Delex Tutorial

Version 1.0, February 10, 2026, by Dev Ahluwalia

Delex is a Python based blocking solution for entity matching. This guide provides step-by-step instructions to write a Python script to perform blocking using Delex. The guide will walk the user through the following steps:

1. Creating a Spark session
2. Loading in data + preprocessing
3. Creating a blocking program
    a. indexable vs. streamable rules
    b. keep rules
    c. drop rules
4. Creating a plan executor and executing it
5. Performing blocking
6. Saving blocking output

Step 1: Creating a Spark Session

Recall that Delex performs blocking on two tables and leverages Spark. Before we can use Delex, we need to set up a SparkSession. A SparkSession defines the location of the Spark master (driver node), name, and other optional settings. There is a link to the documentation at the end of this step. For this guide, we will set the master (driver node) as our local machine, and give it the name 'Delex Tutorial' to identify it. To create the SparkSession, we write the following in our Python script:

```
spark = SparkSession.builder\
          .master('local[*]')\
          .appName('Delex Tutorial’)\
          .getOrCreate()
```

From now on, when we need to use Spark operations in our script, we will use our SparkSession instance, which we called "spark" in the example above. If you think you will need to configure more options for your SparkSession, such as setting storage limits or other SparkSession options, please reference the SparkSession builder documentation directly.

Step 2: Loading in Data + Preprocessing

Assume we have two tables, table *A* and table *B*, which we will use for blocking. Table *A* and table *B* can have different columns and column types. However, both must have an id column.

An id column is a column where each value in the column is unique, and the values in the column are either 32 or 64 bit integers. The id column in each table can have any name.

On a local machine, table *A* and table *B* will be stored on disk initially. However, Delex works by using tables that are stored in DataFrames. So, we need to load the tables from the disk into DataFrames.

We can load the tables into either Spark DataFrames or Pandas DataFrames, but we recommend using Spark DataFrames over Pandas DataFrames. Unlike Pandas DataFrames, Spark DataFrames will create efficient plans for manipulating data and are meant for partitioning across nodes in a Spark cluster. To read more about the differences between Spark DataFrames and Pandas DataFrames, check out this article.

The goal in this step is to show users how to load in data from two popular file types, CSV files and Parquet files, to Spark DataFrames. If you need functionality beyond what is provided here, feel free to reference the Spark API.

Option 1: Loading data from a CSV file

This option is for users who have a table (or tables) stored in CSV file(s). There are two boolean flags that a user may want to set when loading in the dataset, header and inferSchema.

Setting Flag "header": The user should check whether the first row in their CSV is a header row. A header row contains the names of the columns in the dataset.

If the user finds a header row, they will set the header flag to True. This means that the first row in the CSV is a header which contains column names, and Spark will use these column names when creating the DataFrame.

On the other hand, if the header flag is set to False (or the header flag is omitted), the first row (and subsequent rows) will all be treated as data, and Spark will create column names in the following format, from left to right: _c0, _c1, … , _cN.

Setting Flag "inferSchema": A schema in this case is a mapping of every column to the data type that column contains. If you set inferSchema=True, Spark samples the data and guesses each type. If you set inferSchema=False, you need to supply the schema yourself, or Spark will assume every field is a String.

If you already know the column types, you can declare it explicitly with StructType and StructField objects as shown below:

```
schema = StructType([
  StructField("col1", IntegerType(),  True),
  StructField("col2", StringType(),   True),
  StructField("col3", ArrayType(DoubleType()), True),
  # … add one StructField per column …
])
```

The structure of each StructField is (“column_name”, Type, nullable). Nullable is a boolean value, and if it is set to True, it means null values can exist in the column. If it is False, then there will be an error if there are null values in that column.

You can choose the Type from Spark’s full set: BooleanType, ByteType, ShortType, IntegerType, LongType, FloatType, DoubleType, DecimalType(precision, scale), StringType, BinaryType, DateType, TimestampType, TimestampNTZType, ArrayType(elementType), MapType(keyType, valueType), StructType([...]). To read more about the data types and how to use them, you can reference the Spark documentation here.

The way to read in the CSV with your options is as follows:

```
data_path = Path(‘{path to parent directory}’)
table_a = spark.read.csv(str(data_path / '{file name}') , header={True or False}, inferSchema={True or False}, schema={StructType})
```

If inferSchema is True or inferSchema is False and you do not want to provide a schema, you should omit the “schema” option entirely.

Our Suggestion: It is important that Spark recognize the ID column as of type "numeric", because later Delex will hash the values in this column, and the hash function assumes the values to be numeric.

As a result, you should either set inferSchema=True to let Spark guess the types of the columns, including guessing the type of the ID column to be numeric. Or you can set inferSchema=False and then supply the types of the columns yourself, taking care to specify that the type of the ID column is numeric.

Option 2: Loading data from a Parquet file

Unlike CSV’s, Parquet files’ schemas and headers are well defined, so you do not need the “header”, “inferSchema”, or “schema” arguments.

To load in data from a Parquet file, use the following structure:

```
data_path = Path(‘{path to parent directory}’)
table_a = spark.read.parquet(str(data_path / '{file name}'))
```

Additional Guidance

You should repeat these steps for both table *A* and table *B***.** It is okay if one table is in a CSV file and the other table is in a Parquet file, as long as you load each of them individually with the correct method.

In this guide, table *A* will be the smaller table and table *B* will be the larger one.

Preprocessing

Delex does not require a specific schema for the data, with the caveat of requiring an id column, as mentioned earlier. This is particularly important if you set header=False, since Spark generates the header names for you, so you will need to know which column is your id column.

In order to view the schema of your Spark DataFrame, you can add this to your script:

```
      dataframe.printSchema()
```

where dataframe is the name of the DataFrame you loaded your data into.
If you want to rename any column in your DataFrame, you can write this in your script:

```
      dataframe = dataframe.withColumnRenamed('current_column_name',
'new_column_name')
```

where dataframe is the name of the DataFrame variable you created previously. You can repeat this process for each column you want to rename.

In order to ensure that your input tables are correctly formatted, we provide the function:

```
check_tables(table_a, id_col_table_a, table_b,
id_col_table_b)
```

The check_tables function:

- Verifies that id_col_table_a exists in table_a
- Verifies that id_col_table_b exists in table_b
- Confirms that values in id_col_table_a are int32's or int64's, exist for all rows, and are unique
- Confirms that values in id_col_table_b are int32's or int64's, exist for all rows, and are unique

If any of the above conditions are not satisfied, the function will raise a ValueError.
We recommend using these functions immediately after you read table_a and table_b into Spark DataFrames.

At this point, you should note the name of your id columns for each table for use in later steps.

Step 3: Creating a Blocking Program

We now describe how to create a blocking program. For the rest of this guide, we will use a tiny blocking program as a running example.

Consider the DBLP-ACM dataset, which includes two tables A and B where each tuple describes a publication citation, using the attributes **title, authors, venue, year**. Below is an example tuple:

_id	title	authors	venue	year
288	enhancing database correctness : a statistical approach	john smith, adam miller	sigmod conference	1995

The following tiny Delex program can be used to perform blocking on the above two tables. We will explain this program in detail shortly.

```
pub_blocker = BlockingProgram(
    keep_rules = [
        KeepRule([JaccardPredicate('title', 'title', QGramTokenizer(3), operator.ge, .4),
                  ExactMatchPredicate('venue', 'venue', lowercase=True, invert=False)]),      <== Keep rule K1
        KeepRule([BM25TopkPredicate('title', 'title', 'standard', 10)])                       <== Keep rule K2
        ],
    drop_rules = [
        DropRule([CosinePredicate('authors', 'authors', AlphaNumericTokenizer(), operator.lt, .3)]), <== Drop rule D1
        DropRule([ExactMatchPredicate('year', 'year', invert=True)])                           <== Drop rule D2
      ],
  )
```

Step 3.1: Keep Rules & Drop Rules

Delex is a declarative programming language. This means that a user specifies what they want done, not the steps to achieve it. Specifically, a user specifies rules to keep and drop candidates, and Delex will decide how to best execute these rules. The specifications to keep and drop candidates are called “Keep rules” and “Drop rules”, respectively.

- A **Keep rule** specifies a condition (or conditions) that must be met for a tuple pair (x in Table A, y in Table B) to be output by the blocker. For example, the blocker pub_blocker described above has two keep rules K1 and K2. The first keep rule has two conditions. The second keep rule has one condition.

- A **Drop rule** specifies a condition (or conditions) such that if a tuple pair satisfy the condition, then it cannot be in the blocking output. The blocker pub_blocker has two drop rules D1 and D2.

The blocking program must have at least one keep rule, and zero or more drop rules. The meaning of the program is as follows:

- First, we (conceptually) enumerate all tuple pairs (x in A, y in B), then keep only those pairs that satisfy at least one keep rule. In other words, we execute the keep rules and take the union of the outputs.
- Then we drop from the output of the keep rules all tuple pairs that satisfy at least one drop rule.
- Finally, we return all remaining pairs as the output of the blocker program.

Example: Consider again the program pub_blocker. We first conceptually create all pairs in the Cartesian product A x B, then execute keep rule K1 on A x B and output only those pairs satisfying K1. Then we execute keep rule K2 on A x B and output only those pairs satisfying K2. Then we union these two outputs. Next, we execute the two drop rules D1 and D2 on the union and drop all tuple pairs that satisfy D1 or D2. Finally, we return the remaining pairs.

Of course, in practice Delex does not execute by enumerating the entire A x B. It uses a much more efficient solution.

Step 3.2: Term Definitions

Before we can dive into the construction of the rules themselves, we need to define some terms. First, from now on, we will refer to “conditions” as **“predicates”**. A predicate establishes how two records (i.e., tuples) will be compared. The exact components of a predicate will be described in Section 3.3. In this section, we will define the difference between “indexable” and “streamable” predicates, and we will define the difference between “thresholding” and “top-k”.

Step 3.2.1: Indexable vs. Streamable

Indexable Predicates: These are predicates where we can construct an index and then use it to quickly find all tuple pairs satisfying that predicate.

Example: Consider the first predicate of Keep rule K1:

JaccardPredicate('title', 'title', QGramTokenizer(3), operator.ge, .4).

As we will explain later, this predicate keeps all tuple pairs (x in A, y in B) such that the Jaccard score between the two titles (i.e., publication titles x.title and y.title) is greater or equal 0.4, assuming that we tokenize each title into a collection of 3grams.

One way to execute this predicate is to enumerate all pairs (x in A, y in B), then for each pair we compute its Jaccard score and check if the predicate is satisfied. This is obviously very inefficient.

A better way is to construct an index I over the titles in Table A, such that given a tuple y in Table B, we can use the index I to quickly find all tuples in Table A whose titles share at least *one 3gram with the title of tuple y. Then we can pair these tuples with y and keep only those that satisfy the predicate.*

The current version of Delex provides a set of built-in predicate templates. See Section "Similarity Functions" in the Appendix. For example, from the predicate template *JaccardPredicate(index_col=index_column_name, search_col=search_column_name, tokenizer=tokenizer_name, op=operator_name, val=threshold_value)*, we can create the first predicate in keep rule K1: *JaccardPredicate('title', 'title', QGramTokenizer(3), operator.ge, .4).*

For each built-in predicate template, if Delex knows how to build an index for it, as described above, then Delex refers to this predicate template as "indexable" (see the 2nd column of the table under "Similarity Functions" in the Appendix).

Streamable Predicates: Consider again keep rule K1. To execute this rule, we can build an index I for the first predicate, *JaccardPredicate('title', 'title', QGramTokenizer(3), operator.ge, .4),* and use this index to quickly find and keep only those tuple pairs (x in A, y in B) that satisfy this predicate.

Then we can send each of the surviving tuple pairs to the second predicate, *ExactMatchPredicate('venue', 'venue', lowercase=True, invert=False),* to check if the pair satisfies this predicate. The output of executing the entire keep rule K1 will be the tuple pairs that satisfy both predicates of K1.

If we can evaluate *ExactMatchPredicate* on the fly, as described above, then we say that this predicate is streamable, because we can "stream" tuple pairs into it for checking.

The table under "Similarity Functions" in the Appendix shows which built-in predicate templates are streamable, i.e., can be executed in this fashion. All but one built-in predicate templates are streamable. The exception is *BM25TopkPredicate*, which is a top-k predicate and thus is not streamable (as we will elaborate below).

Delex provides an easy way to check if a predicate created by using a built-in predicate template is indexable or streamable. Once you have defined a predicate (say, "pred"), you can check if it is streamable by using `pred.streamable`. You can check if it is indexable by using `pred.indexable`. There are some predicates which are both indexable and streamable (see the Appendix).

Step 3.2.2: Threshold vs. Top-K

In Delex, most predicates need to establish a criteria that must be met to keep or drop a candidate. In the predicates provided by Delex, this is done in one of two ways: thresholding or top-k.

A threshold is a value for which your comparison is being checked against. For example, let’s say we are computing the Jaccard score over two tuples, and the result is 0.8. Let’s say that our keep rule uses a predicate to keep candidates (i.e., tuple pairs) which have a Jaccard score above a threshold of 0.7. Then, this candidate would pass. However, if we set our threshold to 0.9, this candidate would not pass because it has a Jaccard score lower than 0.9.

Top-k searches do not care about what score a record pair has, but rather, how the record pair ranks among all record pairs.

Example: Consider the predicate BM25TopkPredicate('title', 'title', 'standard', 10) in keep rule K2. This predicate says that for each tuple y in Table B, return the top-10 tuples in Table A that has the highest TF/IDF similarity scores with y, pair these tuples with y, and return the 10 pairs. Here, the TF/IDF similarity scores are computed between the titles of the tuples, using the standard tokenizer. Here k = 10, but the user can also set this to other values.

Currently, in Delex, the only predicate which uses top-k is BM25TopkPredicate. But, if you are ever unsure of whether a predicate uses top-k, once you define your predicate (we will call it pred), you can check via `pred.is_topk`.

Step 3.3: Constructing Predicates, Rules, and a Blocking Program

We now show how to construct a predicate, create individual Keep Rules and Drop Rules from these predicates, and package these rules into a Blocking Program. But before that, we highly recommend visiting the Appendix to see the built-in similarity function, tokenizer, and operator options currently offered in Delex.

Step 3.3.1: Constructing a Predicate

Constructing a predicate (using the built-in predicate templates in the Appendix) requires a few key decisions. A predicate is a “blueprint” of how to compare two records. The following questions must be answered when constructing a predicate:

1. Which similarity function is going to be used?
2. Which columns should be compared?
3. (Only necessary for some predicates) Which tokenizer should be applied?
4. Which operator should be used?
5. (For thresholding predicates) What should we set our threshold to be? / (For top-k predicates) What should we set the k value to be?

Once we have all of these components, establishing the predicates themselves is straightforward. So, here is an example of some choices we might make:

1. Which similarity function is going to be used?
    a. Jaccard Score
2. Which columns should be compared?
    a. ‘name’ from table *A*, and ‘name’ from table *B*
3. (Jaccard requires a tokenizer) Which tokenizer should be applied?
    a. 3gram
4. Which operator should be used?

        a. This is a keep rule, so will use a "greater than or equal to" operator
5. (For thresholding predicates) What should we set our threshold to be? / (For top-k predicates) What should we set our k value to be?
    a. We will set the threshold to 0.6

Now that we have each of the questions answered, we can create our predicate, using the template for Jaccard in the Similarity Functions table in the Appendix:

```
JaccardPredicate(col_from_index_table, col_from_search_table,
tokenizer, operator, threshold)
```

So, using our answers, our predicate would be:

```
pred = JaccardPredicate('title', 'title', QGramTokenizer(3),
operator.ge, .6)
```

Step 3.3.2: Constructing Keep Rules and Drop Rules

As mentioned earlier, a keep rule or a drop rule can contain one or more predicates. When multiple predicates are supplied to a single rule, all of the predicates in the rule must be true for the rule to evaluate as true.

Additionally, in Delex, we have the following restrictions on defining keep rules and drop rules in a blocking program:

- A keep rule must contain at least one indexable predicate
- A drop rule must contain at least one predicate
- All predicates in the drop rule must be streamable

Suppose that we have created four predicates: pred1, pred2, pred3, and pred4, and now want to create the following three rules:

1. A keep rule where both pred1 and pred2 must be true
2. A keep rule where just pred3 must be true
3. A drop rule where pred4 must be true

Then we will use the KeepRule and DropRule syntax as follows:

```
keep_rule_1 = KeepRule([pred1, pred2])
keep_rule_2 = KeepRule([pred3])
drop_rule = DropRule([pred4])
```

Note that when we create our keep rules and drop rules, we use lists, even if the rule is evaluating just a single predicate, such as keep_rule_2 and drop_rule.

Step 3.3.3: Constructing the Blocking Program

To create a blocking program, we will create a list of keep rules and a list of drop rules. Each blocking program must have at least one keep rule, and zero or more drop rules.

If your program has zero drop rules, you should still write "drop_rules = [ ]" in the blocking program. For more example blocking programs, see the complex_program_examples.ipynb Python Notebook.

In our example from Step 3.3.2, we have created keep_rule_1, keep_rule_2, and drop_rule. If we wanted to use all of these rules in a blocking program, our blocking program code looks as follows:

```
my_prog = BlockingProgram(
      keep_rules = [keep_rule_1, keep_rule_2],
      drop_rules = [drop_rule]
)
```

As mentioned above, we are not required to have drop_rules, and would use an empty list to indicate that. In that case, our blocking program looks as follows:

```
my_prog = BlockingProgram(
      keep_rules = [keep_rule_1, keep_rule_2],
      drop_rules = []
)
```

Step 4: Creating and executing a Plan Executor

Step 4.1: Creating a Plan Executor Object

In the next step, we must create a Delex object called a PlanExecutor, then use it to execute a blocking program. For example, we can create the following PlanExecutor (which we will explain shortly):

```
my_executor = PlanExecutor(
      index_table=table_A,
      search_table=table_B,
      optimize=True,
      estimate_cost=True,
      ram_size_in_bytes=8589934592,
      build_parallelism=8
)
```

then use it to execute the blocking program my_prog as follows:

```
candidates, stats =
my_executor.execute(prog=my_prog,search_table_id_col='_id')
```

We now describe how to create a PlanExecutor object. At a minimum, we need to provide the plan executor with the index table (table *A* for our example), the search table (table *B* for our example), indicate if we want to optimize the plan, and indicate if we want to estimate the cost of execution.

We can also provide two more optional arguments: build_parallelism and ram_size_in_bytes. We will detail how to set each of these parameters below. Note that for the rest of this section, when we say 'optimal', we are referring to the runtime performance of the blocking program plan.

The first two arguments, the index table and the search table, should be the tables you are blocking on.

Now consider the `estimate_cost` and `optimize` arguments.

- If the `estimate_cost` argument is false, then we will not use any metrics to estimate how much time the query (that is, the blocking program) may take to execute.
- If the `optimize` argument is false, then we will use a default plan for our query execution.
- If the user wants to create a more optimal query plan, the user should set both `estimate_cost` and `optimize` to true.
    - Technically, a user could set just one of these to be true. In the case that just `estimate_cost` is set to true, the cost for the query will be estimated, but it will not be used anywhere; essentially, the program will spend time doing cost estimation, but nothing will come of it. Hence, it is not recommended to only set `estimate_cost` to true.
    - On other hand, if you set `optimize` to true, without setting `estimate_cost` to true, the plan executor will still create an optimized plan, but it is unlikely to be as optimized as possible, since we are not estimating the cost. We will skip the details of how Delex optimizes the plan.
    - If both `estimate_cost` and `optimize` are set to true, the `optimize` function can create an even more optimal execution plan.

Optionally, we can set `build_parallelism`. This argument controls how many parallel threads are used to create a memory map of the data. The memory map is an optimization technique when we have data that exceeds memory, but we still want to access it fast. We suggest setting the value of `build_parallelism` to the number of cores available on your machine. If you set a number below that, you will not be taking full advantage of all of the cores, but if you set a number above that, you will have an "oversubscription", which can actually cause a slowdown. In Delex, the default value for `build_parallelism` is 4.

Next, we can also optionally set `ram_size_in_bytes`. This argument will only be used if `estimate_cost` is also set to true. Utilizing the `ram_size_in_bytes` parameter can help Delex create a more accurate cost estimation when the data does not fit in memory. Specifically, providing `ram_size_in_bytes` when all of the data does not fit in memory allows Delex to estimate how many chunks will be needed to process the data. If there is no number provided, then Delex will assume that all data fits in memory for cost estimation purposes. The ram size is the amount of memory available. For example, if you have a machine with 8 Gigabytes of memory, you would set `ram_size_in_bytes = 8000000000` (which is 8 Gigabytes).

So creating a PlanExecutor object should look as follows, assuming that our index table is table *A*, our search table is table *B*, we want to create the most optimal plan, our memory is 8GB, and we have 8 cores:

```
executor = PlanExecutor(
      index_table=table_A,
      search_table=table_B,
      optimize=True,
      estimate_cost=True,
      ram_size_in_bytes=8000000000,
      build_parallelism=8
)
```

Step 4.2: Executing a Blocking Program

Once we have created our plan executor object, we can execute our blocking program. Executing the program returns two objects: a DataFrame and execution statistics, which we discuss below.

To execute our blocking program, we must provide the plan executor with the blocking program, any columns that we want projected, and the name of our id column in table *B*.

Projecting a column means that the column will appear in the output. We discuss projection more below:

- No matter what the argument for projection is, the resulting DataFrame will have the columns 'id2', the id for a record in table *B*, and 'id1_list', a list of ids from table *A* that are candidates for the record 'id2' in table *B*.
- If you provide no argument to projection, then by default, all of the columns from table *B* will appear in the output. For example, if column *B* has the columns '_id', 'name', 'address', and you provide no value to the projection argument, your resulting DataFrame will have the columns 'id2', 'name', 'address', 'id1_list'. Note that the column '_id' was renamed to 'id2'.

- On the other hand, if you provide an empty list to projection (i.e. projection = []), the output will only contain ‘id2’ and ‘id1_list’.
- If you do provide an argument to projection, it must be a list, even if the list only contains one element. If you just provide a string such as projection=’name’ instead of projection=[‘name’], you will get an error.

We now briefly discuss the statistics that are returned. To help the user understand how their plan is being executed, we provide information about the time it took to optimize the plan, the time it took to conduct the cost estimation, and information about the execution itself. This information may be useful for some power users who want to inspect the execution, but for most users, these statistics can be ignored.

Example: Suppose we want to project all of the columns in our output, use the blocking program we created in Step 3, use the plan executor we created in Step 4.1, project all columns from table B to the output, save the DataFrame in a variable called my_candidates, and the statistics in a variable called my_stats. Then we would execute the program as follows:

```
my_candidates, my_stats =
executor.execute(prog=my_prog,search_table_id_col='_id')
```

Now suppose we only want to project the ‘name’ column from table B in our output, then we would execute the program as follows:

```
my_candidates, my_stats = executor.execute(prog=my_prog,
search_table_id_col='_id',projection=[‘name’])
```

Step 5: Saving Blocking Output

Now that you have performed blocking, you will likely want to save the blocking output DataFrame. We strongly recommend that you use the Parquet file format as it is significantly faster to read and write, has a more expressive data model, and is strongly typed.

If you want to save the output as ‘out.parquet’, you can run:

```
my_candidates.write.mode("overwrite").parquet("./out.parquet")
```

The argument to mode, “overwrite”, tells Spark that if there is already data at that destination, it should overwrite that data.

Other options for the argument include: append (appends the new data to the end of the file), error (errors if there is already data at the destination), or ignore (if there is already data at the destination, it will stay there, and your data will not be written). Additionally, to change the destination, change the argument in .parquet() to your desired path

Closing Notes

This tutorial is meant to give you an introduction to Delex and how to use it on your datasets. If you are unsure about how the methods work, what arguments they can take, or other functionalities not covered here, consult the API docs. If you need further support or have any questions, visit anhaidgroup.github.io/magellan, or email entitymatchinginfo@gmail.com.

Appendix

The information below is current as of February 7, 2026 with Delex version 0.1.0.

Similarity Functions

Here are the similarity functions currently implemented in Delex.

Sim Function Name	Indexable	Streamable	Thresholding	Top-K	Predicate Template
Exact Match[1]	If not inverted	Y	N	N	ExactMatchPredicate(index_col=index_column_name, search_col=search_column_name, invert=should_invert, lowercase=should_lowercase)
Jaccard	If the operator is operator.gt or operator.ge	Y	Y	N	JaccardPredicate(index_col=index_column_name, search_col=search_column_name, tokenizer=tokenizer_name, op=operator_name, val=threshold_value)

Overlap Coefficient	N	Y	Y	N	OverlapCoeffPredicate(index_col=index_column_name, search_col=search_column_name, tokenizer=tokenizer_name, op=operator_name, val=threshold_value)
Cosine	If the operator is operator.gt or operator.ge	Y	Y	N	CosinePredicate(index_col=index_column_name, search_col=search_column_name, tokenizer=tokenizer_name, op=operator_name, val=threshold_value)
Edit Distance	N	Y	Y	N	EditDistancePredicate(index_col=index_column_name, search_col=search_column_name, op=operator_name, val=threshold_value)
Jaro	N	Y	Y	N	JaroPredicate(index_col=index_column_name, search_col=search_column_name, op=operator_name, val=threshold_value)
Jaro Winkler[2]	N	Y	Y	N	JaroWinklerPredicate(index_col=index_column_name, search_col=search_column_name, op=operator_name, val=threshold_value prefix_weight=weight_for_shared_pr efix)
Smith Waterman[3]	N	Y	Y	N	SmithWatermanPredicate(index_col=index_column_name, search_col=search_column_name, op=operator_name, val=threshold_value, gap_cost=insertion_deletion_penalt y)
TF/IDF[4]	Y	N	N	Y	BM25TopkPredicate(index_col=index_column_name, search_col=search_column_name, tokenizer=tokenizer_name, k=number_of_candiates_per_record)

[1]For Exact match, there are two additional optional arguments not previously mentioned: invert and lowercase. Invert means we will look for records that are not an exact match (different values). Lowercase means that we will

lowercase our search (for example john would be an exact match for John).

[2]In Jaro Winkler, a user may set a prefix_weight. By default, this is set to 0.1. This argument determines how much weight to give to a shared prefix. If this is set to 0, then Jaro Winkler becomes the same score as Jaro. This value should not exceed 0.25; otherwise, your similarity score could be greater than 1. For more information on Jaro Winkler, visit the py_stringmatching documentation.

[3]In Smith Waterman, a user may set a gap_cost. By default, this is set to 1.0. This argument determines the penalty for insertions or deletions.

[4]The TF/IDF sim function uses the Sparkly software for execution. Therefore, in the tokenizer field, the user should select a tokenizer from the tokenizers available in Sparkly, rather than a Delex tokenizer. A list of the Sparkly tokenizers is available at the end of the Sparkly Tutorial.

Comparison Operators

Operator	**Function**
operator.ge	Greater than or equal to
operator.gt	Greater than
operator.eq	Equal to
operator.ne	Not equal to
operator.le	Less than or equal to
operator.lt	Less than

Tokenizers

Analyzer Name	**What It Does**	**Example: “Hello world. How are you today? October 21, 2025”**	**How to Use in Delex**
StrippedWhiteSpaceTokenizer	Converts to lowercase, removes characters that are not letters or digits, then splits the remaining tokens by whitespace	{'hello', 'world', 'how', 'are', 'you', 'today', ‘october’, ‘21’, ‘2025’}	StrippedWhiteSpace Tokenizer()
ShingleTokenizer	Applies the StrippedWhiteSpaceTokenizer, and then returns concatenated tokens using n tokens The example uses n=3	{'helloworldhow', 'worldhoware', 'howareyou', 'areyoutoday', ‘youtodayoctober’, ‘todayoctober21’, ‘october212025’}	ShingleTokenizer(n=max_length_of_combos)

WhiteSpaceTokenizer	Converts to lowercase, splits the tokens by whitespace	{‘hello’, ‘world.’, ‘how’, ‘are’, ‘you’, ‘today?’, ‘october’, ‘21,’, ‘2025’}	WhiteSpaceTokenizer()
NumericTokenizer	Creates tokens only using the numbers in the string.	{‘21’, ‘2025’}	NumericTokenizer()
AlphaNumericTokenizer	Lowercases the string, only keeps tokens with letters or digits	{'hello', 'world', 'how', 'are', 'you', 'today', ‘october’, ‘21’, ‘2025’}	AlphaNumericTokenizer()

Qgram	Lowercases the string, splits into tokens of size Q
	
	The example uses q=4
		{'hell', 'ello', 'llo ', 'lo w', 'o wo', ' wor', 'worl', 'orld', 'rld.', 'ld. ', …, 'r 21', ' 21,', '21, ', '1, 2', ', 20', ' 202','2025'}	QgramTokenizer(
			q=token_length,
			use_freqs=order_tokens_by_decreasing_frequency
			)

Stripped_Qgram	Qgram, but it filters out characters that are not letters, digits, or underscores The example uses q=4	{'hell', 'ello', 'llow', 'lowo', 'owor', 'worl', 'orld', 'rldh', 'ldho', 'dhow', …, 'ober', 'ber2', 'er21', 'r212', '2120', '1202', '2025'}	StrippedQgramTokenizer(q=token_length, order_tokens_by_decreasing_frequency)