using the DBLP dataset [37]. There are two tables: *ranking* stores conferences and their ratings, and *publication* stores the number of publications per author and conference. The pipeline involves three main steps: (1) Compute the covariance matrix on *publication*. (2) Join the result with *ranking*. (3) Select the A++ conferences. Step (1) uses LA operators, and Steps (2)(3) use RA operators.

Fig. 10 displays the normalized performance of TensorTable compared to pandas, Spark, AIDA, RMA, and cross-framework implementation using pandas for RA operators and PyTorch for LA operators. TensorTable achieves a 1.5x-1.88x speedup compared with pandas, a 2.51x-31.31x speedup compared with Spark, a 1.27x-5.63x speedup compared with RMA, and a 1.13x-1.58x speedup compared with cross-framework implementation. AIDA loses contextual information when performing LA operators on TabularData containing non-numeric columns. This results in some RA operators failing to execute after LA operators. As a result, this mixed pipeline can only execute the first half, encountering errors in the latter part for AIDA.

Fig. 11 offers a breakdown of execution time into three primary parts: data conversion, LA operators, and RA operators. TensorTable achieves a 1.36x-1.93x speedup for LA operators and a 1.97x-3.47x speedup for RA operators compared with pandas. TensorTable achieves a 2.01x-44.1x speedup, a 1.5x-3.33x speedup, and a 21.62x-249.57x speedup for LA operators, RA operators, and data conversion, respectively, compared with Spark. TensorTable achieves a 1.1x-2.42x speedup for LA operators compared with AIDA, with AIDA's RA operators encountering errors due to contextual information loss caused by preceding LA operators. Against RMA, TensorTable achieves a 1.1x-2.56x speedup for LA operators, a 3.02x-8.4x speedup for RA operators, and a 7.27x-31.66x speedup for data conversion. The LA performance of TensorTable and cross-framework implementation have no significant differences. However, TensorTable achieves a 2.22x-3.62x speedup for RA operators and a 1.23x-4.96x speedup for data conversion in contrast to cross-framework implementation. In summary, TensorTable attains optimal end-to-end performance by leveraging high-performance LA operators, minimizing data conversion overhead, and achieving competitive performance on RA operators.

6. Related work

There are four categories of system implementations to support the mixed RA and LA pipelines.

(1) Extending SQL for LA [24–26,38–42]. These works leverage the mature ecosystem of Relational Database Management Systems (RDBMS) and compile LA computations into SQL statements or RA expressions. While they can ensure good performance for RA operators, the SQL statements often limit the optimization potential for LA computations, leading to suboptimal performance. Additionally, certain LA operators, such as matrix inversion and determinant computation, cannot be implemented using this method.

(2) Using User-Defined Actions (UDAs) or User-Defined Functions (UDFs) to implement LA in RDBMS [27,28,43–45]. However, it requires significant effort and lacks flexibility. UDAs or UDFs provide technical interfaces but do not offer ready-made implementations, leaving users to create high-performance implementations themselves, which can be challenging. Furthermore, these interfaces may struggle to handle the various shapes and dimensions of LA operators effectively, and hinder the users' incremental developments, such as changing algorithms and tuning parameters.

(3) Building cross-framework applications [17–20]. These approaches utilize different systems for specific tasks, combining scientific computing systems for LA operators, and RDBMS for RA operators, and then integrating them. While this method can cover a wide range of LA and RA operators, it introduces extra costs due to data copying and transformations between frameworks and may limit cross-framework optimizations.

(4) Proposing new abstractions for both LA and RA operators. Some of them [22,23,46] propose new abstractions and build dedicated data analysis frameworks from scratch. Others [29,30] propose new abstractions and extend existing frameworks. Most of these systems are based on RDBMS, prioritizing performance for RA operators. However, they often lack optimizations for LA operators, which cover more execution time. In contrast, our work proposes a new abstraction and implements the system on the LA framework, ensuring optimal performance for LA operators while still accommodating RA operators.

7. Conclusion

This paper introduces TensorTable, a novel abstraction designed to seamlessly accommodate mixed pipelines encompassing both relational algebra (RA) and linear algebra (LA) operators. TensorTable can represent relational tables from RA, as well as vectors, matrices, and tensors from LA. we provide TensorTable-based implementations for RA operators and build a system that supports mixed LA and RA pipelines. Built on top of PyTorch, our implementation ensures comparable performance across both RA and LA operators, especially on small datasets. Besides, TensorTable achieves a 1.15x-5.63x speedup for mixed pipelines, outperforming state-of-the-art frameworks—AIDA and RMA.

CRediT authorship contribution statement

Xu Wen: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Data curation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] Per-Åke Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michal Nowakiewicz, Vassilis Papadimos, Real-time analytical processing with SQL server, Proc. VLDB Endow. 8 (12) (2015) 1740–1751.

[2] Babak Yadranjiaghdam, Nathan Pool, Nasseh Tabrizi, A survey on real-time big data analytics: applications and tools, in: 2016 International Conference on Computational Science and Computational Intelligence, CSCI, IEEE, 2016, pp. 404–409.

[3] Arun Kejariwal, Sanjeev Kulkarni, Karthik Ramasamy, Real time analytics: algorithms and systems, 2017, arXiv preprint arXiv:1708.02621.

[4] Erum Mehmood, Tayyaba Anees, Challenges and solutions for processing real-time big data stream: a systematic literature review, IEEE Access 8 (2020) 119123–119143.

[5] Shivani S. Kale, Preeti S. Patil, A machine learning approach to predict crop yield and success rate, in: 2019 IEEE Pune Section International Conference (PuneCon), IEEE, 2019, pp. 1–5.

[6] Marcus D. Bloice, Andreas Holzinger, A tutorial on machine learning and data science tools with python, in: Machine Learning for Health Informatics: State-of-the-Art and Future Challenges, Springer, 2016, pp. 435–480.

[7] M. Vagizov, A. Potapov, K. Konzhgoladze, S. Stepanov, I. Martyn, Prepare and analyze taxation data using the python pandas library, in: IOP Conference Series: Earth and Environmental Science, Vol. 876, (1) IOP Publishing, 2021, 012078.

[8] Ricardo Silva Peres, Andre Dionisio Rocha, Paulo Leitao, Jose Barata, IDARTS–towards intelligent data analysis and real-time supervision for industry 4.0, Comput. Ind. 101 (2018) 138–146.

[9] Chunquan Li, Yaqiong Chen, Yuling Shang, A review of industrial big data for decision making in intelligent manufacturing, Eng. Sci. Technol., Int. J. 29 (2022) 101021.

[10] Dharmitha Ajerla, Sazia Mahfuz, Farhana Zulkernine, A real-time patient monitoring framework for fall detection, Wirel. Commun. Mob. Comput. 2019 (2019) 1–13.

[11] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, Martin L. Kersten, Monetdb: Two decades of research in column-oriented database architectures, IEEE Data Eng. Bull. 35 (2012) 40–45.

[12] Paul DuBois, MySQL, Pearson Education, 2008.

[13] Bruce Momjian, PostgreSQL: Introduction and Concepts, Vol. 192, Addison-Wesley New York, 2001.

[14] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, Travis E. Oliphant, Array programming with NumPy, Nature 585 (7825) (2020) 357–362.

[15] 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, Soumith Chintala, Pytorch: An imperative style, high-performance deep learning library, in: H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, R. Garnett (Eds.), Advances in Neural Information Processing Systems, Vol. 32, Curran Associates, Inc., 2019.

[16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, Xiaoqiang Zheng, TensorFlow: A system for large-scale machine learning, in: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI '16, USENIX Association, USA, 2016, pp. 265–283.

[17] David Kernert, Frank Köhler, Wolfgang Lehner, SLACID-sparse linear algebra in a column-oriented in-memory database system, in: Proceedings of the 26th International Conference on Scientific and Statistical Database Management, 2014, pp. 1–12.

[18] Joseph Vinish D'silva, Florestan De Moor, Bettina Kemme, AIDA: Abstraction for advanced in-database analytics, Proc. VLDB Endow. 11 (11) (2018) 1400–1413.

[19] Stefan Hagedorn, Steffen Kläbe, Kai-Uwe Sattler, Putting pandas in a box, in: CIDR, 2021.

[20] Alekh Jindal, K. Venkatesh Emani, Maureen Daum, Olga Poppe, Brandon Haynes, Anna Pavlenko, Ayushi Gupta, Karthik Ramachandra, Carlo Curino, Andreas Mueller, et al., Magpie: Python at speed and scale using cloud backends, in: CIDR, 2021.

[21] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica, Spark: Cluster computing with working sets, in: 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10), 2010.

[22] Michael Armbrust, Reynolds S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, et al., Spark sql: Relational data processing in spark, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 2015, pp. 1383–1394.

[23] Wes McKinney, et al., Pandas: a foundational python library for data analysis and statistics, Python High Perform. Sci. Comput. 14 (9) (2011) 1–9.
[24] Francesco Del Buono, Matteo Paganelli, Paolo Sottovia, Matteo Interlandi, Francesco Guerra, Transforming ML predictive pipelines into SQL with MASQ, in: Proceedings of the 2021 International Conference on Management of Data, 2021, pp. 2696–2700.
[25] Shangyu Luo, Zekai J. Gao, Michael N. Gubanov, Luis Leopoldo Perez, Christopher M. Jermaine, Scalable linear algebra on a relational database system, IEEE Trans. Knowl. Data Eng. 31 (7) (2019) 1224–1238.
[26] Ying Zhang, Martin Kersten, Stefan Manegold, SciQL: Array data processing inside an RDBMS, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 1049–1052.
[27] Konstantinos Karanasos, Matteo Interlandi, Doris Xin, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Supun Nakandal, Subru Krishnan, Markus Weimer, et al., Extending relational query processing with ML inference, 2019, arXiv preprint arXiv:1911.00231.
[28] Carlos Ordonez, Building statistical models and scoring with UDFs, in: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, 2007, pp. 1005–1016.
[29] Oksana Dolmatova, Nikolaus Augsten, Michael H. Böhlen, A relational matrix algebra and its implementation in a column store, in: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 2573–2587.
[30] Dylan Hutchison, Bill Howe, Dan Suciu, Laradb: A minimalist kernel for linear and relational algebra computation, in: Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and beyond, 2017, pp. 1–10.
[31] Ahmadreza Faghih-Imani, Naveen Eluru, Ahmed M. El-Geneidy, Michael Rabbat, Usama Haq, How land-use and urban form impact bicycle flows: evidence from the bicycle-sharing system (BIXI) in Montreal, J. Transp. Geogr. 41 (2014) 306–314.
[32] Surajit Chaudhuri, An overview of query optimization in relational systems, in: Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS '98, ACM, New York, NY, USA, 1998, pp. 34–43.
[33] Abraham Silberschatz, Henry F. Korth, S. Sudarshan, Database System Concepts, third ed., McGraw-Hill, New York, 2010.
[34] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, Arvind Krishnamurthy, Learning to optimize tensor programs, Adv. Neural Inf. Process. Syst. 31 (2018).
[35] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, Alex Aiken, TASO: optimizing deep learning computation with automatic generation of graph substitutions, in: Proceedings of the 27th ACM Symposium on Operating Systems Principles, 2019, pp. 47–62.
[36] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, Yajuan Wang, Endong Wang, Qing Zhang, Bo Shen, et al., Intel math kernel library, in: High-Performance Computing on the Intel® Xeon Phi™: how to Fully Exploit MIC Architectures, Springer, 2014, pp. 167–188.
[37] University of Trier, DBLP computer science bibliography, 2022, https://dblp.uni-trier.de/.
[38] Vladimir Kotlyar, Keshav Pingali, Paul Stodghill, A relational approach to the compilation of sparse matrix programs, in: European Conference on Parallel Processing, Springer, 1997, pp. 318–327.
[39] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, N. Widmann, The multidimensional database system RasDaMan, SIGMOD Rec. 27 (2) (1998) 575–577.
[40] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, Zekai J. Gao, Declarative recursive computation on an RDBMS: Or, why you should use a database for distributed machine learning, Proc. VLDB Endow. 12 (7) (2019) 822–835.
[41] Umar Syed, Sergei Vassilvitskii, SQML: large-scale in-database machine learning with pure SQL, in: Proceedings of the 2017 Symposium on Cloud Computing, 2017, pp. 659–659.
[42] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, Dan Suciu, SPORES: sum-product optimization via relational equality saturation for large scale linear algebra, 2020, arXiv preprint arXiv:2002.07951.
[43] Xixuan Feng, Arun Kumar, Benjamin Recht, Christopher Ré, Towards a unified architecture for in-RDBMS analytics, in: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, 2012, pp. 325–336.
[44] Joe Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, et al., The MADlib analytics library or MAD skills, the SQL, 2012, arXiv preprint arXiv:1208.4165.
[45] Shreya Prasad, Arash Fard, Vishrut Gupta, Jorge Martinez, Jeff LeFevre, Vincent Xu, Meichun Hsu, Indrajit Roy, Large-scale predictive analytics in vertica: Fast data transfer, distributed model creation, and in-database prediction, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 2015, pp. 1657–1668.
[46] R. Core Team, et al., R: A language and environment for statistical computing, 2013.