Graphical Querying of Model-Driven Spreadsheets [⋆]

Jácome Cunha[1,2], João Paulo Fernandes[1,3],
Rui Pereira[1], and João Saraiva[1]

[1] HASLab/INESC TEC & Universidade do Minho, Portugal
[2] CIICESI, ESTGF, Instituto Politécnico do Porto, Portugal
[3] RELEASE, Universidade da Beira Interior, Portugal
{jacome, jpaulo, ruipereira, jas}@di.uminho.pt

Abstract. This paper presents a graphical interface to query model-driven spreadsheets, based on experience with previous work and empirical studies in querying systems, to simplify query construction for typical end-users with little to no knowledge of SQL. We briefly show our previous text based model-driven querying system. Afterwards, we detail our graphical model-driven querying interface, explaining each part of the interface and showing an example. To validate our work, we executed an empirical study, comparing our graphical querying approach to an alternative querying tool, which produced positive results.

Keywords: Model-driven engineering, graphical querying, spreadsheets

1 Introduction

Spreadsheets are the most successful example of the end user programming approach to software development. Although invented to provide a simple, but powerful, graphical environment to express mathematical formulas, spreadsheet systems quickly evolved into powerful software environments able to manipulate complex and large amount of data. Indeed, spreadsheets are often used to perform operations usually associated to databases. Surprisingly enough, spreadsheet systems lack powerful techniques, researched and developed for decades, that make database systems so powerful to manipulate big data, namely the use of data normalization techniques [1] and the use of query languages to filter and transform data [2]. And even then, the construction of a textual query language is difficult for end-users.

The purpose of this paper is three-fold:

[⋆] This work is part funded by the ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-022701. The first author was funded by FCT: SFRH/BPD/73358/2010.

- Firstly, we introduce a single, but powerful, graphical model-driven query language: *Graphical-QuerySheet*. Like in databases, we use models to express the business logic of the data. As a consequence, we can express the query based on those models, rather than on large and complex data. We use well-known spreadsheet models, namely ClassSheets [3–5], where databases use the relational models. Our queries are then expressed on those models and not on the data, exactly as in databases. Querying data, using for example SQL [6], is not a simple task, even for professional programmers. To make querying data available to end users, we define a visual querying language. Moreover, we hid from end users the data (de)normalization tasks.
- Secondly, we present a complete graphical model-driven spreadsheet querying architecture and tool. We detail the graphical tool, and through an example, show how a user would build his/her query.
- Thirdly, we present an empirical study where we compare our graphical approach to spreadsheet querying with the language provided by Google on its spreadsheets. A group of end users was asked to perform a series of tasks using both query systems. We present the first results of such a study, showing that *Graphical-QuerySheet* increases productivity, is intuitive and human-friendly, and is easy to use for someone with little or no SQL knowledge.

This paper is organized as follows: Section 2 briefly presents Google's QUERY function, along with some of its disadvantages, and a textual model-driven querying system. Section 3 introduces our graphical model-driven spreadsheet query interface. Section 4 presents our empirical study. Finally, in Section 5 we conclude the paper, and mention some future work.

2 Querying Spreadsheets

Before presenting techniques to query spreadsheets, let us introduce a spreadsheet to be used as a running example throughout this paper.

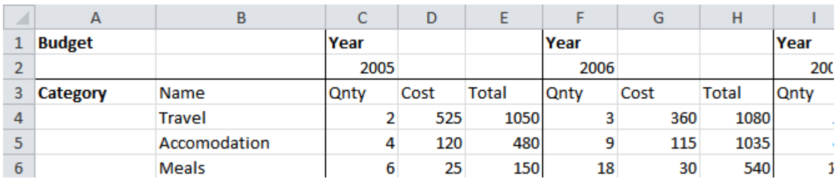
Figure 1 shows part of a spreadsheet to store information about the budget of a company. In this spreadsheet, we have information about the Category of the budget used (such as Travel or Accommodation) and the Year. The relationship between these two entities gives us information on the Quantity, Cost, and the Total Costs, per year per category, defined by spreadsheet formulas.

Although spreadsheet systems do not provide mechanisms to query the data, the fact is that we often need to answer simple questions like:

> **Question** *What was the total per year, ordered descendantly, from 2010 on wards?*

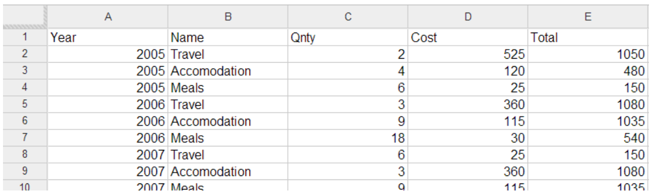
2.1 Google QUERY Function

Google provides a querying function, the Google QUERY function [7], which uses a SQL-like syntax [6], to perform a query over an array of values. An example would be the Google Docs spreadsheets, where the querying function is built in.



	A	B	C	D	E	F	G	H	I
1	**Budget**		**Year**			**Year**			**Year**
2			2005			2006			20(
3	**Category**	Name	Qnty	Cost	Total	Qnty	Cost	Total	Qnty
4		Travel	2	525	1050	3	360	1080	
5		Accomodation	4	120	480	9	115	1035	
6		Meals	6	25	150	18	30	540	1

Fig. 1: Part of the spreadsheet data for a Budget example



	A	B	C	D	E
1	Year	Name	Qnty	Cost	Total
2	2005	Travel	2	525	1050
3	2005	Accomodation	4	120	480
4	2005	Meals	6	25	150
5	2006	Travel	3	360	1080
6	2006	Accomodation	9	115	1035
7	2006	Meals	18	30	540
8	2007	Travel	6	25	150
9	2007	Accomodation	3	360	1080
10	2007	Meals	9	115	1035

Fig. 2: Budget data (denormalized)

This function needs two arguments, consisting of a range as its first argument, to state the range of data cells to be queried (for example A1:Q13), and the actual query string.

So if we wanted to answer our previous question, we would have to write in a cell, the formula expressed as the pre-defined *query* function, as displayed in Listing 1.1, after denormalizing the data from Figure 1 to Figure 2:

Listing 1.1: Google QUERY function for our example Question

```
=query(A1:58; "SELECT A, sum(E) WHERE A >= 2010
               GROUP BY A ORDER BY sum(E) DESC")
```

While being a powerful query function, it still has its flaws. To run this function, the user needs to represent his spreadsheet information in a single table. This means that someone who has their spreadsheet information divided as entities with relations, would need to first denormalize the data (as shown in Figure 2).

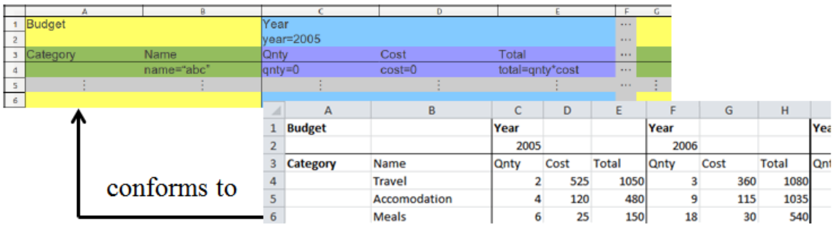
Afterwards, the user would need to write the query string, and here comes the second flaw: instead of writing the query using column names/labels, one must use the column letters. As one would expect, this can get confusing, counter-intuitive, and almost impossible to understand what the query is supposed to do, as shown in [8].

2.2 QuerySheet

We believe that querying spreadsheets should be simple and intuitive. This motivated us to design and implement a querying language simpler than Google's querying function, based on using some form of labels or descriptive tags to point to attributes and entities, as is in the database realm.

We turned to model-driven engineering [9, 10], a methodology in software development that uses and exploits domain models, or abstract representations of software. This has been successfully applied to spreadsheets, making model-driven spreadsheets [11, 12] and a model-driven spreadsheet environment (MD-Sheet) possible [13]. One of such spreadsheet models is ClassSheets [3, 4], a high-level and object-oriented formalism, using the notion of classes and attributes, to express business logic spreadsheet data. ClassSheets allow us to define the business logic of a spreadsheet in a concise and abstract manner, resulting in users being able to understand, evolve, and maintain complex spreadsheets.

To showcase ClassSheets, we present the ClassSheet model for our example spreadsheet from Figure 1. This ClassSheet model, named **Budget**, has a **Category** class (with a **Name** attribute) and a **Year** class (with a **Year** attribute) expanding vertically and horizontally, respectively. The joining of these gives us a **Quantity**, a **Cost**, and the **Total** of a **Category** in a given **Year**, each with its own default value. The corresponding spreadsheet instance conforms to the ClassSheet model as shown in Figure 3.



	A	B	C	D	E	F	G
1	Budget		Year			⋯	
2			year=2005			⋯	
3	Category	Name	Qnty	Cost	Total	⋯	
4		name=“abc”	qnty=0	cost=0	total=qnty*cost	⋯	
5	⋮	⋮	⋮	⋮	⋮	⋯	⋮
6							

	A	B	C	D	E	F	G	H	
1	Budget		Year			Year			Yea
2			2005			2006			
3	Category	Name	Qnty	Cost	Total	Qnty	Cost	Total	Qnt
4		Travel	2	525	1050	3	360	1080	
5		Accomodation	4	120	480	9	115	1035	
6		Meals	6	25	150	18	30	540	

Fig. 3: ClassSheet model and conforming instance for the Budget example

Having ClassSheet models available, we designed a textual querying language to write the queries based on those models [14], allowing descriptive and human-friendly query construction, in contrast to Google's approach. Moreover, we implemented a query framework, called *QuerySheet* [15, 16], that automatically denormalizes the data (as shown in Figure 2), translates it to a Google QUERY, and executes it in Google's engine. Thus, answering the previous question would be as simple as looking at the ClassSheet and writing the query:

Listing 1.2: QuerySheet query for our example Question

```
SELECT Year, sum(Total) WHERE Year >= 2010
GROUP BY Year ORDER BY sum(Total)
```

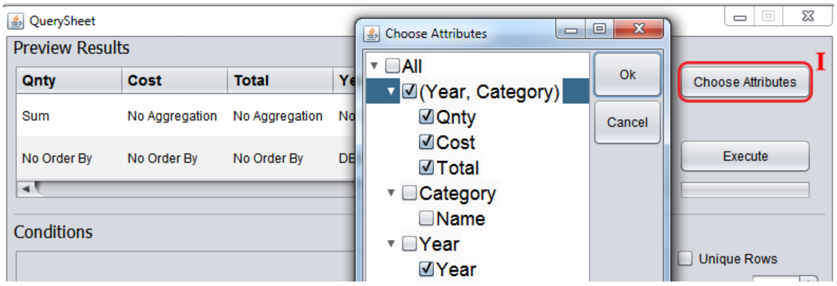
3 Graphical Model-Driven Spreadsheet Query Language

In order to validate the model-driven query language, we have performed an empirical study with real spreadsheet users [8], and realized that we can simplify querying spreadsheets even further, especially for end-users.

Indeed, while the participants in our study who were experienced in SQL had no problems, all others expressed frustration with writing SQL queries, due to having to remember the syntax, forgetting a group by clause after an aggregation, or even simple typos. This in turn motivated us to design a way to abstract the users from the textual query language, to a simple point and click query construction interface, where we could once again take advantage of spreadsheet models, and our previous experience with *QuerySheet*. What we designed was a simple, interactive visual language for querying model-driven spreadsheets, called *Graphical-QuerySheet*.

3.1 *Graphical-QuerySheet*

To try to shorten, or even eventually eliminate, the knowledge of SQL needed to correctly construct queries in our original system, we began building a graphical interface for *QuerySheet*. The focus of this interface was to be as simple as possible, displaying all the information in our query language, but in a human-friendly way. The interface also had to be intuitive to use, both for an experienced SQL user, and an end-user. We also wanted the interface to reduce the amount of errors (at least in the query syntax and attributes' names), and let the user choose the attributes based on the spreadsheet's model.



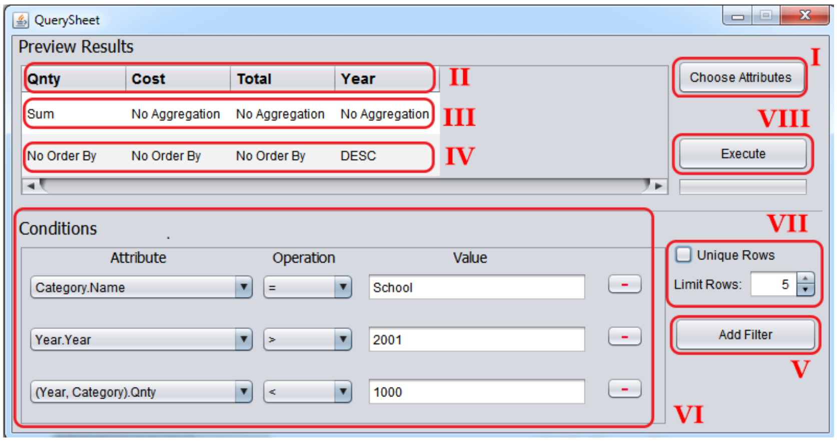
Qnty	Cost	Total	Ye
Sum	No Aggregation	No Aggregation	No
No Order By	No Order By	No Order By	DE

▾ □ All
▾ ☑ (Year, Category)
☑ Qnty
☑ Cost
☑ Total
▾ □ Category
□ Name
▾ □ Year
☑ Year

□ Unique Rows

Fig. 4: Attribute selection in the graphical interface

What resulted was an interactive graphical query building interface named *Graphical-QuerySheet* integrated into the MDSheet framework, and launched by a simple button. This interface allows a less experienced user to use a series of drop-down boxes to select his/her filter conditions, attribute orders, aggregations, and other querying conditions, to easily construct the queries, eliminating any possible syntax errors. The actual attribute selection (or Select clause) is presented by a tree-list based on a ClassSheet model (as shown in Figure 4), where the user may choose (by checking the corresponding check-boxes) all the attributes, all from a specific class, or individual ones. These chosen attributes are then displayed in a *Preview Results* panel, each attribute in its own column, showing the user how the result is to be returned, and allowing the user to drag the columns left or right to organize how he or she desires.



Qnty	Cost	Total	Year
Sum	No Aggregation	No Aggregation	No Aggregation
No Order By	No Order By	No Order By	DESC

Attribute	Operation	Value
Category.Name	=	School
Year.Year	>	2001
(Year, Category).Qnty	<	1000

☐ Unique Rows

Limit Rows: 5

Fig. 5: *Graphical-QuerySheet* (The boxes and numbers do not belong to the interface and are only shown for identifying the various areas)

In Figure 5 we see the various areas, identified by the red[4] boxes and Roman numerals. Each area is as follows:

I *Choose Attributes.* This button opens the Choose Attributes tree-list panel, where the user may check off which attributes to display.

II *Column Headers.* Display the chosen attribute names. Dragging this column, allows the user to rearrange the columns.

III *Aggregation.* This row displays which attributes have aggregations. Clicking on the cell displays a drop-down box with all the possible aggregations, or no aggregation.

IV *Order.* This row displays which attributes have an order clause. Clicking on the cell displays a drop-down box with the three possible options: `ASC`, `DESC`, or `No Order By`.

V *Add Filter.* This button adds a new row in the Conditions panel, to allow the user to add a new condition (or Where clause).

VI *Conditions Panel.* Displays all the conditions in the query. Each row is made up of two combo-boxes (allowing the user to respectively choose which attributes and operations to be used per condition), a text box to state the

[4] We assume colors are visible through the digital version of this document.

value of the condition, and a remove button (displayed as a red minus sign) to remove the condition from the panel.

VII *Unique/Limit.* A check-box to choose to display unique rows, and a scroll panel to state how many rows to display in the results, respectively. If the scroll panel value is 0, there is no limit.

VIII *Execute.* This button automatically translates the visual language to our model-driven query language, and executes the query, displaying the results in the user's spreadsheet. Below this button is a progress bar to give the user visual feedback of the process.

The interface also displays tool-tips when hovering over the buttons, check boxes, etc, to help the user understand the various parts. Along with the helpful tool tips, if one were to hover their mouse over the selected attribute headers, the attribute's class name is displayed (as shown in Figure 6). This is useful for when an attribute has the same name as another, while also reducing the amount visual information presented to the user all at once (for example showing another row to display the class names).

Another useful addition is the automatic calculation of when a group by is needed. In other words, when an aggregation is detected with other selected attributes, the visual language automatically produces a grouping. This automatic calculation not only is practical in query construction, but also made it so one less query clause needed to be presented in the graphical interface.

3.2 Building a Query in *Graphical-QuerySheet*

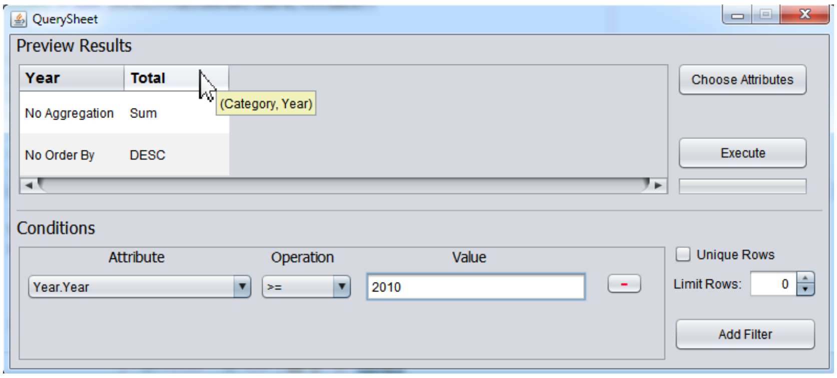
We will explain how one would construct the query from Listing 1.2 using *Graphical-QuerySheet*, as shown in Figure 6. The steps to construct this query are as follows:

1. Click on *Choose Attributes* and check *Year* and *Total*
2. Click on the *aggregation combo box* and choose *Sum*
3. Click on the *order by combo box* and choose *DESC*
4. Click on *Add filter*
5. Select the *Year.Year* attribute and *greater or equal to* operation using the combo boxes, and fill in *2010* in the *text box*
6. Click *Execute*

With this graphical interface guiding the user in his or her query construction, we are able to reduce a number of possible errors, and simplfy the user's experience. Using this interface, the user can have little to no SQL experience, and still perform queries.

3.3 Architecture

Since *Graphical-QuerySheet* builds upon *QuerySheet*, the only necessary extra work to build the former was to translate the visual query language that we introduce in this paper to our textual model-driven query language. The remaining steps of the process remain the same.



Year	Total
No Aggregation	Sum
No Order By	DESC

(Category, Year)

Attribute	Operation	Value
Year.Year	>=	2010

□ Unique Rows

Limit Rows: 0

Fig. 6: *Graphical-QuerySheet*

When the user clicks on the Execute button, the visual language is translated, and the data is denormalized. Using our previous model-driven query techniques, we produce the appropriate Google QUERY function string, with the corresponding data, and send both to Google to be executed by its query engine.

The results are then passed through our model inference technique, generating a ClassSheet-driven spreadsheet, with the resulting model and instance. In fact, two new worksheets are added to the original spreadsheet: one containing the spreadsheet data that results from the query, and the other containing the ClassSheet model.

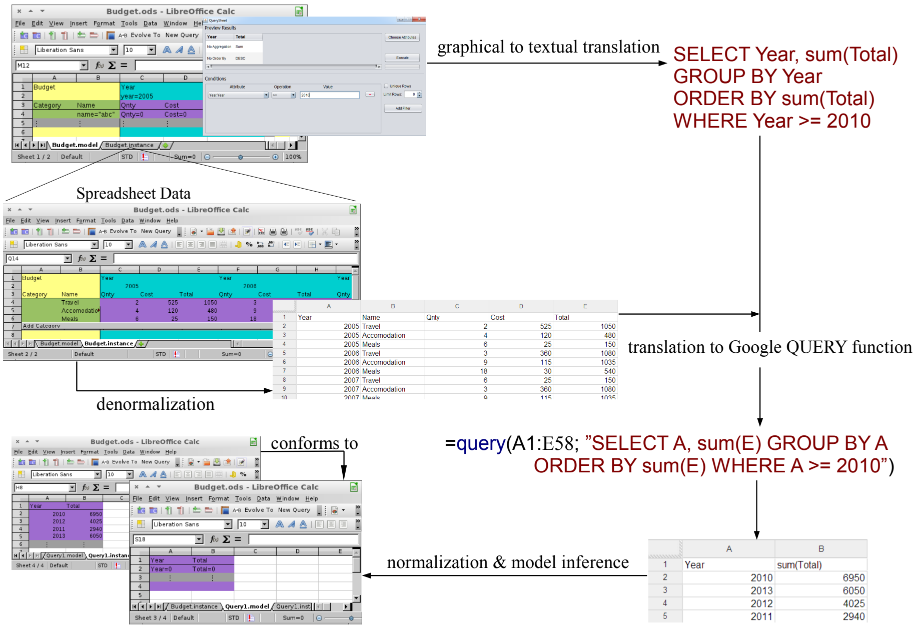
A complete illustration of the architecture that we have devised for *Graphical-QuerySheet* is shown in Figure 7. Indeed, we sketch how our tool produces the result of executing the query in Listing 1.2 on the spreadsheet of Figure 3.

An important aspect to note about our approach is that the result of executing a query is not only the data that it asks for, but also the ClassSheet model that such data conforms to, which is automatically inferred using a technique from our previous work [5]. This means that this result can be further queried in a model-driven fashion.

4 Empirical Study

In order to assess the use of *Graphical-QuerySheet* in practice, we planned and executed an empirical study with end-users. With this study, we wanted to obtain concrete feedback on our query system and to assess the productivity associated with its use.

The study was done one participant at a time, in a think-aloud session. By doing this, we were able to see each participant using our system and learn the difficulties they were having, and how to improve the system to overcome them.


graphical to textual translation
Spreadsheet Data
denormalization
translation to Google QUERY function
conforms to
normalization & model inference

```
SELECT Year, sum(Total)
GROUP BY Year
ORDER BY sum(Total)
WHERE Year >= 2010
```

```
=query(A1:E58; "SELECT A, sum(E) GROUP BY A
    ORDER BY sum(E) WHERE A >= 2010")
```

	A	B	C	D	E
1	Year	Name	Qnty	Cost	Total
2	2005	Travel	2	525	1050
3	2005	Accomodation	4	120	480
4	2005	Meals	6	25	150
5	2006	Travel	3	360	1080
6	2006	Accomodation	9	115	1035
7	2006	Meals	18	30	540
8	2007	Travel	6	25	150
9	2007	Accomodation	3	360	1080
10	2007	Meals	9	115	1035

	A	B
1	Year	sum(Total)
2	2010	6950
3	2013	6050
4	2012	4025
5	2011	2940

Fig. 7: The architecture of *Graphical-QuerySheet*

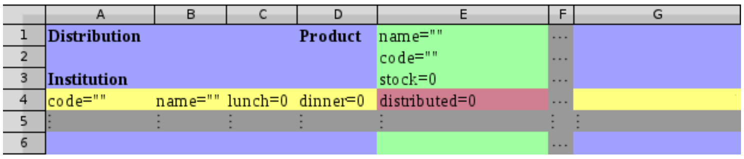
We ran this study with seven students, ranging from Bachelor to PhD students. Before running the actual study, we prepared a tutorial to teach the students how to use Google's QUERY function and the *Graphical-QuerySheet* system, with a series of exercises using both systems. When the students were comfortable with each system, the actual study was performed.

In the study, we used a real-life spreadsheet which we obtained, with permission, from a local food bank in our hometown of Braga. We thoroughly explained how the information was represented to the students, and how to properly interpret the spreadsheet information. This spreadsheet stored information regarding the distributions of basic products and other food bank institutions. This spreadsheet had information on 85 institutions and 14 different types of basic products, giving way to over 1190 lines of unique information.

We also denormalized the information for the students (to use with Google's QUERY function), and also prepared the spreadsheet model and conformed instance in the MDSheet environment. Since we can not show the actual spreadsheet data due to revealing private information, only the spreadsheet model is presented in Figure 8.

During the study, we asked participants to implement queries to answer the following four questions:

1. What is the total distributed for each product?



	A	B	C	D	E	F	G
1	**Distribution**			**Product**	name=""	...	
2					code=""	...	
3	**Institution**				stock=0	...	
4	code=""	name=""	lunch=0	dinner=0	distributed=0	...	
5	⋮	⋮	⋮	⋮	⋮	⋮	⋮
6						...	

Fig. 8: A model-driven spreadsheet representing Dstributions

2. What is the total stock?
3. What are the names of each institution without repetitions?
4. Which were the products with more than 500 units distributed, and to which institution were they delivered to?

For each question, users had to implement a query in both systems, alternating between starting with one and then the other (initial starting system was chosen by each student). Users were also asked to write down the time after carefully reading each question, and the time after the queries were executed (the differences in the running performance of *Graphical-QuerySheet* compared to the standalone Google QUERY function are negligible). They would then once again read the question and write down the initial and final times, but for the opposite system.

At the end of each question, participants were asked to choose which system they felt was more: Intuitive, Faster (to construct the queries), Easier (to construct the queries) and Understandable (being able to fully explain and understand the constructed queries).

At the end of the study, participants answered which system they preferred and why, and what advantages/disadvantages existed between the systems.

4.1 Results

The results we obtained from our study were gathered and analyzed, and are presented in this section. In Figure 9, we analyze the differences in terms of performance between *Graphical-QuerySheet* and the Google QUERY function. The left side (Y-Axis) represents the average number of minutes the students took to answer the questions. The bottom side (X-Axis) represents the question the students answered. The green bars represent the Google QUERY function, and the blue bars represent the *Graphical-QuerySheet* system.

As we can see, users spent substantially less time to construct the queries using the *Graphical-QuerySheet* system, ranging from as much as approximately 65% to approximately 90%, averaging out to an overall of 80%. In the cases where the queries or the results were incorrect, almost all (6 out of 7 errors) were with the Google QUERY function, varying between incorrect column letters chosen, bad query construction, and incorrect ranges.

Almost all (104 out of 112) chose our system in regards to the four previously mentioned points (Intuitive, Faster, Easier, and Understandable). The few cases

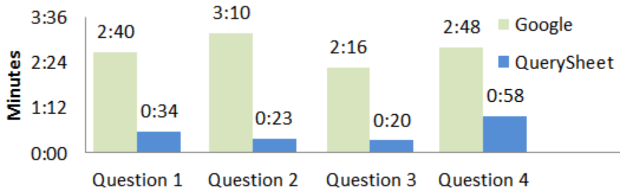


Fig. 9: A chart detailing the information gathered from the empirical evaluation

where they preferred Google's approach, or neither, provided us with interesting information, allowing us to detect some of the drawbacks of this system.

The comments written by the students were also very positive. All preferred our *Graphical-QuerySheet* system over Google's QUERY function, finding the interface extremely intuitive and query construction facilitated. Some of the comments can be seen below:

- *Graphical-QuerySheet is very intuitive and quick to use, presenting an attracting interface.*
- *Graphical-QuerySheet allows me to complete my tasks much more quickly.*
- *It was easy to construct queries using labels instead of column letters, and ordering, grouping, and aggregations are much simpler with Graphical-QuerySheet.*
- *No need to know SQL, a normal user like myself can quickly and easily construct queries.*
- *I do not need to worry about using group by when I aggregate, Graphical-QuerySheet does it automatically for me.*

5 Conclusion

In this paper, we presented the design, implementation, and validation of a graphical query language interface for model-driven spreadsheets. The focus of our design for the graphical query interface was to provide a human-friendly, easy to use, interactive environment to quickly construct ClassSheet-driven queries, for users with different SQL skills.

We have implemented our graphical, model-driven query environment in a model-driven spreadsheet environment. The *Graphical-QuerySheet* was used in an empirical study where we were able to increase productivity by approximately 80%, while also meeting our goals of balancing simplicity with expressability.

Even with the good results and responses towards our graphical querying system, some interesting directions of future research were identified. Although the empirical results we have presented are interesting, they were the result of a study with a relatively small group participants. Thus, we plan to execute a second study, this time with more participants, and more end-users.

Acknowledgments: We would like to thank Professor José Creissac Campos for his helpful comments and insight regarding the graphical query interface.

References

1. Maier, D.: The Theory of Relational Databases. Computer Science Press (1983)
2. Hainaut, J.L.: The transformational approach to database engineering. [17] 95–144
3. Engels, G., Erwig, M.: ClassSheets: automatic generation of spreadsheet applications from object-oriented specifications. In: Proc. of the 20th IEEE/ACM Int. Conf. on Aut. Sof. Eng., ACM (2005) 124–133
4. Bals, J.C., Christ, F., Engels, G., Erwig, M.: Classsheets - model-based, object-oriented design of spreadsheet applications. In: TOOLS Europe Conference (TOOLS 2007), Zürich (Swiss). Volume 6., Journal of Object Technology (October 2007) 383–398
5. Cunha, J., Erwig, M., Saraiva, J.: Automatically inferring classsheet models from spreadsheets. In: IEEE Symp. on Visual Languages and Human-Centric Computing, IEEE CS (2010) 93–100
6. Melton, J.: Database language sql. In Bernus, P., Mertins, K., Schmidt, G., eds.: Handbook on Architectures of Information Systems. International Handbooks on Information Systems. Springer Berlin Heidelberg (1998) 103–128
7. Google: Google query function. `https://developers.google.com/chart/interactive/docs/querylanguage` (2013) [Accessed on November 2013].
8. Cunha, J., Mendes, J., Fernandes, J.P., Pereira, R., Saraiva, J.: Design and implementation of queries for model-driven spreadsheets. In: Proceedings of the Domain-Specific Language Summer School 2013. (2014) (submitted).
9. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. Computer **39**(2) (February 2006) 25–31
10. Bézivin, J.: Model driven engineering: An emerging technical space. [17] 36–64
11. Ireson-Paine, J.: Model master: an object-oriented spreadsheet front-end. Computer-Aided Learning using Technology in Economies and Business Education (1997)
12. Abraham, R., Erwig, M., Kollmansberger, S., Seifert, E.: Visual specifications of correct spreadsheets. In: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing. VLHCC '05, Washington, DC, USA, IEEE Computer Society (2005) 189–196
13. Cunha, J., Fernandes, J.P., Mendes, J., Saraiva, J.: Mdsheet: A framework for model-driven spreadsheet engineering. In: ICSE. (2012) 1395–1398
14. Pereira, R.: Querying for model-driven spreadsheets. Master's thesis, University of Minho (2013)
15. Cunha, J., Fernandes, J.P., Mendes, J., Pereira, R., Saraiva, J.: Querying model-driven spreadsheets. [18] 83–86
16. Belo, O., Cunha, J., Fernandes, J.P., Mendes, J., Pereira, R., Saraiva, J.: Querysheet: A bidirectional query environment for model-driven spreadsheets. [18] 199–200
17. Lämmel, R., Saraiva, J., Visser, J., eds.: Generative and Transformational Techniques in Software Engineering, International Summer School, Braga, Portugal, July 4-8, 2005. Revised Papers. In Lämmel, R., Saraiva, J., Visser, J., eds.: GTTSE 2005. Volume 4143 of Lecture Notes in Computer Science., Springer (2006)
18. Kelleher, C., Burnett, M.M., Sauer, S., eds.: 2013 IEEE Symposium on Visual Languages and Human Centric Computing, San Jose, CA, USA, September 15-19, 2013. In Kelleher, C., Burnett, M.M., Sauer, S., eds.: VL/HCC, IEEE (2013)