Prosto Documentation

Release latest

Prosto Data

Nov 21, 2021

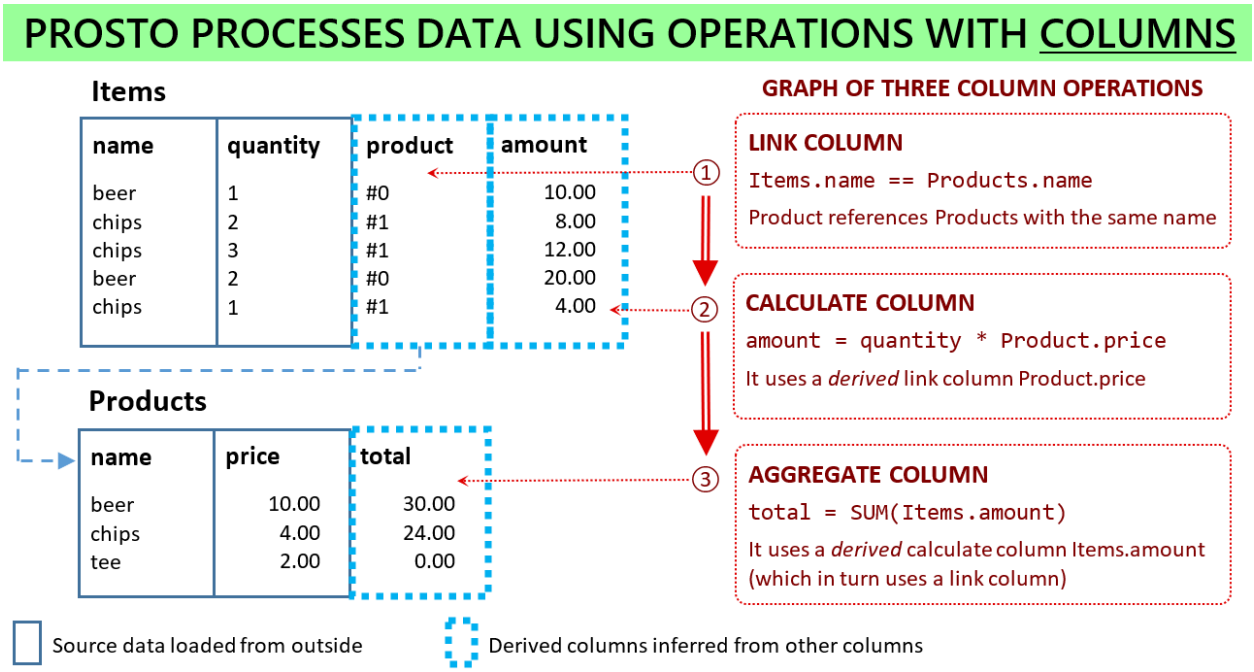
Contents

Prosto is a Python data processing toolkit to (programmatically or using *Column-SQL*) author and execute complex data processing workflows. Conceptually, it is an alternative to purely *set-oriented* approaches to data processing like map-reduce, relational algebra, SQL or data-frame-based tools like *pandas*.

Prosto radically changes the way data is processed by relying on a novel data processing paradigm: *concept-oriented model* of data [1]. It treats columns (modelled via mathematical functions) as first-class elements of the data processing pipeline having the same rights as tables. If a traditional data processing graph consists of only set operations than the *Prosto* workflow consists of two types of operations:

- *Table operations* produce (populate) new tables from existing tables. A table is an implementation of a mathematical *set* which is a collection of tuples.
- *Column operations* produce (evaluate) new columns from existing columns. A column is an implementation of a mathematical *function* which maps tuples from one set to another set.

An example of such a *Prosto* workflow consisting of 3 column operations is shown below. The main difference from traditional approaches is that this *Prosto* workflow will not modify any table - it changes only columns. Formally, if traditional approaches apply set operations by generating new sets from already inferred sets, then *Prosto* derives new *functions* from existing functions. In many cases, using functions (column operations) is much simpler and more natural.



Prosto provides two ways to define its operations:

- *Programmatically* by calling function with parameters specifying an operation
- *Column-SQL* by means of syntactic statements with all operation parameters. Column-SQL is a new way to define a column-oriented data processing workflow. It is a syntactic alternative to programmatic operations. Read more here: *Column-SQL*

Prosto operations are demonstrated in notebooks which can be found in the "notebooks" folder in the main repo. Do your own experiments by tweaking them and playing around with the code: https://github.com/asavinov/prosto/tree/master/notebooks

The column-oriented approach was used in the Intelligent Trading Bot for deriving new features: https://github.com/asavinov/intelligent-trading-bot

CHAPTER 1

Contents

1.1 Quick start (Column-SQL)

1.1.1 Creating a workflow

All data elements (tables and columns) as well as operations for data generation are defined in a workflow object (interpreted as a context):

```
import prosto as pr
prosto = pr.Prosto("My Prosto Workflow")
```

More info: Workflow and operations

1.1.2 Populating a source table

Each table has some structure which is defined by its *attributes*. Table data is defined by the tuples it consists of and each tuple is a combination of some attribute values.

The simplest way to populate a source table is to create or load a `pandas` data frame and then pass it to a Column-SQL statement:

```
sales_data = {
    "product_name": ["beer", "chips", "chips", "beer", "chips"],
    "quantity": [1, 2, 3, 2, 1],
    "price": [10.0, 5.0, 6.0, 15.0, 4.0]
}
sales_df = pd.DataFrame(sales_data)

prosto.column_sql("TABLE Sales", sales_df)
```

The Column-SQL statement `TABLE Sales` will create a definition of a source table with data from the `sales_df` data frame.

In more complex cases, we could pass a *user-defined function* (UDF) instead of the data frame. This function is supposed to "know" where to load data from by returning a `pandas` data. For example, it could load data from a `csv` file.

More info: Table operations

1.1.3 Defining a calculate column

A column is formally interpreted as a mathematical function which maps tuples (defined by table attributes) of this table to output values. The simplest column operation is a `calculate` column which *computes* output values using the values of the specified input columns of the same table:

```
prosto.column_sql(
    "CALCULATE  Sales(quantity, price) -> amount",
    lambda x: x["quantity"] * x["price"]
)
```

This new `amount` column will store the amount computed for each record as a product of `quantity` and `price`. The `CALCULATE` statement consists of two parts separated by arrow:

- First, we define the source table and its columns that we want to process as input: `Sales(quantity, price)`
- Second, we define a column to be created: `amount`

This use of arrows is an important syntactic convention of Column-SQL which informally represent a flow of data within one table or between tables.

More info: Column operations

1.1.4 Executing a workflow

A workflow object stores only operation *definitions*. In order to really process data, the workflow has to be executed:

```
prosto.run()
```

`Prosto` translates a workflow into a graph of operations (topology) taking into account their dependencies and then executes each operation.

Now we can explore the result by reading data form the table along with the calculate column:

```
df = prosto.get_table("Sales").get_df()
print(df)
```

```
   product_name quantity price amount
0  beer         1        10.0  10.0
1  chips        2        5.0   10.0
2  chips        3        6.0   18.0
3  beer         2        15.0  30.0
4  chips        1        4.0   4.0
```

The `amount` column was derived from the data in other columns. If we change input data, then we can again run this workflow and the derived column will contain updated results.

The full power of `Prosto` comes from the ability to process data in multiple tables by defininig derived links (instead of joins) and then aggregating data based on these links (without groupby). Note that both linking and aggregation

do not require and will not produce new tables: only columns are defined and evaluated. For example, we might use column paths like `my_derived_link::my_column` in operations in order to access data in other tables.

More info: Column-SQL

1.2 Quick start (programmatic)

1.2.1 Importing Prosto

`Prosto` is a toolkit and it is intended to be used from another (Python) application. Before its data processing functions can be used, the module has to be imported:

```
import prosto as pr
```

1.2.2 Defining a workflow

A workflow contains *definitions* of data elements (tables and columns) as well as *operations* for data generation. Before data processing operations can be defined, a `Prosto` workflow has to be created:

```
prosto = pr.Prosto("My Prosto Workflow")
```

More info: Workflow and operations

1.2.3 Defining a table

Each table has some structure which is defined by its *attributes*. Table data is defined by the tuples it consists of and each tuple is a combination of some attribute values.

There exist many different ways to populate a table with tuples (attribute values). One of the simplest one is a table `populate` operation. It relies on a *user-defined function* which is supposed to "know" how to populate this table by returning a `pandas` data frame with the data.

Below we define a table with three attributes which will be populated by the specified user-defined function:

```
sales_data = {
    "product_name": ["beer", "chips", "chips", "beer", "chips"],
    "quantity": [1, 2, 3, 2, 1],
    "price": [10.0, 5.0, 6.0, 15.0, 4.0]
}

sales = prosto.populate(
    # Table definition consists of a name and list of attributes
    table_name="Sales", attributes=["product_name", "quantity", "price"],

    # Table operation is an UDF, list of input tables and model (parameters for UDF)
    func=lambda **m: pd.DataFrame(sales_data), tables=[], model={}
)
```

The user-defined function in this example returns a `pandas` data frame with in-memory sales data. In a more realistic case, the data could be loaded from a CSV file or database. This data frame has to contain all attributes declared for this table.

Other table operations like `project`, `product` and `filter` allow for processing table data from already existing input tables which in turn could be populated using other operations.

More info: Table operations

1.2.4 Defining a column

A column is formally interpreted as a mathematical function which maps tuples (defined by table attributes) of this table to tuples in another table.

There exist many different ways to compute a mapping form one table to another table. One of the simplest column operations is a `calculate` column which *computes* output values of the mapping using the values of the specified input columns of the same table:

```
calc_column = prosto.calculate(
    # Column definition consists of a name and table it belongs to
    name="amount", table=sales.id,

    # Column operation is UDF, list of input columns and model (parameters for UDF)
    func=lambda x: x["quantity"] * x["price"], columns=["quantity", "price"],␣
↪model=None
)
```

This new column will store the amount computed for each record as a product of quantity and price. Note that the input columns could be also derived columns computed from some other data in this or other tables.

Other column operations like `link`, `aggregate` or `roll` allow for producing link columns referencing records in other tables and aggregate data.

More info: Column operations

1.2.5 Executing a workflow

When a workflow is defined it is not executed - it stores only operation definitions. In order to really process data, the workflow has to be executed:

```
prosto.run()
```

`Prosto` translates a workflow into a graph of operations (topology) taking into account their dependencies and then executes each operation: table operations will populate tables and column operations will evaluate columns.

Now we can explore the result by reading data form the table along with the calculate column:

```
df = table.get_df()
print(df)
```

```
   product_name quantity price amount
0  beer         1        10.0  10.0
1  chips        2        5.0   10.0
2  chips        3        6.0   18.0
3  beer         2        15.0  30.0
4  chips        1        4.0   4.0
```

Although it looks like a normal table, the last column was derived from the data in other columns. If we change input data, then we can again run this workflow and the derived column will contain updated results.

The full power of `Prosto` comes from the ability to process data in multiple tables by definining derived links (instead of joins) and then aggregating data based on these links (without groupby). Note that both linking and aggregation do not require and will not produce new tables: only columns are defined and evaluated. For example, we might use column paths like `my_derived_link::my_column` in operations in order to access data in other tables.

1.3 Column-SQL

1.3.1 What is Column-SQL

Column-SQL is a query language based on the principles of column-orientation where both columns and tables are main elements of data representation and processing (as opposed to using only tables in the relational and other set-oriented models). In Column-SQL, we process data by mainly deriving new columns from the data stored in other columns in this or other tables. This approach is simpler and more natural than set-oriented query languages where we derive tables from other tables. The main problem with traditional set-oriented query languages is that we frequently do not need to derive new tables at all and do it just because there is no choice.

1.3.2 Column-SQL structure

Column-SQL syntax at high level is intended for describing a flow of data from source columns (or tables) to newly defined target columns (or tables). It is not intended for describing complex workflows but rather short fragments of such workflows. If we define many such short statements then they together are treated as a complex workflow with dependencies. At high level, any Column-SQL statement starts from an operation name followed by a sequence of data elements (table and/or column) separated by the arrow symbol ->.

```
<OPERATION NAME> <TABLE NAME> ( <COLUMN NAME>, ...) -> <TABLE NAME> ( <COLUMN NAME>, .
↪..) ->
```

Note that between arrows we use a generic syntax for specifying data elements we want to process which is a table name followed by a sequence of its column names in parentheses:

```
<TABLE NAME> ( <COLUMN NAME 1>, <COLUMN NAME 2>, ...)
```

Note that both table name and column list could be empty and there could be single name specified treated either as column or table depending on the operation.

How the data elements between arrows are interpreted depends on the operation, and it will be described in the next sections. Here we give only one example of how a new calculated column could be defined which derives its values from two other columns in this same table:

```
CALCULATE  My_existing_table(A, B) -> my_new_column
```

This statement will add a new column to the `My_existing_table` by processing data in columns `A` and `B`. Note that the arrow here means that data from these two source columns flows to the new target column. In this case, this flow is very simply (each output value is computed from two input values) but other operations allows us to link tables and aggregate data from other tables.

According to the concept-oriented model of data, column-orientation means using mathematcial *functions* for representing the data and inferring new data. In particular, this means that how exactly we compute values of new columns is specified by functions. In Prosto, functions needed to compute columns are Python functions which are associated with each Column-SQL statement. Attaching a (Python) function is performed when we add a statement to the Prosto context:

```
ctx.column_sql(
    "CALCULATE  My_existing_table(A, B) -> my_new_column",
    lambda x: x['A'] + x['B']
)
```

The Python function passed to this statement expects a row with two fields in its data argument which are added and the result returned as a new column value.

Alternatively, the function can be passed within Column-SQL statement after the `FUNC` keyword:

```
ctx.column_sql(
    "CALCULATE  My_existing_table(A, B) -> my_new_column FUNC lambda x: x['A'] + x['B
↪']"
)
```

Functions may take an additional (static) argument which can be as simple as one number and as complex as a neural network (trained) model for computing forecasts:

```
ctx.column_sql(
    "CALCULATE  My_existing_table(A, B) -> my_new_column",
    lambda x, **m: x['A'] + x['B'] + param,
    {'param': 1.0}
)
```

It is similar to how `apply` works in `pandas` (and actually it relies on it in its implementation) but it is different from how `map` operation works because a calculated column does not add any new table while `map` computes a new collection (which makes computations less efficient).

This approach is somewhat similar to spreadsheets with the difference that new columns depend on only one coordinate - other columns - while cells in spreadsheets depend on two coordinates - row and column addresses. The columns defined in some Column-SQL statements can be then used in other statements and the system will evaluate them based on these dependencies.

Column-SQL statements are not executed immediately but rather are simply translated and added to the context. This workflow is evaluated as follows:

```
ctx.run()
```

In the next sections, we describe operations provided by Column-SQL.

1.3.3 POPULATE operation for importing data

```
df = pd.DataFrame({'A': [1, 2, 3]})
ctx.column_sql("TABLE  My_table (A)", lambda **m: df)
```

1.3.4 CALCULATE operation (instead of map operation)

The purpose of the CALCULATE operation is to create a new column in a table using data in other columns in this same table.

In this example, a `new_column_name` will be created and attached to the existing `My_table`:

```
ctx.column_sql(
    "CALCULATE  My_table (A) -> new_column",
    lambda x: float(x)
)
```

Note that we specify the input column `A` for our function and its value will be passed to the lambda function. The lambda function will be evaluated for each row of the table and its outputs will be stored as a new column.

We can specify more input columns for calculating values of the new output column. In addition, it is possible to pass an object with parameters to the function:

```
ctx.column_sql(
    "CALCULATE  My_table (A, B) -> new_column",
    lambda x, **m: x['A'] + x['B'] + param, model={"param": 5}
)
```

This query will compute a column where each value is the sum of values in columns `A`, `B` plus constant 5.

1.3.5 Compute column

The `COMPUTE` operation does the same as the `CALCULATE` except that its function gets whole columns rather than individual rows. The only difference is that the lambda function has to be implemented differently because its arguments are pandas Series.

1.3.6 LINK operation (instead of join)

The purpose of a link column is to store references to records in another table. Link columns are not valuable by themsevles but they can be then used in other operations to access data from different tables. In this sense, it is a main means of connectivity analogous to joins in the relational model.

Given two existing tables `Facts` and `Groups`, we can define a link from the first one to the second one as follows:

```
ctx.column_sql("LINK  Facts (A) -> link_column -> Groups (A)")
```

The `link_column` will be created in the `Facts` table by storing references to the records in the `Groups` table. The criterion of matching records is equality of columns `A` in these two tables.

The main use of link columns is in *column paths* which are sequences of simple column names following links between tables. For example, now we could use the column path `Facts::link_column::target_column` to reference `target_column` from table `Groups` in the context of table `Facts`. Link columns are also used as grouping criteria for aggregation.

1.3.7 ROLL operation (instead of over-partition)

Like `CALCULATE` operation, `ROLL` operation adds a new column to the same table where input columns are. However, each value of the new column is computed from many rows of this table and not one row. Thus `ROLL` operation aggregates data in many rows in the selected columns.

```
ctx.column_sql(
    "ROLL  My_table (A) -> roll_column WINDOW 2",
    lambda x: x.sum()
)
```

Each value in the `roll_column` will be computed as the sum of 2 values in the `A` column: one from this record and one from the previous record. The window length is specified in the `WINDOW` parameter. Currently, the logic of grouping logic is equivalent to that of the rolling aggregation in `pandas`.

1.3.8 AGGREGATE operation (instead of groupby)

The purpose of the `AGGREGATE` operation is to create a column each value of which aggregates data in several rows of another table. In this sense, it is an analogue of the `groupby` operation in SQL. Its main difference form `groupby` is that a new aggregated column is added directly to the table with groups and no new table is created.

```
ctx.column_sql(
    "AGGREGATE  Facts (M) -> link_column -> Groups (Aggregate)",
    lambda x, bias, **model: x.sum() + bias,
    {"bias": 0.0}
)
```

This statement adds an `Aggregate` column to the existing table `Groups`. Each value of this aggregate column is the sum of values in the `M` column for several records. All these records belonging to one group reference same record in the `Facts` table using the existing `link_column`.

1.3.9 FILTER operation (instead of select)

This operation is intended for filtering a table. However, its main difference form the conventional `SELECT` is that a new (filtered) table does not include any columns from the original table. Instead, it it creates a link column and references the selected records from the original (base) table.

In the current implementation, filter conditions are not specified in the operation itself and a boolean column in the base table is needed. The filtered table will include only records for which this column stores true values. In the `FILTER` statement, it is necessary to specify the base table name and the boolean column used for selection:

```
ctx.column_sql("FILTER BaseTable (filter_column) -> super -> FilteredTable")
```

This operation creates a new `FilteredTable` and a link column from the `FilteredTable` to the `BaseTable`. This link column in this example is called `super` (because the filtered table is a subset of the base table).

Note that we can treat the filtered table as a subset of the base table with all the original columns although they are not copied to the new table. We say that the base table columns are inherited by filtered tables.

1.3.10 PRODUCT operation

The purpose of the `PRODUCT` operation is to create a new table with all combinations of records from the source tables. It also will support filter which is currently not implemented. The new product table will create link columns to every of the source tables and will not contain the source columns.

```
ctx.column_sql("PRODUCT  Table_1; Table_2 -> t1; t2 -> Product")
```

The first part of the statement (before first arrow) is a list of source tables (separated by a colon). The second part (between arrows) is a list of the link column names which will be created. The last element `Product` is a name of the product table.

If columns from a source table need to be accessed in some other operation then it is done by means of the link columns as a column path like `Table_1::t1::source_column`.

Although the product operation looks analogous to join, it is has much narrow application scope. It is used mainly for multidimensional analysis (OLAP) and not for connectivity like join. If it is necessary to connect tables, then LINK operation should be used. It is a conceptual difference between the concept-oriented model relying on mathematical functions and the relational model relying on mathematical sets.

1.3.11 PROJECT operation

The main purpose of the `project` operation is to create a new table with unique combinations of attributes from the source table. For example, given a source table `Facts` with attributes `A` and `B` we can produce a new table `Groups` with all unique combinations of these attributes:

```
ctx.column_sql("LINK  Facts (A, B) -> link_column -> Groups (A, B)")
```

In addition to a new project table, this operation automatically creates a new link column in the source table which links records of the new project table. This link column can be then used for aggregation.

1.4 Motivation: Why Prosto?

1.4.1 Why functions and column-orientation?

In traditional approaches to data processing, we frequently need to produce a new table even though we need to define a new attribute. For example, in SQL, a new relation has to be produced even if we want to define a new calculated attribute. We also need to produce a new relation (using join) if we want to add a column with data from another table. Data aggregation by means of groupby operation produces a new relation too, although the goal is to compute a new attribute.

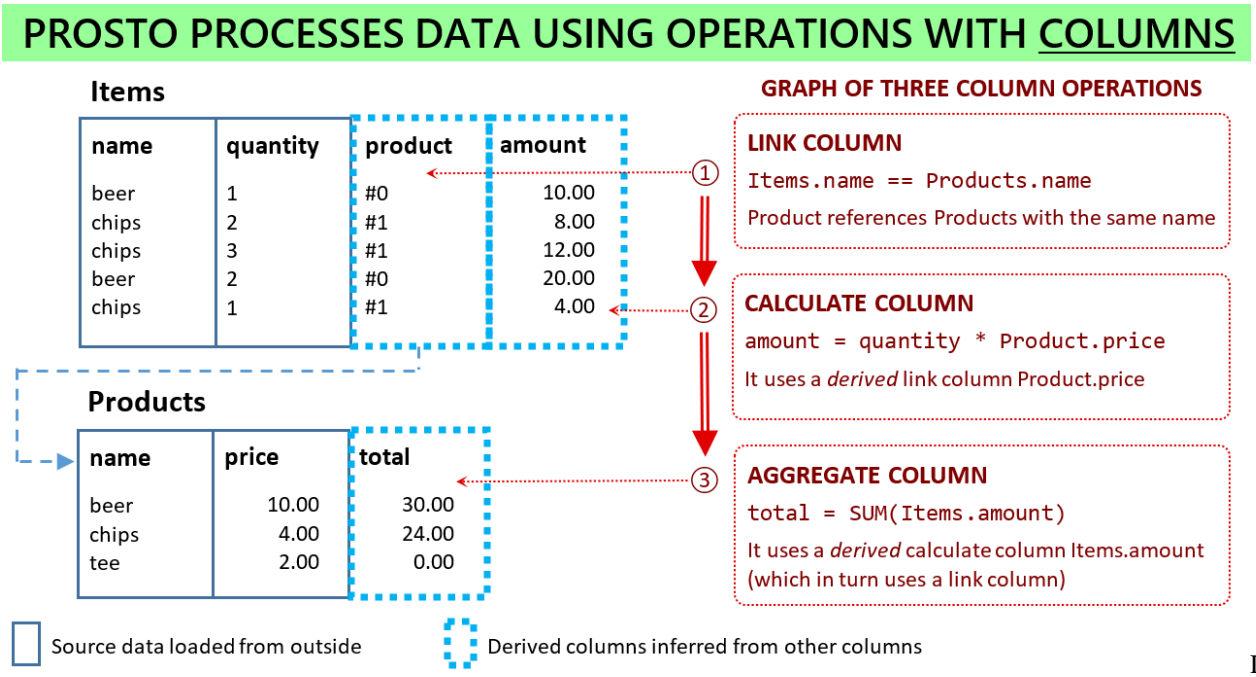
> In many important cases, processing data using *only* set operations is counter-intuitive, and this is why map-reduce, join-groupby (including SQL) and similar set-oriented approaches require high expertise and are error-prone

The main unique novel feature of `Prosto` is that it relies on a different formal basis:

> `Prosto` adds mathematical *functions* (implemented as columns) to its model by significantly simplifying data processing and analysis

Now, if we want to define a new attribute then we can do it directly without defining new unnecessary tables, collections or relations. New columns are defined in terms of other columns (in different tables) and this makes this model similar to how spreadsheets work but instead of cells we use columns. For comparison, if in spreadsheets we could define a new cell as `A1=B2+C3`, then in `Prosto` we could define a new column as `Column1=Column2+Column3`. The main theoretical challange is in introducing a set of column operations between columns in *multiple* tables in such a way that these operations effectively replace relational operations (join and groupby) and cover most important use cases. How it is done is described in *[2]*. `Prosto` with `Column-SQL` is one possible implementation of this model.

Below we describe three use cases where applying set operations is an unnecessary and counter-intuitive step. The example data model shown in this diagram is used for demonstration purposes. (Note however that it does not exactly corresponds to the use cases.)



Data processing workflow

Calculating data

One of the simplest operations in data processing is computing a new attribute using already existing attributes. For example, if we have a table `Items` containing orders characterized by `quantity` and `price` then we could compute a new attribute `amount` as their arithmetic product:

```
SELECT *, quantity * price AS amount FROM Items
```

This wide spread data processing pattern may seem very natural and almost trivial but it actually has one significant conceptual flaw:

> the task was to compute a new *attribute* while the query produces a new *table*

Although the result table does contain the required attribute, the question is why not to do exactly what has been requested? Why is it necessary to produce a new table if we actually want to compute only an attribute?

The same problem exists in map-reduce. If our goal is to compute a new field then we apply the map operation which will emit completely new collection of objects having this new field. Here again the same problem: our intention was not to create a new collection with new objects – we wanted to add a new computed property to already existing objects. However, the data processing framework forces us to describe this task in terms of operations with collections. We simply do not have any choice because such data models provide only sets and set operations, and the only way to add a new attribute is to produce a new set with this attribute.

An alternative approach consists in using *column operations* for data transformations so that we can do exactly what is requested: adding (calculated) attributes to existing tables.

Linking data

Another wide spread task consists in computing links or references between different tables: given an element of one table, how can I access attributes in a related table? For example, assume that `price` is not an attribute of the `Items` table as in the above example but rather it is an attribute of a `Products` table. Here we have two tables, `Items` and `Products`, each having `name` attribute which relates their records. If now we want to compute the amount for

each item then the price needs to be retrieved from the related `Products` table. The standard solution is to copy the necessary attributes into a *new table* by using the relational (left) join operation for matching the records:

```
SELECT item.*, product.price FROM Items item
JOIN Products product ON item.name = product.name
```

This new result table can be now used for computing the amount precisely as we described earlier because it has the necessary attributes copied from the two source tables. Let us again compare this solution with the problem formulation. Do we really need a new table? No. Our goal was to have a possibility to access attributes from the second `Products` table (while computing a new attribute in the first table). Hence this again can be viewed as a workaround rather than a solution:

> a new set is not needed for the solution and it is produced just because there is no possibility not to produce it

A principled solution to this problem would be a data model which uses *column operations* for data processing so that a link can be defined as a new column in an existing table.

Aggregating data

Assume that for each product, we want to compute the total number of items ordered. This task can be solved using group-by operation:

```
SELECT name, COUNT(name) AS totalQuantity
FROM Items GROUP BY name
```

Here again we see the same problem:

> a new unnecessary *table* is produced although the goal is to produce a new (aggregated) attribute in an existing table

Indeed, what we really want is to add a new attribute to the `Products` table which would be equivalent to all other attributes (like product `price` used in the previous example). This `totalQuantity` could be then used to compute some other properties of products. Of course, this also can be done using set operations in SQL but then we will have to again use join to combine the group-by result with the original `Products` table followed by producing yet another table with new calculated attributes. It is apparently not how it should work in a good data model because the task formulation does not mention and does not actually require any new tables - only attributes. Thus we see that the use of set operations in this and above cases is a problem-solution mismatch.

A solution to this problem again would be a column oriented data model where aggregated columns could be defined without adding new tables.

1.4.2 Benefits of Prosto

Easily processing data in multiple tables

New derived columns are added directly to tables without creating multiple intermediate tables.

We can easily implement calculate columns using `apply` method of `pandas`. However, we cannot use this technique in the case of multiple tables. `Prosto` is intended for and makes it easy to process data stored in many tables by relying on `link` columns which are also evaluated from the data.

Getting rid of join and group-by

Column definitions such as link columns and aggregate columns are used instead of join and groupby set operations.

Data in multiple tables can be processed using the relational join operation. However, it is tedious, error prone and requires high expertise especially in the case of many tables. `Prosto` does not use joins. Instead, it relies on `link` columns which also have definitions and are evaluated during workflow execution.

Data aggregation is typically performed using some kind of group-by operation. `Prosto` does not use this relational operation by providing column operations for that purpose which are simpler and more natural especially in describing complex analytical workflows.

Flexibility and modularization via user-defined functions

UDFs describe what needs to be done with the data only in this operation using arbitrary Python code. If UDF for an operation changes then it is not necessary to update other operations.

`Prosto` is very flexible in defining how data will be processed because it relies on user-defined functions which are its minimal units of data processing. They provide the logic of processing at the level of individual values while the logic of looping through the sets is implemented by the system according to the type of operation applied. User-defined functions can be as simple as format conversion and as complex as as a machine learning algorithm.

Parameterization of operations by a model object

A model can be as simple as one value and as complex as a trained deep neural network. This feature leads to a novel view of how data analysis should be organized by combining feature engineering and machine learning so that both model training and model use (predictions or transformations) are part of one data processing workflow. Currently models are supported only as static parameters but in future there will be a possibility to train model within the same workflow

Future directions

- In future, `Prosto` will implement such features as *incremental evaluation* for processing only what has changed, *model training* for training models as part of the workflow, data/model persistence and other data processing and analysis operations.
- *Data Dictionary* (DD) for declaring schema, tables and columns, and *Feature Store* (FS) for defining operations over these data objects

1.4.3 References

[1]: A.Savinov. On the importance of functions in data modeling, Eprint: arXiv:2012.15570 [cs.DB], 2019. https://www.researchgate.net/publication/348079767_On_the_importance_of_functions_in_data_modeling

[2]: A.Savinov. Concept-oriented model: Modeling and processing data using functions, Eprint: arXiv:1606.02237 [cs.DB], 2019. https://www.researchgate.net/publication/337336089_Concept-oriented_model_Modeling_and_processing_data_using_functions

[3]: A.Savinov. From Group-By to Accumulation: Data Aggregation Revisited, Proc. IoTBDS 2017, 370-379. https://www.researchgate.net/publication/316551218_From_Group-by_to_Accumulation_Data_Aggregation_Revisited

[4]: A.Savinov. Concept-oriented model: the Functional View, Eprint: arXiv:1606.02237 [cs.DB], 2016. https://www.researchgate.net/publication/303840097_Concept-Oriented_Model_the_Functional_View

[5]: A.Savinov. Joins vs. Links or Relational Join Considered Harmful, Proc. IoTBD 2016, 362-368. https://www.researchgate.net/publication/301764816_Joins_vs_Links_or_Relational_Join_Considered_Harmful

1.5 Concepts behind Prosto

1.5.1 Matrixes vs. sets

It is important to understand the following crucial difference between matrixes and sets expressed in terms of multidimensional spaces:

> A cell of a matrix is a point in the multidimensional space defined by the matrix axes - the space has as many dimensions as the matrix has axes. Values are defined for all points of the space. A tuple of a set is a point in the space defined by the table columns - the space has as many dimensions as the table has column. Values are defined only for a subset of all points of the space.

It is summarized in the table:

| Property | Matrix | Set | | — | — | — | | Dimension | Axis | Attribute | | Point coordinates | Cell axes values | Tuple attribute values | | Dimensionality | Number of axes | Number of attributes | | Represents | Distribution | Predicate | | Point | Value of distribution | True of false |

The both structures can represent some distribution over a multidimensional space but do it in different ways. Obviously, these differences make it extremely difficult to combine these two semantics in one framework.

`Prosto` is an implementation of the set-oriented approach where a table represents a set and its rows represent tuples. Note however that `Prosto` supports an extended version of the set-oriented approach which includes also functions as first-class elements of the model.

1.5.2 Sets vs. functions

Tuples are a formal representation of data values. A tuple has structure declared by its *attributes*.

A *set* is a formal representation of a collection of tuples representing data values. Tuples (data values) can be only added to or removed from a set. In `Prosto`, sets are implemented via table objects.

A *function* is a mapping from an input set to an output set. Given an input value, the output value can be read from the function or set for the function. In `Prosto`, functions are implemented via column objects.

1.5.3 Attributes vs. columns

Attributes define the structure of tuples and they are not evaluated. Attribute values are set by the table population procedure.

Columns implement functions (mappings between sets) and their values are computed by the column evaluation procedure.

1.5.4 `Pandas` vs. `Prosto`

`Pandas` is a very powerful toolkit which relies on the notion of matrix for data representation. In other words, matrix is the main unit of data representation in `pandas`. Yet, `pandas` supports not only matrix operations (in this case, having `numpy` would be enough) but also set operations, relational operations, map-reduce, multidimensional and OLAP as well as some other conceptions. In this sense, `pandas` is a quite eclectic toolkit.

In contrast, `Prosto` is based on a solid theoretical basis: the concept-oriented model of data. For simplicity, it can be viewed as a purely set-oriented model (not the relational model) along with a function-oriented model. Yet, `Prosto` relies on `pandas` in its implementation just because `pandas` provides a powerful set of highly optimized operations with data.

1.6 Workflow and operations

1.6.1 Structure of workflow

TBD

1.6.2 List of operations

`Prosto` provides two types of operations which can be used in a workflow:

- A *table population operation* adds new records to the table given records from one or more input tables
- A *column evaluation operation* generates values of the column given values of one or more input columns

`Prosto` currently supports the following operations:

- Column operations
  - `compute`: A complete new column is computed from the input columns of the same table. It is analogous to the table `populate` operation
  - `calculate`: New column values are computed from other values in the same table and row
  - `link`: New column values uniquely represent rows from another table
  - `merge`: New columns values are copied from a linked column in another table
  - `roll`: New column values are computed from the subset of rows in the same table
  - `aggregate`: New column values are computed from a subset of row in another table
  - `discretize`: New column values are a finite number of groups like numeric intervals
- Table operations
  - `populate`: A complete table with all its rows is populated and returned by the specified UDF similar to the column `compute` operaiton
  - `product`: A new table consists of all combinations of rows in the inputs tables
  - `filter`: A new table is a subset of rows from another table selected using the specified UDF
  - `project`: A new table consists of all unique combinations of the specified columns of the input table

Examples of these operations can be found in unit tests or Jupyter notebooks in the `notebooks` project folder.

1.6.3 Operation parameters

An operation in Prosto provides a general logic of data processing and it does not do anything by itself. An operation needs additional parameters which specify what exactly has to be done with the data. Below we describe parameters which are common to almost all operation types.

- Data elements and operations. It is important to understand that data elements and operations are different types of objects and they are managed separately in `Prosto`. We can create, update and delete them separately. Yet, for simplicity, Prosto provides functions which create an operation along with the corresponding new data element. For example, we call the `calculate` function then it will define one column and one operation. A new data element and a new operation are described by different parameters of the function.

- Data element definition. First two parameters of an operation define a data element. If it is a column operation like `link` then it defines a new column using its `name` and (existing) `table`. If it is a table operation like `project` then it is its `table_name` and a list of `attributes`. The rest of the operation parameters define an operation.
- Function. Most operations have a `func` argument which provides a user-defined function (UDF). This function "knows" what to do with the data. There are two types of functions: (i) functions which are called in an internal loop and take/return data values, (ii) functions which are called only once and take/return collections of values (columns or tables). For each operation it is specified which kind of UDF it uses.
- Data. Here we can specify what data has to be processed by the operation (and the corresponding UDF). For many column operations, it is a list of `columns` of the input table. It is assumed that only these columns have to be processed. For many table operations, it is a list of `tables`.
- Model. This argument of an operation is intended for providing additional parameter for data processing. The model object is passed to UDF which has to know how to use it. It can be as simple as one value and as complex as a trained data mining model. It can be a tuple, dictionary or an arbitrary Python object. A tuple will be unpacked in a list of positional arguments of UDF. A dictionary will be unpacked into a list of keyword arguments. An object will be passed as one positional argument.

1.7 Table operations

1.7.1 Populate table

A new table will be populated with data returned by the specified user-defined function. This operation is analogous to `compute` column operation with the difference that a complete table is returned rather than a complete column.

1.7.2 Product of tables (instead of join)

This table is intended to produce all combinations of rows in other tables. Its main difference from the relational model is that the result table stores links to the rows of the source tables rather than copies of its rows. The result table has as many attributes as it has source tables in its definition. (In contrast, the number of attributes in a relational product is equal to the sum of attributes in all source tables.)

Uses:

- Creating a cube table from dimension tables for multi-dimensional analysis. It is typically followed by aggregating data in the fact table.

1.7.3 Filter table (instead of select-where)

It is one of the most frequently used operations. The main difference form conventional implementations is that the result never includes the source table columns. Instead, the result (filtered) table references the selected source rows using an automatically created link column. If it is necessary to use the source table data (and it is almost always the case) then they are accessible via the created link column.

1.7.4 Project table (instead of select-distinct)

This operation has these important uses:

- Creating a table with group elements for aggregation because (in contrast to other approaches) it must exist
- Creating a dimension table for multi-dimensional analysis in the case it does not exist

Check out the `project.ipynb` notebook for a working example of the `project` operaiton.

1.7.5 Range table

Not implemented yet.

This operation populates a table with one attribute which contains values from a range described in the model. A range specification typically has such parameters as `start`, `end`, `step` size (or frequency), `origin` and others depending on the range type.

Links:

- https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.date_range.html
- https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#generating-ranges-of-timestamps

1.8 Column operations

1.8.1 Compute column

A `compute` column is intended for computing a new column based the values in other columns in the same row. It is defined via a Python user-defined function which gets several input columns and returns one output column which is then added to the table.

1.8.2 Calculate column (instead of map operation)

Probably the simplest and most frequent operation in `Prosto` is computing a new column of the table which is done by defining a `calculate` column. The main computational part of the definition is a (Python) function which returns a single value computed from one or more input values in its arguments.

This function will be evaluated for each row of the table and its outputs will be stored as a new column.

It is similar to how `apply` works in `pandas` (and actually it relies on it in its implementation) but it is different from how `map` operation works because a calculated column does not add any new table while `map` computes a new collection (which makes computations less efficient).

The `Prosto` approach is somewhat similar to spreadsheets with the difference that new columns depend on only one coordinate - other columns - while cells in spreadsheets depend on two coordinates - row and column addresses. The both however are equally simple and natural.

Check out the `calculate.ipynb` notebook for a working example of the `calculate` operaiton.

1.8.3 Link column (instead of join)

We can define and evaluate new columns only in individual tables but we cannot define a new column which depends on the data in another table. Link columns solve this problem. A link column stores values which uniquely represent rows of a target (linked) table. In this sense, it is a normal column with some values which are computed using some definition. The difference is how these values are computed and their semantics. They do not have a domain-specific semantics but rather they are understood only by the system. More specifically, each value of a link column is a reference to a row in the linked table or None in the case it does not reference anything.

The main part of the definition is a criterion for finding a target row which matching this row. The most wide spread criterion is based on equality of some values in two rows and the definition includes lists of the columns which have to be equal in order for this row to reference the target row.

Link columns have several major uses:

- Data in other (linked) tables can be accessed when doing something in this table, say, when defining its calculate columns
- Data can be grouped using linked rows interpreted as groups, that is, all rows of this table referencing the same row of the target table are interpreted as one group
- Link columns are used when defining aggregate columns

There could be other criteria for matching rows and defining link columns which will be implemented in future versions.

Check out the `link.ipynb` notebook for a working example of the `link` operaiton.

1.8.4 Merge column (instead of join)

Once we have defined link columns and interlinked our (initially isolated) set of tables, the question is how we can use these links? There are two major conceptual alternatives:

- move the whole linked columns to the source tables as one dedicated operation performed before the data in this column is used
- do not move the whole column but rather use this link to access individual data values in the linked column from within each operation which needs this data.

The second approach requires less memory because the (linked) data is used where it resides but it is less efficient because each value is accessed via the link. The first approach require more memory because we duplicate the linked column by moving it to the table where it will be used. However, access to this data will be as fast as to all other columns within this source table.

Currently, the first approach is implemented via the dedicated `merge` column operation. This operation specified a sequence of link columns from the source table to a target column in another table. Its result is a new column in this source table which contains the same data as in the target column and it can be used precisely as any other column in this table. The merged (copied) column can be then used in other operations like calculate columns or aggregate columns.

It is important that it is not necessary to use this operation explicitly. In other words, if we want to use a linked column in some operation, then we could merge it first but it is an explict and optional way. A simpler approach is to specify a *column path* as our column name in the operation. A column path is a sequence of simple column names separated by some symbol, '::' (two colons) by default. The translator will find such column paths and automatically insert the necessary merge operation.

1.8.5 Rolling aggregation (instead of over-partition)

This column will aggregate data located in "neighbor" rows of this same table. These rows to be aggregated are selected using criteria in the `window` object. For example, we can specify how many previous rows to select.

Currently, its logic is equivalent to that of the rolling aggregation in `pandas` with the difference that the result column is immediately added to the table and this operation is part of the whole workflow.

The `roll` operation can distinguish different groups of rows and process them separately as if they were stored in different tables. We refer to this mode as rolling aggregation with grouping. If the `link` parameter is not empty then its value specifies a column or attribute used for grouping.

Check out the `roll.ipynb` notebook for a working example of rolling aggregation.

1.8.6 Aggregate column (instead of groupby)

This column aggregates data in groups of rows selected from another table. The selection is performed by specifying an existing link column which links the fact table with this (group) table. The new column is added to this (group) table.

Currently, its logic is equivalent to that of the groupby in `pandas` with the difference that the result column is added to the existing table and the two tables must be linked beforehand.

Check out the `aggregate.ipynb` notebook for a working example of aggregation.

1.8.7 Discretize column

Let us assume that we have a numeric column but we want to partition it into a finite number of intervals and then use these intervals intead of numeric values. The `discretize` coumn produces a new column with a finite number of values where each such value represents a group the input value belongs to.

How the groups are identified and how the input space is partitioned is defined in the model. In the simplest case, there is one numeric column and the model defines intervals with equal length. These intervals are identified by their border value (left or right). The output columm will contain border values for the intervals input values belong to. For example, if we have temperature values in the input column like 21.1, 23.3, 22.2 etc. but we want to use discrete values like 21, 23, 22, then we need to define a `discretize` column. In this case, it is similar to rounding (which can be implemented using a `calculate` column) but the logic of discretization can be more complicated.

Links:

- https://numpy.org/doc/stable/reference/generated/numpy.digitize.html

1.9 Column paths

1.9.1 Link columns

A link column stores row identifiers of another table. Its purpose is to provide access to the data stored in another table. For example, a `Persons` table might store such attributes like name directly in this table while address information could be stored in another table called `Addresses`. In this case, the `Persons` table should have a column which references records of the `Addresses` table. For example, this link column could be called `address`. In contrast to other columns, its values are not used directly but rather are used to access values in the referenced table.

1.9.2 Column paths

If a table has a link column then it can be used to access columns in the referenced table. This is done by specifying a column path which is a sequence of link columns where each next column starts from the table where previous column ends. The last column in a column path is a normal table column or attribute the data of which has to be processed. Syntactically, `Prosto` uses double colon to separate segments in a column path. For example, in order to access a street column from the `Persons` table we write the following column path: `address::street`. This column path starts from the `Persons` table and ends in the `Addresses` table.

1.9.3 Defining link columns

Link columns are defined by means of the `link` operation. Given two tables, this operation will produce a new link column the values of which will reference the second (target) table from the first table. The criterion of matching is equality of the specified columns. For example, in Column-SQL a link column could be defined as follows:

```
ctx.column_sql("LINK  SourceTable(A, B) -> link_column -> TargetTable (A, B)")
```

Once this link column has been defined, it can be used in all operations where the `SourceTable` columns are used in the same way as simple columns, for example: `link_column::C` for accessing the `C` columns of the `TargetTable`.

Link columns are also automatically created by the `project` operation. This operation not only projects a source table by producing a new target table, but also creates a link between these two tables. For example, if the `TargetTable` was not available, then it could be created by projecting the `SourceTable` by finding all unique combinations of the `A` and `B` attributes:

```
ctx.column_sql("PROJECT  SourceTable(A, B) -> link_column -> TargetTable (A, B)")
```

1.9.4 Use of column paths

Column paths can be used where a computation needs to be done with some data in another table. For example, we might want to define a calculate column using data in a referenced table:

```
ctx.column_sql(
    "CALCULATE  Persons(position::salary) -> adjusted_salary",
    lambda x: x['position::salary'] * 1.1
)
```

Here the `salary` is stored in a table with all existing positions while `Persons` have a link to the position.

Link columns are used also in `aggregate` columns for grouping by assuming that all referencing one record means belonging to one group. For example, we could find mean age for each position as follows:

```
ctx.column_sql(
    "AGGREGATE  Persons (age) -> position -> Positions (mean_age)",
    lambda x: x.mean()
)
```

1.10 Design

1.10.1 Data schema

Prosto stores two lists as members of the `Prosto` class‘:

- List of table definitions in the `tables` field. A table object is represented by an instance of the `Table` class.
- List of column definitions in the `columns` field. A column object is represented by an instance of the `Column` class.

A table definition involves such fields as table name and a list of its attributes:

```
definition = {
    "id": "My table",
    "attributes": ["A", "B"],
}
```

A column definition involves such fields as column name and table name:

```
definition = {
    "id": "My column",
    "table": "My table",
}
```

1.10.2 Data

Real data being processed is stored in the `data` field of the `Table` class. The data is represented by an instance of the `Data` class. It relies on `pandas` `DataFrame` object for data representation by storing data for all columns of this table (so `Column` objects do not store any data).

The `Table` class stores also `pandas` groupby object for each link column which is then used in operations with the link columns.

1.10.3 Operations

Prosto stores all operation definitions in the `operations` field of the `Prosto` class‘. Each operation is a dictionary object with certain structure which is interpreted depending on the operation.

A operation definition includes the following fields:

```
definition = {
    "id": "My operaiton",
    "operation": "operation_name",  # Operation name (supported by Prosto)

    "outputs": ["My table"],  # What this operation produces (table or column)

    "tables": ["Table"],  # Source table
    "columns": ["A"],  # Source columns

    "function": func,  # UDF
}
```

There can be other fields depending on the operation.

A base class for all operations is `Operation`. It has two subclasses: `TableOperation` and `ColumnOperation`. These classes implement the logic of execution of each operation.

1.10.4 Dependencies

Each operation has dependencies as tables and columns which must be available before this operation can be executed. The dependencies are computed and returned depending on the operation type and its definition. These methods are implemented in the `Operation` class and its child classes.

1.10.5 Topology and translation

Topology represents a graph of operations which are ready to be executed and this object can execute them as one workflow. Translating a topology means generating such a list of operations from their definitions stored in the Prosto context. The translation procedure analyzes the list of operations with their dependencies and produces a graph. This procedure may also add new operations.

1.11 Install and test

1.11.1 Install from source code

Check out the source code and execute this command in the project directory (where `setup.py` is located):

```
$ pip install .
```

Or alternatively:

```
$ python setup.py install
```

1.11.2 Install from PYPI

This command will install the latest release of `Prosto` from PYPI:

```
$ pip install prosto
```

1.11.3 How to test

Run tests from the project root:

```
$ python -m pytest
```

or

```
$ python setup.py test
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search