Sparkly Tutorial

Version 1.1, February 10, 2026, by Dev Ahluwalia

This tutorial describes how to perform blocking for entity matching using Sparkly. We first discuss how Sparkly works, and then provide a step-by-step guide to walk you through creating your own blocking program. You should first read through the "How Sparkly Works" section in its entirety because it will define terms and concepts that are used in the step-by-step guide.

Additionally, Sparkly leverages Spark to speed up blocking, and Spark can be used either on a local machine or on a cluster of machines. On a cluster of machines, Spark treats each machine as a worker node to distribute work to. On a local machine, Spark treats each CPU core as a worker node instead. For this guide, we are using Spark on a local machine. So, a Spark cluster will refer to all of the CPU cores on a local machine, not a cluster of machines.

How Sparkly Works

Sparkly focuses on blocking, which aims to reduce the search space for finding matching records across two tables. In Sparkly, blocking works by finding possible matching records from the smaller table *for each record in the larger table*. We will establish the following terminology for the guide:

- The possible matching records from the smaller table are referred to as *candidates* or *candidate records*.
- The smaller table is the *indexing table*, since it will have an index built for it. If you are unfamiliar with what an index is, that is okay. We will discuss what an index is, why we use them, and how to use them.
- The larger table is the *search table*, since for each record in this table, we will search the smaller table (using an index) to find records that can potentially match this record.
- A *record* is a row of data. We may also refer to it as a *tuple.*
- A *value* is a piece of information from one of the columns in a row of data.

Now that we have defined our terms, let's look at how Sparkly works. As mentioned above, Sparkly performs blocking on two tables. For this tutorial, we will assume two tables, A and B, as illustrated below.

- Table *A* will be the indexing table that contains candidate records. It has the following columns:
    - _id (a unique identifier for the row),
    - Name (a person's name),
    - DOB (a person's date of birth, in the format MM/DD),

    - Postal Code (the postal code from a person's address), and
    - Degree (the highest degree a person has obtained).
- Table *B* will be the search table. It has the following columns:
    - _id (a unique identifier for the row),
    - Name (a person's name),
    - Birthday (a person's date of birth, in the format DD/MM), and
    - Postal Code (the postal code from a person's address).

Table *A* (Indexing Table)

_id	**Name**	**DOB (MM/DD)**	**Postal Code**	**Degree**
0	John M. Doe	01/10	12345	B.A.
1	Jane N. Doe	07/16	53127	B.S.
2	Jim L. Smith	03/13	86941	Ph.D.
…	…	…	…	…
999	Dave M. Smith	09/07	35274	M.S.

Table *B* (Search Table)

_id	**Name**	**Birthday (DD/MM)**	**Postal Code**
0	Aaron Jones	16/07	53127
1	David Smith	07/09	35275
2	Jonathon Doe	10/01	12345
…	…	…	…
9999	Janet Smith	18/04	57234

Blocking requires that Table A is indexed and Table B is searched (that is, for each record x in Table B, find records in Table A that can potentially match x). The index of Table A will be sent to each Spark worker in our cluster.

There are forward indexes and inverted indexes. A forward index is a mapping of records to terms. A term is a part of a value from a record. For example, consider the value "John M. Doe" from the record with _id 0 in table A. We can tokenize the value to be the terms 'john', 'm', and 'doe'. There are several ways to tokenize values, as described in the Appendix. If we tokenize "John M. Doe" as 'john', 'm', and 'doe', a forward index for Table A, using just column "Name", would look like this:

Record	Term
0	‘john’
0	‘m’
0	‘doe’
…	…
999	‘m’

Building a forward index like the one above would be inefficient for our use because we would end up looking through every record’s list of tokens. Instead, *we want to build a mapping of terms to records*. This is called an inverted index. An inverted index for this scenario would look like this:

Term	Document
‘john’	0
‘m’	0, 999
‘doe’	0
…	…

Now, instead of going into each record and searching for the term, we will search over the terms, and find the documents that contain that term.

Now let's discuss how we will build an inverted index for Table A (so that we can do blocking). This will be done in two parts:

1. Choosing the fields (that is, the columns) to build the index with
2. Choosing the analyzers to apply to those fields (that is, deciding how to tokenize the fields).

Selecting Fields in Table A for the Index: Since the index is going to be what we search through, we will want to determine the pairs of fields that have similar data values between table *A* and table *B*.

In your data, fields that have similar data values could have the same name, like “Name” and “Postal Code” in our tables, but it is okay if they have different names. For example, table *A* contains “DOB” (MM/DD) and table *B* contains “Birthday” (DD/MM). Although the columns have different names, they contain similar information (birthdays).

We do not need to include every field of Table A in our index. In our example, we will not want to build an index using the “Degree” field from table *A* since there is no similar field in table *B*. Additionally, we do not want to include the “_id” field when building the index because the “_id” values do not contain information that is relevant for matching, they just help identify the record. So, we will build our index using the following fields from table *A*: “Name”, “DOB”, “Postal Code”. *Intuitively, we should use a small set of fields in Table A that (a) we can find similar fields in Table B to compare with, and (b) we believe that matching records between Tables A and B will have similar values for those fields.*

Deciding How to Tokenize the Selected Fields: After choosing our index fields, we will decide on the analyzers (that is, tokenizers) we want to use to break each of these fields into a bag of terms. *From this section on, we will use the words “terms” and “tokens” interchangeably.* A bag of terms means that the order the terms are in is irrelevant. Using the tokenization of “John M. Doe” from earlier, it would not matter if the name was written as “John M. Doe” or “Doe M. John”; both of these would result in the bag of terms “john”, “m”, and “doe”.

Further, in our example, we can see that the “DOB” and “Birthday” fields have similar data, but the “DOB” field is MM/DD and the “Birthday” field is DD/MM. So, the value 09/07 in the “DOB” column and the value 07/09 in the “Birthday” column are actually the same date.

In order to account for this, we want to convert the field into tokens. Tokens are smaller representations of the values in a field, and Sparkly currently offers 9 options, summarized in the Appendix. One of the tokenizer options is “2gram”, which converts a field value into a set of tokens of size two, then keeps only the alphanumeric ones.

So, for the DOB field, it would convert the value 09/07 into tokens '09', '9/', '/0', '07', then keep only ‘09’ and ‘07’. Then, in the Birthday field, it would convert the value 07/09 into tokens ‘07’, ‘7/’, ‘/0’, ‘09’, then keep only ‘07’ and ‘09’. Although the dates were in different orders, they are both broken into the terms ‘07’ and ‘09’, and the order of the terms does not matter. For each field that we are indexing, we can choose to use one or more tokenizers.

After creating the index, we need to decide which fields in table *B* we should use to search over the index in table *A*. Using a field in table *B* to search over the index in table *A* means that for each value in the field from table *B*, we will compare it to the values in the indexed field from table *A* (*Note: This comparison is what drives our score for how similar two records are. Sparkly scoring will be covered in the next paragraph*). In our example, we will want to search over the “Name” index created from table *A* with the “Name” field in table *B*. Additionally, we will want to search the “DOB” index created from table *A* with the “Birthday” field in table *B*. Lastly, we will want to search the “Postal Code” index from table *A* with the “Postal Code” field in table *B*.

Sparkly computes scores using BM25, which is a popular keyword search score based on TF/IDF (term frequency/inverse document frequency). TF/IDF works by giving higher scores to records in A and B which share terms, while discounting terms which are common in records. If you are interested in reading about the details of the scoring mechanism, see Section 3.1 in the

Sparkly Technical Report. However, a general understanding that unique terms that are shared by records in table *A* and table *B* helps increase a pair score, and generic terms that are shared by records in table *A* and table *B* discounts a pairs score is sufficient for creating your own program.

When we create our index, if we choose multiple field names and tokenizers, we will end up with scores for each (field name, tokenizer) pair we choose. The default behavior in Lucene is to add up the score for each field and tokenizer pair we choose to compute the final score between two records.

For example, let's say there is a record in *A*, a_1, being compared to a record in *B*, b_1. Then, if s_1 is the score of the "Name" field with the "3gram" tokenizer for (b_1, a_1), s_2 is the score of the "Name" field with the "standard" tokenizer for (b_1, a_1), and s_3 is the score of the "Postal Code" field with the "3gram" tokenizer for (b_1, a_1), the final score for (b_1, a_1) is $s_1+s_2+s_3$.

Sometimes, this behavior is okay. However, we could look at our data and say that the "standard" tokenizer on the "Name" field should have more weight than the "3gram" tokenizer on the "Name" field, and that the "3gram" tokenizer on the "Postal Code" field should have more weight than the "standard" tokenizer on the "Name" field. This could lead to a decision that the "3gram" tokenizer on the "Postal Code" field should hold 50% weight, the "standard" analyzer on the "Name" field should hold 30% weight, and the "3gram" analyzer on the "Name" should only hold 20% weight. Then, the score for (b_1, a_1) would be $.5*s_3 + .3*s_2+.2*s_1$. This concept is referred to as a *boost map*, and Sparkly supports users creating their own boost map, as we will discuss later.

Now that we know how to build our index and how scores are calculated, we can talk about setting the limit of candidates records from table *A* for each record in table *B*. Recall that blocking is the first part of entity matching, and the second part of entity matching is the matching part. We perform blocking to reduce the search space for the matching step. If we do not reduce the search space, we would compare each record in B to each record in A during matching. If we take the size of B to be |B| and the size of A to be |A|, our search space would be |B|x|A|. For example, let's take two small tables with |B| = 10,000 and |A| = 1,000. Without blocking, the search space size would already be 10,000*1,000 = 10,000,000. However, in reality, we could have |B| = 10,000,000 and |A| = 1,000,000, yielding a search space of 10^{13}.

We want to pick a limit that is low enough to significantly reduce the size of our search space, but large enough to increase our *recall* - the percentage of actual matches which still exist after the blocking step. If you have labeled data (gold), you can experiment with the limit to find a suitable limit. However, if you do not have gold data, your limit choice should aim to reduce your search space to a certain fraction of the worst-case search space. If you have small datasets, like |B| = 10,000 and |A| = 1,000, your choice of limit could aim to reduce the search space to 10% of the full search space, so you would choose a limit of 100. This means that for each record in B, it will find the 100 highest scoring records in A as the candidates. However, if |B| is 10,000,000 and |A| is 1,000,000, reducing the search space to 10% of the worst-case search space (limit = 1,000,000 * .1 = 100,000) would still leave a search space for matching of 10^{12}, so you would likely want to reduce your limit further.

Finally, at the end of blocking your output schema will be id2, id1_list, scores, search_time. There will be one record for each record in table *B*. id2 is the id of that record in table *B*. id1_list is a list with its maximum length equal to your limit, and contains the ids of candidate records in table *A* (that Sparkly believe potentially match the target record in Table B). scores is a list of maximum length equal to your limit and contains the scores of the candidate records in table *A*. search_time is the time it took to complete the search for candidates for the record in table *B* over the index from table *A*.

Lastly, you can save your blocking output. Now that you have an understanding of how Sparkly works, use the tutorial below to start blocking on your own data.

Step-by-Step Walkthrough

As a reminder, Sparkly is a Python library, and this guide will provide step-by-step instructions to write a Python script to perform blocking using Sparkly. The guide will walk the user through the following steps:

1. Creating a Spark Session
2. Loading in Data + Preprocessing
3. Building an Index + Creating a Query Specification
    a. Option 1: Sparkly Manual
        i. Choosing Index Fields
        ii. Choosing Analyzers
        iii. Creating an Index
            1. Creating the Index Configuration
            2. (Optional) Concatenating Fields
        iv. Applying the Index
        v. Creating the Query Specification
    b. Option 2: Sparkly Auto
        i. Creating an Index
        ii. Creating the Query Specification
    c. (Optional) Creating a Boost Map
4. Performing Blocking
5. Saving Blocking Output

Step 1: Creating a Spark Session

Before we can use Sparkly, we need to set up a SparkSession. A SparkSession defines the location of the Spark master (driver node), name, and other optional settings. For this guide, we will set the master (driver node) as our local machine, and give it the name ‘Sparkly Tutorial’ to identify it. To create the SparkSession, we write the following in our Python script:

```
spark = SparkSession.builder\
                .master('local[*]')\
                .appName('Sparkly 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, see the SparkSession builder documentation.

Step 2: Loading in Data + Preprocessing

Important Assumptions about the Schemas of Tables A and B: 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:

- Each table must have an id column. An id column is a column where each record must have a value for that column, the values must be unique, and the values in the column are either 32 or 64 bit integers. The id column can be named anything, but you will need to know what its name is.
- If you use Sparkly Manual (discussed below), then Tables A and B do not have to share any columns, because you will *manually* relate the columns, for example, specifying that Column 'Birthday' of Table B will be compared with Column 'DOB' of Table A.
- However, if you use Sparkly Auto (discussed below), then you do not have an opportunity to manually relate the columns. Instead, Sparkly Auto will automatically relate the columns. As such, Sparkly Auto requires all columns of Table A (except the id column) to appear as columns in Table B. Sparkly Auto will work on these shared columns (and ignore all columns in Table B that do not appear in Table A).

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

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, see 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). Users may want to set the boolean option ‘header’ when loading in the dataset. First, before setting this option, 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’ option 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’ option is set to False (or the ‘header’ option 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. You may change these names after you load the data into your DataFrame. The steps to do so are located below in the “Preprocessing” section.

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})
```

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” argument.

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 which will have an inverted index built for it. Table *B* will be the larger table which will be our searching table.

Preprocessing

Sparkly does not require a specific schema for the data, with the caveat of requiring an id column, which was 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 two functions:

1. check_tables_manual(table_a, id_col_table_a, table_b, id_col_table_b). Use check_tables_manual if you plan to use Sparkly Manual (described in Step 3, Option 1).
2. check_tables_auto(table_a, id_col_table_a, table_b, id_col_table_b). Use check_tables_auto if you plan to use Sparkly Auto (described in Step 3, Option 2).

The check_tables_manual 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_column_from_table_a are int32's or int64's, exist for all rows, and are unique
- Confirms that values in id_column_from_table_b are int32's or int64's, exist for all rows, and are unique

The check_tables_auto function performs all of the checks from the check_tables_manual_function and:

- Confirms the columns available in table_b a superset of the columns available in table_a

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: Building an Index + Creating a Query Spec

After loading the data into Spark DataFrames, we can prepare for blocking. Our blocking preparation consists of the following steps: choosing fields to build an inverted index on (including concatenated fields optionally), choosing analyzers which create tokens for the data, building the inverted index using the chosen analyzers and fields, and creating a query spec to use for searching.

Sparkly offers two options for building the index and creating a query spec: Sparkly Manual and Sparkly Auto. Sparkly Manual requires the user to specify the analyzers, index fields and query spec. Sparkly Auto, on the other hand, configures these options for the user. Although Sparkly Auto can be less work, it is not guaranteed to outperform Sparkly Manual. *Additionally, as* *mentioned earlier, unlike Sparkly Manual, Sparkly Auto requires that all fields of Table A (except the id field) exist in Table B. It will only do indexing and searching on these shared fields (thus ignoring all fields of Table B that do not exist in Table A).*

The goal of this step is to walk the user through choosing index fields, analyzers, building the index, creating the query specification, and optionally changing the weighting of field/analyzer combinations using a boost map.

Option 1: Sparkly Manual

As mentioned above, Sparkly Manual requires the user to specify the analyzers, index fields, and query spec. The following steps will walk the user through these options.

Step 3.1: Choosing Index Fields

Choosing index fields requires the user to understand the values in their tables. The user will want to choose index fields such that the index field in table *A* contains similar values to a field in table *B*. For example, if you have the fields (‘name’, ‘dob’) in the table you are indexing (Table A) and the fields (‘name’, ‘address’) in the table you are searching (Table B), it may not make sense to build an index using ‘dob’ since there is not a similar field in table *B*.

However, if the fields in the searching table were instead (‘name’, ‘birthday’), it could make sense to build an index on ‘dob’ since it may have similar information to the ‘birthday’ field in table *B*. As shown by this example, it is not required that the fields in table *A* and table *B* that share similar values have the same name. You will want to note the index field(s) you choose for use in step 3.3.1.

In Sparkly, we also offer the ability to create an index on a concatenated field. The concatenated field will combine two or more fields into a single attribute for indexing and search. For example, if table *A* has “fName” and “lName” as columns you may want to combine them into a single attribute for a name. Creating a concatenated field is covered further in step 3.3.2.

Step 3.2: Choosing Analyzers

Analyzers are used to split data into tokens for indexing. In the Appendix, the Analyzers Table lists the analyzers that are supported by sparkly-em:0.1.0 and Lucene 9.12.0. If the version of Sparkly or Lucene differ from this, the analyzers below may work differently or may be deprecated entirely.

These analyzers will be used along with the index fields to create an inverted index on table *A*. Note your analyzer(s) for step 3.3.1.

Step 3.3: Create an Index

Steps 3.1 and 3.2 should have led the user to select the index fields and analyzers they want to build the inverted index from. If you have not chosen your index fields and analyzers yet, you can not proceed, as this step deals with creating the index configuration.

Step 3.3.1: Creating the Index Configuration

Before we can create our inverted index, we need to create an Index Configuration object. In the IndexConfig constructor, we will specify our id_col. The id_col should be the name of the id column in table *A*, the table you are indexing. Then, you can instantiate an Index Configuration object with the following:

```
config = IndexConfig(id_col="name_of_your_id_column")
```

Next, to add your chosen fields to your Index Configuration, you should use the add_field method:

```
config.add_field("field_name", analyzers)
```

where analyzers are either a string (like "3gram") or a list (like ["3gram", "standard"]). The list will apply each analyzer to the field. So, the following two methods are equivalent:

```
config.add_field("name", "3gram")
config.add_field("name", "standard")

AND

config.add_field("name", ["3gram", "standard"])
```

You should repeat this for each field you want to create an index on. Your analyzers do not have to be the same for each field. For example, you could choose to apply the "3gram" and "standard" tokenizer to "name", but only apply the "standard" tokenizer to "address". This would be written as:

```
config.add_field("name", ["3gram", "standard"])
config.add_field("address", "standard")
```

In general, if the field's values are relatively long strings, then you may want to consider using the 'standard' analyzer. If the values are relatively short or have misspellings, then you may consider using a '2gram' or '3gram' analyzer.

Step 3.3.2: Concatenating Fields

As noted in step 3.2, it may be useful to concatenate fields. To concatenate fields, we offer a helper method in the IndexConfig field called 'add_concat_field'. To add a concatenated field to your index config, use the following syntax:

```
config.add_concat_field("concat_column1_column2", ['column1', 'column2', analyzers]
```

If your IndexConfig object was called config and you wanted to create a concatenated field for 'fName' and 'lName', it would look like this:

config.add_concat_field("concat_fName_lName", ['fName', 'lName'], analyzers)

The first argument will be the name of your concatenated field. We recommend setting it to be something like ‘concat_field1_field2’ to make it clear what the concatenated field consists of, but you may choose any name you want. The second argument contains a list of fields that you want to concatenate. All of the fields in the list must exist in table *A*. In the third argument, just as in the non-concatenation case, the argument 'analyzers' can be either a string or a list.

You can concatenate more than 2 fields. In that case, the above three arguments should be modified accordingly. For example, the first argument can be "concat_field1_field2_field3". The second argument is a list of three column names from Table A, and so on.

Step 3.4: Creating the Index

Now that you have configured your index, we can proceed with creating the index for table *A*. We will need to build the index on our local filesystem, and we will add the location where we want it built as an argument to our index object constructor.

Additionally, there is an optional argument in the constructor called delete_if_exists. This is default True. If you have already created an index at this location and delete_if_exists is False, it will not clear the old index, so you will be using the index that already exists at that location. If you had already created an index at that location and did not want to specify a new one, this is okay.

To create the index object, you can use the LuceneIndex constructor as follows:
index = LuceneIndex(index_path=“{path_to_index}”, config=index_config_from_above, delete_if_exists={True or False})

If we wanted to build our index at “/tmp/example_index/”, use the index configuration from earlier (config), and delete an index if it already exists at “/tmp/example_index/”, our implementation would look like this:
index = LuceneIndex(index_path="/tmp/example_index/", config=config, delete_if_exists=True)

Finally, we can create the index on table *A* using the upsert_docs method. We will pass in the DataFrame with the table we want indexed, and three optional boolean arguments. The three optional arguments are disable_distributed, force_distributed, and show_progress_bar.

- If disable_distributed is set to True, the index is built on a single node. If it is set to False, Sparkly will decide whether it is built on a single node or across multiple nodes.
- Conversely, if force_distributed is set to True, the index will be built across multiple nodes. If it is set to False, Sparkly will decide whether it is built on a single node or across multiple nodes. You cannot set both disable_distributed and force_distributed to True since that would imply you want to disable a distributed build while also forcing a distributed build, which does not make sense.

- Finally, if show_progress_bar is set to True, there will be a progress bar in the command line showing how many documents have been indexed and how many total documents need to be indexed. If show_progress_bar is False, this progress bar will not appear.

If we wanted to create our index on table *A*, force it to be distributed among several nodes, and we wanted to track the progress, we would use the upsert_docs method as follows:
index.upsert_docs(df=table_a, force_distributed=True, show_progress_bar=True)

Step 3.5: Creating the Query Specification

After creating the index on table *A*, we need to establish the fields from table *B* that we will search the index on table *A* for. The query spec is where you define which attributes and analyzers you want to use for searching. The format is "table_*B*_column_name": {indexed_field_fromA.analyzer1, indexed_field_fromA.analyzer2} with whichever analyzers you choose to include. The analyzers used in the query spec can either be a subset or the full set of the analyzers used to create the index.

For example, if the index on table *A* consisted of the fields "name" and "address", and the "name" field had the "3gram" analyzer applied to it and the "address" field had the "standard" and "3gram" analyzers applied to it, your query spec could look like this:

```
query_spec = QuerySpec( {
      "name": {"name.3gram"},
       "address": {“address.standard”, “address.3gram”}
}
)
```

In the above query spec, the "name" and "address" fields are the fields from table *B*. Then, "name.3gram", "address.standard", and "address.3gram" are the fields and analyzers used to create the index for table *A*.

When we do "name": {"name.3gram"}, we are saying that we want to compare the "name" field in table *B* with the "3gram" representation of the "name" field from table *A*. However, doing a comparison between the "name" field of *B* with the "3gram" representation of the "name" field from table *A* would yield poor results since the "name" field from table *B* does not have any analyzers applied to it. Therefore, Sparkly will tokenize the field from table *B* with the analyzer that is being used on the indexed field. In this case, that would mean the "name" field of *B* will be tokenized using "3gram" and then will be compared to the tokenized "name" field in the index.

Then, for "address": {“address.standard”, “address.3gram”}, we treat this as equivalent to "address": {“address.standard”} and "address": {“address.3gram”}. So, the "address" field from table *B* will be tokenized with the "standard" analyzer to be compared to the "standard" tokenization of "address" in the index of *A*, and then the "address" field from table *B* will be tokenized with the "3gram" analyzer to be compared with the "3gram" tokenization of "address" in the index of *A*.

Now, let’s take a look at another query spec for two new tables. Let’s say that tables *A* and *B* each have the fields “_id”, “name”, and “definition”. In our index configuration, we created an index with the “standard” tokenizer on “name”, the “3gram” tokenizer on “definition”, and the “standard” tokenizer on “definition”. However, we might decide we only want to make comparisons using the “standard” tokenizer on “definition” and ignore the “3gram” tokenization from our index. Then, our query spec would look like this:

```
query_spec = QuerySpec( {
      "name": {"name.standard"},
       "definition": {'definition.standard'}
}
)
```

This example demonstrates that we do not need to use all of the field + analyzer combinations we defined in our index configuration.

Next, we will demonstrate how you can use concatenated fields in your query spec. For this example, let’s assume the same setup as the last example: we have tables *A* and *B* which each have the fields “_id”, “name”, and “definition”. In our index configuration, we created an index with the “standard” tokenizer on “name”, the “3gram” tokenizer on “definition”, and the “standard” tokenizer on “definition”. Additionally, assume we created a concatenated field of name and definition in our index, “concat_name_definition” and tokenized it with the “standard” tokenizer and with the “3gram” tokenizer. Then, you may choose to create the following query spec:

```
query_spec = QuerySpec( {
      "name": {"name.standard"},
       "definition": {'definition.standard'},
       "concat_name_definition": {'concat_name_definition.standard',
'concat_name_definition.3gram'}
}
)
```

In the above example, we chose to only use the standard analyzer for the ‘name’ and ‘definition’ attributes, while we chose to keep both the ‘standard’ and ‘3gram’ analyzers for ‘concat_name_definition’. You may add whichever analyzers to each field, as long as the field + analyzer combination have been included in your index configuration for Table A.

Considering again the above example, note that the “concat_name_definition” field might not exist as a field in table *B*, and that is okay. Please refer to the following note on the limitations/expectations of using a concatenated field in your query spec.

Note: If you choose to use a concatenated field, you must use a mapping in one of the following ways:

1. "concat_field1_field2": {'concat_field1_field2.standard', 'concat_field1_field2.3gram'} if "field1" and "field2" exist in table *B* (the above example shows this)
2. "single_field": {'concat_field1_field2.standard', 'concat_field1_field2.3gram'}, where "single_field" refers to the name of an existing field in table *B*; in our example it could look like this:
   "definition": {'concat_name_definition.standard', 'concat_name_definition.3gram'}
3. "concat_field3_field4": {'concat_field1_field2.standard', 'concat_field1_field2.3gram'} if "concat_field3_field4" already exists in table *B* as a single field (this is really just Case 2 discussed above).

This, however, would be invalid, and would not generate the field on the fly:

1. "concat_field3_field4": {'concat_field1_field2.standard', 'concat_field1_field2.3gram'} where "field3" and "field4" exist in table *B*, but "concat_field3_field4" does not exist in table *B*.

Option 2: Sparkly Auto

If you do not want to build the index or query spec manually, we offer Sparkly Auto. Sparkly Auto creates the index with standard and 3gram analyzers for each field in your indexing table as well as one concatenated field, which concatenates all of the fields in your indexing table.

Then, the query spec is generated with combinations of the index fields, analyzers, and your columns from table *B*. Sparkly Auto searches the space of all possible query specs to find a query spec judged to be optimal for blocking purposes.

Step 3.1: Creating the Index

To use Sparkly Auto, you should use the IndexOptimizer constructor. The IndexOptimizer constructor has several options that are included in the API. However, the only required argument is called is_dedupe, which should be True if the indexed table and the search table are the same. Otherwise, it should be False.

So, if the indexed table and search table are the same, the constructor should be:
index_optimizer = IndexOptimizer(True)
And if the indexed table and search table are different, the constructor should be:
index_optimizer = IndexOptimizer(False)

Next, to create our index, we can call the make_index_config method with our DataFrame and the id column of our DataFrame. In this step, the DataFrame is the DataFrame which you want to build an index on (the guide has been referencing this DataFrame as table *A*)
config = index_optimizer.make_index_config(table_a, id_col="name_of_your_id_column")

Now we can proceed with creating the inverted index for table *A*. We will need to build the index on our local filesystem, and we will add the location where we want it built as an argument to our index object constructor.

Additionally, there is an optional argument in the constructor called delete_if_exists. This is default True. If you have already created an index at this location and delete_if_exists is False, it will not clear the old index, so you will be using the index that already exists at that location. If you had already created an index at that location and did not want to specify a new one, this is okay.

To create the index object, you can use the constructor as follows:
```
index = LuceneIndex(index_path=“{path_to_index}”, config=index_config_from_above,
delete_if_exists={True or False})
```

For example, if we wanted to build our index at “/tmp/example_index/”, use the index configuration from earlier (config), and delete an index if it already exists at “/tmp/example_index/”, our implementation would look like this:
```
index = LuceneIndex(index_path="/tmp/example_index/", config=config, delete_if_exists=True)
```

Finally, we can create the index on table *A* using the upsert_docs method. We will pass in the DataFrame with the table we want indexed, and three optional boolean arguments: disable_distributed, force_distributed, and show_progress_bar.

- If disable_distributed is set to True, the index is built on a single node. If it is set to False, Sparkly will decide whether it is built on a single node or across multiple nodes.
- Conversely, if force_distributed is set to True, the index will be built across multiple nodes. If it is set to False, Sparkly will decide whether it is built on a single node or across multiple nodes. You cannot set both disable_distributed and force_distributed to True since that would imply you want to disable a distributed build while also forcing a distributed build, which does not make sense.
- Lastly, if show_progress_bar is set to True, there will be a progress bar in the command line showing how many documents have been indexed and how many total documents need to be indexed. If show_progress_bar is False, this progress bar will not appear.

If we wanted to create our index on table *A*, force it to be distributed among several nodes, and we wanted to track the progress, we would use the upsert_docs method as follows:
```
index.upsert_docs(df=table_a, force_distributed=True, show_progress_bar=True)
```

Step 3.2: Creating the Query Specification

After creating the index, we can create the query spec by calling the 'optimize' function with our index and search table. If our index is held in the “index” variable, and our search table is called table_b, we can find the optimal query spec using the following command:
```
query_spec = index_optimizer.optimize(index=index, table=table_b)
```

If you are curious about the fields the optimizer chose, you can display them with:
```
print(f"spec: {query_spec.to_dict()}")
```

(Optional Step) Boost Map Usage

After establishing the fields and analyzers you are using for blocking from either Sparkly Manual or Sparkly Auto, you may decide to re-weigh the fields. By default, Lucene calculates the score for a document by adding together the scores of each field. So all fields have equal weights.

However, you may know that one of your fields should have more of an impact in the scoring than another. For example, if you have a “definition” field and a “name” field, the entries in your “definition” field could be very similar while the entries in the “name” field are very different. In this case, you may decide that the “name” field deserves more weight than the “definition” field. To accomplish this, you can use a boost map. The boost map can be added as:

```
query_spec.boost_map = {
("name", "name.standard"): 0.5,
("definition", "definition.standard"): 0.25
("'concat_name_definition", "'concat_name_definition.standard"): 0.25,
("'concat_name_definition", "'concat_name_definition.3gram"): 0.25,
}
```

In this example, we are using a similar syntax to what we used in our query spec where we take a field from table *B*, an indexed field and analyzer from table *A*, but now we are adding a weight. In this example, we are comparing the “name” field from *B* with the “standard” tokenization of the “name” field from *A*, and we are giving this comparison a weight of 0.5. Then, we are comparing the “definition” field from *B* with the “standard” tokenization of the “definition” field from *A*, and we are giving this comparison a weight of 0.25. Next, we are comparing the concatenated field with “name” and “definition” from *B* with the “standard” tokenization of the concatenated field with “name” and “definition” from *A*, and we are giving this comparison a weight of 0.25. Lastly, we are comparing the concatenated field with “name” and “definition” from *B* with the “3gram” tokenization of the concatenated field with “name” and “definition” from *A*, and we are giving this comparison a weight of 0.25.

Note: the weights do not need to add to 1.

To summarize, each field can be added as:
(“table_B_field_name”, “indexed_field_from_A.analyzer”): weight

where weight is a float.

Step 4: Performing Blocking

After you have created the inverted index and built your query spec (including optionally adding a boost map), you can block by searching for candidates. That is, for each record in Table B, you will search the index to find k records in Table A with the highest similarity scores (where k is a pre-specified limit).

You do this by creating a Searcher object. To create a Searcher, you need to create a Search object and pass in your index like this:
searcher = Searcher(index)

Next, to search over table *B*, you need to pass in table *B*, your query spec, the id column for table *B*, and value k, that is, the maximum number of candidates you want for each record in table *B* (aka the limit). Recall that the goal for blocking is to reduce the search space for the matching problem. Sparkly does this by using top-k, where it will return the top-k records from table *A* for each record in table *B*, ranked by score in descending order.

For example, if you had 1000 records in table *B* and 1000 records in table *A*, matching without blocking would result in a search space of 1000*1000=1,000,000 checks. As demonstrated in the "How Sparkly Works" section, it is important to choose a good value k. Setting k too high reduces the benefits of blocking, and setting it too low can lead to low recall. You may start with trying to reduce the search space to 10% of the Cartesian product between A and B. So, if the indexed table (table *A*) initially has 100 records, you would want to set the limit to 10.

As an example, if the id column in Table B is '_id' and we wanted a maximum of 10 candidates for each record in table *B*, you would use the following:
candidates = searcher.search(table_b, query_spec, id_col="_id", limit=10)

The blocking output in 'candidates' will have the following columns:
id2, id1_list, scores, search_time

id2 is the id of the record in table *B*. id1_list is a list with its maximum length equal to your limit, and contains the ids of records in table *A*. scores is a list of maximum length equal to your limit and contains the scores of the candidate records in table *A*. search_time is the time it took to complete the search for candidates for the record in table *B* over the index from table *A*.

Step 5: Saving Blocking Output

Now that you have performed blocking, you will likely want to save your blocking output. 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.

In the case that you want to save it as 'out.parquet' in the same directory as your Sparkly Python Script, you can run:
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 Sparkly and how to use it on your datasets. If you are unsure about how the methods work, what arguments they can take in, or other functionality not covered here, please head over to the API docs. If you need further support or have any questions, please visit anhaidgroup.github.io/magellan, or email us at entitymatchinginfo@gmail.com.

Appendix

Analyzers Table

The analyzers below are current as of February 10, 2026. These analyzers are supported by sparkly-em:0.1.0 and Lucene 9.12.0. If the version of Sparkly or Lucene differs from this, the analyzers below may work differently or be deprecated entirely.

Analyzer Name	**What it does**	**Example: "Hello world. How are you today?"**
standard	Tokenizes + lowercases the field, creates tokens by using whitespace and punctuation as delimiters. It will not contain tokens with punctuation.	{ 'hello', 'world', 'how', 'are', 'you', 'today' }
3gram	Breaks the string into words of size 3, filters to alphanumeric tokens only, then lowercases the field	{ 'hel', 'ell', 'llo', 'wor', 'orl', 'rld', 'how', 'are', 'you', 'tod', 'oda', 'day' }
4gram	Breaks the string into words of size 4, filters to alphanumeric tokens only, then lowercases the field	{ 'hell', 'ello', 'worl', 'orld', 'toda', 'oday' }
2gram	Breaks the string into words of size 2, filters to	{ 'he', 'el', 'll', 'lo', 'wo', 'or', 'rl', 'ld', 'ho', 'ow', 'ar', 're', 'yo',

	alphanumeric tokens only, then lowercases the field	'ou', 'to', 'od', 'da', 'ay' }
shingle	Applies the standard analyzer, and then returns combinations of 1, 2, 3 tokens	{ 'hello', 'world', 'how', 'are', 'you', 'today', 'hello world', 'world how', 'how are', 'are you', 'you today', 'hello world how', 'world how are', 'how are you', 'are you today' }
stripped_3gram	3gram, but it filters out the punctuation within the tokens (ie punctuation is removed, but is not a delimiter)	{ 'hel', 'ell', 'llo', 'low', 'owo', 'wor', 'orl', 'rld', 'ldh', 'dho', 'how', 'owa', 'war', 'are', 'rey', 'eyo', 'you', 'out', 'uto', 'tod', 'oda', 'day' }
unfiltered_3gram	3gram, but there is no filtering based on punctuation. The tokens can contain punctuation.	{ 'hel', 'ell', 'llo', 'lo ', 'o w', ' wo', 'wor', 'orl', 'rld', 'ld.', 'd. ', '. h', ' ho', 'how', 'ow ', 'w a', ' ar', 'are', 're ', 'e y', ' yo', 'you', 'ou ', 'u t', ' to', 'tod', 'oda', 'day', 'ay?' }
standard36edgegram	Uses a standard tokenizer, filters it to lowercase, keeps only alphanumeric tokens. Then, it creates tokens of size 3-6, but it does not have a sliding approach like other analyzers. See example.	{ 'hel', 'hell', 'hello', 'wor', 'worl', 'world', 'how', 'are', 'you', 'tod', 'toda', 'today' }
unfiltered_5gram	5gram, but there is no filtering based on punctuation. The tokens can contain punctuation.	{ 'hello', 'ello ', 'llo w', 'lo wo', 'o wor', ' worl', 'world', 'orld.', 'rld. ', 'ld. h', 'd. ho', '. how', ' how ', 'how a', 'ow ar', 'w are', ' are ', 'are y', 're yo', 'e you', ' you ', 'you t', 'ou to', 'u tod', ' toda', 'today', 'oday?' }