Optimizing internal data representation in Jayvee

BACHELOR THESIS

Jonas Zeltner

Submitted on 21 August 2024

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Johannes Jablonski, M.Sc.
Prof. Dr. Dirk Riehle, M.B.A.

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Erlangen, 21 August 2024

License

Erlangen, 21 August 2024

Abstract

Jayvee is a simple language for describing data pipelines. The execution of these pipelines necessitates the handling of tabular data. Previously, *Jayvee* utilized *TypeScript*'s data structures to represent such data. This thesis develops a new table implementation for the *Jayvee* interpreter.

We present the architectural design and implementation of such a prototype. It uses the *Polars* library to adhere to the *Apache Arrow* specification. Additionally, the library *sqlite-loader-lib*, written in *Rust*, is integrated into this architecture, to accelerate the export of tables.

The evaluation demonstrates, that the new implementation has the potential to increase the *Jayvee* interpreter's maximum input size from 475 Megabyte above 2.5 Gigabyte and its processing speed by a factor of between 3.60 and 18.22.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

ETL extract, transform, load

URL uniform resource locator

CSV Comma-seperated values

IPC Inter-process communication

SQL Structured Query Language

ODbL Open Data Commons Open Database License

API application programming interface

JSON JavaScript Object Notation

regex regular expression

CLI command-line interface

MB Megabyte

GB Gigabyte

UTF-8 8-Bit Universal Coded Character Set Transformation Format

1 Introduction

Each year the European Union publishes a report on the maturity of open data offerings by its member states. Publications Office of the European Union et al. (2023) show an increase in average maturity scores since 2018. The use of this open data is diminished by the often significant technical knowledge required to work with it. Lowering this barrier of entry is the major motivation behind the JValue Project (JValue Contributors, n.d.-c) in general and its language, *Jayvee*, specifically (JValue Contributors, n.d.-b).

1.1 The *Jayvee* language

The *Jayvee* language enables the description and subsequent execution of data pipelines. A pipeline is a sequence of executable blocks to perform operations on data. These blocks are categorized in extractors, transformers and loaders.

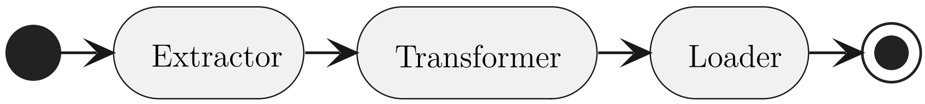


Figure 1.1: The basic structure of a pipeline.

Extractors bring the data into the pipeline and make it available for the other types of blocks. *Jayvee* supports retrieving data from a local file (`LocalFileExtractor`) or a uniform resource locator (URL) (`HttpExtractor`).

Transformers modify the data between extractor and loader.

Loaders export data to somewhere outside the pipeline, such as a *SQLite* file (`SQLiteLoader`), *postgres* database (`PostgresLoader`) or Comma-seperated values (CSV) file (`CSVFileLoader`).

A more detailed description of *Jayvee*'s core concepts has been provided by JValue Contributors (n.d.-a).

Jayvee's goal is to enable everyone to describe extract, transform, load (ETL) pipelines (JValue Contributors, n.d.-b).

2 Literature Review

Ahmad et al. (2021) created their own memory format based on *Apache Arrow* to process genome data. They found that *Arrow* improved the use of the available hardware, and led to fewer cache misses. *Arrow* performed better than both *ramDisk* and *Unix* pipes. The overall execution time was 4.85 times faster with *Arrow*.

Peltenburg et al. (2021) developed *Fletchgen*, which can generate hardware interfaces designed for *Arrow* workloads.

Grossman et al. (2022) used *Apache Arrow* to build *SHMEM-ML*, a machine learning library. *Arrow* enabled ”zero copy data sharing” with other libraries. They were able to accelerate model training by a factor of 38, compared to the industry average.

2.1 Tabular data memory layout

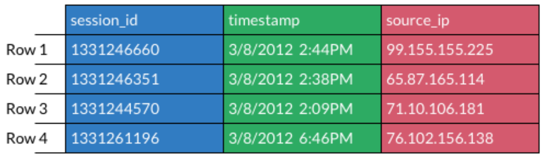
When sequentially processing tabular data, a fundamental challenge arises: how to organize the two-dimensional data onto a linear layout, either on disk, or in memory. Two primary approaches have been developed to address this challenge (Floratou, 2019):

row-oriented, row-storage The values are saved one row after another.

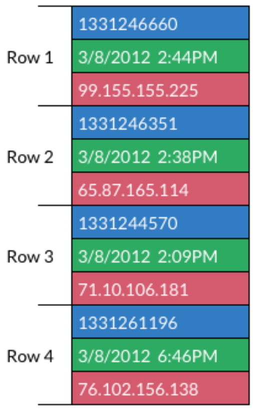
columnar, column-storage The values are saved column after column.

See Figure 2.1 for a visual comparison.

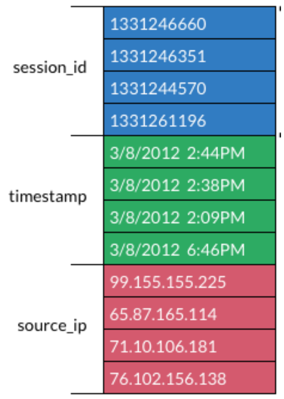
The earliest instances of columnar storage formats were initially developed for use in databases, with *MonetDB* (Boncz, 2002) being a notable example. By 2013 these formats had become widely implemented across the industry (Abadi et al., 2013). In approximately 2011 initial research was conducted into the potential applications of columnar storage in the context of Big Data frameworks, which culminated in the development of *Apache Parquet* and *Apache ORC*, both disk-based columnar formats (Floratou, 2019).



(a) An example table in ...



(b) ...row-oriented memory.



(c) ...columnar memory.

Figure 2.1: A table's is representation in row-oriented and columnar memory (The Apache Software Foundation, n.d.-a).

In 2016, The Apache Foundation initiated *Apache Arrow*, which specifies a columnar storage layout *in memory* (Ahmad et al., 2021).

Columnar storage has advantages over row-oriented storage:

- column specific compression (Abadi et al., 2013)
- less memory usage (Abadi et al., 2013)
- faster read times (Floratou, 2019)

2.2 *Apache Arrow*

Apache Arrow is often used synonymously with the memory format it defines. This format is *language agnostic*, *columnar* and *in memory*. It has implementations in many languages (The Apache Software Foundation, n.d.-a) (see subsection C.1 for a list).

In the *Arrow* specification (The Apache Software Foundation, n.d.-b), a table is called *record batch*. They are comprised of *Arrow* arrays which represent columns. These *Arrow* arrays are different from *TypeScript* arrays, as they cannot contain different data types. An *Arrow* array's actual data is split across a series of *Arrow* buffers, which represent continuous space in memory. *Arrow* record batches also have *Arrow* schemas, which contain the batch's metadata (see Figure 2.2).

Arrow has a more granular type system than *TypeScript*. This allows for a more precise control of memory. *Arrow* types can have parameters. For example instead of *TypeScript*'s type `number`, *Arrow* has (un)signed integers with a bit width parameter, decimals with width, scale and precision as parameters, and floating point numbers with a precision parameter. The specification includes a physical memory layout for each type (The Apache Software Foundation, n.d.-b).

Additionally, the *Arrow* specification defines an Inter-process communication (IPC) file format, that allows saving *Arrow* record batches on disk (see Figure 2.3).

At the time of writing this thesis, there are 48 projects powered by *Apache Arrow* (The Apache Software Foundation, n.d.-d).

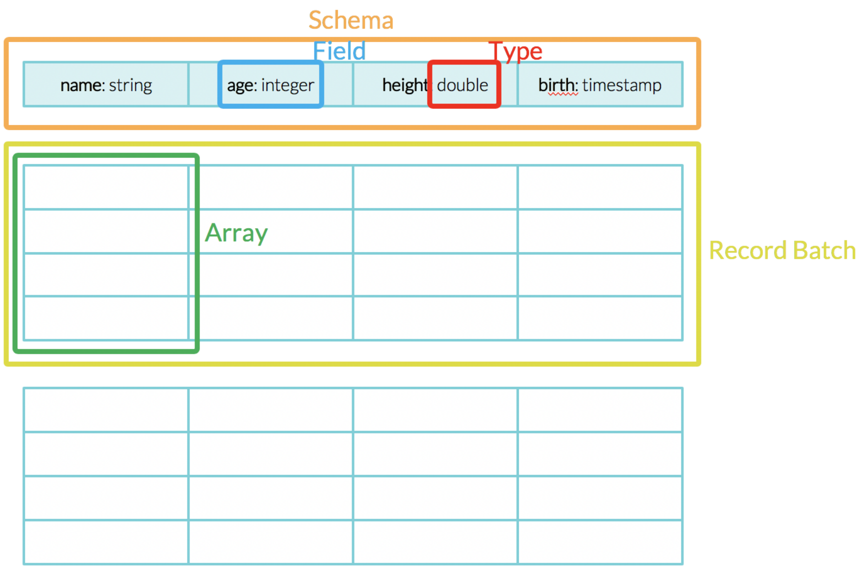


Figure 2.2: Key Concepts of an *Arrow* Table (Shiran, 2019).

2.3 *Polars*

Polars is built on top of the *Rust* implementation of the *Arrow* specification. It incorporates convenient abstractions such as `DataFrame` and `Series`. Additionally, it uses *NAPI-RS* to provide a *TypeScript* application programming interface (API) that preserves the performance of the *Rust* library. It promises "up to 50x performance" (Polars Contributors, n.d.-e).

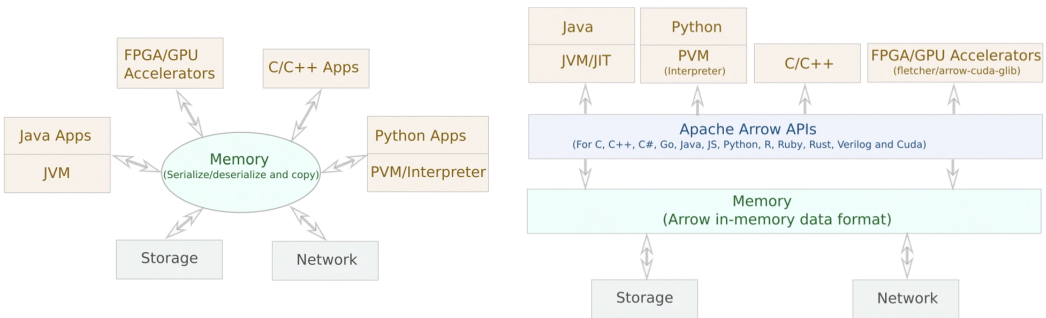


Figure 2.3: Illustration demonstrating *Arrow* enhances interoperability between processes (Ahmad et al., 2021).

3 Requirements

3.1 Functional requirements

interoperability This allows the data to be used by multiple different applications without complex conversion.

columnar A columnar format has advantages outlined in section 2.1, including faster read times and less memory usage.

feature toggle We must be able to (de)activate the new features at runtime. This makes comparing the new features easier.

compatibility The electric vehicles example must complete without errors. This example is executed, by running the command-line interface (CLI) command `npm run example:vehicles-polars` inside the project directory. The `.sqlite` files produced by the different implementations must be identical.

modularization Should it be necessary to write code in a language other than *TypeScript*, this code must be placed in an external library usable from *TypeScript*.

extensibility The functionality of the chosen data representation must be extensible.

3.2 Non-functional requirements

performance The main goal of this thesis is to optimize *Jayvee*'s in-memory representation. We expect this to result in either a decrease in the execution time for *Jayvee* models, or an extension of the *Jayvee* interpreter's capabilities.

code style The source code adheres to the project's code style. This is enforced by running `npm run lint`. For *Rust* source code, we use the defaults from *rustfmt* and *clippy*.

maturity We aim to create a prototype, not a mature implementation.

The fulfillment of these requirements is evaluated in section 6.7.

4 Architecture

The objective of this thesis is to optimize *Jayvee*'s internal data representation. This necessitates modifying the *Jayvee* interpreter, the program, that parses *Jayvee* source files and executes the defined pipelines.

In order to improve the class diagrams' readability, some parameters and return types have been abbreviated with "...".

4.1 Interpreter Overview

This section provides an overview of how the interpreter executes a *Jayvee* pipeline and the involved classes, illustrated by Figure 4.1. How the interpreter parses a *Jayvee* source file is not relevant for the purpose of this thesis.

Once the parsing process is complete, the interpreter obtains a series of pipeline definitions, which consist of block definitions. Block definitions include a *block type*. The interpreter uses this block type to find an executor for the block, which is a class that implements the interface `BlockExecutor`. The interpreter then executes these executors in the order specified in the pipeline definition. The output of the preceding block executor is always the input for the subsequent one.

Inputs and outputs are instances of classes that implement the interface `IOTypeImplementation`. This includes the `Table` class, which will be used to integrate *Apache Arrow* into the *Jayvee* interpreter.

One of *Jayvee*'s core features is the transformation of tables with the `TableTransformer` block type. Consequently, we will devote particular attention to this functionality. The executor for blocks of type `TableTransformer` is `TableTransformerExecutor`. It creates and executes the `TransformExecutor`, which evaluates the transform's expression for every row of the table and returns the resulting column. This column is then added to the input table by `TableTransformerExecutor`

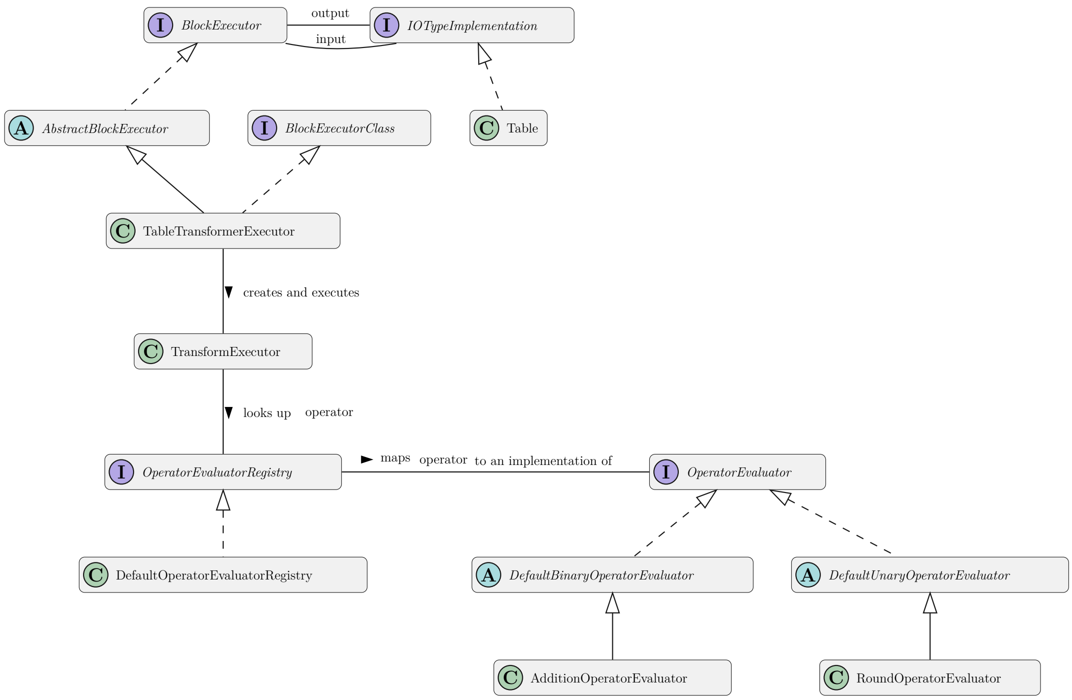


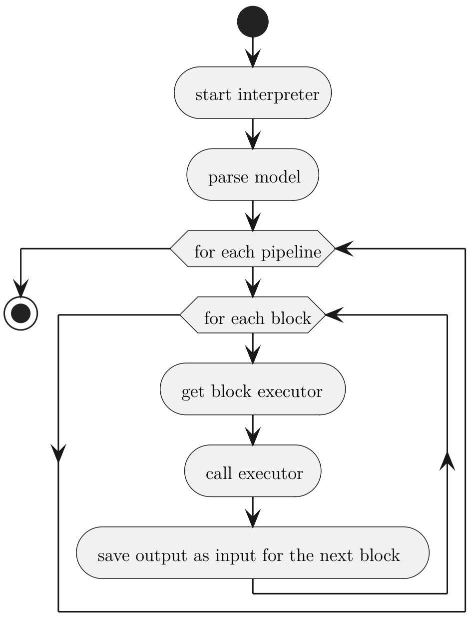
Figure 4.1: The general structure of the *Jayvee* interpreter. Many subclasses and implementations have been omitted for readability.

Expressions are evaluated by the function `evaluateExpression(...)`. Expressions with parameter(s) (unary, binary, ternary) are called operators. They have evaluator classes that implement the interface `OperatorEvaluator`. The function `evaluateExpression(...)` finds the correct evaluator with the help of the `OperatorEvaluatorRegistry`.

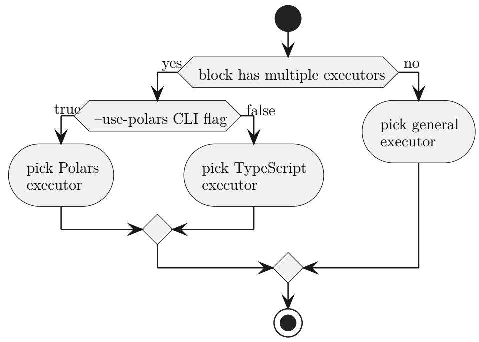
4.2 General approach

We add an alternative table implementation, based on the *Apache Arrow* framework (section 2.2), to the interpreter. This means, the interpreter will contain two table implementations at the same time. Dooley and Kazakova (2024) recommend the strategy pattern for this situation, because it will ensure that both implementations share a common interface and that the different implementations can be chosen dynamically.

We also adapt the interpreter to take advantage of the new features introduced by the new table (see Figure 4.2b). But, because the original implementation needs to be preserved here too, the strategy pattern is a good approach again.



(a) Main diagram



(b) `get block executor` picks the specified implementation

Figure 4.2: The activity diagram overview of the interpreter.

4.2.1 Possible implementations of *Arrow*

Implementation from scratch

The most direct approach would be a new implementation of the *Arrow* specification. This would afford the advantage of having full control over the implementation.

However, it is not guaranteed, that a sufficient amount can be implemented within the timeframe of the thesis. Furthermore, this implementation may not perform as well as an existing, more mature implementation, which may also be more correct.

Apache Arrow in *TypeScript*

There is an *Arrow* library written in *TypeScript* (The Apache Software Foundation, n.d.-a), which is the language of the *Jayvee* interpreter. This implementation is comparable to those in other languages regarding supported data types, but it lags behind with support for other Apache frameworks (The Apache Software Foundation, n.d.-c), which takes away one of *Arrow*'s strengths.

Polars

As described in section 2.3, *Polars* is a high level library built on top of *Apache Arrow*, which also offers useful abstractions such as `DataFrames`, that hide the *Arrow* data types. These abstractions make it easier for developers unfamiliar with *Arrow* to contribute to *Jayvee*.

Polars is written in *Rust*, but offers a *TypeScript* API, which enables the *Jayvee* interpreter to use *Polars*. For these reasons, we choose to build the new table implementation on top of *Polars*.

4.3 Type conversion

Jayvee types, e.g. text, boolean or decimal (JValue Contributors, n.d.-a), are represented by the interface `ValueType`, They are useful, because *Jayvee* types do not map directly to *TypeScript* types. For example, when a value inside a table has the *TypeScript* type `number`, an object implementing the interface `ValueType` denotes, whether the value is a *Jayvee* `integer` or a *Jayvee* `decimal`.

Polars' types are represented by the class `DataType`. The class `DataType` is not compatible with the interface `ValueType`, which leaves us with two possible approaches.

Replace `ValueType` with `DataType`. We decided against this approach, because it enforces the *Polars* implementation at compile time, which breaks the *feature toggle* requirement.

Convert between `ValueType` and `DataType` when needed. The *TypeScript* implementation can ignore the conversion and keep using `ValueType` as before. Because of the *feature toggle* requirement, we choose this approach.

For details on how this conversion mechanism is implemented, see section 5.1.

4.4 Creating a table implementation based on *Polars*

The usual application of the strategy pattern would create an interface (Dooley & Kazakova, 2024) that both the old and the new table classes implement. Instead, we utilize an abstract class, to allow for code-sharing between the table subclasses. We can still define functionality for both subclasses using *abstract methods*.

Jayvee already contains a class `Table` that implements the interface `IOTypeImplementation<IOType.TABLE>`. We move all the functionality from `Table` to the class `TsTable` and create a new abstract class `Table`.

Figure 4.3 visualizes the table architecture described in the following subsections.

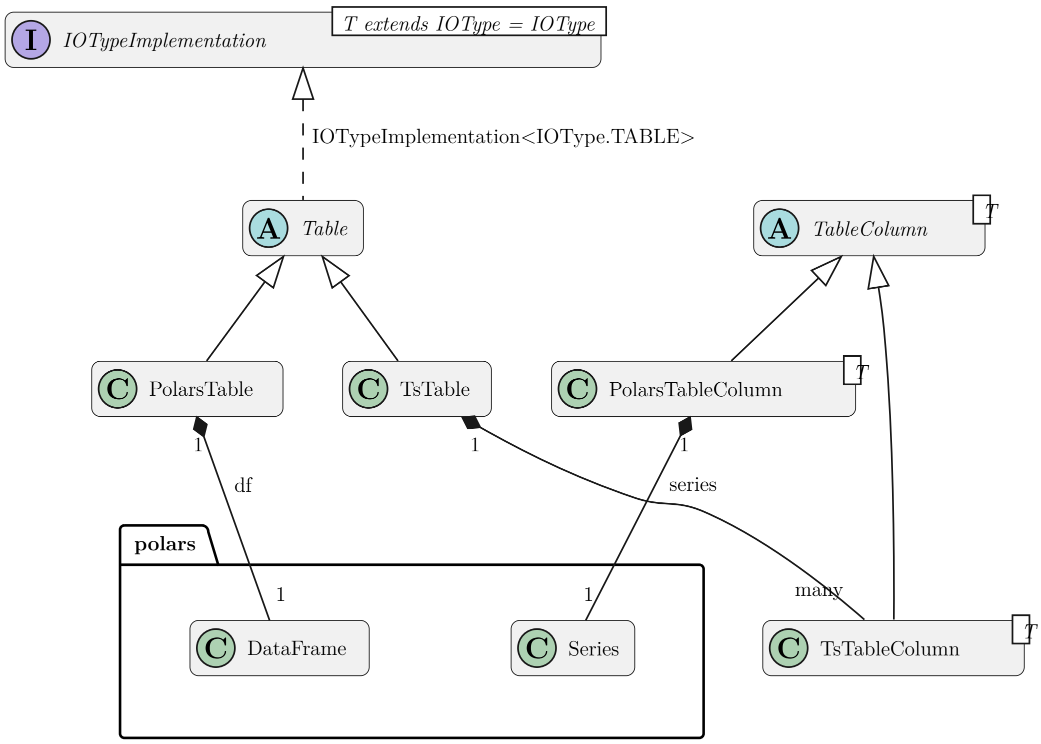


Figure 4.3: Overview of the classes relevant for the table implementation.

4.4.1 Table

This class implements the interface `IOTypeImplementation<IOType.TABLE>` to be allowed as an input or output for blocks. Its methods are based on those of the original class `Table`. See Figure 4.4 for a class diagram.

Differences between the old class Table and the new abstract class Table

The methods `getRow(...)`, `addRow(...)`, `dropRow(...)` and `dropRows(...)` are not part of the abstract class `Table`. These methods work with rows, so their usage would make the columnar data representation obsolete. They still exist in the `TsTable` class to satisfy the *compatibility* requirement.

In order to meet the requirement *feature toggle*, it is necessary to distinguish the concrete table implementation. Therefore, we have implemented the type guards `isPolars()` and `isTypescript()`.

Given the difficulties in executing operations with side effects in parallel (Gordon

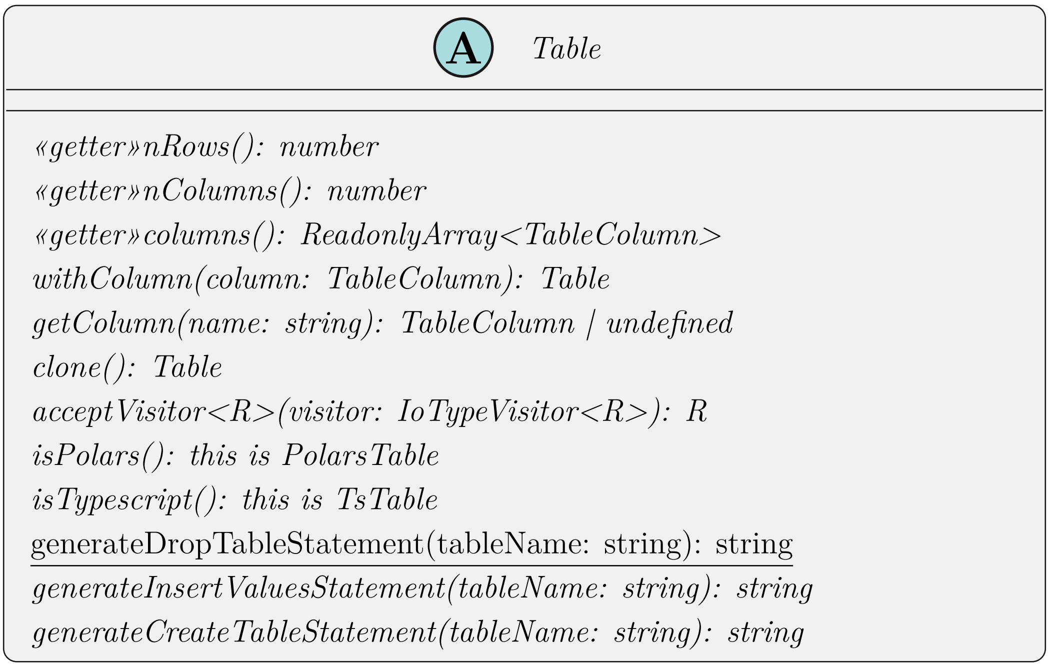


Figure 4.4: The abstract class Table.

et al., 2021), it is preferable to clone tables rather than mutate them. To satisfy the compatibility requirement, a new method, withColumn(...) is created to implement this behavior. The method TsTable.addColumn(...) preserves the old behavior.

Because of the fact, that the *Polars* and *TypeScript* implementations of the method generateDropTableStatement(...) are identical, we implement it as a non-abstract method in Table.

4.4.2 PolarsTable

Polars' DataFrame already has table functionality, such as the ability to add, remove and transform columns. However, DataFrame's methods aren't the same as those required by Table. To solve this, we create a wrapper class PolarsTable, that extends Table and thereby implements IOTypeImplementation<IOType.TABLE>, allowing PolarsTable to be used as an input or output for block executors.

PolarsTable extends Table's abstract methods by calling DataFrames methods, which is a characteristic of the *adapter pattern* (Dooley & Kazakova, 2024). In the terminology of the adapter pattern, Table is the target, PolarsTable the adapter and DataFrame the adaptee.

It is notable, that when `Table` is parameter or return type in an abstract method in `Table`, the overriding method in `PolarsTable` replaced that with `PolarsTable`. For example,

```
abstract clone(): Table
```

becomes

```
override clone(): PolarsTable
```

New functionality enabled by *Polars*

In addition to the abstract methods from `Table`, `PolarsTable` also implements methods that take advantage of *Polars*' features unavailable to the *TypeScript* implementation.

The method `withColumn(...)` is overloaded with a second signature, that can handle *Polars* expressions. *Polars* expressions are a way to describe a series of operations, that result in one or more columns, with automatic optimization and parallelization (Polars Contributors, n.d.-c). Listing 1 contains an example of how *Polars* expressions can be chained.

```
use { pl } from 'nodejs-polars';
// Creation of the DataFrame was omitted
const dataFrame = ...;
// When the expression is applied to a DataFrame, `pl.col(...)`
↪  selects a column, based on the name
const first = pl.col("a")
const second = pl.col("b")
// When the expression is applied, `mul` multplies all values
↪  from expression `first` with those from `second`
const product = first.mul(second)
// When the expression is applied, `alias("c")` renames the
↪  column to "c".
const renamed = product.alias("c");
// withColumn applies the expression and returns a DataFrame,
↪  containing the column computed by the expression.
const newDataFrame = dataFrame.withColumn(renamed)
// `newDataFrame` contains a column "c". "c" contains the values
↪  of column "a" multplied with the values from column "b"
```

Listing 1: Multiplying the values in the columns ”a” and ”b” of a `DataFrame`.

The methods `writeIpc(...)` and `writeIpcTo(...)` convert the table to the *Arrow* IPC format, writing either into a buffer or a file.

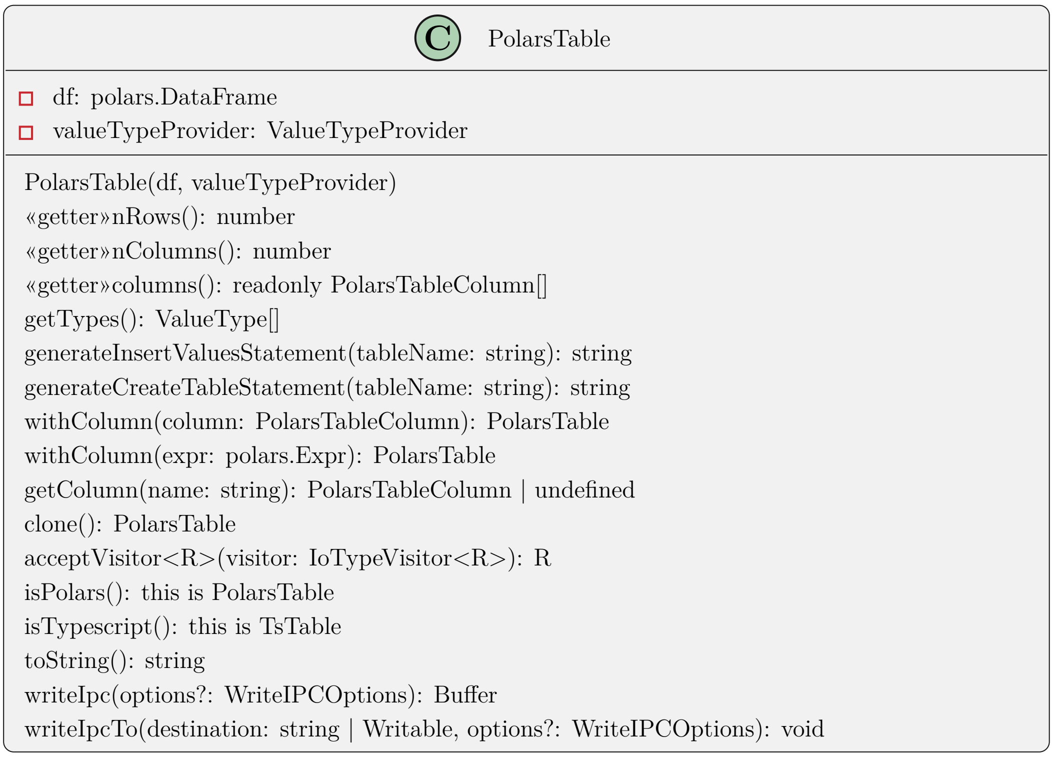


Figure 4.5: The class `PolarsTable`.

One of the challenges is, that adding a column into a `DataFrame` object loses that column's `ValueType`, because the `DataFrame` uses *Polars*' `DataType`. This is problematic, because the class `TableColumn` expects a `ValueType` for its constructor.

The solution is to convert the *Polars* `DataType` object into a `ValueType` object using the conversion mechanism in section 5.1. This conversion requires a `ValueTypeProvider` object, which is saved as one of `PolarsTable`'s properties. Refer to Figure 4.5 for the class diagram.

4.4.3 TableColumn

Analogous to the class `Table`, we create an abstract class `TableColumn` that has two subclasses, `TsTableColumn` and `PolarsTableColumn`

Compared to the old interface `TableColumn` (Figure 4.6), the new abstract class no longer defines how the values of the column are saved. It used to be a *TypeScript* array, but now this decision is left to the subclasses of `TableColumn`.

We utilize a generic type parameter, to remain close to the original implementation and to be more specific on the return type of the method `valueType()`.

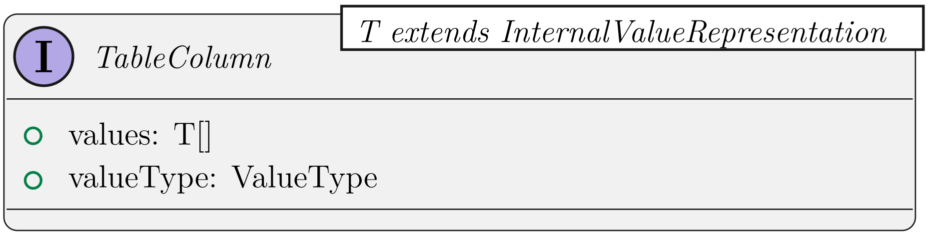


Figure 4.6: The replaced interface `TableColumn`.

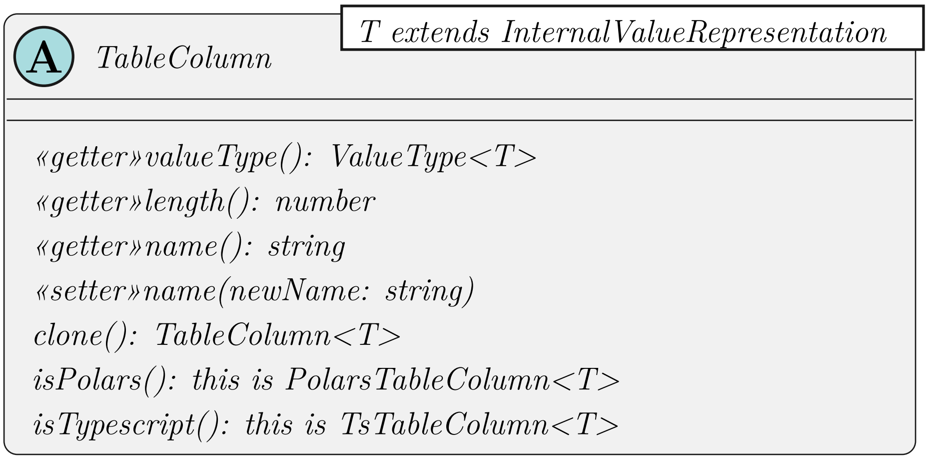


Figure 4.7: The class `TableColumn`.

The type guards `isPolars()` and `isTypescript()` were added.

Figure 4.7 contains a diagram of this class.

PolarsTableColumn

The method `DataFrame.getColumn(...)` returns an instance of the *Polars* class `Series`, which means that `Series` represents a column in *Polars*. To make `Series` useful, we create a class adapter with the target `TableColumn` and the adaptee `Series`.

One challenge here is, that `Series` does not have a generic type parameter. This limitation can be addressed in two ways:

- Create the class `PolarsTableColumn` without a type parameter and extend the abstract class `TableColumn<InternalValueRepresentation>`.
- Create the class `PolarsTableColumn<T>` with a type parameter and extend the abstract class `TableColumn<T>`.

The second approach is preferable, because it allows us to preserve the generic type parameter of the property `_valueType: ValueType<T>` (see Figure 4.8).

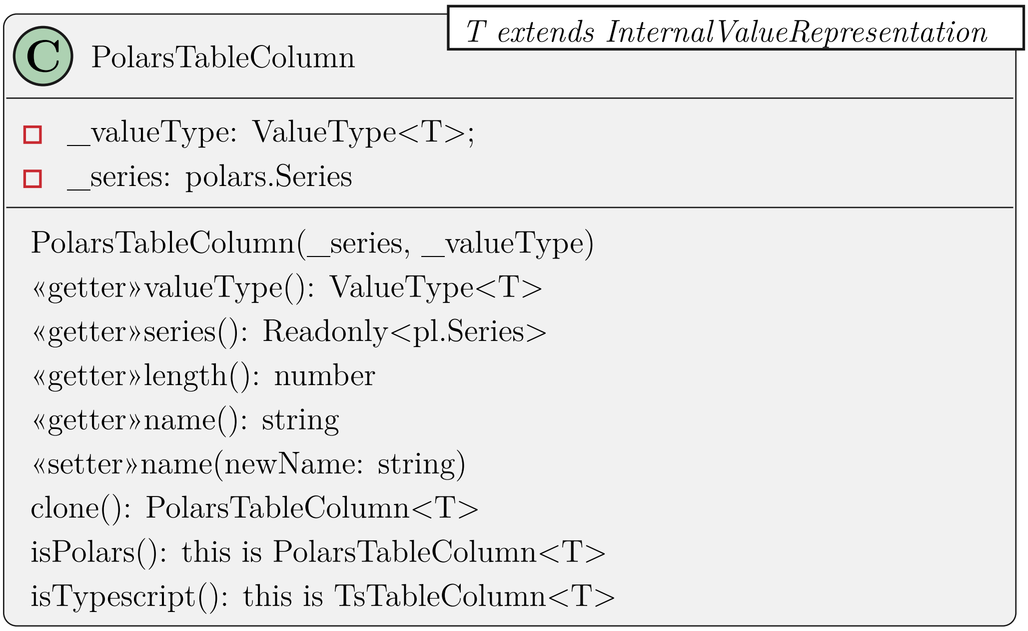


Figure 4.8: The class `PolarsTableColumn`.

It should be noted, that `T` does not indicate the actual type of the values stored in the column, as would be the case for `Array<T>`. This detail is left to `Series`. `T` specifies what type the column's values would be, if it were implemented in *TypeScript*. This is used to narrow down the type of `_valueType`.

TsTableColumn

For the class `TsTableColumn` (see Figure 4.9), the type parameter `T` behaves in the expected way, it denotes the type of the values saved in the column. `TsTableColumn` can also be looked at through the lens of the adapter pattern. Then, `TableColumn` is the target, `TsTableColumn` the adapter and `T[]` the adaptee.

In addition to the abstract methods of `TableColumn`, `TsTableColumn` also has methods that allow users to use the wrapped array functionality, `push(...)`, `at(...)` and `drop(...)`.

4.5 Adapting and creating block types

The executors for *Jayvee* blocks are organized in extensions that extend the abstract class `JayveeExecExtension` (see Figure 4.10). The *Jayvee* interpreter uses the method `createBlockExecutor(...)` to get an implementation of the interface `BlockExecutor`. This is the method, that chooses either the existing block

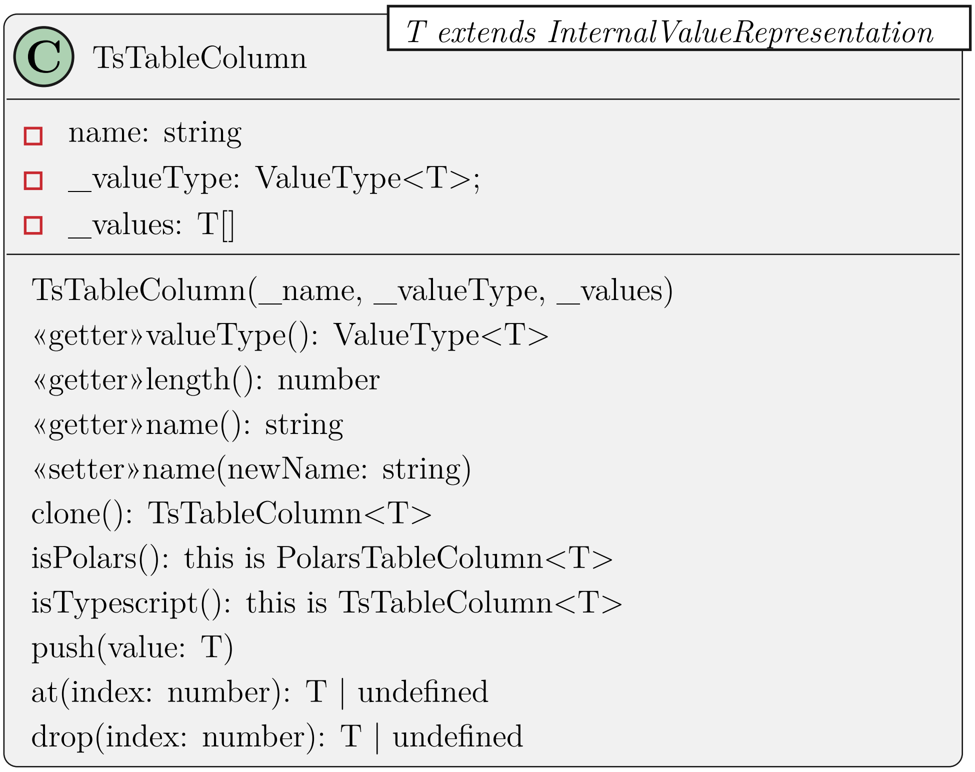


Figure 4.9: The class `TsTableColumn`.

executors or the new block executors that are described in this section. For details on the implementation of this method, see subsection 5.4.1.

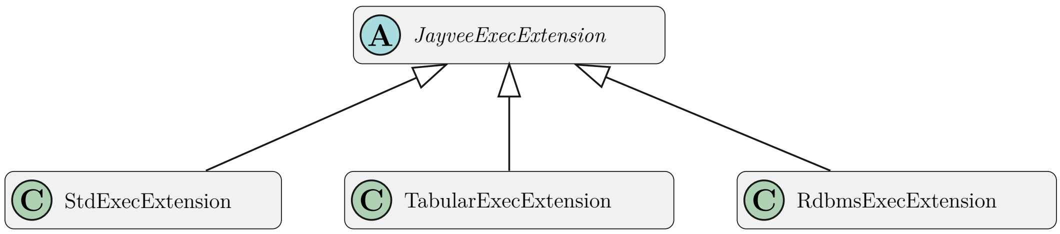


Figure 4.10: The *Jayvee* execution extensions.

The only class implementing `BlockExecutor` is the abstract class `AbstractBlockExecutor`. It exists as an intermediary between `BlockExecutor` and the concrete classes implementing it. Classes extending `AbstractBlockExecutor`, instead of implementing `BlockExecutor`, benefit from its general implementation of the method `execute(...)`, that leaves the execution behavior to the method `doExecute(...)` (see Figure 4.11).

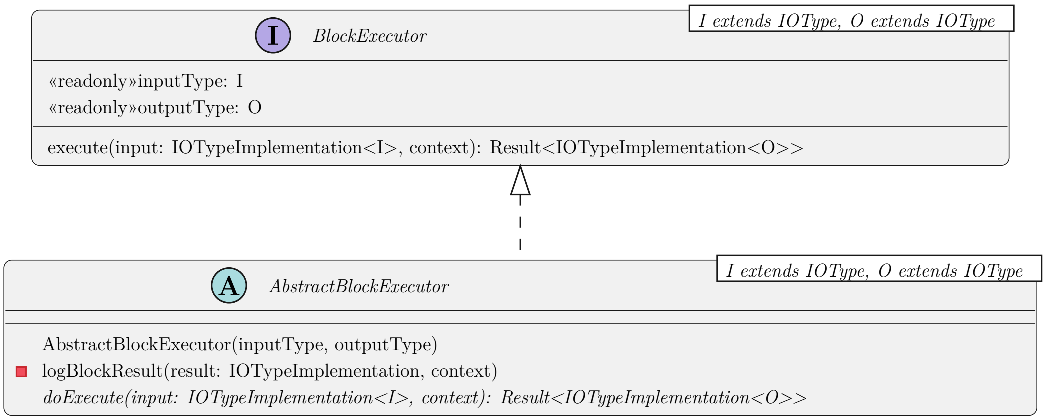


Figure 4.11: The class diagram of `AbstractBlockExecutor`.

4.5.1 TableInterpreter

The block type `TableInterpreter` converts a Sheet, *Jayvee*'s representation for CSV data, into a table. We create an abstract class `TableInterpreterExecutor` that extends `AbstractTableInterpreter<IOType.SHEET, IOType.TABLE>` (Figure 4.12). For implementation details, see subsection 5.4.2. `TableInterpreter` has two subclasses, `TsTableInterpreter`, which contains the old behavior, and the new `PolarsTableInterpreter`.

This architecture, like the table architecture in section 4.4, follows the strategy pattern from Dooley and Kazakova (2024). The usual strategy interface is split up into `BlockExecutor`, `AbstractBlockExecutor` and `TableInterpreterExecutor`, while the concrete strategy is either `TsTableInterpreterExecutor` or `PolarsTableInterpreterExecutor`.

In order to facilitate code sharing, we use static methods to implement behavior that is useful for other executors as well.

4.5.2 FileToTableInterpreter

Blocks with the type `FileToTableInterpreter` receive a `BinaryFile`, which contains a *TypeScript* `ArrayBuffer` and converts that into a table. This combines the block types `TextFileInterpreter`, `CSVInterpreter` and `TableInterpreter`.

Merging these block types allows us to utilize *Polars*' builtin CSV parsing functionality, instead of relying on the `fast-csv` library, like `CSVInterpreter`, which means less indirection when reading a CSV file.

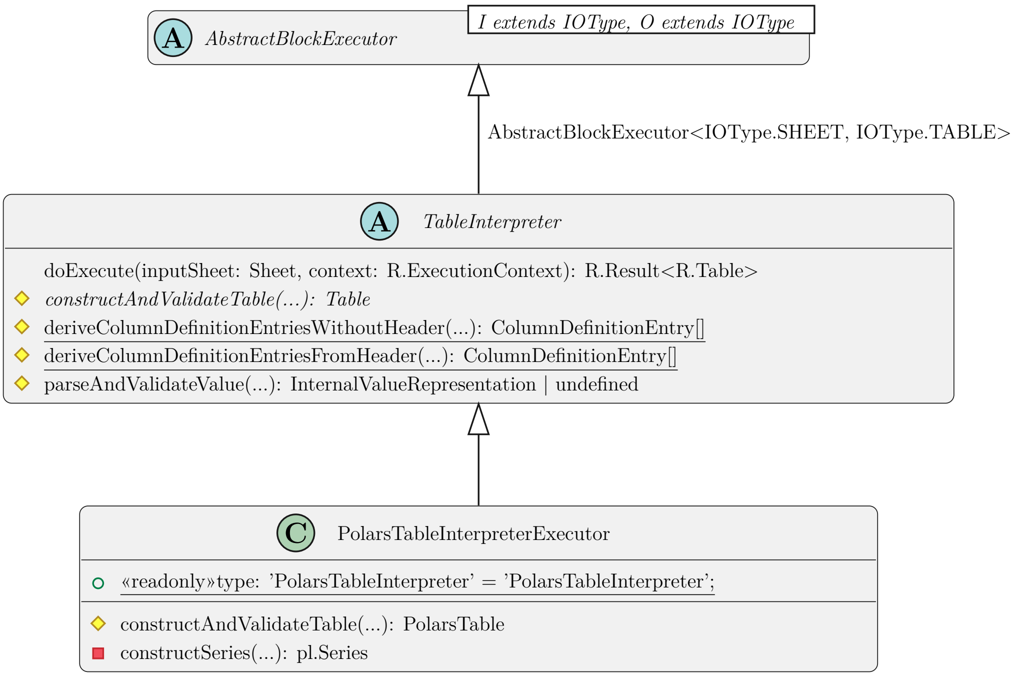


Figure 4.12: The class diagram of `TableInterpreterExecutor`.

Adding a block type

To add this block type to the *Jayvee* language, its definition is put in the file `FileToTableInterpreter.jv` (refer to Table 8 for the full path). Running the command `npm run generate` creates class definitions necessary for this block type to be used in the *Jayvee* interpreter.

We also create an executor for the new block type (Figure 4.13). All methods except `doExecute(...)` are static, making them reusable by other executors.

In practice, the only executor using these, is `LocalFileToTableExtractorExecutor`. Because there is no preexisting implementation of this executor, we do not need an additional abstract class and can extend `AbstractBlockExecutor` directly.

4.5.3 LocalFileToTableExtractor

The block type `LocalFileToTableExtractor` combines `LocalFileExtractor` and `FileToTableInterpreter`. This removes another layer of indirection for parsing CSV data, letting *Polars* handle the entire the process.

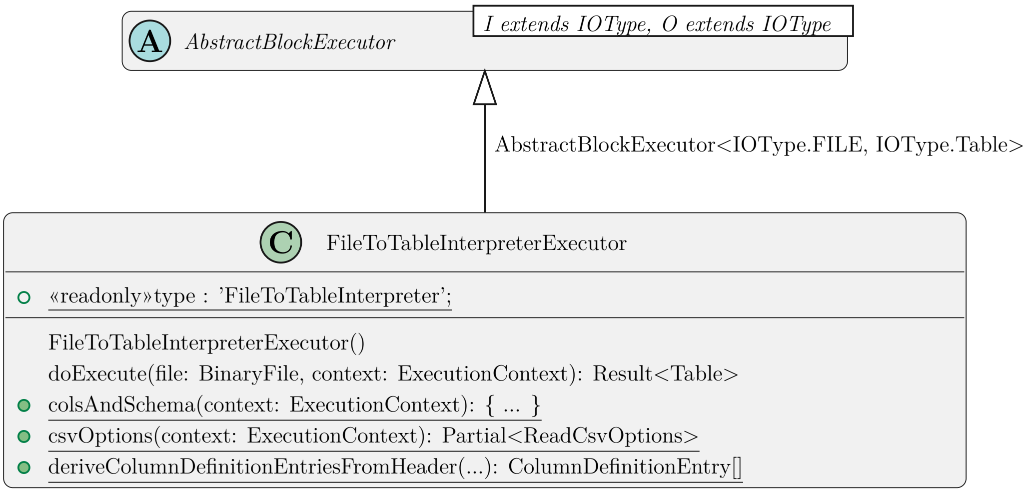


Figure 4.13: The class diagram of `FileToTableInterpreter`.

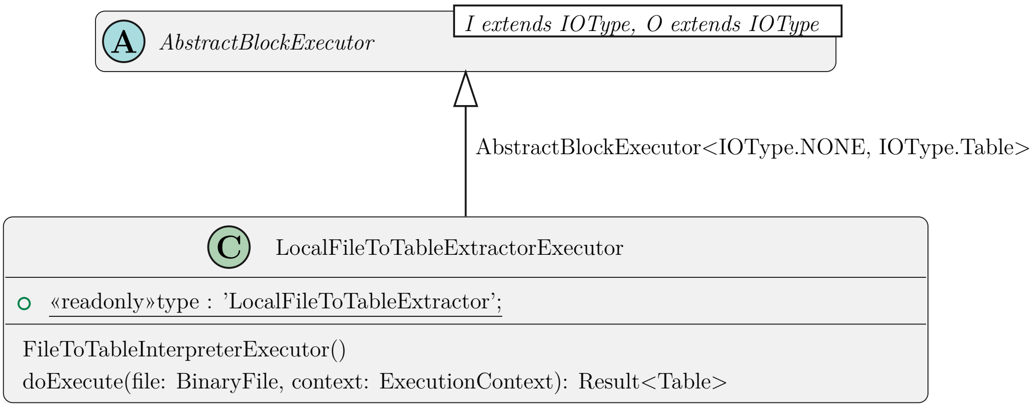


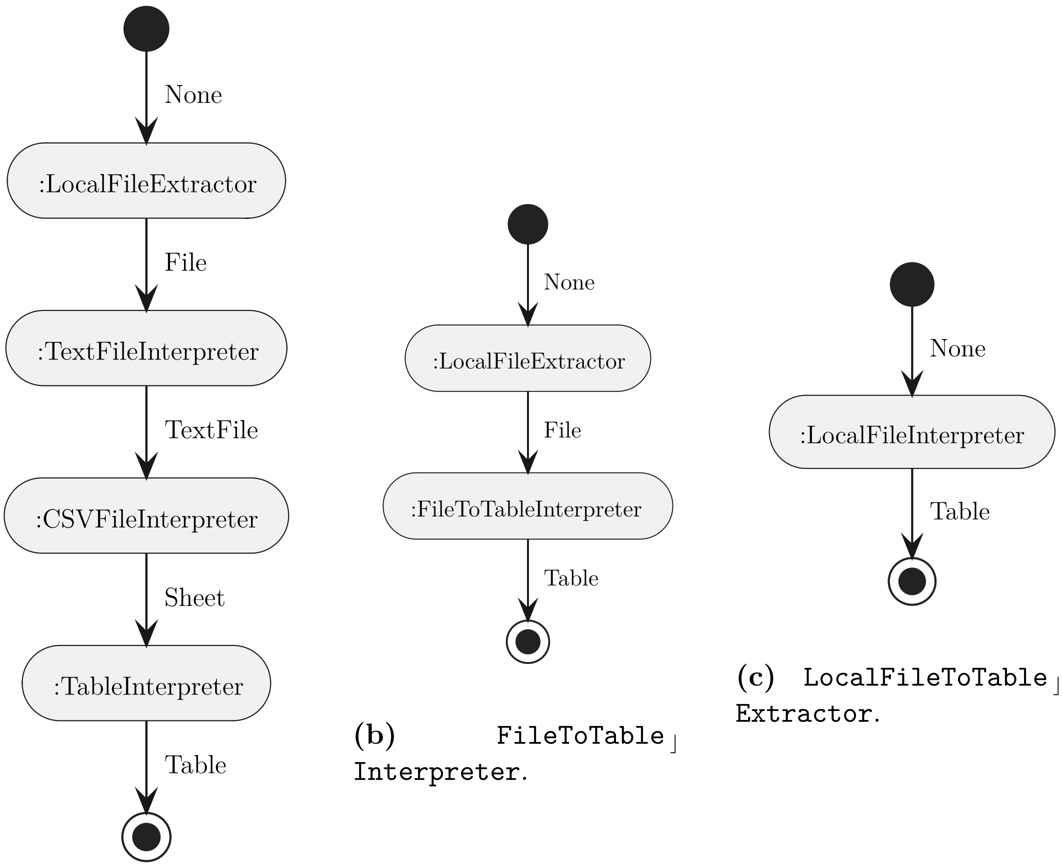
Figure 4.14: The class diagram of `LocalFileToTableExtractorExecutor`.

The process of adding the block type to *Jayvee* is the same as described in subsection 4.5.2, the only difference being that the definition file is placed in `LocalFileToTableExtractor.jv` (see Table 8). Figure 4.14 contains the executor's class diagram.

Figure 4.15 illustrates, how the block types introduced in subsection 4.5.2 and subsection 4.5.3 transform CSV input into a table.

4.5.4 TableTransformer

We still follow the strategy pattern (section 4.2), so, we create another abstract class `TableTransformer` (Figure 4.16) with two subclasses `TsTableTransformer` and `PolarsTableTransformer` (Figure 4.17).



(a) Only original block types.

(b) `FileToTableInterpreter`.

(c) `LocalFileToTableExtractor`.

Figure 4.15: *Jayvee* pipelines using either only original block types, `FileToTableInterpreter` or `LocalFileToTableInterpreter`.

As usual the created abstract class contains some shared behavior, in this case `logColumnOverwriteStatus(...)` and `checkInputColumnsExist(...)`.

Blocks with type `TableInterpreter` use *Jayvee* transforms to compute an output column from a series of input columns. *Jayvee* transforms are executed by `TransformExecutors` (see section 4.6).

4.5.5 SQLiteLoaderExecutor

Jayvee blocks with type `SQLiteLoader` load their input table into a *SQLite* database. Its output `IOType` is `None`, so no blocks in the pipeline can come after a block with type SQLiteLoader. The abstract `SQLiteLoaderExecutor` (Figure 4.18) offers a general implementation, utilizing methods defined in the abstract `Table` class. This approach includes serializing all the data contained in

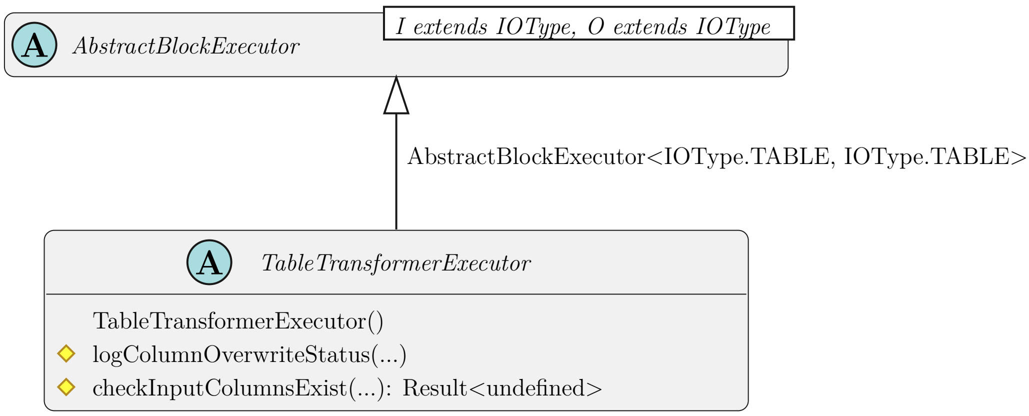


Figure 4.16: The class diagram of `TableTransformerExecutor`.

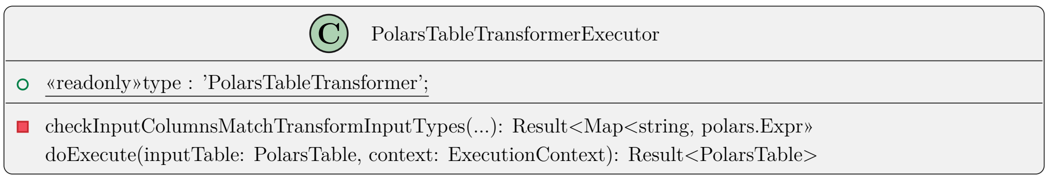


Figure 4.17: The class diagram of `PolarsTableTransformerExecutor`.

the input table into a Structured Query Language (SQL) query with type string. We consider this to be a non-optimal representation of the data and improve it in subsection 4.5.6.

`SQLiteLoaderExecutor` has three subclasses: `TsSQLiteLoaderExecutor` doesn't override the super class' methods to preserve the original *TypeScript* implementation. `PolarSQLiteLoaderExecutor` doesn't override the super class' methods, because *Polars*' *NodeJS* API does not offer any database functionality at the time of writing this thesis. The different behavior results from different overrides of the methods `generateCreateTableStatement` and `generateInsertValuesStatement` in the classes `PolarsTable` and `TsTable`.

RustSQLiteLoaderExecutor

`RustSQLiteLoaderExecutor` avoids serializing the input table's data into a SQL query string, by relying on an external library, *sqlite-loader-lib*.

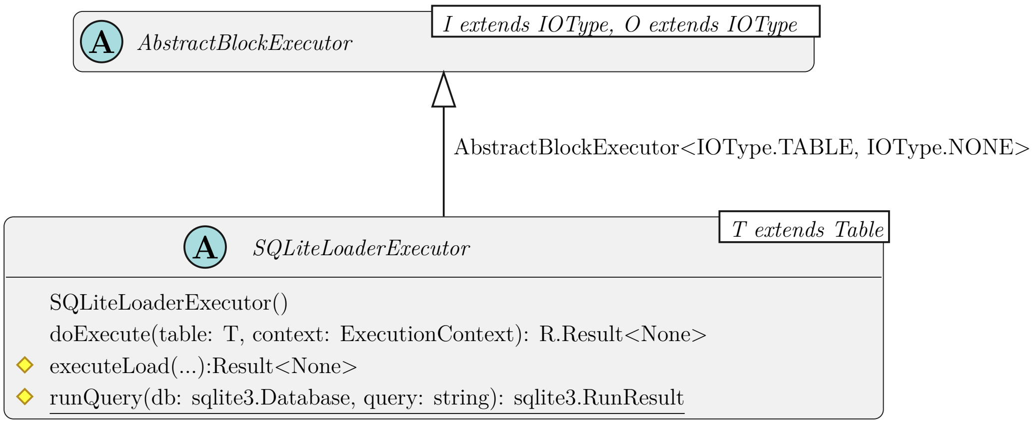


Figure 4.18: The class diagram of `SQLiteLoaderExecutor`.

4.5.6 *sqlite-loader-lib*

Polars' core functionality is written in *Rust*, which Polars Contributors (n.d.-e) describe as allowing "for high performance with fine-grained control over memory". For this reason, we choose *Rust* to implement *sqlite-loader-lib* (see Figure 4.19).

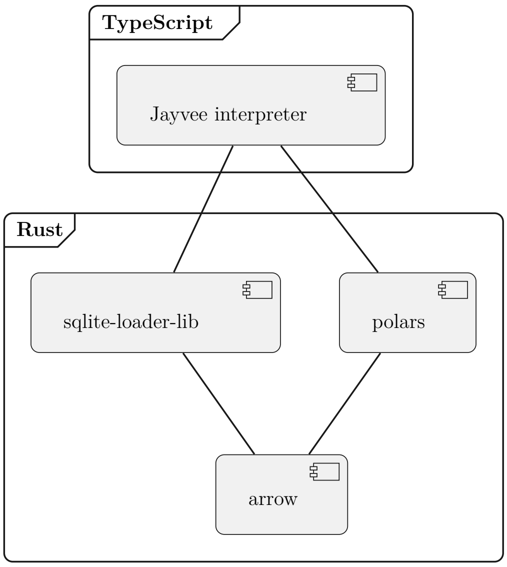


Figure 4.19: The *Rust* components used by the interpreter.

Polars uses *NAPI-RS* (Polars Contributors, n.d.-d), which compiles a *Rust* library to a *NodeJS* addon (NAPI-RS Contributors, n.d.-a). With this, the *Jayvee* interpreter, written in *TypeScript*, can use functions from a library written in *Rust*.

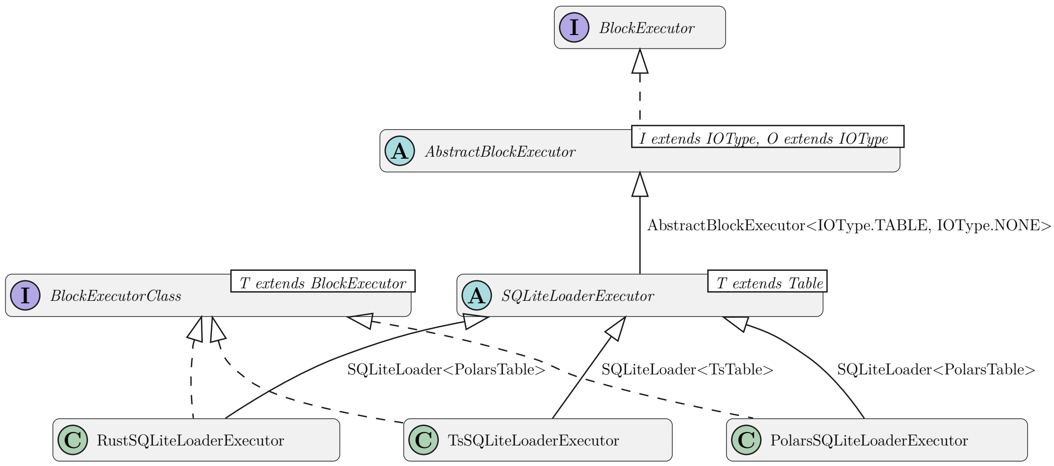


Figure 4.20: The executors for the class `SQLiteLoader`

We were not able to pass the table to the library as a function parameter. This would require redefining the class `PolarsTable` inside *sqlite-loader-lib* using *NAPI-RS*. Instead, the class `RustSQLiteLoaderExecutor` saves its input table into an *Arrow* IPC file on disk. *sqlite-loader-lib* can then read this file and reconstruct the table.

Because the library only exists to execute a single task, it only exposes one function:

```
loadSqlite(ipcPath: string, tableName: string, sqlitePath: string, dropTable: boolean): void
```

Its parameters are the path of the *Arrow* IPC file and the properties of the `SQLiteLoader` block.

At the time of writing this thesis, *Polars*' *Rust* implementation doesn't offer database export either. As a consequence, *sqlite-loader-lib* uses a separate library to interface with a *SQLite* database. This library, known as *Connector Arrow*, enables the writing of *Arrow* record batches into a *SQLite* database table (Eržen, 2024). Connector *Arrow* relies on the *rusqlite* library to support *SQLite* databases (Eržen, 2024).

See section 5.9 for the implementation.

Figure 4.20 contains a diagram of all classes connected to loading the table into *SQLite*.

```
newColumn = ...  # An empty column
expression = transform.expression
for row in inputColumns:
        ...  # add row to evaluation context
        let value = evaluateExpression(expression, context)
        newColumn.add(newColumn)
        ...  # remove row from evaluation context
return newColumn
```

Listing 2: Pseudocode of the old algorithm the interpreter used, to execute transforms.

4.6 Transforms

The `TableTransfromerExecutor` class creates an instance of a concrete subclass of `TransformExecutor` to compute a new column for it's input table. For `PolarsTableTransformerExecutor` this is an instance of `PolarsTransformExecutor` (Figure 4.21), for `TsTableTransformerExecutor` this is `TsTransformExecutor`.

Originally, `TransformExecutor` would compute the new column by following the algorithm outlined in Listing 2. It applies the *Jayvee* expression to each row in the input columns and returns a column.

The new approach transforms the *Jayvee* expression into a *Polars* expression, that the `PolarsTableTransformer` can apply to its input table. So, we add the new function `jayveeExpressionToPolars(...)` (see section 4.7), which is similar to the existing `evaluateExpression(...)`, but it returns a *Polars* expression, not a final value.

A detailed explanation of this process can be found at section 5.7.

4.7 Expressions

Polars Contributors (n.d.-c) separate *Jayvee* expressions into three categories: *literals*, *variables*, *operators*.

literal Concrete values, e.g. `5`, `"sometext"` or `true`. Their evaluation does not differ from the original implementation.

variable Represents a value defined in the evaluation context (Figure 4.22). Variables are used to refer to columns inside of transforms.

operator Transforms one to three *Jayvee* expressions into one result. The evaluation context contains a registry of operator evaluators (see subsection 4.7.1)

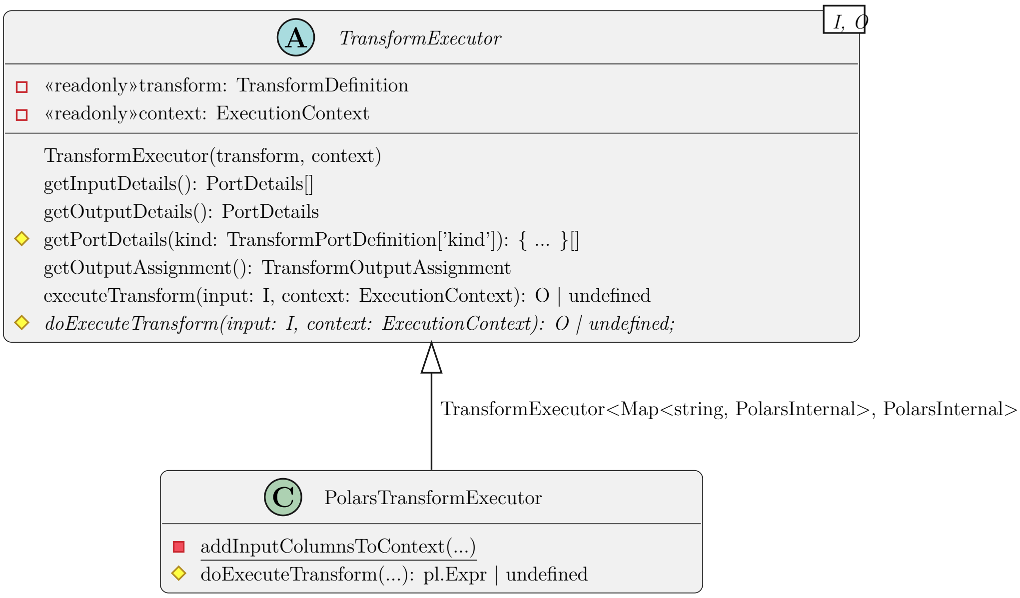


Figure 4.21: The class diagram of `PolarsTransformExecutor`.

Jayvee expressions are transformed to *Polars* expressions using the function `jayveeExpressionToPolars(...)` (see subsection 5.7.2).

4.7.1 Operator evaluators

Evaluators implement the interface `OperatorEvaluator`.

We create a new interface `PolarsOperatorEvaluator`, that defines one method, `polarsEvaluate(...)` and extends the existing `OperatorEvaluator`. As a consequence, all classes that implement `PolarsOperatorEvaluator` are also required to implement the properties defined by `OperatorEvaluator`. This ensures, that the existing *TypeScript* implementation remains available.

All classes, that previously implemented `OperatorEvaluator` interface, now implement `PolarsOperatorEvaluator` (see Figure 5.7.2).

We deliberately did not follow the strategy pattern approach here. *Jayvee* currently has 32 evaluator operators. The strategy pattern requires two additional classes per evaluator, one for the abstract class and one for the *Polars* implementation. This would increase the number of evaluator classes to 96. Because of this, we considered implementing one additional method per evaluator the simpler option.

Figure 4.23 visualizes the inheritance structure for operator evaluators.

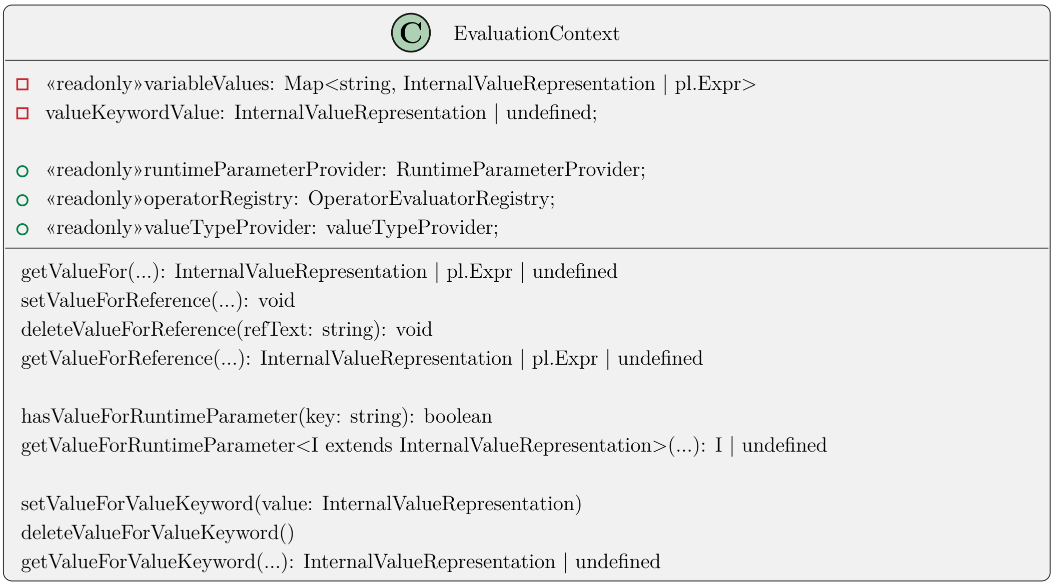


Figure 4.22: The class diagram of `EvaluationContext`.

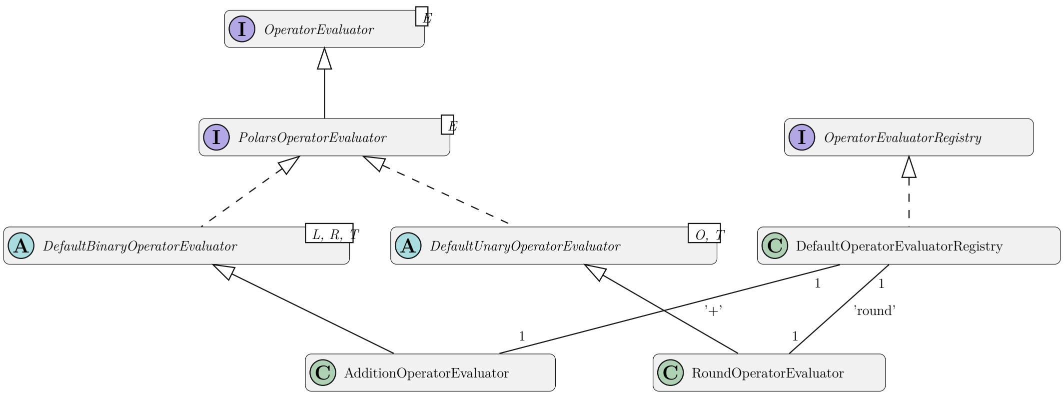


Figure 4.23: Assuming the only operations were `round` and `plus`, this is how the evaluators would be structured.

5 Implementation

This chapter presents implementation of the architecture described in chapter 4. In instances where this thesis makes reference to filenames, the corresponding path can be found in Table 8.

In general, we prefer methods, such as `map(...)`, over for loops, because they are, in our experience, less error-prone and result in more readable code.

5.1 Type Conversion

As described in section 4.3, we implement a conversion mechanism between a *Polars*' `DataType` and *Jayvee*'s `ValueType`.

5.1.1 Conversion from DataType to ValueType

The class `ValueTypeProvider`, located in `primitive-value-type-provider.ts`, implements this method:

```
fromPolarsDType(dtype: polars.DataType): ValueType
```

It uses the method `DataType.equals(...)`, instead of the operator `===`, for a more accurate comparison of between instances of `DataType`. To make this apparent immediately, *if statements* instead of *switch-case* are used.

The *Polars* type system is more granular than that of *Jayvee*, which results in some instances of `DataType`, that yield the same `ValueType`. For instance, both `polars.Float32` and `polars.Float64` return `Decimal`.

Not every *Polars* `DataType` has a corresponding *Jayvee* `ValueType`. Due to the limited timeframe of this thesis, those are unsupported and throw an error.

5.1.2 Conversion from ValueType to DataType

The interface `ValueType`, located in `value-type.ts`, defines this method:

```
toPolarsDataType(): polars.DataType | undefined
```

Classes, that implement `ValueType`, but are not supported by *Polars*, implement this method to return `undefined`.

5.1.3 InternalValueRepresentation

In many cases, the return type of *Polars* methods, that return concrete values, is `any`. However, the *Jayvee* interpreter expects such a value to have the type `InternalValueRepresentation`. To narrow down the returned type `any`, we expand the existing type guards, implemented in `typeguards.ts`, to handle inputs with the type `unknown`. Table 9 contains a list of internal types and the mechanism their type guard uses.

5.2 Table

The overview of existing classes and their relations is given in section 4.4. In this section, we describe the new table implementation and the changes to the original one:

To reduce the code in the source file, `table.ts` and the modularization requirement, we move the abstract class `TableColumn` and its subclasses `PolarsTableColumn` and `TsTableColumn` into a new file, `table-column.ts`.

Wrapping methods

The following `PolarsTable` methods wrap methods of the *Polars*' class `DataFrame`:

- `writeIpc(...)`, `writeIpcTo(...)`
- `nRows()`, `nColumns()`
- `clone()`
- `toString()`

Non wrapping methods

Besides these wrapping methods, it is necessary for `PolarsTable` to implement the following additional methods to remain compatible with the original implementation:

- `getTypes()`
- `withColumn(...)`
- `generateInsertValuesStatement(...)`

- `generateCreateTableStatement(...)`
- `columns()`, `getColumn(...)`

5.2.1 Implementation details

`PolarsTable.clone()` does *not* clone the attribute `PolarsTable.valueTypeProvider`, because it should be treated as a Singleton (see Listing 3).

```
/**
* Should be created as singleton due to the equality comparison
↪  of primitive value types.
* Exported for testing purposes.
*/
export class ValueTypeProvider {
```

Listing 3: Excerpt from `primitive-value-type-provider.ts`.

Because SQL insert values statements expect rows of data, the method `PolarsTable.generateInsertValuesStatement()` transposes the columnar `DataFrame` into a row-oriented format, before converting that to SQL.

The methods `PolarsTable.columns()` and `PolarsTable.getColumn(...)` return object(s) of the class `PolarsTableColumn`. Their construction uses the conversion method from subsection 5.1.1 using the class `PolarsTables`'s attribute, `valueTypeProvider`.

The method `withColumn(...)` faces the challenge of having to discern, whether its parameter is of the type `PolarsTableColumn` or `pl.Expr`. However, the *TypeScript* compiler is aware, that the property `series` only existsn in the class `PolarsTableColumn` and not `pl.Expr`. Consequently, this property in combination with the *TypeScript* operator `in` (MDN Contributors, 2024b) is used to differentiate the parameter's type.

5.3 TableColumn

As described in section 5.2, the class `TableColumn` and its subclasses are implemented in `table-column.ts`.

5.3.1 PolarsTableColumn

Some attribute names are prefixed with an underscore to prevent naming conflicts between the attribute and its getter and setter.

`PolarsTableColumn` is a thin wrapper around `polars.Series`. It doesn't have any methods, that aren't getters, setters or wrappers around equivalent `Series` methods or attributes.

The method `PolarsTableColumn.clone()` excludes the attribute `PolarsTableColumn.valueType`, because, as explained in subsection 5.2.1, objects of the class `ValueType` should not be cloned.

TsTableColumn

All properties are prefixed with an underscore to prevent a naming conflict between the property and its getter and setter.

`TsTableColumn.clone()` produces a *deep clone* of the attribute `_values`. Based on our experience, the usual way to accomplish this is the global function `structuredClone(...)`. This approach caused the interpreter to crash at runtime, reporting a `DataCloneError`. We suspect, that the cause has to do with limitations of `structuredClone(...)` described by MDN Contributors (2024a).

This issue was addressed by serializing the column to JavaScript Object Notation (JSON), and subsequently parsing it, thereby creating deep copies. To retrieve the type information lost during this process, the type guards described in section 5.1 were utilized.

5.4 New block executors

5.4.1 Selecting the correct block executor

The method `getExecutorForBlockType(...)` in `extension.ts` is modified, to look for a different block type than its parameter specifies. Refer to Figure 5.1 for a diagram.

5.4.2 TableInterpreterExecutor

`TableInterpreterExecutor` is implemented in `table-interpreter-executor.ts`. This file also contains the function

```
toPolarsDataTypeWithLogs(valueType: ValueType, logger: Logger):
↪  polars.DataType
```

Its purpose is, to convert a *Polars* `DataType` into a *Jayvee* `ValueType`. It was determined, that an unsupported `ValueType` should be included in the table as an unparsed string, while still logging an error. To this end, the function converts an unsupported `ValueType` to the default value of `pl.Utf8` (see Listing 4).

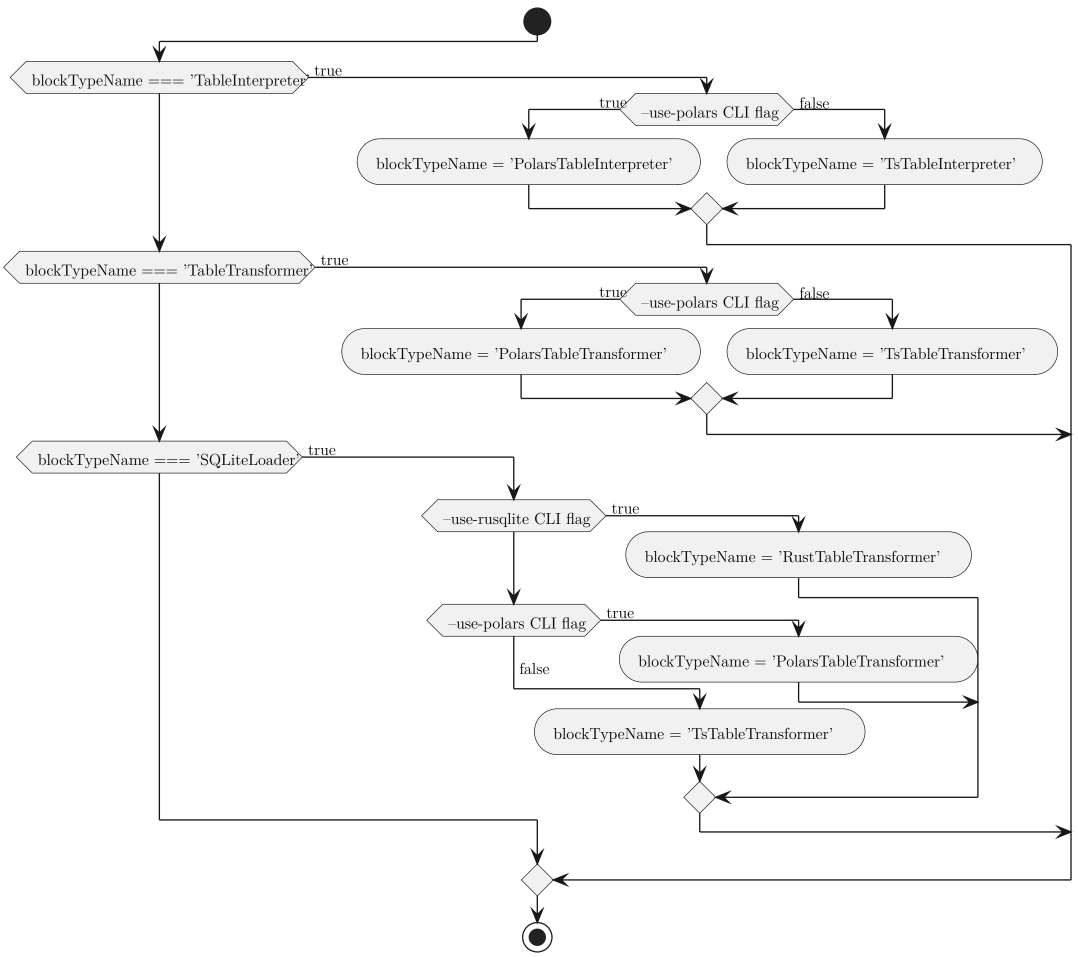


Figure 5.1: How the correct block executor is selected at runtime.

```
def toPolarsDataTypeWithLogs(valueType, logger):
        dataType = valueType.toPolarsDataType()
        if dataType is undefined:
                ... # Log the error using logger
                return polars.Utf8
        return dataType
```

Listing 4: Pseudocode of the function `toPolarsDataTypeWithLogs(...)`.

```
def constructSeries(rows, columnEntry, context):
        valueType = columnEntry.valueType
        dataType = toPolarsTypeWithLogs(valueType, context.logger)
        values = []  # Empty list
        for row in rows:
                cell = row[columnEntry.sheetColumnIndex]
                value = this.parseAndValidateValue(cell,
                 ↪  valueType, context)
                columnData.add(value)

        return polars.Series(columnEntry.ColumnName, values,
         ↪  dataType)
```

Listing 5: Pseudocode of the method `constructSeries(...)`.

The methods `deriveColumnDefinitionEntriesWithoutHeader(...)`, `deriveColumnDefinitionEntriesFromHeader(...)` and `parseAndValidateValue(...)` remain unchanged, because they do not rely on the table implementation.

The method `doExecute(...)` merely preprocesses the block's properties, leaving the concrete algorithm to `TableInterpreterExecutor`'s subclasses.

PolarsTableInterpreterExecutor

The static attribute `type` is set to the value `'PolarsTableInterpreter'`. The method `getExecutorForBlockType(...)` compares this attribute to the block type from the jayvee source file. As a consequence of the alterations, made to the block executor selection process in subsection 5.4.1, the interpreter is now searching for the `PolarsTableInterpreter` instead of `TableInterpreter`, if the `--use-polars` command line interface CLI flag is enabled.

`PolarsTableInterpreterExecutor`'s objective is, to transform the existing, row-oriented, CSV data into a columnar table format. This is addressed by iterating over all rows multiple times, with each iteration constructing only one column (see Listing 5). This process yields a list of *Polars* `Series`, which are then utilized to construct a columnar *Polars* `DataFrame`.

The method `constructAndValidateTable(...)` skips the first row of the input data, if the block property `header` is set to `true`. The use of the ternary if statement is justified by the fact that, it allows `rows` to be declared as a constant.

5.5 FileToTableInterpreter

As the block type `FileToTableInterpreter` is a combination of `TextFileInterpreter`, `CSVInterpreter` and `TableInterpreter` (see subsection 4.5.2), the block type definition (`FileToTableInterpreter.jv`) includes the properties from these blocks. One property, `TextFileInterpreter`'s `lineBreak`, is no longer supported, due to the fact, that line splitting is handled by *Polars*, which does not support line breaks based on a regular expression (regex).

5.5.1 FileToTableInterpreterExecutor

The block executor `FileToTableInterpreterExecutor` is implemented in `file-to-table-interpreter-executor.ts`.

Given the close relationship between the block `FileToTableInterpreter` and `TableInterpreter` (see subsection 4.5.2), it is reasonable to share code with `TableInterpreterExecutor`, in order to enhance both consistency and maintainability. Specifically, the method `TableInterpreterExecutor.deriveColumnDefinitionEntriesWithoutHeader(...)` and the function `toPolarsDataTypeWithLogs(...)` were reused.

Polars only supports the 8-Bit Universal Coded Character Set Transformation Format (UTF-8) encoding, with the option to either crashing when an error occurs, or incorrectly decoding the text. This is different from the original implementation, which supports more encodings. Because this block's objective is to let *Polars* handle text parsing, this is also a limitation of the block type `FileToTableInterpreter`. Given that this is a new block type, we do not consider this limitation to violate the *compatibility* requirement.

The method `doExecute(...)` is deemed sufficiently straightforward, to not need a separate method, such as `TableInterpreterExecutor.constructAndValidateTable(...)`.

5.5.2 LocalFileToTableExtractor

As described in subsection 4.5.3, the block type `LocalFileToTableExtractor` is based on `FileToTableInterpreter`. As a consequence, the block type definition, located in `LocalFileToTableExtractor.jv` includes all of `FileToTableInterpreter`'s properties.

The class `LocalFileToTableExtractorExecutor`, implemented in `local-file-to-table-extractor-executor.ts`, reuses the method `csvOptions(...)` from the class `FileToTableInterpreterExecutor`.

It is not feasible to share code with `LocalFileExtractorExecutor`, because the

TypeScript compiler identifies a circular dependency. As a result, the method `doExecute(...)` has to reimplement a check preventing upward path traversal.

5.6 TableTransformer

The executors for the block type `TableTransformer` are implemented in `table-transformer-executor.ts`.

5.6.1 PolarsTableTransformer

In *Jayvee*, the name of a transform input variable may differ from the name of corresponding column. Additionally, the transform's input variables may have a `ValueType` that is incompatible with the column of the input table.

These issues are addressed by the method `checkInputColumnsMatchTransformInputTypes(...)`. It verifies that the variable and column `ValueType` is compatible and links each input variable to the corresponding *Polars* column expression. When applied to a `DataFrame` this `pl.col(name)` expression enables the transformation of the values in column `name` (Polars Contributors, n.d.-b).

The method `PolarsTransformExecutor.executeTransform(...)` returns a *Polars* expression, not a table column. This has two consequences:

1. `alias(outputColumnName)` is appended to the *Polars* expression, to ensure the final column has the correct name.
2. The method `PolarsTable.withColumn(...)` is used to apply the expression to the table. This is possible, because one of the overload signatures, that allows passing *Polars* expressions.

TsTableTransformerExecutor

Although the *TypeScript* implementation is beyond the scope of this thesis, it is noteworthy, that the method `TsTableTransformerExecutor.createOutputTable(...)` utilizes `TsTable.addColumn(...)` rather than the new `TsTable.withColumn(...)`. This approach is necessitated by the *compatibility* requirement.

5.7 TransformExecutor

Transform executors are implemented in `transform-executor.ts`.

As explained in section 4.6 we will use *Polars* expressions to execute transforms.

```
def doExecuteTransform(variableToColumnName, context):
        inputDetails = this.getInputDetails()
        outputDetails = this.getOutputDetails()
        this.addInputColumnsToContext(
                inputDetails,
                variableToColumnName,
                context.evaluationContext
        )

        try:
                expr = jayveeExpressionToPolars(
                        this.getOutputAssignment().expression,
                        context.evaluationContext
                )
        except Error:
                return

        targetPolarsDataType =
        ↪  outputDetails.valueType.toPolarsDataType()

        # This casts the type of the resulting column to the type
        ↪  defined for the output.
        expr = expr.cast(targetPolarsDataType)

        return expr
```

Listing 6: Pseudocode of the method `doExecuteTransform(...)`.

5.7.1 PolarsTransformExecutor

Subsection 5.6.1 describes, how a transform's variable input name is linked with a *Polars* expression representing the corresponding column. `addInputColumnsToContext(...)` adds these links to the evaluation context (see Figure 4.22), which allows them to be used when evaluating the transform's expression.

The transform executor has to ensure, that the computed column has the correct `ValueType`. To this end, the method `doExecuteTransform(...)` converts this target `ValueType` to a target `DataType` and appends `cast(target)` to the *Polars* expression (see Listing 6).

Figure 5.2 contains a visualization of the calls between `PolarsTableTransformerExecutor` and `PolarsTransformExecutor`.

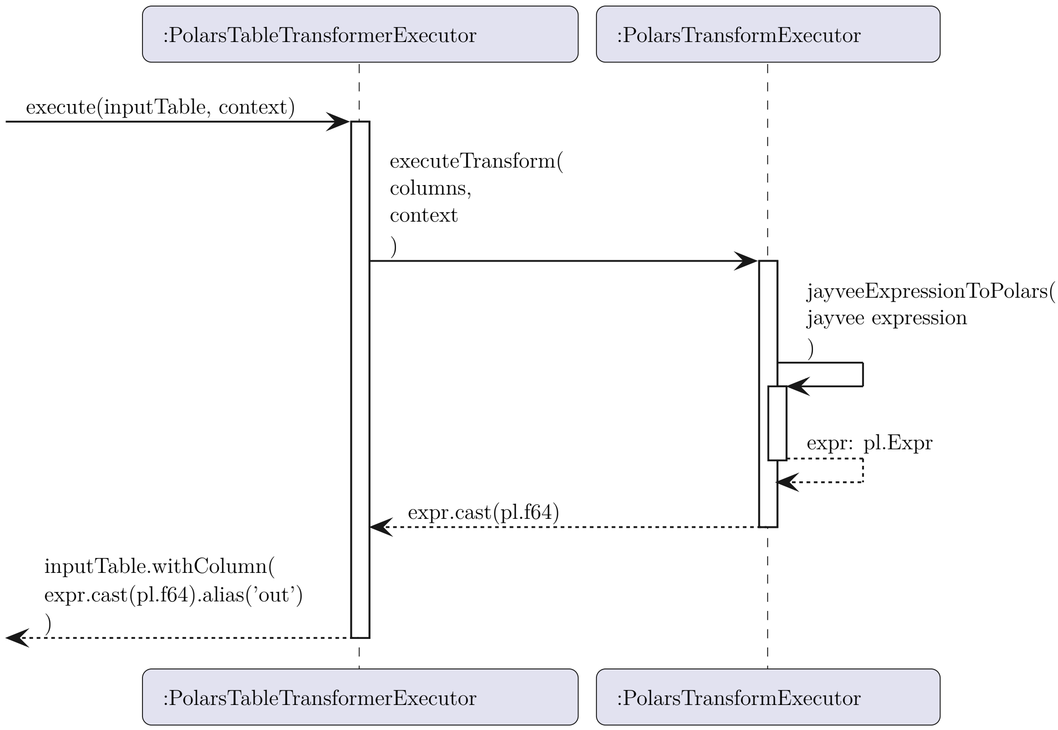


Figure 5.2: Sequence diagram of the method calls between `PolarsTableTransformerExecutor` and `PolarsTransformExecutor`.

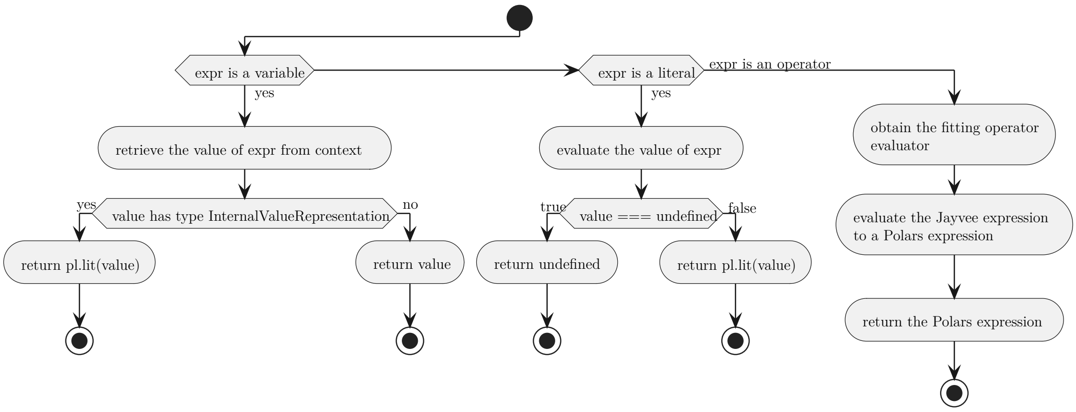


Figure 5.3: Activity diagram of the function `jayveeExpressionToPolars(...)`. `expr` represents the input *Jayvee* expression.

5.7.2 Expressions

As explainded in section 4.7, *Jayvee* expression are either literals, variables or operators. The function `jayveeExpressionToPolars(...)`, located in `evaluate-expression.ts`, transforms *Jayvee* expressions into *Polars* expressions depending on this category (see Figure 5.3).

It utilizes an instance of the class `EvaluationContext` (Figure 4.22) to retrieve a variable's value, or to find an operator's evaluator. `pl.lit` represents a value literal that can be uses as a *Polars* expression. It is useful in expressions such as:

```
pl.lit(10).minus(pl.col("somecolumn"))
```

Operator evaluators

Table 7 contains a table linking every *Jayvee* operator with the *Polars* expression it evaluates to. Many *Jayvee* operators have an equivalent *Polars* expressions. For those that do not, we provide a list of explanations here:

xor Use a logically equivalent term comprised of the expressions `and(...)`, `or(...)` and `not()`.

sqrt Use the mathematically equivalent expression `pow(1/2)`.

round *Polars*' expression `round` expects the number of digits after the comma as a parameter. In order to remain compatible with the *TypeScript* implementation, this number is set to `0`.

```
transform tr {
        from x oftype integer;
        from y oftype integer;
        to z oftype decimal;
        z: x + y;
}
block B oftype TableTransformer {
        inputColumns: ['a', 'b'];
        outputColumn: 'c';
        uses: tr;
}
```

Listing 7: A *Jayvee* snippet defining a block with type `TableTransformer` and its transform

There are also *Jayvee* operators that are not supported by the new implementation:

root, pow, replace, matches The *Polars* expressions for these operators expect single values and do not support columns as parameters.

asBoolean *Polars* supports converting between strings and numeric types with the expression `cast(...)`. This approach is not supported for booleans (Polars Contributors, n.d.-a).

The sequence diagram in Figure 5.4 depicts, how the block `B` from Listing 7 would execute its transform.

5.8 SQLiteLoaderExecutor

`SQLiteLoaderExecutor` is implemented in `sqlite-loader-executor.ts`.

The method `executeLoad(...)` provides a general implementation that works for the both classes `TsTable` and `PolarsTable`, because it uses the abstract `Table` class' methods to generate SQL queries. These methods are:

- `Table.generateDropTableStatement(...)`
- `Table.generateCreateTableStatement(...)`
- `Table.generateInsertValuesStatement(...)`

`PolarsSQLiteLoaderExecutor` and `TsSQLiteLoaderExecutor` do not override this default implementation.

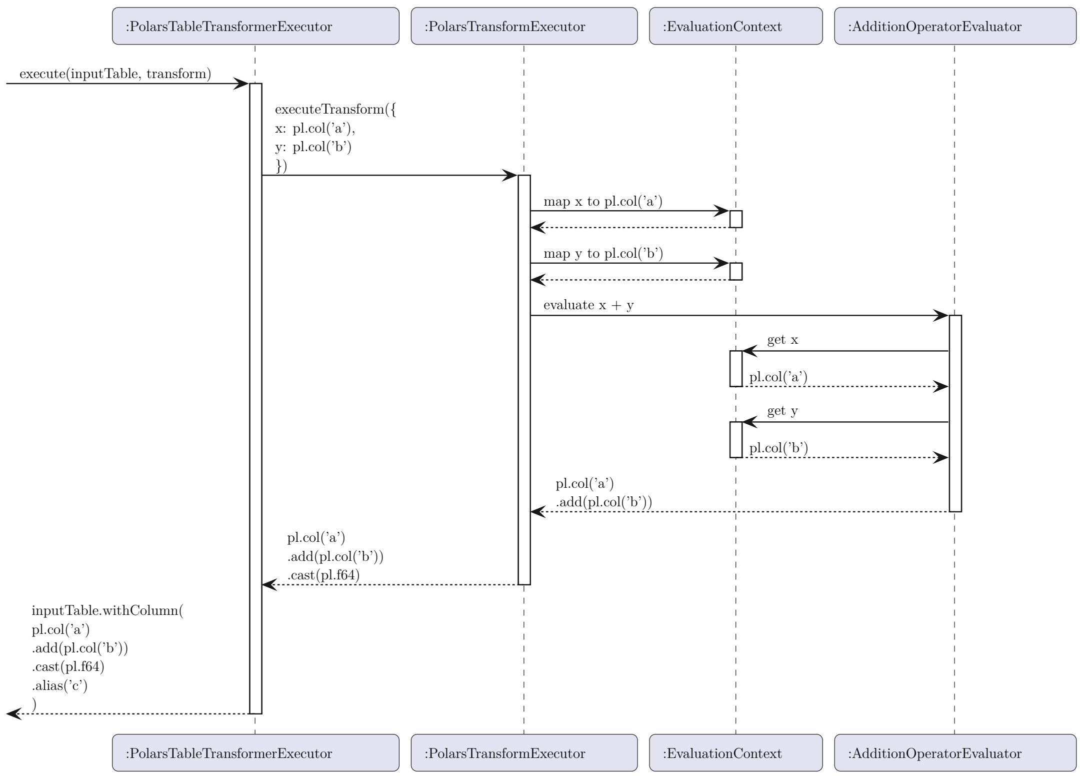


Figure 5.4: The method calls relevant to transforming a table.

The subclass `RustSQLiteLoaderExecutor` overrides the method `executeLoad(...)`. Its implementation writes the table to `dataframe.arrow` (the `.arrow` extension is recommended by the *Arrow* specification (The Apache Software Foundation, n.d.-b)), using `PolarsTable`'s `writeIpcTo(...)` method. Then it calls the `loadSQLite(...)` function implemented in *sqlite-loader-lib*.

5.9 *sqlite-loader-lib*

The library *sqlite-loader-lib* is implemented in a new directory outside the *Jayvee* repository. Attempts were made to implement *sqlite-loader-lib* within the *Jayvee* repository. Regrettably, these efforts were unsuccessful. The reason for this is, that interpreter's build system was unable to integrate with *NAPI-RS*.

NAPI-RS Contributors (n.d.-b) provide simple setup instructions for a library built with *NAPI-RS*. `lib.rs` contains the functionality. To compile the library, the CLI command:

```
npm run build
```

is executed.

As described in subsection 4.5.6, *sqlite-loader-lib* only exposes the function `loadSqlite(...)` (see 5.5). Due to the experimental nature of this library, we do not implement complex error recovery. When an error occurs, it is used to construct an instance of the `napi::Error` struct. *NAPI-RS* can convert this instance into an Error object usable by *TypeScript*.

subsection 4.5.6 describes, why tables cannot be passed to *sqlite-loader-lib* as function parameters. This also applies to *Jayvee*'s class `Logger`, meaning it is inaccessible from the *Rust* library. As an alternative, log messages are printed using the *Rust* macro `println!(...)`, which is similar to *TypeScript*'s `console.log(...)`.

Integrating *sqlite-loader-lib* into the interpreter

sqlite-loader-lib is made available to the *Jayvee* interpreter by the CLI command

```
npm install --save <PATH TO sqlite-loader-lib>
```

The *Jayvee* interpreter's build process includes merging the entire source code into a single file. This is not feasible, given that the library is only accessible to the build process as a binary file, rather than as source code.

In order to instruct the build process to exclude *sqlite-loader-lib* from the final source code file, we append the string `"sqlite-loader-lib"` to the JSON array `targets.build.options.external` inside the files listed in subsection C.2.

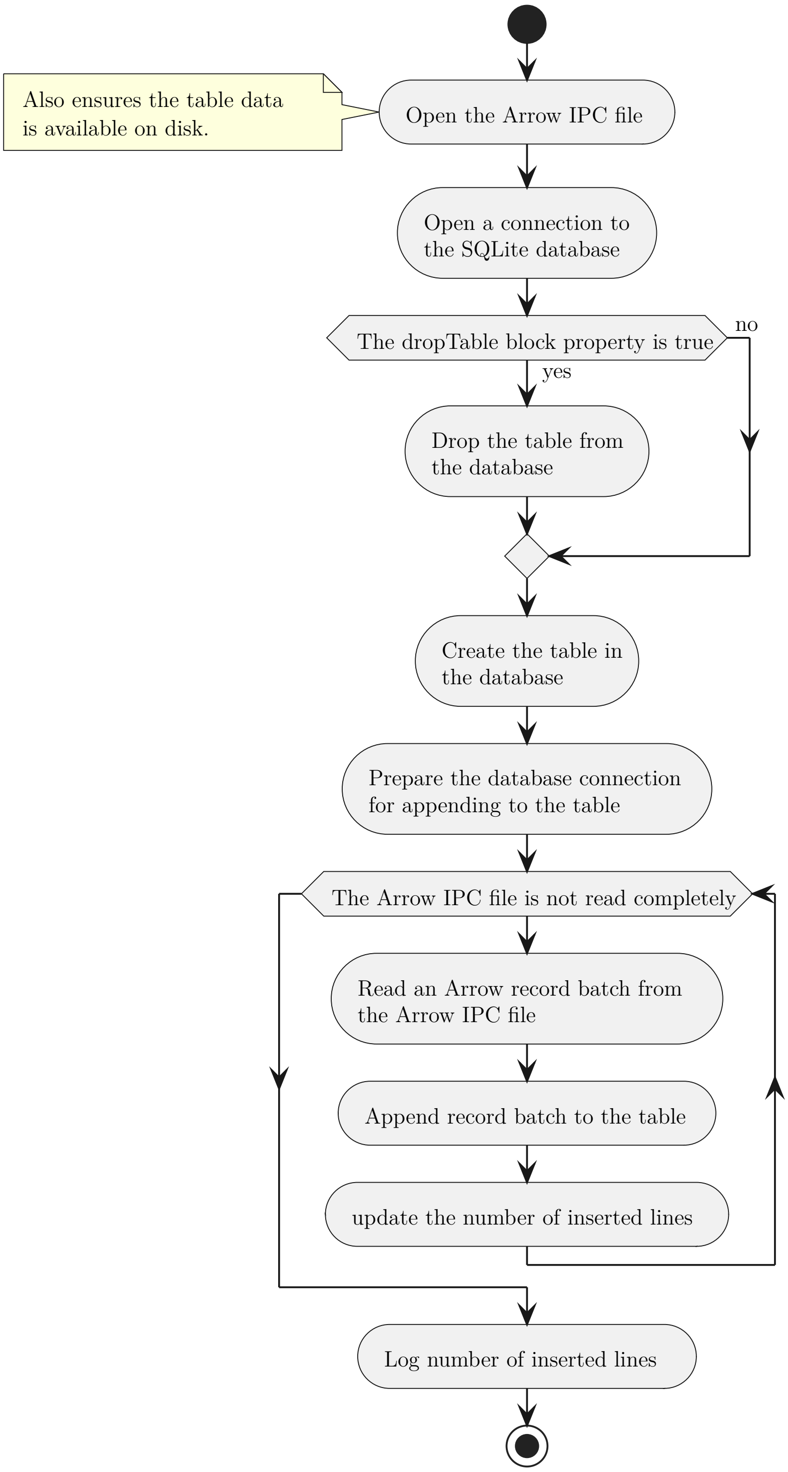


Figure 5.5: The activity diagram of the function `loadSqlite(...)`.

6 Evaluation

The Evaluation will focus on the execution time of *Jayvee* pipelines, and how it has changed due to the optimizations described in previous chapters.

6.1 Data source

Jayvee pipelines require a data input with the following requirements.

6.1.1 Requirements

CSV format The dataset has to be available in a CSV format, because the interpreter can only create tables from CSV data.

openness The dataset must have a license compliant with the Open Definition (Open Knowledge Foundation, n.d.-b). Open Knowledge Foundation (n.d.-a) provides a list of recommended and compliant licenses.

6.1.2 Chosen dataset

The selected dataset is entitled "Brewery Operations and Market Analysis" (Napa, 2023). The dataset does not contain real world data. The dataset is licensed under the Open Data Commons Open Database License (ODbL). All data is contained in a single CSV file, eliminating the need for table joins, which are not supported by *Jayvee*.

Dataset biases

An examination of the dataset revealed no evidence of bias that would distort the results.

6.2 Parameters

To measure the performance of the new implementation, the evaluation includes the execution of the *Jayvee* interpreter with different configurations. A configuration is characterized by the enabled optimizations, the amount of transforms in the pipeline, and the number of rows in the input CSV file.

We define the following backends:

TS TypeScript. The baseline for the evaluation. The terms "TS backend" and "baseline" are used interchangeably.

PL Polars. Enables the *Polars* implementations of tables, columns, block executors and transforms.

PLOB Polars-One-Block. Enables PL, plus the `LocalFileToTableExtractor` block (see subsection 4.5.3).

PLRS Polars-Rusqlite. Enables PL, plus the `RustSQLiteLoaderExecutor` executor and the *sqlite-loader-lib* library (see subsection 4.5.6).

PLOBRS Polars-One-Block-Rusqlite. Enables PL, plus `LocalFileToTableExtractor`, plus `RustSQLiteLoaderExecutor`.

This backend is not usable for every pipeline. For example, pipelines that download files instead of reading a local one, it cannot use the block `LocalFileToTableExtractor`. However, more blocks analogous to this one could be created to be used in such cases.

We create three pipelines with the following amount of transforms:

none The pipeline does not transform the data.

some There are four transforms in the pipeline.

many There are eight transforms in the pipeline.

In the remainder of this chapter, the term "some transforms" refers to the pipeline with four transforms. Similarly, the term "many transforms" refers to a pipeline with eight transforms.

We set six values for the amount of input rows: 56250, 112500, 225000, 4500000, 900000 and 1800000

A configuration picks one value from each category.

6.3 The evaluation tool

In order to automate the execution of the *Jayvee* interpreter with the correct configurations, a CLI program is created, the evaluation tool (see Figure 2). The tool creates a list of all possible configurations and runs each of them 10 times, saving the execution duration. It also uses `sqldiff` CLI program (SQLite Contributors, n.d.), to verify that the created databases are identical. Barton (2022) provides example code, showing how to time execution duration in *Rust*.

6.3.1 Running a configuration

Running a configuration involves creating a source file with the correct amount of lines, selecting the correct *Jayvee* source file, and passing the location of the source and destination files as command line arguments.

The `head` CLI program outputs the first lines of a file (MacKenzie & Meyering, 2024). Except for the header line, one line in the CSV file represents one row of data. Consequently, executing the CLI command

```
head --lines=<NUMBER_OF_ROWS + 1> brewery_data_all.csv >
↪  l-<NUMBER_OF_ROWS + 1>.csv
```

generates a CSV file with the requisite number of rows.

Given that there are three distinct numbers of transform amounts, three separate *Jayvee* source files are created. For each of these, a variant incorporating the `LocalFileToTableExtractor` block is required, resulting in a total of six. Figure 1 outlines the process by which the evaluation tool select the correct source file.

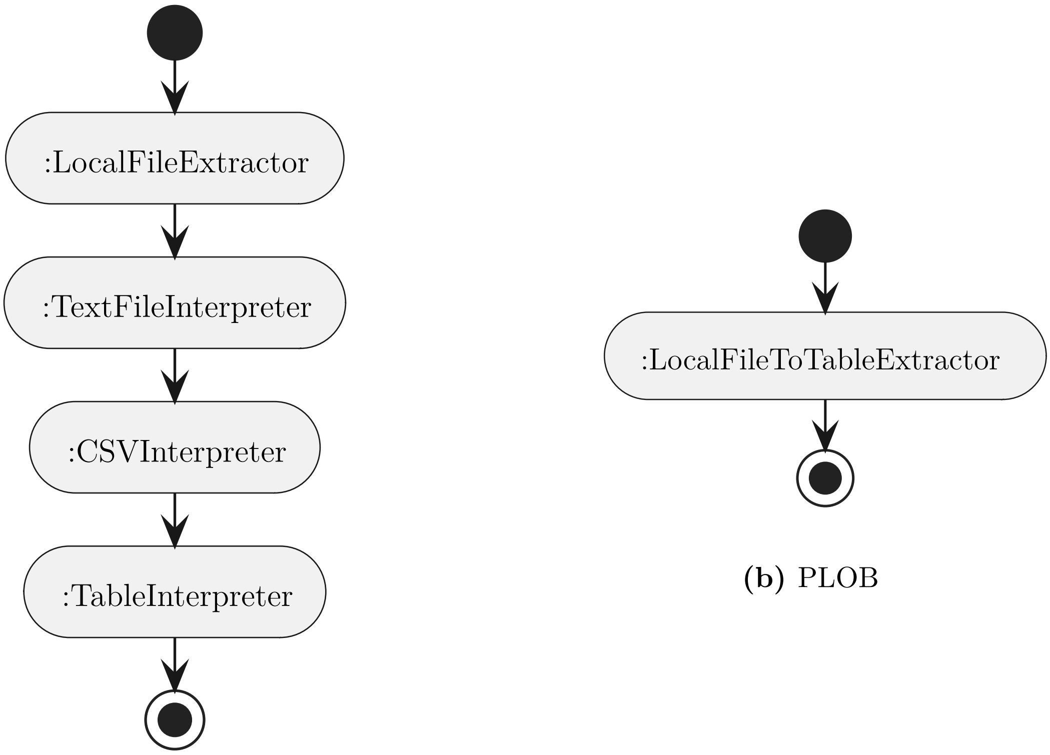
The path of the CSV data source, as well as the path of the destination database, are passed to the *Jayvee* interpreter via runtime parameters (JValue Contributors, n.d.-d).

Given that the interpreter is required to be executed via a shell command, it was determined, that shell commands would be preferred over libraries for other functionality, such as comparing `.sqlite` files, or creating the input data files.

Listing 8 outlines the whole process of running a configuration.

6.3.2 Evaluation pipelines

This subsection describes the *Jayvee* pipelines that get executed during the evaluation.



(a) TS

(b) PLOB

Figure 6.1: The initial section of *Jayvee* pipelines in *Jayvee* files starting with TS or PLOB. Presents the block types and their associated `IOType`.

In total, there are six different pipelines, each with its own file. The pipelines differ in two ways: how they extract the data from the data file and how many transforms are in the pipeline. The former way is visualized in Figure 6.1, the latter in Figure 6.2.

The following is an explanation of the transforms used inside the evaluation pipelines.

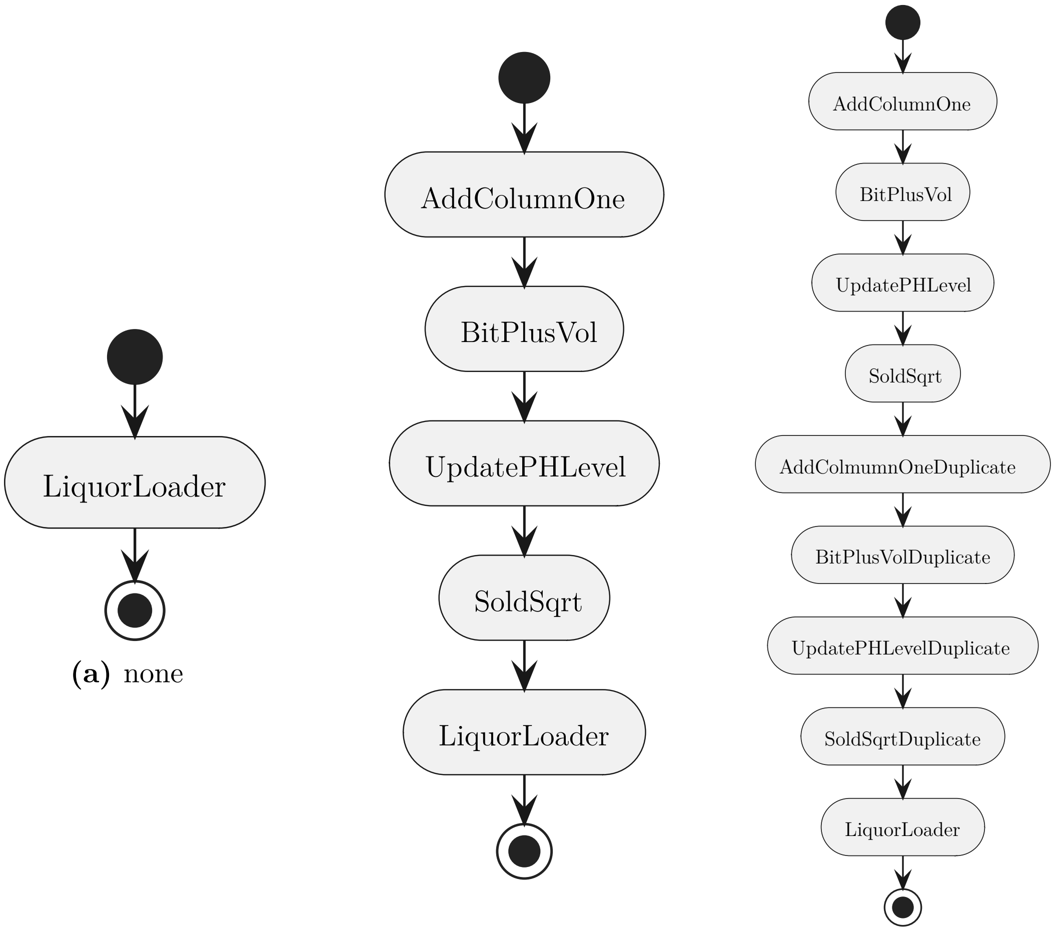
AddColumnOne Adds a column filled with the value one.

BitPlusVol Adds a column containing the sum of the columns ”Bitterness” and ”Volume_Produced”.

UpdatePHLevel Multiplies the ”ph_Level” column by 10000.

SoldSqrt Adds a column containing the square root values of the ”Total Sales” column.

The implementation of string operations, such as `lowercase`, was not completed until after the conclusion of the evaluation process. Consequently, the aforementioned pipelines only contain numeric operations.



(a) none

(b) some

(c) many

Figure 6.2: The transform section of pipelines with none, some or many transforms. The block types have been omitted for readability.

6.4 Maximum size of the input data

During the evaluation process, we encountered *Jayvee* interpreter crashes (see Table 6). We observed, that reducing the size of the input file appeared to prevent these crashes. We subsequently narrowed down the exact limit to a range of 100000 lines, or 25 Megabyte (MB). This limit depends on the enabled optimizations and the amount of transforms in the pipeline (see Figure 6.3).

Table 6 contains a table of maximum file sizes for each configuration and a snippet of the crash's error message. Because the biggest data input (1800000 rows) exceeds the limits for pipelines with some or many transforms, we introduce a seventh data input with 1300000 rows. The configurations with many transforms will only process data inputs with 900000 rows or lower.

Based on our knowledge of the interpreter, the "Invalid string length" error originates from a *NodeJS* limit on the maximum length of a string. The observation, that the error occurs during the loading of the table into the *SQLite* database, suggests, that the string limit may be exceeded during the generation of the SQL queries.

The error message "JavaScript heap overflow", which also occurs is the `SQLite␣Loader` block, indicates that the heap can not contain the data and the SQL queries.

The error message "Cannot make a string longer than 0x1fffffe8 characters." is emitted, during the execution of a block with type `TextFileInterpreter`.

The PLOBRS backend processed 10000000 rows (2.5 Gigabyte (GB)) without crashing.

Considering these results, we conclude, that enabling optimizations increases the amount of data, that the *Jayvee* interpreter is able to process.

6.5 Execution Duration

In this section, we will compare the execution duration of the configurations defined in section 6.2. We group the configurations based on the number of transforms in the executed pipeline.

Figure 6.4, Figure 6.5 and Figure 6.6 illustrate the average execution time for each backend. The precise numbers can be found in subsection A.1.

PL It is somewhat unexpected, that the PL backend performs worse than the TS baseline twice. To investigate this, the debug outputs of both these configurations (Table 2) were compared. It was observed, that the duration

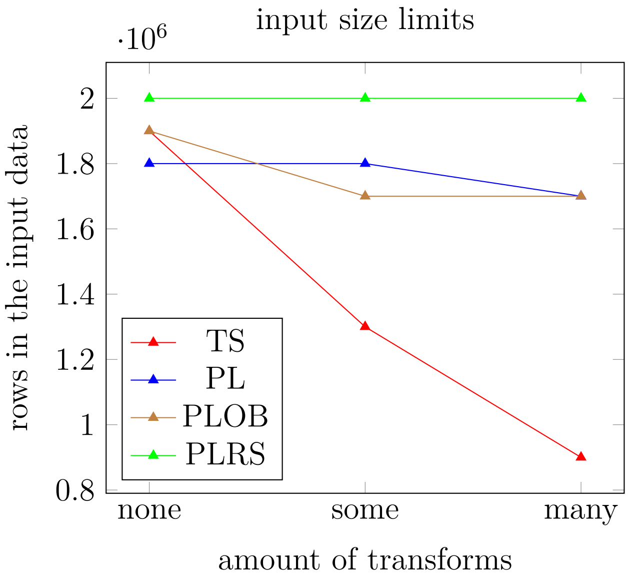


Figure 6.3: Maximum input sizes for each backend.

of the block `LiquorLoader` approximately doubled when the backend PL was enabled. We determined this to be the cause of the worse performance.

We observed, that with no transforms, PL was 1.33 times slower than the baseline TS. With some transforms, this factor decreased to 1.09. With many transforms PL was 1.07 times faster than the baseline TS. This trend suggests, that the *Polars* implementation of table transforms is superior to the original *TypeScript* one.

PLOB The PLOB backend performs better than the baseline on all tested pipelines and inputs. This indicates, that the block type `LocalFileTo`⌋`TableExtractor` (subsection 4.5.3) is a more effective method of parsing local CSV data, than the existing block types.

PLRS The PLRS backend also performs better than the baseline on all tested pipelines and inputs. This is compelling evidence, that the combination of the executor `RustSQLiteLoaderExecutor` and the library *sqlite-loader-lib* is superior to the original implementation. Furthermore, PLRS outperforms than PL in all tested situations. This means, that the external library is more effective than the approach described in section 5.8.

PLOBRS The PLOBRS backend combines the optimizations from the PLOB and PLRS backends, thereby achieving the highest processing speed.

Figure 6.7 shows the factor, by which PLOBRS is faster than the baseline. The three curves are monotonously rising, which evidences the assertion, that as the number of rows increases, the PLOBRS backend's processing speed compared to TS increases. Furthermore, it can be observed, that the curve representing a pipeline with many transforms, lies above the curve representing some transforms, which itself lies above the curve representing no transforms. This evidence substantiates the claim, that the greater the number of transforms in a pipeline, the faster the PLOBRS backend is in comparison to the TS baseline.

Due to the fact, that the PLOB, PLRS, and PLOBRS backends have been demonstrated to be faster than the TS baseline in all tested circumstances, we conclude that they are successful optimizations. The PL backend can only be considered an optimization, if the pipeline has sufficient transforms.

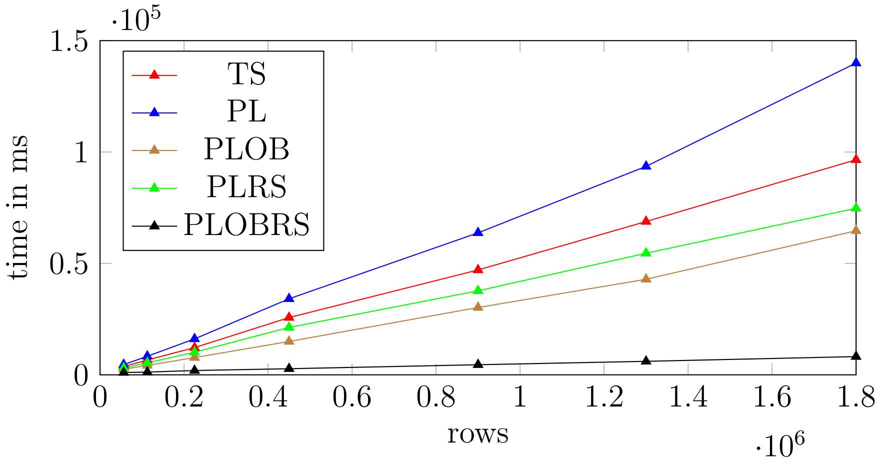


Figure 6.4: no transforms.

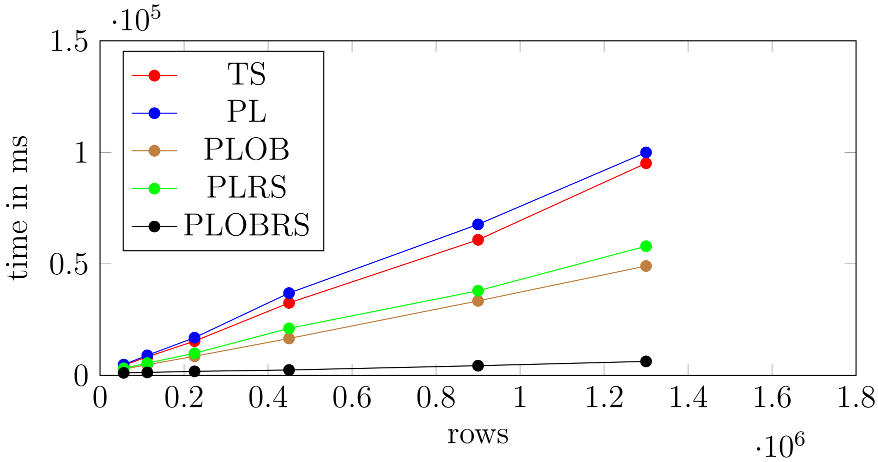


Figure 6.5: some transforms.

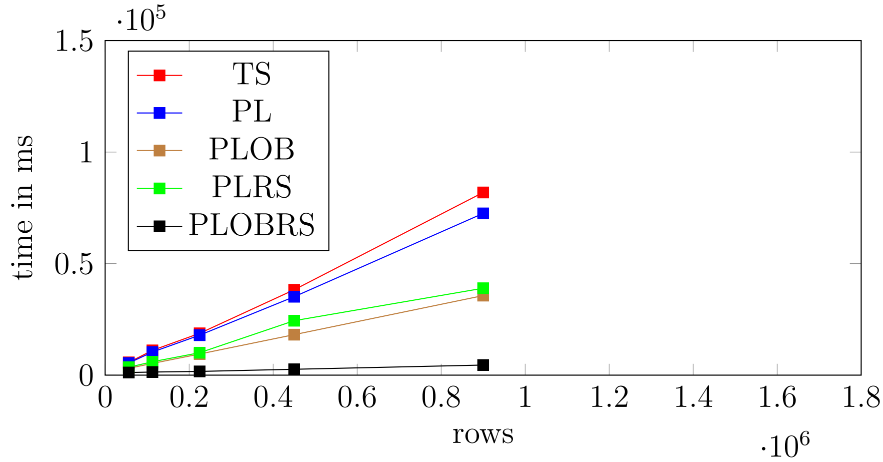


Figure 6.6: many transforms.

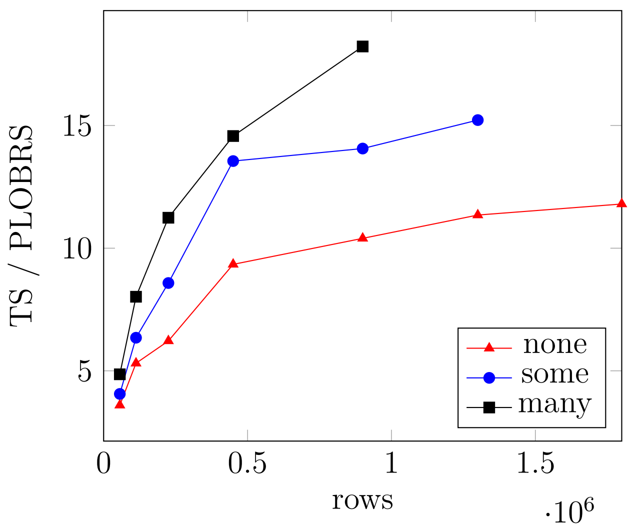


Figure 6.7: The processing speed of TS compared to PLOBRS.

6.6 Differences in the resulting tables

The backends PL, PLOB, PLRS and PLOBRS yield tables that differ from those produced by the original implementation in two notable ways: floating point numbers and rows with `NULL`.

6.6.1 Floating point values

During the evaluation process, it was observed, that seemingly random floating-point numbers in the final tables exhibited slight discrepancies. To investigate this phenomenon, a pipeline with some transforms and an input of 900000 rows, was executed, once with the TS backend and once with the PL backend. The PL backend was selected, because it includes the least amount of changes compared to the baseline.

A comparison of the resulting tables revealed a total of 838 differing values, indicating that one such value occurs approximately every 1074 rows. Additionally, it was observed, that the differing values were confined to columns generated by transforms. The value emitted by the *Polars* backend was found to deviate from the value emitted by the *TypeScript* backend by an average of 8.53×10^{-9}. This led to the conclusion that there are discrepancies in the floating point operations implemented by *TypeScript* and *Polars*. However, the underlying cause and potential solution to this discrepancy remain uncertain.

6.6.2 Rows including `NULL`

The original *TypeScript* implementation either discards rows containing `NULL`, or replaces them with an empty string. This is inconsistent with the new implementation, which allows for `NULL` values in tables. While this issue has not been resolved due to the time constraints of this thesis, but we are optimistic that it can be addressed in the future.

6.7 Reevaluating the requirements

In order to evaluate the success of this thesis, it is necessary to revisit the defined requirements and determine whether they have been met.

6.7.1 Functional requirements

interoperability The standardized *Apache Arrow* IPC format is used by `Rust`⌋`SQLiteLoaderExecutor` and *sqlite-loader-lib* (see subsection 4.5.6). This requirement is fulfilled.

columnar The class `PolarsTable` uses the *Polars* library, which implements the *Apache Arrow* specification, a columnar format (see subsection 4.4.2). This requirement is fulfilled.

feature toggle The implementation can be chosen with the interpreter's CLI flags `--use-polars` and `--use-rusqlite` (see subsection 5.4.1). This requirement is fulfilled.

compatibility The electric vehicles example completes without errors. However, discrepancies still exist:

- Not all operators are supported by the new implementation (see Table 7).
- It is not possible to convert every *Jayvee* `ValueType` to a *Polars* `DataType`, nor vice versa (see section 5.1).
- The tables produced by the new implementations are not consistent with the old implementation (see section 6.6).

Consequently, this requirement remains unfulfilled.

modularization The *Rust* library *sqlite-loader-lib* is usable from *TypeScript* through *NAPI-RS* (see subsection 4.5.6). This requirement is fulfilled.

extensibility `RustSQLiteLoaderExecutor`, using *sqlite-loader-lib*, exports a `DataFrame` into an *SQLite* database. Such functionality is not yet implemented in *Polars* (see subsection 4.5.6). This requirement is fulfilled.

We conclude, that all functional requirements, except for compatibility, have been met.

6.7.2 Non-functional requirements

performance The optimizations afforded by the new implementation enhance processing speeds (see section 6.5) and the new implementation is capable of processing more input data without crashing (see section 6.4). This requirement is fulfilled.

code style `npm run lint` only throws errors for files out of the scope of this thesis. *Rust*'s linter *clippy* and formatter *rustfmt* do not report any issues. This requirement is fulfilled.

maturity This thesis implemented a prototype. This requirement is fulfilled.

We conclude, that all non-functional requirements have been met.

7 Conclusions

In this thesis, we created an alternative table implementation for the *Jayvee* interpreter. This prototype is based on the *Polars* library. By using this library, the prototype implements the *Apache Arrow* specification.

The strategy pattern is used, to allow for the original implementation to be preserved alongside the implementation presented in this thesis. It also allows the interpreter's users to choose the implementation at runtime.

The external library *sqlite-loader-lib* was developed. This library implements database functionality that is not present in *Polars*. *sqlite-loader-lib* was shown to be more performant than the other approaches. However, we suspect, that an implementation of database functionality within the *Polars* library, may potentially be even more performant.

The new optimizations, enabled by the new implementation, were shown to increase the maximum input size, 475 MB of the *Jayvee* interpreter. With all optimizations enabled, the 2.5 GB evaluation dataset could be completely processed.

We also demonstrated, that simply replacing the table implementation is not enough to reduce the average duration of a pipeline. When more optimizations were enabled, the new implementation was always faster than the old one. With all optimizations enabled, the speedup factor was between 3.60 and 18.22. This factor increased with a larger input and more pipelines.

Further work is required to address the identified compatibility issues. It is believed that the remaining unsupported *Jayvee* operators can be implemented and that `NULL` values can be handled appropriately. However, the cause and potential solution for the seemingly random differences in floating-point numbers remain unclear.

Appendices

A Tables

A.1 Evaluation results

Table 1: Average execution duration and standard deviation for a pipeline with no transforms (10 repetitions).

rows	TS	PL	PLOB	PLRS	PLOBRS
56250	3724 ms 43 ms	4581 ms 54 ms	2531 ms 35 ms	3102 ms 45 ms	1035 ms 24 ms
112500	6488 ms 80 ms	8323 ms 78 ms	4231 ms 52 ms	5390 ms 83 ms	1221 ms 29 ms
225000	12202 ms 174 ms	16118 ms 513 ms	7739 ms 69 ms	10126 ms 1148 ms	1963 ms 362 ms
450000	25701 ms 1305 ms	34125 ms 1314 ms	14976 ms 196 ms	21270 ms 1121 ms	2740 ms 608 ms
900000	47064 ms 696 ms	63728 ms 1157 ms	30211 ms 533 ms	37705 ms 932 ms	4526 ms 1270 ms
1300000	68801 ms 1091 ms	93559 ms 823 ms	42888 ms 335 ms	54562 ms 1116 ms	6060 ms 1837 ms
1800000	96513 ms 1785 ms	139939 ms 9652 ms	64660 ms 4557 ms	74755 ms 7137 ms	8184 ms 2081 ms

Table 2: Average duration of the block `LiquorLoader` for a pipeline with no transforms (10 repetitions).

rows	TS	PL	PL / TS
56250	850 ms	1857 ms	2.18
112500	1892 ms	3677 ms	1.94
225000	3591 ms	7356 ms	2.05
450000	7341 ms	14549 ms	1.98
900000	14830 ms	30934 ms	2.09
1300000	22516 ms	46691 ms	2.07
1800000	24720 ms	61527 ms	2.49

Table 3: Average execution duration and standard deviation for a pipeline with some transforms (10 repetitions).

rows	TS	PL	PLOB	PLRS	PLOBRS
56250	4495 ms 88 ms	4825 ms 124 ms	2804 ms 97 ms	3144 ms 128 ms	1106 ms 108 ms
112500	8283 ms 187 ms	8952 ms 243 ms	4756 ms 175 ms	5479 ms 198 ms	1305 ms 143 ms
225000	15346 ms 563 ms	16881 ms 165 ms	8544 ms 106 ms	9894 ms 111 ms	1789 ms 286 ms
450000	32467 ms 1359 ms	36874 ms 2415 ms	16524 ms 164 ms	21076 ms 191 ms	2396 ms 90 ms
900000	60755 ms 581 ms	67674 ms 761 ms	33375 ms 389 ms	37910 ms 465 ms	4323 ms 804 ms
1300000	95102 ms 2985 ms	99959 ms 1243 ms	49007 ms 576 ms	57910 ms 2558 ms	6252 ms 1487 ms

Table 4: Average execution duration and standard deviation for a pipeline with some transforms (10 repetitions).

rows	TS	PL	PLOB	PLRS	PLOBRS
56250	5722 ms 206 ms	5421 ms 158 ms	3064 ms 46 ms	3545 ms 103 ms	1178 ms 22 ms
112500	11097 ms 450 ms	10348 ms 401 ms	5248 ms 84 ms	5997 ms 134 ms	1383 ms 36 ms
225000	18750 ms 155 ms	17914 ms 889 ms	9470 ms 87 ms	10051 ms 195 ms	1668 ms 35 ms
450000	38264 ms 992 ms	35151 ms 954 ms	18147 ms 537 ms	24406 ms 712 ms	2627 ms 111 ms
900000	81900 ms 1168 ms	72515 ms 2056 ms	35656 ms 2845 ms	38963 ms 1246 ms	4518 ms 1017 ms

Table 5: The result of the average execution time of TS divided by that of PLOBRS.

rows	no transforms	some transforms	many transforms
56250	3.60	4.06	4.86
112500	5.31	6.35	8.02
225000	6.22	8.58	11.24
450000	9.34	13.55	14.57
900000	10.40	14.06	18.22
1300000	11.35	15.22	-
1800000	11.80	-	-

Table 6: *Jayvee* interpreter crashes.

configuration	number of rows	file size	error message
TS-no	2000000 rows	500 MB	Invalid string lenght.
TS-so	1400000 rows	350 MB	JavaScrip heap overflow.
TS-ma	1000000 rows	250 MB	JavaScrip heap overflow.
PL-no	1900000 rows	475 MB	JavaScrip heap overflow.
PL-so	1900000 rows	475 MB	JavaScrip heap overflow.
PL-ma	1800000 rows	450 MB	Invalid string length.
PLOB-no	2000000 rows	500 MB	Invalid string lenght.
PLOB-so	1800000 rows	450 MB	Invalid string length.
PLOB-ma	1800000 rows	450 MB	Invalid string length.
PLRS	2100000 rows	525 MB	Cannot make a string longer than 0x1fffffe8 characters.

A.2 Other tables

Table 7: *Jayvee* operators and the *Polars* expressions they are transformed to. a represents the first parameter, b the second, and c the third.

Jayvee operator	*Polars* expression
a + b	a.plus(b)
asText a	a.cast(Text.toPolarsDataType())
asDecimal a	a.cast(Decimal.toPolarsDataType())
asInteger a	a.cast(Integer.toPolarsDataType())
a and b	a.and(b)
ceil a	a.ceil()
a / b	a.div(b)
a == b	a.eq(b)
floor a	a.floor()
a >= b	a.gtEq(b)
a > b	a.gt(b)
a in b	a.isIn(b)
a != b	a.neq(b)
a <= b	a.ltEq(b)
a < b	a.lt(b)
lowercase a	a.str.toLowerCase()
-a	a.mul(-1)
a % b	a.modulo(b)
a * b	a.mul(b)
!a	a.not()
a or b	a.or(b)
+a	a
round a	a.round(0)
sqrt a	a.pow(1/2)
a - b	a.minus(b)
uppercase a	a.str.toUpperCase()
a xor b	a.and(b.not()).or(a.not().and(b))
asBoolean a	unsupported
a matches b	unsupported
a pow b	unsupported
a replace b with c	unsupported
a root b	unsupported

Table 8: Source code references.

file name	containing folder relative to the project root
electric-vehicles.jv	example/
table.ts	libs/execution/src/lib/types/io-types/
table-colummn.ts	libs/execution/src/lib/types/io-types/
primitive-value-type-provider.ts	libs/language-server/src/lib/ast/wrappers/value-type/primitive/
value-type.ts	libs/language-server/src/lib/ast/wrappers/value-type/
typeguards.ts	libs/language-server/src/lib/ast/expressions/
table-interpreter-executor.ts	libs/extensions/tabular/exec/src/lib/
extension.ts	libs/execution/src/lib/
FileToTableInterpreter.jv	libs/language-server/src/stdlib/builtin-block-types/
file-to-table-interpreter-executor.ts	libs/extensions/tabular/exec/src/lib/
LocalFileToTableExtractor.jv	libs/language-server/src/stdlib/
local-file-to-table-extractor-executor.ts	libs/extensions/tabular/exec/src/lib/
table-transformer-executor.ts	libs/extensions/tabular/exec/src/lib/
transform-executor.ts	libs/execution/src/lib/transforms/
evaluate-expression.ts	libs/language-server/src/lib/ast/expressions/
sqlite-loader-executor.ts	libs/extensions/rdbms/exec/src/lib/
lib.rs	src/

Table 9: The type guard mechanism of each `InternalValueRepresentation`.

type	mechanism
string	typeof
number	typeof
boolean	typeof
RegExp	instanceof
CellRangeLiteral	langium generated type guard
ConstraintDefinition	langium generated type guard
ValuetypeAssignment	langium generated type guard
BlockTypeProperty	langium generated type guard
TransformDefinition	langium generated type guard
AtomicInternalValueRepresentation	any of the above
InternalValueRepresentation[]	Both Array.isArray(...) and the Internal-ValueRepresentation type guard
InternalValueRepresentation	Either AtomicInternalValueRepresentation or InternalValueRepresentation[]

B Software bill of materials (SBOM)

Table 10: SBOM.

name	version	used in	url
nodejs-polars	0.11.0	jayvee	https://www.npmjs.com/package/nodejs-polars
arrow	51	sqlite-loader-lib	https://crates.io/crates/arrow
connector_arrow	0.4.2	sqlite-loader-lib	https://crates.io/crates/connector_arrow
napi	2	sqlite-loader-lib	https://crates.io/crates/napi
napi-derive	2	sqlite-loader-lib	https://crates.io/crates/napi-derive
napi-build	2	sqlite-loader-lib	https://crates.io/crates/napi-build
rusqlite	0.31.0	sqlite-loader-lib	https://crates.io/crates/rusqlite
package-template	f653a34	sqlite-loader-lib	https://github.com/napi-rs/package-template
clap	4.5.7	evaluation tool	https://crates.io/crates/clap
itertools	0.13.0	evaluation tool	https://crates.io/crates/itertools
head	9.5	evaluation tool	https://www.gnu.org/software/coreutils/
sqldiff	3.46.0	evaluation tool	https://sqlite.org/sqldiff.html
texlive	2024.2	thesis document	https://tug.org/texlive/
plantuml	1.2023.13	thesis document	https://plantuml.com/

C Lists

C.1 Languages with *Apache Arrow* libraries

C++, C#, Go, Java, JavaScript, Julia, Rust, Swift, C, MATLAB, Python, R and Ruby (The Apache Software Foundation, n.d.-a).

C.2 Configuration files modified to successfully build the interpreter

- `libs/interpreter-lib/project.json`
- `libs/extensions/std/exec/project.json`
- `apps/interpreter/project.json`
- `libs/extensions/rdbms/exec/project.json`

D Figures

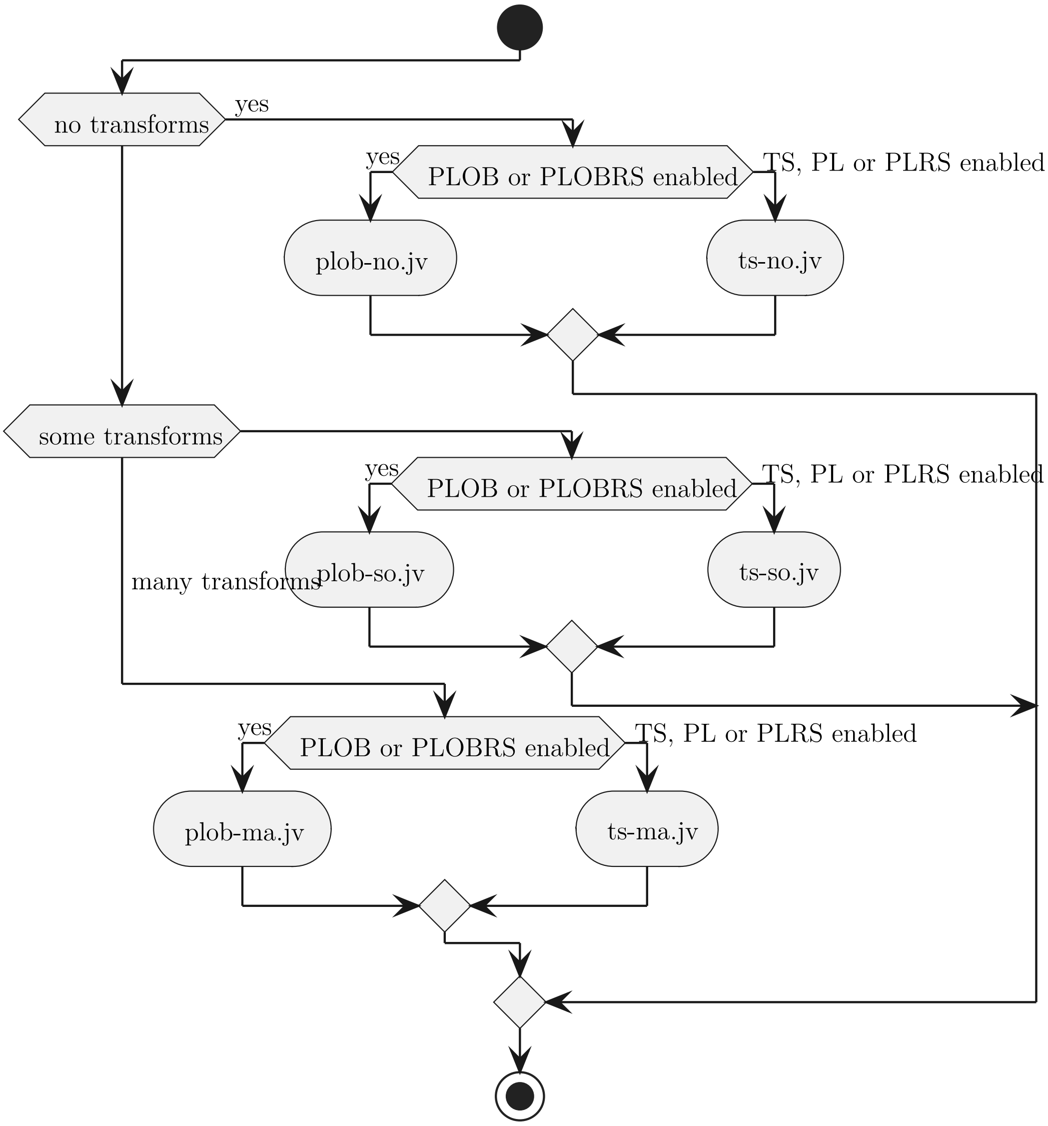


Figure 1: The process, by which the evaluation tool identifies the correct source file.

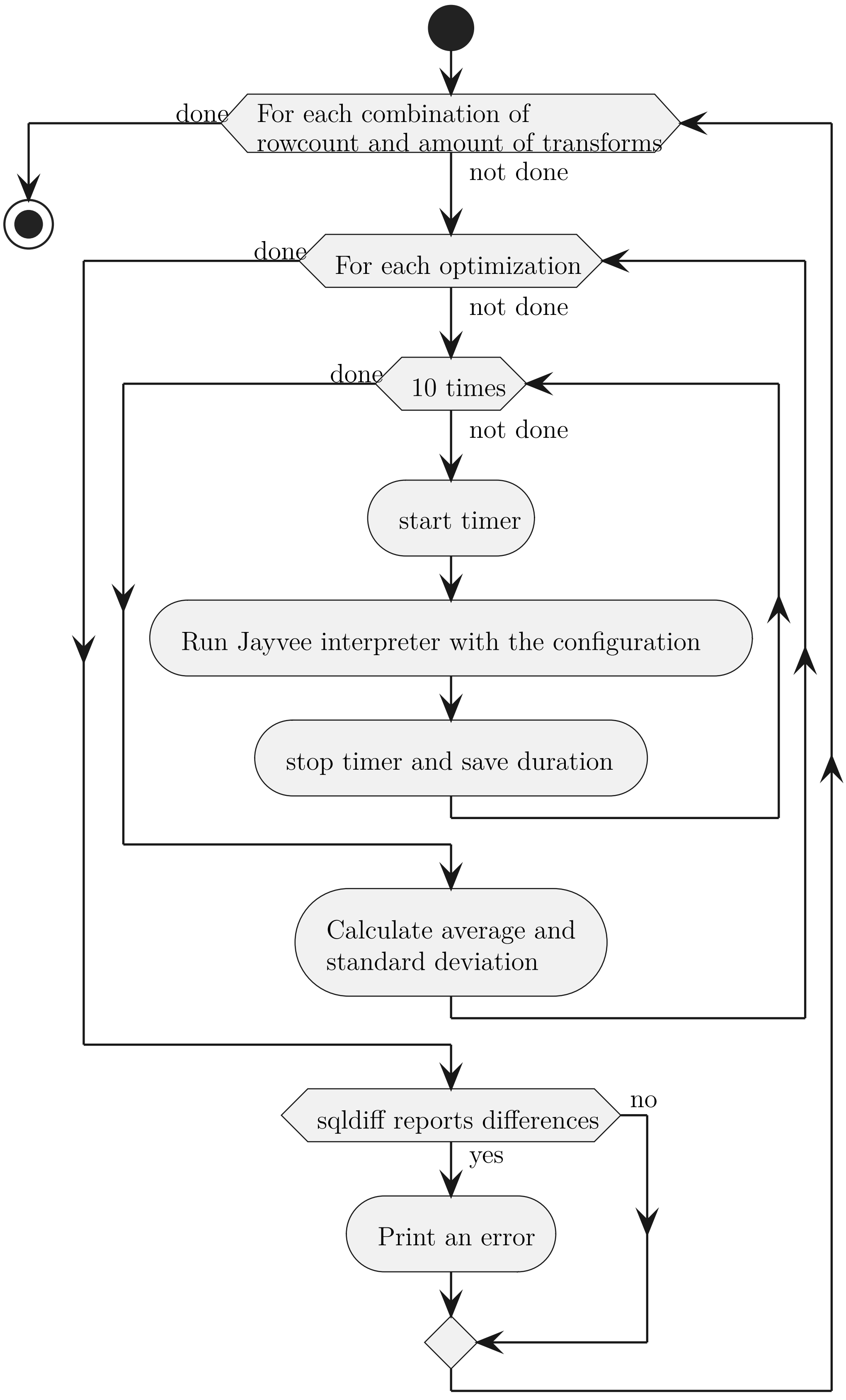


Figure 2: The evaluation tool's activity diagram.

E Listings

```
def run_config(interpreter_dir, rowcount, transforms, backend):
                source = f"l-{rowcount}.csv"
                run f"head --lines=${rowcount}
                ↪  brewery_data_all.csv > {source}"
                source_file = ... # Omitted the source file
                ↪  selection.
                destination =
                ↪  f"{backend}-{transforms}-{rowcount}.sqlite"

                command = f"node dist/apps/interpreter/main.js
                ↪  {source_file} -e SRC={source} -e SRC
                ↪  {destination}";

                if backend != "TS":
                        command += " --use-polars"

                if backend == "PLOBRS" or backend == "PLRS":
                        command += " --use-rusqlite"

                start = now()
                execute(command)
                duration = now() - start
                return duration
```

Listing 8: Pseudocode illustrating the manner in which the evaluation tool executes a configuration

References

Abadi, D., Boncz, P., Harizopoulos, S., Idreos, S., & Madden, S. (2013). The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases*, *5*(3), 197–280. https://doi.org/10.1561/1900000024

Ahmad, T., Ahmed, N., Al-Ars, Z., & Hofstee, H. P. (2021). Optimizing performance of gatk workflows using apache arrow in-memory data framework. *BMC Genomics*, *21*(10), 683. https://doi.org/10.1186/s12864-020-07013-y

Barton, C. (2022, January 4). *How to benchmark programs in rust?* Retrieved August 9, 2024, from https://stackoverflow.com/a/40953863

Boncz, P. (2002, May). *Monet: A next-generation database kernel for query-intensive applications* [Doctoral dissertation].

Dooley, J. F., & Kazakova, V. A. (2024). Design patterns. In *Software development, design, and coding: With patterns, debugging, unit testing, and refactoring* (pp. 275–311). Apress. https://doi.org/10.1007/979-8-8688-0285-0_13

Eržen, A. M. (2024, June 20). *Connector arrow*. Retrieved August 8, 2024, from https://crates.io/crates/connector_arrow

Floratou, A. (2019). Columnar storage formats. In S. Sakr & A. Y. Zomaya (Eds.), *Encyclopedia of big data technologies* (pp. 464–469). Springer International Publishing. https://doi.org/10.1007/978-3-319-77525-8_248

Gordon, C. S., Parkinson, M. J., Parsons, J., Bromfield, A., & Duffy, J. (2021). Uniqueness and reference immutability for safe parallelism. *SIGPLAN Not.*, *47*(10), 21–40. https://doi.org/10.1145/2398857.2384619

Grossman, M., Poole, S., Pritchard, H., & Sarkar, V. (2022). Shmem-ml: Leveraging openshmem and apache arrow for scalable, composable machine learning. In S. Poole, O. Hernandez, M. Baker & T. Curtis (Eds.), *Openshmem and related technologies. openshmem in the era of exascale and smart networks* (pp. 111–125). Springer International Publishing. https://doi.org/10.1007/978-3-031-04888-3_7

JValue Contributors. (n.d.-a). *Core concepts*. Retrieved August 5, 2024, from https://jvalue.github.io/jayvee/docs/user/core-concepts

JValue Contributors. (n.d.-b). *Jayvee*. Retrieved July 13, 2024, from https://jvalue.com/jayvee

JValue Contributors. (n.d.-c). *The jvalue project* [Open data, easy and social]. Retrieved July 13, 2024, from https://jvalue.com/

JValue Contributors. (n.d.-d). *Runtime parameters*. Retrieved August 9, 2024, from https://jvalue.github.io/jayvee/docs/user/runtime-parameters

MacKenzie, D., & Meyering, J. (2024, March). *Head(1)* [User commands]. Retrieved August 9, 2024, from https://man.archlinux.org/man/head.1

MDN Contributors. (2024a, May 31). *The structured clone algorithm*. Retrieved August 4, 2024, from https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm#things_that_dont_work_with_structured_clone

MDN Contributors. (2024b, July 30). *Expressions and operators*. Retrieved August 14, 2024, from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_operators#in

Napa, A. (2023). *Brewery operations and market analysis*. Retrieved July 12, 2024, from https://www.kaggle.com/datasets/ankurnapa/brewery-operations-and-market-analysis-dataset/data

NAPI-RS Contributors. (n.d.-a). *Napi-rs*. Retrieved August 8, 2024, from https://napi.rs/

NAPI-RS Contributors. (n.d.-b). *Napi-rs/package-template*. Retrieved August 8, 2024, from https://github.com/napi-rs/package-template

Open Knowledge Foundation. (n.d.-a). *Conformant licenses*. Retrieved July 19, 2024, from http://opendefinition.org/licenses/

Open Knowledge Foundation. (n.d.-b). *Open definition 2.1*. Retrieved August 9, 2024, from https://opendefinition.org/od/2.1/en/

Peltenburg, J., van Straten, J., Brobbel, M., Al-Ars, Z., & Hofstee, H. P. (2021). Generating high-performance fpga accelerator designs for big data analytics with fletcher and apache arrow. *Journal of Signal Processing Systems*, *93*(5), 565–586. https://doi.org/10.1007/s11265-021-01650-6

Polars Contributors. (n.d.-a). *Casting*. Retrieved August 7, 2024, from https://docs.pola.rs/user-guide/expressions/casting/

Polars Contributors. (n.d.-b). *Column selections*. Retrieved August 6, 2024, from https://docs.pola.rs/user-guide/expressions/column-selections/

Polars Contributors. (n.d.-c). *Expressions*. Retrieved August 3, 2024, from https://docs.pola.rs/user-guide/concepts/expressions/

Polars Contributors. (n.d.-d). *Nodejs-polars/cargo.toml*. Retrieved August 17, 2024, from https://github.com/pola-rs/nodejs-polars/blob/main/Cargo.toml#L18-L21

Polars Contributors. (n.d.-e). *Polars* [Dataframes for the new era]. Retrieved July 31, 2024, from https://pola.rs/

Publications Office of the European Union, Page, M., Hajduk, E., Lincklaen Arriëns, E., Cecconi, G., & Brinkhuis, S. (2023). *Open data maturity re-*

port 2023 (tech. rep.). Publications Office of the European Union. https://doi.org/doi/10.2830/384422

Shiran, T. (2019). It's time to replace odbc & jdbc. Retrieved July 13, 2024, from https://www.dremio.com/blog/is-time-to-replace-odbc-jdbc

SQLite Contributors. (n.d.). *Sqldiff.exe* [Database difference utility]. Retrieved August 9, 2024, from https://sqlite.org/sqldiff.html

The Apache Software Foundation. (n.d.-a). *Apache arrow overview*. Retrieved July 30, 2024, from https://arrow.apache.org/overview/

The Apache Software Foundation. (n.d.-b). *Arrow columnar format*. Retrieved July 31, 2024, from https://arrow.apache.org/docs/format/Columnar.html

The Apache Software Foundation. (n.d.-c). *Implementation status*. Retrieved July 14, 2024, from https://arrow.apache.org/docs/status.html

The Apache Software Foundation. (n.d.-d). *Project and product names using "apache arrow"*. Retrieved July 14, 2024, from https://arrow.apache.org/powered_by/