ISBN 978 -987-633-574-4

TreeSpark: A Distributed Tool for Progeny Analysis based on Spark

Paula López [1], Waldo Hasperué [1,2], Facundo Quiroga [1] and Franco Ronchetti [1,3]

[1] Instituto de Investigación en Informática LIDI. Facultad de Informática. Universidad Nacional de La Plata
[2] Investigador asociado - Comisión de Investigaciones Científicas (CIC-PBA)
[3] Investigador asistente - Comisión de Investigaciones Científicas (CIC-PBA)
{pdlopez,whasperue,fquiroga,fronchetti}@lidi.info.unlp.edu.ar

Abstract. Progeny analyses are useful in biological sciences for various purposes, such as improving individuals in new generations or carrying out molecular analysis of the transmission of genetic characteristics. Analyzing these data by making comparisons between individuals of a generation with their offspring is not a trivial task, and increases in complexity as more and more generations are incorporated. In this article, we present TreeSpark, an open source tool to carry out progeny analysis and provides functionality that allows simple access to the information of the individuals and their relations both as progenitors and descendants. This tool is developed as a Python module, which in turn inherits the distributed processing features of Spark, allowing it to process large volumes of progeny information. TreeSpark is compared with other similar tools, finding TreeSpark much simpler to use.

Keywords: Spark, big data, progeny analysis, genealogy, analytics.

1 Introduction

Various biological sciences carry out progeny analyzes looking for different objectives with the goal of comparing some characteristic of an individual with that of their offspring. For example, when analyzing and monitoring cattle, studies are aimed at establishing the magnitude of the improvement in milk production in the new generations, and thus be able to estimate its possible association with reproductive indicators in the offspring [1].

Progeny analyses are carried out both in animal [2][3] and plant species [4][5][6]. There are different works that range from the manual selection of breed individuals aimed at producing better individuals in the new generations based on a given characteristic of interest [7], to the molecular analysis of the transmission of genetic properties [8]. To carry out these analyses, a database prepared for this purpose is required. Above all, this database should have kinship relationship information between two individuals (descendant-parent). Crossing the information of an individual with that of its progeny or its progenitors quickly becomes complicated if many generations are included in the analysis. Researchers usually lack the required expertise in programming to use scripts developed for this types of tasks.

ISBN 978 -987-633-574-4

In this article, we present TreeSpark, an open source tool that facilitates progeny analysis by introducing a working mechanism based on Spark. TreeSpark is a Python module that includes, as part of its API, a set of variables and functions that facilitate access to information on the progeny or progenitors of any individual analyzed. TreeSpark is developed on the Spark framework, so it can be used both on individual computers as well as in distributed environments.

The present work is organized as follows: In Section 2, the problem of accessing the progeny information of an individual is described in detail. Some current tools that allow progeny analysis are detailed in Section 3. In Section 4, the TreeSpark tool is described. In Section 5, TreeSpark is compared with other state-of-the-art developments, analyzing the code that each of these requires solving various problems. Finally, conclusions are presented in Section 6.

2 Progeny Tree

In progeny analyses, the evolution of an individual with respect to its parents and progeny is studied. In other words, the focus of interest is analyzing the evolution of a branch (or a tree) of genealogical descent. To carry out these analyses, kinship relationship information between individuals is needed. All individuals that are part of the database must have information about their progenitors. In studies where it is only important to know a single progenitor (mother or father in the case of sexual species), then the set of individuals make up what is known as a progeny tree.

In this work, individuals with no parent information are called "root individuals". "Leaf" individuals are those that do not have offspring and, following this same logic, an individual is said to be "parent" of another individual, called "child". Figure 1 shows an example of a progeny database and the corresponding progeny trees of the two "root" individuals in the database.

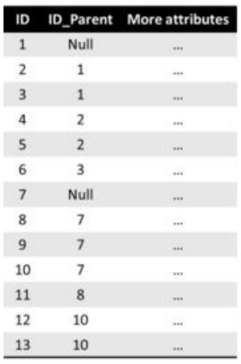
2.1 Building Progeny Trees

To carry out a progeny analysis for different generations of the same family from a dataset like the one shown in Figure 1, the following steps must be completed: 1) obtaining the root individuals; 2) obtaining the children from the root individuals; 3) obtaining the children of the individuals identified in the previous step; continuing recursively with this procedure until all the leaf individuals of each family are included.

In a database where progeny information is stored in a table like the one shown in Figure 1, which has the columns ID and ID_Parent, root individuals are obtained through a filter operation (SELECT), while for each generation that is to be included in the tree, a JOIN operation between the filtered result and the table of individuals must be performed. This operation is then repeated as many times as necessary until the entire family tree is formed. This way of working is inherent to the data that make up a tree structure. The following pseudo-SQL script allows obtaining the number of individuals in each family of the dataset.

```
Gen1 = SELECT ID FROM Table WHERE ID_Parent = Null
Gen2 = SELECT ID, Table.ID_Parent AS Family FROM Table
          INNER JOIN Gen1 ON Tabla.ID_Parent = Gen1.ID
Gen3 = SELECT ID, Family FROM Table INNER JOIN Gen2
          ON Table.ID_Parent = Gen2.ID
Res = SELECT Family, Count(Family) FROM Gen3
          GROUP BY Family
```

This script only allows working with three-generation trees like the ones shown in Figure 1. Developing a generic script that allows the treatment of *N* generations requires more sophisticated code, since a control structure of the WHILE type that evaluates some condition that detects if all individuals have been processed (allowing it to end the loop) is required.



ID	ID_Parent	More attributes
1	Null	...
2	1	...
3	1	...
4	2	...
5	2	...
6	3	...
7	Null	...
8	7	...
9	7	...
10	7	...
11	8	...
12	10	...
13	10	...

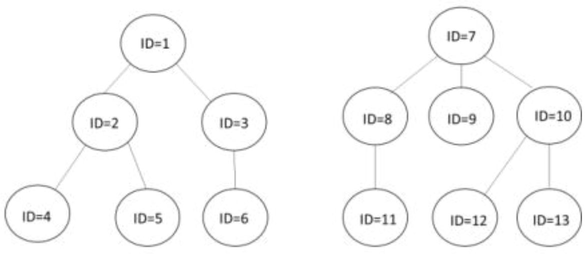


Fig. 1. On the left, a table with progeny information where the ID column (corresponding to the identifier of the individual) and the ID_Parent column (corresponding to the identifier of the parent individual) are highlighted. On the right, the graphic representation of the 13 individuals in the table, forming two independent trees or families.

3 Progeny Analysis Tools

Currently, there are several tools that allow analyzing progeny database. Some have the disadvantage of having a paid license, while others require a special pre-treatment of the data, and yet some others are outdated.

GraphFrames [9] and GraphLab [10] are two frameworks for treating graphs. GraphFrame is an integrated system that combines graphing algorithms, pattern matching, and relational queries. It is implemented on Spark SQL, it allows running processes in parallel, and it is compatible with the Spark dataframe API. To use it, data must be split into two tables: one with vertices (individuals) and another one with edges (kinship relationships).

On the other hand, GraphLab is a framework for machine learning written in C++ that has libraries for data transformation and manipulation, as well as model visualization. One of its various functionalities is to create graphs, with requirements similar to those of GraphFrames.

ISBN 978 -987-633-574-4

Both tools allow data to be loaded from various sources (JSON, CSV, etc.), but they only work with graphs in a generic way, i.e., they do not deal specifically with tree-shaped structures. Even though it is true that a tree is a particular type of directed graph, the functions that these tools provide, being graph-based, make it difficult to treat a tree-shaped graph. In order to work with these tools, at least two relationships (sets of edges) between individuals must be specified – parent → child and child → parent. If sibling information is also to be considered, then a third relationship has to be added between these individuals.

A previous version of the tool presented in this paper, [11] published a tool that allows using tree information to analyze progeny. This tool allows establishing ancestry and descent relationships between individuals, generating the progeny tree, providing functions to process their information. Even though the tool allows assembling and processing progeny trees, it does not support distributed execution.

Among commercial tools, ChromoSoft[1] and Breeders Assistant [2] stand out. Both work only with animal information and are designed especially for breeders. Even though they allow analyzing ancestors and descendants, in addition to calculating genetic or consanguinity coefficients, these tools support limited data formats, are tied to the payment of an annual membership, and cannot be run in distributed environments.

Finally, PedHunter [3] (which focuses on processing people information in large genealogies), PEDSYS [12] (which is designed to analyze individuals of any species), ENDOG [13] (which only focuses on analyzing information from animals), and InterHerd [4] (which is intended for dairy and meet cattle producers that wish to carry out progeny analyses, in addition to monitoring production, among other functionalities) are all tools that are currently outdated or obsolete.

4 TreeSpark

In this section, we introduce TreeSpark[5], an open source tool that allows progeny database analysis in Python using a simple and friendly syntax, as it provides variables and functions for this purpose.

The use of TreeSpark consists of, as a first stage, creating the progeny tree from a database and then, using several filtering operations "pruning" the family trees in the dataset based on the analysis to be carried out. For example, keeping individuals with more than three children, individuals that are "only children", individuals with a certain number of siblings, and so on, in addition to being able to use the data from the dataset as filters (days of longevity, milk production, number of eggs laid, etc.). Also,

[1] www.chromosoft.com/en

[2] www.tenset.co.uk/ba/

[3] www.ncbi.nlm.nih.gov/CBBresearch/Schaffer/pedhunter.html

[4] https://www.compuagro.net/interherd.htm

[5] https://gitlab.com/Danikawaii/treespark

TreeSpark allows obtaining dataframes for later analysis, made up of individuals with a given kinship relationship, such as all parents and their children, all siblings, etc.

4.1 Creating Family Trees

TreeSpark works with Spark dataframes; therefore, the first step is to create a dataframe by retrieving the data from any source supported by Spark. This dataframe must have at least two columns, as shown in the example in Figure 1: one with the identifier of the individual (ID) and the other with the identifier of the individual's parent (ID_parent). Optionally, the database can have a field that has the birth order of an individual. This must be a number that is interpreted as follows: 1 represents the first child, 2 represents the second child, and so forth. Having this information allows querying individuals sequences when they have a sibling relationship.

Once the dataframe is created, family trees must be assembled. This is done by creating the *TreeContext* object:

```
tc = TreeContext(DataFrame, "ID", "ID_PARENT", "ORDER")
```

where *DataFrame* is the dataframe with the progeny database retrieved from some data source, "ID" is the name of the column in the dataframe that stores the identification of the individuals, and "ID_PARENT" is the identifier of the parent individual. "ORDER", which is an optional parameter, is the name of the column that stores individual's birth order information. The object that represents all family trees is stored in the variable *tc*.

4.2 Filtering Family Trees

Once family trees are created, they can be "pruned" so that only those individuals that are of interest for a given analysis are retained (for example, individuals with more than four children or those that were born third), and use only the data corresponding them.

To carry out this task, TreeSpark provides a filter function called *filter* that allows “pruning” the trees. It is used as follows:

```
pruning1 = tc.filter(functionFilter1)
```

where *tc* is the *TreeContext* and *functionFilter1* is a function that will be evaluated for each of the individuals found in *tc*. *functionFilter1* is a function that takes all the information associated with an individual and returns a Boolean value. TreeSpark's *filter* function works similarly to Spark's *filter* function [14].

To simplify the code for the filters, TreeSpark incorporates special variables that refer to progeny information (Table 1) and are used with dot notation, as shown in the following examples.

```
fil1 = tc.filter(lambda ind: ind.childrenCount <= 3)
fil2 = tc.filter(lambda ind: ind.parentExists)
fil3 = tc.filter(lambda ind: ind.parent.parentExists)
```

ISBN 978 -987-633-574-4

```
    fil4 = tc.filter(lambda ind: ind.childrenOrder == 1)
    fil5 = tc.filter(lambda ind: ind.siblingsCount > 4)
```

In these filter functions, all the attributes found in the database can be accessed, as shown in the following example that selects all the individuals born in the year 2003:

```
   pruning = tc.filter(lambda ind: ind["Year"] == 2003)
```

Pruning Results from Previous Prunings. The result of the filter function is an object that represents all the individuals that met the filter condition and therefore can be used to apply a new filter:

```
pruning2 = pruning1.filter(functionFilter2)
```

where *pruning1* is the result of a *filter* function and *functionFilter2* is another function with the characteristics mentioned above. Thus, the results from a previous "pruning" operations can be "pruned" again, and different pruning "paths" can be built as needed (Figure 2).

4.3 Lazy Evaluation

Since TreeSpark is developed using the RDDs API and Spark DataFrames, the evaluation of all the defined filters is not performed until some action is executed. Filters are not applied when the *filter* function is invoked, but the Spark RDD dependency graph (RDDs lineage) is generated internally. The graph is executed at the time of invoking any action [14]. The only action available in TreeSpark is *collect*, which retrieves all the information of the individuals that resulted from the filters applied. It is used as follows:

```
result = pruning.collect()
```

where *pruning* is the result of any previous *filter* function or the entire *TreeContext* itself. The result returned by *collect* is a Spark dataframe, or None if there are no results. The resulting dataframe will have one row for each individual that met all filter conditions, and it will also include the same columns as the original dataframe.

Table 1. Special variables that can be used in *filter* functions.

Variables	Typo of return	Description
parent	individual	Reference to the parent individual.
childrenCount	int	Number of children
siblingCount	int	Number of siblings
grandchildrenCount	int	Number of grandchildren
parentExists	bool	True if the individual has a parent individual
childrenOrder	int	Number representing sibling order
hasPrevSibling	bool	True if the individual is not the first child
hasNextSibling	bool	True if the individual is not the last child

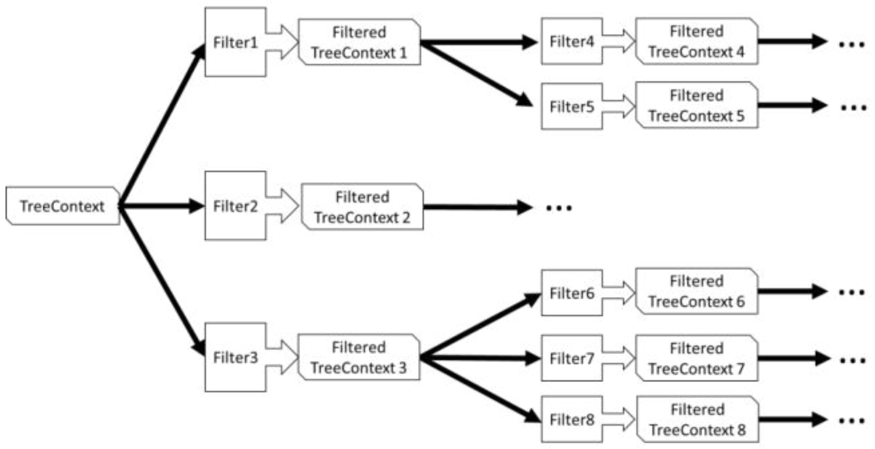


Fig. 2. Different filter "paths" stemming from a single TreeContext.

4.4 Obtaining Progeny Information

Using the results obtained after applying the filters, progeny relationships can be obtained for the resulting individuals. TreeSpark provides a set of functions that allow obtaining a collection with family relationships. These are three functions: *siblings, descendants* and *ascendants*.

The *siblings* function allows obtaining the relationships between an individual and its siblings. For example, if an individual *i,* obtained after applying a filter had three siblings h_1, h_2 and h_3 and they were born in the order h_1, i, h_2 and h_3, it would be possible to obtain the consecutive siblings in order of birth as follows:

```
result = pruning.siblings(2)
```

obtaining as a result the relations (h_1, i) and (i, h_2). The relationship (h_2, h_3) is not obtained as a result since, in this example, neither h_2 nor h_3 were the result of applying the filter. The relation (i, h_3) is not included either, because these are not consecutive siblings.

The value 2 used as a parameter in the *siblings* function indicates the number of individuals in the returned relationships. A value of 3 would return the relations (h_1, i, h_2) and (i, h_2, h_3). A value of 4 or greater would return the relationship (h_1, i, h_2, h_3).
Similarly to the *siblings* function, TreeSpark provides the *descendants* function to obtain the descendants of an individual. It is used as follows:

```
result = pruning.descendants(2)
```

where the value of the parameter indicates the number of generations to be obtained. A value of 1 would only get the individual itself. With a value of 2, the function returns all children relationships; a value of 3, returns all children and grandchildren relationships, and so forth.

ISBN 978 -987-633-574-4

Finally, the tool provides the *ascendants* function, which allows obtaining the ancestors of an individual. Its use is similar to that of the *descendants* function:

```
result = pruning.ascendants(2)
```

where the value of the parameter indicates the number of generations to be obtained. Given the nature of the ancestry relationship, a relationship is obtained for each requested generation: (individual, parent), (individual, parent, grandparent), (individual, parent, grandparent, great-grandparent), etc.

5 Tool Comparison

In this section, the simplicity of the code that has to be written in TreeSpark to solve a progeny problem is analyzed. This comparison is made between TreeSpark and GraphFrames, since the latter is also a Spark-based tool.

As regards data sources, TreeSpark only needs a dataframe that contains the ID and ID_PARENT fields. On the other hand, GraphFrames requires a dataframe with the data of the vertices (the individuals) and another dataframe with the information of all the edges (parent-child relationships). For example, to get all parent-child relationships from the database in TreeContext, in TreeSpark, the following code must be run:

```
result = tc.descendants(2)
```

while in GraphFrames, the following statement must be executed:

```
result = graph.find("(n1)-[e]->(n2)")
```

where "(n1)-[e]->(n2)" is an expression that returns all edges *e* originating at node *n*1 (parent) and reaching node *n*2 (child). This way of retrieving relations from a graph becomes more complicated if we look for more complex relations such as grandparent-child-grandchild. In TreeContext, only *descendants (3)* is required, while in GraphFrames, the following statement has to be executed:

```
result = graph.find("(n1)-[e1]->(n2);(n2)-[e2]->(n3)")
```

Another example is obtaining sibling relationships. In TreeSpark, if the birth-order field is available, then it is a matter of simply executing the sentence:

```
df = tc.siblings(2)
```

while in GraphFrames, the parent-child relationship between vertices is required, as mentioned in Section 2. However, after this, the resulting graph must be converted to a DataFrame and a search for siblings using the DataFrame API (that is, externally to GraphFrame), must be carried out. Part of the code will be as follows:

ISBN 978 -987-633-574-4

```
g1 = graph.filterEdges("relationship = parent_child")
      v_df = g1.vertices() ; e_df = g1.edges()
 all_df = v_vf.join(e_df, e_df(“dst”) === v_df(“id”))
```

Then, to obtain the siblings of an individual, a search in the *all_df* dataframe using the dataframe API must be carried out. Alternatively, to carry out this search the edges corresponding to the relationships between siblings should be added to the graph, as follows:

```
g1 = graph.filterEdges("relationship = siblings")
      motifs = g1.find(“(n1)-[e1]->(n2)”)
```

As it can be seen, the GraphFrames code in this second example is simpler, but it requires adding more edges to the graph with the relationships between siblings. In the case of performing a search with a higher value, in TreeSpark the sentence and its complexity will remain the same, while in GraphFrames the situation will be similar to that of the descendants calculation.

An important point to note is that, even though in GraphFrames the vertex filter is supported through the *filterVertices* function, its execution not only filters individuals but also all their relationships (it eliminates those edges whose vertices no longer exist in the subgraph). This behavior means that, when applying the filter, the relationship between a child and its parent is lost if the condition provided is not met. In TreeSpark, on the other hand, the reference to the individual parent does not disappear regardless of the number of filters applied to the tree, meaning that information can be queried at any filtering instance.

6 Conclusions

We presented TreeSpark, a tool that facilitates progeny analysis through the use of specific variables and functions provided by the tool itself. TreeSpark inherits the simplicity of Python, hiding the complexity of an iterative process of multiple JOINs, thus allowing any researcher with little knowledge of Python to take advantage of all its functionality by simply searching for progenies to carry out their analyses.

TreeSpark is implemented on the Spark framework, and inherits two very important functionalities from it: On the one hand, it can retrieve data from various sources, since data must be retrieved through a DataFrame in order to use TreeSpark. On the other, and most importantly, the filters and operations carried out with TreeSpark can be executed in a distributed manner using a cluster of computers. If the database volume is very large, then TreeSpark can be run in a distributed manner. This processing is transparent to the user, since all distributed execution is carried out internally by Spark. TreeSpark tests have been carried out on a single node with a database of hundreds of individuals. As future work, we plan to study the performance of this tool in a cluster of nodes, considering how data distribution affects task execution performance. More complex filters than those shown in the examples included in this article are also pending testing. Finally, it should be noted that TreeSpark is in development, meaning that it is still going through testing and debugging stages.

References

1. Rearte, R., LeBlanc, S. J., Corva, S. G., de la Sota, R. L., Lacau-Mengido, I. M., Giuliodori, M. J.: Effect of milk production on reproductive performance in dairy herds. Journal of Dairy Science 101(8), 7575–7584. doi: 10.3168/jds.2017-13796 (2018).
2. Lopera-Barrero, N. M., Vargas, L., Nardez-Sirol, R., Pereira-Ribeiro, R., Aparecido-Povh, J., Streit Jr, D. P., Cristina-Gomes, P.: Diversidad genética y contribución reproductiva de una progenie de *Brycon orbignyanus* en el sistema reproductivo seminatural, usando marcadores microsatélites. Agrociencia 44(2), 171-181 (2010)
3. Domínguez Viveros, J., Rodríguez Almeida, F. A., Núñez Domínguez, R., Ramírez Valverde, R., Ortega Gutierrez, J.A., Ruíz Flores, A.: Análisis del pedigrí y efectos de la consanguinidad en el comportamiento del ganado de lidia mexicano. Archivos de Zootecnia 59(225), 63-72 (2010)
4. Salomón, J. L, Castillo, J. G, Arzuaga, J. A, Torres, W, Caballero, A, Varela, M., Hernández Betancourt, V. M.: Análisis de la interacción progenie-ambiente con minitubérculos a partir de semilla sexual de papa (*Solanum tuberosum*, L.) en Cuba. Cultivos Tropicales 36(2), 83-89 (2015).
5. Kolvalsky, I. E., Solís Neffa, V. G.: Análisis de la progenie de individuos productores y no productores de gametos masculinos no reducidos de *Turnera sidoides* (Passifloraceae). Boletín de la Sociedad Argentina de Botánica 50(1), 23-33 (2015)
6. Gutiérrez Vázquez, B. N., Cornejo Oviedo, E. H., Zermeño González, A., Valencia Manzo, S., Mendoza Villarreal, R.: Conversión de un ensayo de progenies de *Pinus greggii* var. greggii a huerto semillero mediante eigen-análisis. Bosque (Valdivia) 31(1), 45-52 (2010)
7. Guitou, H. R, Monti, A., Sutz, G., Baluk, I.: Interpretación y uso correcto de las diferencias esperadas entre progenie (DEP´s) como herramienta de selección para la calidad de carne: Segunda parte. Revista Colombiana de Ciencias Pecuarias 20(3), 363-376 (2007)
8. Luévanos-Escareño, M. P., Reyes-Valdés, M. H., Villarreal-Quintanilla, J. Á., Rodríguez-Herrera, R.: Obtención de híbridos intergenéricos *Helianthus annuus* x *Tithonia rotundifolia* y su análisis morfológico y molecular. Acta botánica mexicana (90), 105-118 (2010)
9. Dave, A., Jindal, A., Li, L., Xin, R., Gonzalez, J., Zaharia, M.: GraphFrames: an integrated API for mixing graph and relational queries. 1-8. 10.1145/2960414.2960416 (2016).
10. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.: GraphLab: A New Framework for Parallel Machine Learning. Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence, UAI 2010 (2010).
11. López, P., Hasperué, W., Rearte, R., de la Sota, R. L.: Herramienta informática para el análisis de progenie. Innovación y Desarrollo Tecnológico y Social 2(1), 35-54 (2020).
12. Dyke B.: PEDSYS: a pedigree data management system user's manual. San Antonio: Texas Southwest Foundation for Biomedical Research, Population Genetics Laboratory Technical Report No. 2. 1999;368 (1999).
13. Gutiérrez, J., Goyache, F.: A note on ENDOG: A computer program for analysing pedigree information. Journal of animal breeding and genetics. 122. 172-6. 10.1111/j.1439-0388.2005.00512.x (2005).
14. Scott, J. A.: Getting started with Apache Spark. MapR Technologies, Inc., San Jose, CA (2015)