RASL: Relational Algebra in Scikit-Learn Pipelines

Chirag Sahni RPI
Kiran Kate IBM Research
Avraham Shinnar IBM Research
Hoang Thanh Lam IBM Research
Martin Hirzel IBM Research

Abstract

Integrating data preparation with machine-learning (ML) pipelines has been a long-standing challenge. Prior work tried to solve it by building new data processing platforms such as MapReduce or Spark, and then implementing new libraries of ML algorithms for those. But despite the availability of these platforms, many ML practitioners continue to use scikit-learn instead, owing to its clean design and rich set of algorithms. Therefore, this paper proposes a different approach: instead of extending a data processing platform for ML, extend an ML library for data processing. Specifically, this paper proposes RASL, an open-source library of relational algebra (RA) operators for scikit-learn (SL). We illustrate RASL with a detailed case study involving joins and aggregation across multi-table input data. We hope our approach will lead to cleaner integration of data preparation with machine learning in practice.

1 Introduction

Data science code often consists of two separate pieces: a data preparation pipeline in a framework such as pandas [23] or Spark [3] and a machine-learning pipeline in a framework such as scikit-learn [10]. For the first piece, data preparation frameworks are optimized for data filtering, mapping, and data integration via joins and aggregates, all of which are essentially relational-algebra operators. For the second piece, machine-learning frameworks contain large collections of transformers (e.g., imputation, normalization, or feature selection) and estimators (e.g., classifiers or regressors) that can be tuned and trained to maximize predictive performance. Unfortunately, when a data scientist maintains separate pipelines based on different frameworks, the ensuing context switches make them less productive, the code becomes more prone to errors and overfitting, and they can face substantial re-work when trying to scale their code or deploy it as a service. Separating the pipelines for data preparation and machine learning is problematic for productivity, transparency, and robustness.

One prior solution to this problem is to build a combined platform for both pieces. For example, Hive [32] and Mahout [2] offer relational data preparation and machine learning on top of MapReduce [13]. Similarly, Spark SQL [3] and MLlib [25] do the same on top of Spark [34]. Unfortunately, while they combine relational algebra and machine learning on the same platform, they still expose separate programming models for them. Furthermore, since many machine-learning algorithms are easier to implement stand-alone than on bulk-synchronous frameworks such as MapReduce or Spark, Mahout and MLlib have relatively small collections of algorithms compared to scikit-learn. Hence, a common practice is to use Spark for preprocessing large data, using aggregation to reduce its size, followed by scikit-learn for machine learning in a separate pipeline. Another solution from prior work is to tightly integrate relational algebra with linear algebra [17, 21, 28]. While this yields elegant algebraic systems that are well-suited to rewrite-based optimizers, the linear-algebra part is too low-level for most users. It is easier for a data scientist to pick an off-the-shelf algorithm from scikit-learn than to build up that same algorithm at the level of linear algebra.

This paper explores a different solution: adding relational-algebra operators to scikit-learn [10]. Scikit-learn comes with well over a hundred transformers and estimators for machine learning. Other machine-learning libraries provide scikit-learn compatible APIs, such as XGBoost [12], Snap

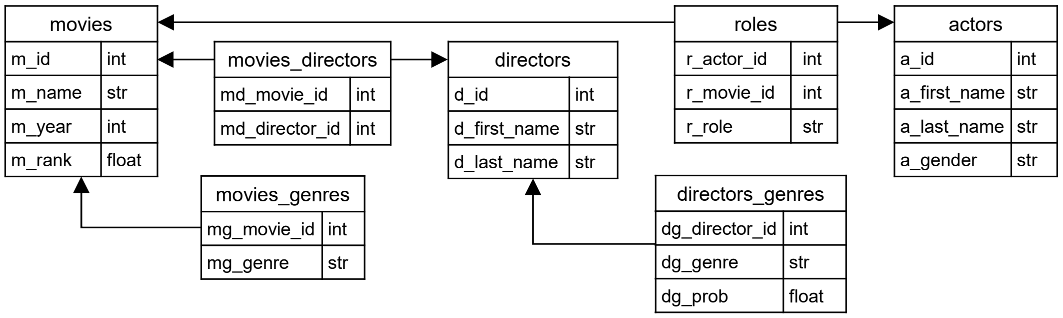


Figure 1: IMDB dataset entity-relationship diagram.

ML [14], and AIF360 [7]. Between scikit-learn itself and compatible libraries, data scientists have access to hundreds of high-level algorithms. This makes scikit-learn immensely popular: as of this writing, its GitHub page indicates that it is used by 255k other projects. Importantly for our work, most data scientists are familiar with scikit-learn's well-designed pipeline API. Adding the well-understood operators from relational algebra thus makes it possible to perform both data preparation and machine learning in a single pipeline, using a single uniform programming model.

This paper describes RASL, the first design of relational-algebra operators with scikit-learn syntax and semantics. It discusses programming-model questions and solutions for cleanly integrating relational algebra with high-level machine learning algorithms. This paper offers a detailed usage example. We have implemented our operators with back-ends on both pandas and Spark. Pipelines can thus run on either of these frameworks without code changes. The implementation is available open-source (https://github.com/ibm/lale) as part of the Lale library [5], which extends scikit-learn for automated machine learning. Overall, this paper introduces a consistent end-to-end programming model for integrating relational-algebra based preprocessing into scikit-learn pipelines for machine learning, which we hope will make data scientists more productive and pipelines more robust.

2 Overview and Example

This section uses a detailed example to give an overview of our relational-algebra operators. The example task is to predict movie ratings based on the IMDB dataset [19]. Figure 1 describes the dataset, comprising seven tables. We will predict the `movies.m_rank` attribute, which rates movies on a scale from 1 to 10. The difficulty is that not all relevant features are available in the movies table, since the database is in normal form. For instance, a useful feature might be *"Given a new movie, what is the average rating of earlier movies by the same directors?"* Since the `movies` table contains no information about directors, computing this feature requires data integration with the `movies_directors` table, in other words, denormalization. The following code accomplishes that:

```
1 join_dir = (
2     (Scan(table=it.movies) & Scan(table=it.movies_directors))
3    >> Join(pred=[it.movies.m_id == it.movies_directors.md_movie_id],
4            join_type="left", name="j_dir"))
```

Line 2 uses two instances of the `Scan` operator to select the `movies` and `movies_directors` table, respectively. The `&` combinator arranges these two operators side-by-side and independently. The `>>` combinator on Line 3 pipes both their output to a `Join` operator. The join predicate `pred` finds rows with matching movie IDs. We use a left join, to ensure each row from the `movies` table is represented in the output, even if there are no corresponding rows in the `movies_directors` table. And we set a name for the resulting output table so we can refer to it later. A data scientist can run this code interactively in a notebook and can print the resulting dataframe for exploratory data analysis:

```
join_dir.transform(imdb_train).toPandas()
```

	m_id	m_name	m_year	m_rank	md_director_id	md_movie_id
0	2	$	1971	6.4	9970.0	2.0
1	6	$1,000,000 Duck	1971	5.0	52153.0	6.0
...	...	...	...	...	...	...
58503	378610	nc sayfa	1999	7.0	19451.0	378610.0
58504	378614	. 19,99	1998	6.3	20640.0	378614.0

58505 rows × 6 columns

As expected, columns `m_id` and `md_movie_id` match. Of course, this table does not yet have the desired information, which is ratings of earlier movies by the same directors. This requires another join: each row of the table is a movie, and we want to join each row with other rows whose director ID matches. We could do this with a self-join, but this would not work as expected with train-test splits. Specifically, the set of movies being tested may be small and disjoint from the movies for training, from which the join should discover earlier movies by the same director. Therefore, we assume two versions of the movies table: `movies` (the set of current movies under test) vs. `other_movies` (the set of other movies to join against). We first join `other_movies` with directors, renaming its columns with `Map` to avoid conflicts:

```
join_other_dir = (
    (Scan(table=it.other_movies) & Scan(table=it.movies_directors))
   >> Join(pred=[it.other_movies.m_id == it.movies_directors.md_movie_id],
          join_type="left", name="j_o_dir")
   >> Map(columns={"o_md_director_id": it.md_director_id,
                  "o_m_year": it.m_year, "o_m_rank": it.m_rank},
          remainder="drop"))
```

Now, we are ready to join movies with earlier movies by the same director, using a `Join` by director ID followed by a `Filter` to ensure that the other movie is indeed earlier. This filter prevents label leakage and is common sense, since we cannot use information from the future to predict the present.

```
self_join_dir = (
    Join(pred=[it.j_dir.md_director_id == it.j_o_dir.o_md_director_id],
         join_type="left", name="sj_dir")
   >> Filter(pred=[it.m_year > it.o_m_year]))
```

As before, while developing the pipeline, the data scientist can inspect the intermediate dataframe.

	m_id	m_name	m_year	m_rank	md_director_id	md_movie_id	o_md_director_id	o_m_year	o_m_rank
0	240335	Omerip	2003	3.1	959.0	240335.0	959.0	1977	6.1
1	240335	Omerip	2003	3.1	959.0	240335.0	959.0	1978	6.9
...	...	...	...	...	...	...	...	...	...
1155837	364161	Who Framed Roger Rabbit	1988	7.4	88102.0	364161.0	88102.0	1980	6.5
1155838	299343	Shit Happens Again	1994	1.2	88761.0	299343.0	88761.0	1993	7.2

1155839 rows × 9 columns

As expected, this dataframe is much larger than the original, since it has potentially multiple rows for each current `m_id`. For example, in the screenshot, the 2003 movie "Omerip" was joined against at least two earlier movies from 1977 and 1978 by the same director. Each row also has the rank of the other movie, `o_m_rank`. The next step is to compute the average and maximum rank of each other movie per current movie. This is a straight-forward `GroupBy` and `Aggregate`:

```
features_dir = (
    GroupBy(by=[it.m_id])
   >> Aggregate(columns={"dir_max_rank": max(it.o_m_rank), "dir_mean_rank": mean(it.o_m_rank),
                         "m_rank": first(it.m_rank)}))
```

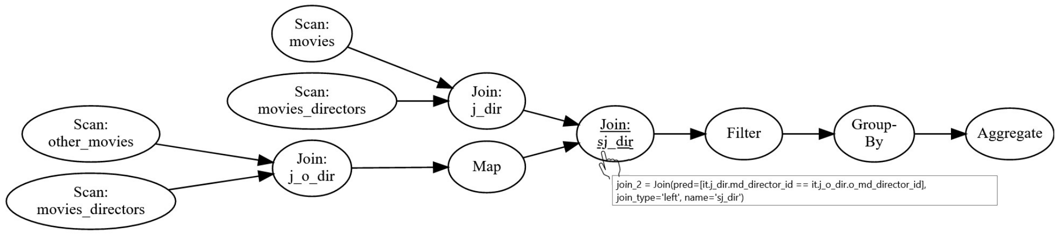
The above code also holds on to the rank of the current movie, which will serve as the ground-truth target label for which we want to train a predictive machine-learning model. Before getting to that, the data scientist can inspect the intermediate data:

	m_id	m_rank	dir_max_rank	dir_mean_rank
0	341515	5.1	4.8	4.80
1	185096	6.4	6.5	5.76
...	...	...	...	...
36241	39592	9.1	7.4	6.35
36242	290699	6.1	6.6	6.60

36243 rows × 4 columns

Also, the data scientist can visualize the pipeline so far:

```
prefix_dir = (join_dir & join_other_dir) >> self_join_dir >> features_dir
prefix_dir.visualize()
```



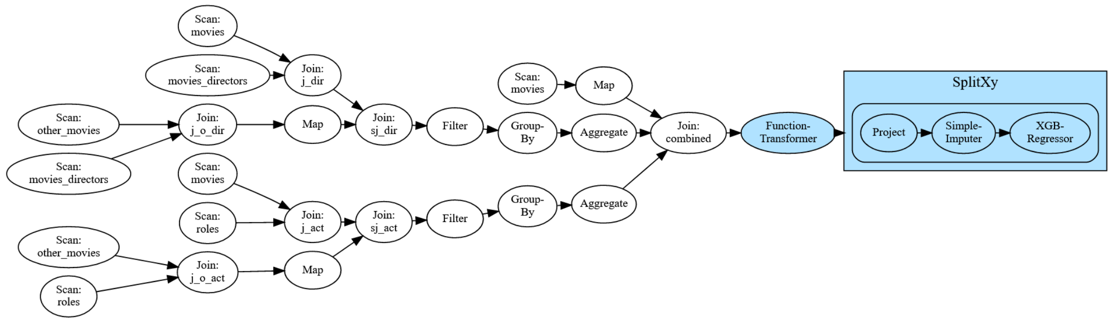
```
join_2 = Join(pred=[it.j_dir.md_director_id == it.j_o_dir.o_md_director_id],
join_type='left', name='sj_dir')
```

As shown in the screenshot, hovering the mouse pointer over an operator in the visualization reveals its configuration. Looking back at the description of the IMDB dataset in Figure 1, another useful feature might be *"Given a new movie, what is the average rating of earlier movies by the same actors?"* This feature can be calculated with a similar pipeline as before, this time joining `movies` against `roles`. Here, we elide the code for that, and assume it defines a sub-pipeline `prefix_act`. In practice, the data scientist may add more features, but for our example, we will leave it at that. When we combine the features from both preprocessing sub-pipelines, we also include the movie IDs from the original movie table in the join to ensure no movies are accidentally dropped:

```
movie_ids = Scan(table=it.movies) >> Map(columns={"row_m_id": it.m_id}, remainder="drop")
combine_features = (
    (movie_ids & prefix_dir & prefix_act)
   >> Join(pred=[it.movies.row_m_id==it.sj_act.m_id, it.movies.row_m_id==it.sj_dir.m_id],
            join_type="left", name="combined"))
```

That completes the relational-algebra part of the example pipeline. We are now ready to add the machine-learning part. First, since scikit-learn estimators can handle dataframes from pandas but not Spark, we add a scikit-learn `FunctionTransformer` that converts them if necessary. Next, preprocessing produces a single table with both features and the prediction target, but scikit-learn estimators require those to be separated into `X` and `y`, respectively. Therefore, we add a simple higher-order operator `SplitXy` that takes another operator or sub-pipeline as an argument, and splits its input appropriately as the input to that operator. Next, ID columns are, at best, useless features and can, at worst, cause over-fitting, so we drop them with a `Project`. Also, due to the left outer joins, the preprocessed data contains null values, so we add a scikit-learn `SimpleImputer`. Finally, since the prediction target `m_rank` is a continuous variable, we need a regressor. Specifically, we pick `XGBRegressor` from the popular XGBoost library [12], which offers a scikit-learn compatible API. Putting this all together yields:

```
estimator = Project(drop_columns=["row_m_id", "m_id"]) >> SimpleImputer() >> XGBRegressor()
pipeline = (
    combine_features
   >> FunctionTransformer(func=lambda X: X if isinstance(X, pd.DataFrame) else X.toPandas())
   >> SplitXy(operator=estimator, label_name="m_rank"))
pipeline.visualize()
```



The visualization colors the machine-learning part of the pipeline light blue to indicate that it has learnable coefficients that have yet to be trained. At this point, we have a single end-to-end pipeline, which supports the standard scikit-learn `fit` and `predict` API:

```
trained_pipeline = pipeline.fit(imdb_train)
predictions = trained_pipeline.predict(imdb_test)
```

If the input consists of pandas dataframes, the relational-algebra operators in the pipeline use pandas internally. On the other hand, if the input consists of Spark dataframes, they use Spark SQL internally.

3 Syntax and Semantics

This section discusses the user-facing design of our relational-algebra operators, leaving implementation details to the next section.

Scikit-learn adopts a different syntax from data preprocessing libraries such as pandas and Spark SQL. Code that uses pandas or Spark SQL conflates operator configuration with execution, invoking an operator directly on a dataframe (e.g., `transformed_data = input_data.agg(**agg_hyperparams)`). In contrast, code for scikit-learn first configures the operator while being *tacit* about data (e.g., `agg_op = Aggregate(**agg_hyperparams)`). Separating configuration from execution lets scikit-learn support not just one, but two different execution modes. Specifically, the two quintessential machine-learning execution modes are training and predicting, and a configured scikit-learn transformer supports them with two methods, `fit` and `transform`. Since our relational-algebra operators have no learnable coefficients, their `fit` is a no-op, so the relevant interface is `transformed_data = configured_op.transform(input_data)`. Scikit-learn carries this different syntax to the pipeline level as well. In scikit-learn, the code `op123 = make_pipeline(op1, op2, op3)` only mentions configured operators, not data, and the resulting pipeline `op123` supports `fit` and `transform`. This paper uses the `>>` combinator, so the above code becomes `op123 = op1 >> op2 >> op3`. Being tacit about data during configuration allows using different data during executions with different modes. For example, k-fold cross-validation calls `fit` with $k - 1$ folds and `transform` with the remaining fold.

For ease of use (and ease of implementation), we support a standard set of familiar text-book relational algebra operators, shown in Table 1, and demonstrated by the example in Section 2. These operators work on *tables*, which are data structures with rows and columns. Each column has a name and

Table 1: Relational algebra operators for scikit-learn.

Name	Description	Hyperparameters		Transform	
		Name	: Type	Input	→ Output
`Filter`	Drop non-matched rows	`pred`	: `List[expr]`	`Table`	→ `Table`
`Map`	Assign columns,	`columns`	: `Dict[str, expr]`	`Table`	→ `Table`
	one row at a time	`remainder`	: `Enum["passthrough","drop"]`		
`Join`	Combine columns of	`pred`	: `List[expr]`	`List[Table]`	→ `Table`
	matching rows	`join_type`	: `Enum["inner","left","right"]`		
		`name`	: `str`		
`GroupBy`	Create groups of rows	`by`	: `List[expr]`	`Table`	→ `Grouped`
`Aggregate`	Reduce group to row	`columns`	: `Dict[str, expr]`	`Grouped`	→ `Table`
`OrderBy`	Sort by columns	`by`	: `List[expr]`	`Table`	→ `Table`
`Scan`	Pick out a table	`table`	: `expr`	`List[Table]`	→ `Table`
`Alias`	Rename table	`name`	: `str`	`Table`	→ `Table`

each row is a record with one value for every column. Our current design leaves the order of rows unspecified and the types of columns dynamic. Besides tables, some operators also work on *grouped* data, which is like a table, except that some columns are designated as a grouping key and sets of rows with the same values for the grouping key form groups.

Most relational-algebra operators need to be configured with code to be executed per row, e.g., filter conditions, mapping expressions, or join predicates (see column Hyperparameters of Table 1). There are some scikit-learn operators whose hyperparameters accept functions, which may be anonymous with Python's `lambda` keyword. But that is verbose and hard to translate to non-Python backends, so we designed a more restrictive expression language instead. While one option would have been to represent that as a Python string, that would have been less familiar to Python users and would have required its own parser [4]. Instead, we use Python's operator overloading to build up expressions.

Since scikit-learn code is tacit about data, expressions need a placeholder to refer to the implicit input data of the operator. We picked the neutral third person singular pronoun `it` for this. For example, `max(it.o_m_rank)` overloads Python's dot notation for attribute access to create a symbolic expression that accesses the `o_m_rank` column of the current table. While most operators only have a single input table, `Join` inherently has multiple. We support this with an extra level of attribute access. For example, in `it.movies.m_id == it.movies_directors.md_movie_id`, the attribute `movies` refers to a table and `m_id` refers to a column of that table. To make this work, operators `Scan`, `Alias`, and `Join` assign table names, and operators `Filter`, `Map`, `GroupBy`, and `Aggregate` propagate table names from their input to their output. These table names are piggy-backed on the data itself, so the operators do not need to register them into some catalog as a side-effect. Sometimes, table or column names are not valid Python identifiers, so we also overload `it["attr"]` as an alternative syntax for `it.attr`.

Python does not allow overloading logical connectives, so we used lists for `and`, and so far, we do not yet provide syntax for `or` or `not`. The `Join` operator currently only supports equi-joins, by checking that the predicate uses `==` comparisons between columns. For non-equi-joins, users can write `Join(..) >> Filter(..)`, keeping the `Join` operator simple. Similarly, separate `GroupBy(..) >> Aggregate(..)`. The `Join` operator requires column names that do not participate in `==` comparisons to be unique, i.e., the output table does not have any duplicate columns with the same name but different values from different input tables.

4 Implementation and Results

This section describes our implementation of the design from Section 3, using two back-ends: pandas and Spark SQL. We plan to add additional back-ends in future work. The implementation represents tables as `pd.DataFrame` or `pyspark.sql.DataFrame` objects and grouped data as `pd.core.groupby.DataFrameGroupBy` or `pyspark.sql.GroupedData` objects. In other words, it directly uses the core data structures of the supported back-ends. The implementation represents operators

Table 2: Translating operators to back-ends (examples).

scikit-learn	pandas	Spark SQL
`Filter(pred=[it.a > it.b])`	`it[it["a"] > it["b"]]`	`it.filter(col("a") > col("b"))`
`Map(columns={ "a": day_of_month(it.a)})`	`pd.to_datetime( it["a"]).dt.day()`	`it.withColumn( "a", to_timestamp(it["a"]) ).select(dayofmonth("a"))`
`Join(pred=[ it.t1.a1 == it.t2.a2])`	`pd.merge(t1, t2, left_on="a1", right_on="a2")`	`t1.join(t2, col("a1") == col("a2"))`
`GroupBy(by=[it.a])`	`it.groupby("a")`	`it.groupby("a", sort=False)`
`Aggregate(columns={ "a": max(it.a)})`	`it.agg({"a": "max"})`	`it.agg({"a": "max"})`
`OrderBy(by= [it.a, desc(it.b)])`	`it.sort_values(by=["a","b"], ascending=[True, False])`	`it.orderBy(["a","b"], ascending=[True, False])`
`Scan(table=it.a)`	`next(filter(lambda d: d.table_name == "a", it))`	`next(filter(lambda d: get_alias(d) == "a", it))`
`Alias(name="a")`	`set_attr(it, "table_name", "a")`	`it.alias("a")`

as objects that conform to scikit-learn conventions, i.e., with methods `__init__`, `fit`, and `transform`. The `__init__` method validates and stores the hyperparameters; the `fit` method is a no-op; and the `transform` method dispatches to the corresponding back-end based on the type of its input.

Table 2 gives examples for `transform` for simple cases; for other supported cases, see our open-source code. The main difference between the back-ends is that pandas is eager and Spark SQL is lazy. Operations on pandas dataframes eagerly return a new dataframe or grouped data that materializes the effect of the operator right away. In contrast, operations on Spark SQL merely return a Spark dataframe proxy object that contains a query, which is a promise for the effect of the operator. In the case of a scikit-learn pipeline of relational operators, that laziness gets chained: the last operator returns a dataframe with a promise for the effect of the entire pipeline. Consider the sub-pipeline `join_other_dir` from Section 2. The Spark SQL dataframe resulting from its `transform` method provides an `explain` method that prints the optimized query plan shown below:

```
join_other_dir.transform(train).explain("extended")
```

```
== Parsed Logical Plan ==
...
== Optimized Logical Plan ==
Project [md_director_id#4 AS o_md_director_id#6, m_year#2 AS o_m_year#7, m_rank#3 AS o_m_rank#8]
+- Join LeftOuter, (m_id#0 <=> md_movie_id#5)
   :- Project [m_id#0, m_year#2, m_rank#3]
   :  +- Relation[m_id#0,m_name#1,m_year#2,m_rank#3] csv
   +- Relation[md_director_id#4,md_movie_id#5] csv
```

In this example, while the original scikit-learn pipeline uses a `Map` operator for a projection after the join only, the optimized query plan contains a `Project` operator before the join. This is because of a query rewrite that hoists the projection to drop the unneeded `m_name` column early, to save space and time during the join. Continuing with the example, at the end of Section 2, the result of the relational sub-pipeline flows into a scikit-learn `FunctionTransformer`, configured to call `X.toPandas()`. That call triggers Spark SQL to rewrite the entire relational part of the pipeline with its optimizer and then execute the optimized query to materialize its result.

Our implementation currently supports functions for the `Map` operator (including several date-time functions), the `Aggreegate` operator (including `sum`, `max`, `mean`, etc.); and the `Filter` operator (`isnan`, `isnotnan`, etc.). The implementation factors framework-specific glue code in such a way that functions can reuse it without much repetitive code, so adding a new function only takes a few lines of code each. We are planning to add many more functions in future work.

One reason for choosing these two back-ends, which are both popluar for data preprocessing, was their differing execution performance characteristics. Pandas is used when the dataset fits in memory, while the distributed data processing capabilities of Spark are required for larger data sets. To compare these implementations for the example in this paper, we created different sized datasets by down-sampling and up-sampling the `movies` table. Figure 2 shows runtime in seconds for `fit` and `predict` with pandas and Spark. These are average runtimes over 5 runs, and were executed on a single-node machine with 16GB RAM and a 2.7 GHz processor. Spark was run in local mode using 4 cores. For 8 times the data, pandas ran out of memory, so only Spark execution times are reported. Not surprisingly, pandas does better for smaller datasets while Spark scales better for larger datasets. This demonstrates that our implementation provides a unified interface for these operators, while maintaining the performance characteristics of the back-ends. Moreover, one can seamlessly switch between back-ends based on the size of the datasets without any changes to the pipeline code.

5 Related Work

There have been several APIs for pipelines of machine-learning transformers and estimators over the years, including WEKA [16], PMML [15], scikit-learn [10], and Spark MLlib [25]. At the time of this writing, scikit-learn is the most popular, thanks to its clean design, large collection of operators, and ecosystem of compatible libraries. Before this paper, scikit-learn lacked relational-algebra operators.

The detailed example in Section 2 is inspired by prior work on relational preprocessing for machine learning, including by hand [27] and with automated pipeline discovery in Deep Feature Synthesis [20] and One Button Machine [22]. None of these papers offered scikit-learn extensions with relational algebra. In future work, we will explore targeting our operators with automated pipeline discovery.

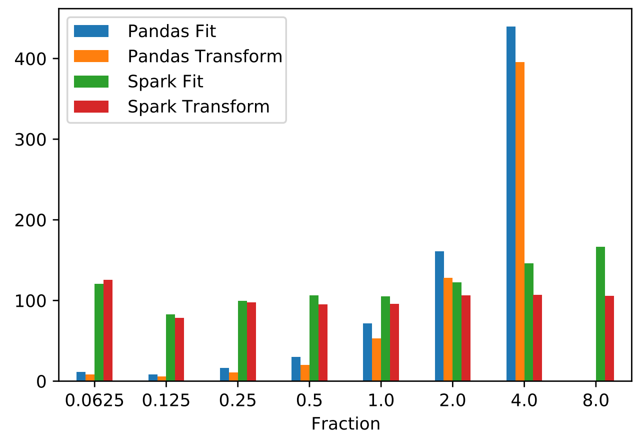


Figure 2: Runtime in seconds of pandas and Spark SQL for our IMDB example. The x-axis shows the dataset sample size, a value greater than 1 indicates up-sampling with replacement.

Both back-ends in our implementation, pandas [23] and Spark ML [3], are based on dataframes, which are essentially tables for data science. Dataframes originated in the R programming language [18], and there have been several efforts to optimize their computational performance [3, 29, 30, 32]. Our implementation relies on existing approaches for computational performance, and innovates beyond existing approaches by providing a scikit-learn interface to relational-algebra operators. Both KeystoneML [31] and Helix [33] extend Spark with Scala-based pipeline APIs for joint data preparation and machine learning. RASL instead uses scikit-learn for its large operator library and Python for its popularity. In future work, we may apply some of their caching ideas to RASL.

One might think that the path to integrating machine learning with relational algebra should be integrating linear algebra with relational algebra. Linear algebra underlies much of machine learning, especially deep learning as pioneered by Theano [8] and later frameworks such as TensorFlow [1]. This inspired efforts towards implementing linear algebra on a platform that initially focused on relational algebra, such as Mahout [2], SystemML [9], and SparkNet [26]. For even tighter integration, recent efforts propose joint linear-and-relational algebras, such as Weld [28], Lara [21], and LaraDB [17]. While this is useful for computational performance, the success of scikit-learn indicates that most data scientists prefer not to work directly with low-level linear algebra. Hence, in contrast to this line of work, our paper integrates relational algebra with high-level scikit-learn pipelines.

LINQ integrates relational algebra directly with a general-purpose programming language [24]. This differs from our approach of integrating relational algebra with high-level machine-learning pipelines. The sklearn2sql project translates scikit-learn estimators to SQL [11]. This differs from our approach of adding relational-algebra operators to scikit-learn. TFX extends Tensorflow with data transformations for feature wrangling [6]; in contrast, we extend scikit-learn with relational algebra.

6 Conclusion

This paper presents the first scikit-learn compatible API for relational-algebra operators. Our API enables data scientists to include advanced multi-table relational feature preprocessing in their scikit-learn pipelines, rather than performing this as a separate step. Having everything in a single pipeline improves transparency; for instance, data scientists can visualize the entire pipeline, and it omits nothing they need to know for training and scoring. While data cleaning and feature engineering remain a demanding task for data scientists, our API puts them on a solid relational foundation and integrates them cleanly with machine learning. We have implemented our design with two interchangeable back-ends, one for pandas and one for Spark SQL. Initial experiments indicate that the pandas back-end allows for interactive speed while iterating on small data, whereas the Spark SQL back-end allows for better scaling. In future work, we will explore additional operators, additional back-ends, and automation to further assist data scientists in creating well-performing pipelines.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283, 2016.

[2] Robin Anil, Gokhan Capan, Isabel Drost-Fromm, Ted Dunning, Ellen Friedman, Trevor Grant, Shannon Quinn, Paritosh Ranjan, Sebastian Schelter, and Ozgur Yilmazel. Apache Mahout: Machine learning on distributed dataflow systems. *Journal of Machine Learning Research (JMLR)*, 21(127):1–6, 2020.

[3] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational data processing in Spark. In *International Conference on Management of Data (SIGMOD)*, pages 1383–1394, 2015.

[4] Guillaume Baudart, Martin Hirzel, Kiran Kate, Louis Mandel, and Avraham Shinnar. Machine learning in Python with no strings attached. In *Workshop on Machine Learning and Programming Languages (MAPL)*, pages 1–9, 2019.

[5] Guillaume Baudart, Martin Hirzel, Kiran Kate, Parikshit Ram, Avraham Shinnar, and Jason Tsay. Pipeline combinators for gradual AutoML. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[6] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, Chiu Yuen Koo, Lukasz Lew, Clemens Mewald, Akshay Naresh Modi, Neoklis Polyzotis, Sukriti Ramesh, Sudip Roy, Steven Euijong Whang, Martin Wicke, Jarek Wilkiewicz, Xin Zhang, and Martin Zinkevich. TFX: A TensorFlow-based production-scale machine learning platform. In *Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1387–1395, 2017.

[7] Rachel K. E. Bellamy, Kuntal Dey, Michael Hind, Samuel C. Hoffman, Stephanie Houde, Kalapriya Kannan, Pranay Lohia, Jacquelyn Martino, Sameep Mehta, Aleksandra Mojsilovic, Seema Nagar, Karthikeyan Natesan Ramamurthy, John Richards, Diptikalyan Saha, Prasanna Sattigeri, Moninder Singh, Kush R. Varshney, and Yunfeng Zhang. AI Fairness 360: An extensible toolkit for detecting, understanding, and mitigating unwanted algorithmic bias, 2018.

[8] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A CPU and GPU math compiler in Python. In *Python in Science Conference (SciPy)*, pages 3–10, 2010.

[9] Matthias Boehm, Douglas R. Burdick, Alexandre V. Evfimievski, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Shirish Tatikonda, and Yuanyuan Tian. SystemML's optimizer: Plan generation for large-scale machine learning programs. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 37(3):52–62, 2014.

[10] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: Experiences from the scikit-learn project, 2013.

[11] Antoine Carme. sklearn2sql, 2017. https://github.com/antoinecarme/sklearn2sql-demo (Retrieved September 2021).

[12] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Conference on Knowledge Discovery and Data Mining (KDD)*, pages 785–794, 2016.

[13] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

[14] Celestine Dünner, Thomas Parnell, Dimitrios Sarigiannis, Nikolas Ioannou, Andreea Anghel, Gummadi Ravi, Madhusudanan Kandasamy, and Haralampos Pozidis. Snap ML: A hierarchical framework for machine learning. In *Conference on Neural Information Processing Systems (NIPS)*, pages 252–262, 2018.

[15] Alex Guazzelli, Michael Zeller, Wen-Ching Lin, and Graham Williams. PMML: An open standard for sharing models. *The R Journal*, 1(1):60–65, 2009.

[16] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: An update. *SIGKDD Explorations Newsletter*, 11(1):10–18, November 2009.

[17] Dylan Hutchison, Bill Howe, and Dan Suciu. LaraDB: A minimalist kernel for linear and relational algebra computation. In *Workshop on Algorithms and Systems for MapReduce and Beyond (BeyondMR)*, 2017.

[18] Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

[19] The IMDb database, from the Relational Dataset Repository. https://relational.fit.cvut.cz/dataset/IMDb (Retrieved September 2021).

[20] James Max Kanter and Kalyan Veeramachaneni. Deep feature synthesis: Towards automating data science endeavors. In *Conference on Data Science and Advanced Analytics (DSAA)*, pages 1–10, 2015.

[21] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Bress, Tilmann Rabl, and Volker Markl. An intermediate representation for optimizing machine learning pipelines. In *Conference on Very Large Data Bases (VLDB)*, pages 1553–1567, 2019.

[22] Hoang Thanh Lam, Johann-Michael Thiebaut, Mathieu Sinn, Bei Chen, Tiep Mai, and Oznur Alkan. One button machine for automating feature engineering in relational databases, 2017.

[23] Wes McKinney. pandas: a foundational Python library for data analysis and statistics. *Workshop on Python for High Performance and Scientific Computing (PyHPC)*, pages 1–9, 2011.

[24] Erik Meijer, Brian Beckman, and Gavin M. Bierman. LINQ: Reconciling objects, relations, and XML in the .NET framework. In *Industrial Sessions at the International Conference on Management of Data (SIGMOD-Industrial)*, pages 706–706, 2006.

[25] Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. MLlib: Machine learning in Apache Spark. *Journal of Machine Learning Research (JMLR)*, 17:34:1–34:7, 2016.

[26] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I. Jordan. SparkNet: Training deep networks in Spark. In *Workshop on Machine Learning Systems (LearningSys)*, 2015.

[27] Carlos Ordonez. Data set preprocessing and transformation in a database system. *Intelligent Data Analysis (IDA)*, 15(4):613–631, December 2011.

[28] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, and Matei Zaharia. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*, 2017. http://cidrdb.org/cidr2017/papers/p127-palkar-cidr17.pdf.

[29] Devin Petersohn, William W. Ma, Doris Jung Lin Lee, Stephen Macke, Doris Xin, Xiangxi Mo, Joseph Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya G. Parameswaran. Towards scalable dataframe systems. In *Conference on Very Large Data Bases (VLDB)*, pages 2033–2046, 2020.

[30] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Python in Science Conference (SciPy)*, pages 130–136, 2015.

[31] Evan R. Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J. Franklin, and Benjamin Recht. KeystoneML: Optimizing pipelines for large-scale advanced analytics. In *International Conference on Data Engineering (ICDE)*, 2017.

[32] Ashish Thusoo, Sen Joydeep Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - a warehousing solution over a Map-Reduce framework. In *Demo at Conference on Very Large Data Bases (VLDB-Demo)*, pages 1626–1629, 2009.

[33] Doris Xin, Litian Ma, Jialin Liu, Stephen Macke, Shuchen Song, and Aditya Parameswaran. Helix: Accelerating human-in-the-loop machine learning. In *Conference on Very Large Data Bases (VLDB)*, pages 1958–1961, 2018.

[34] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.