Universidade do Minho
Escola de Engenharia

Rui Alexandre Afonso Pereira

Querying for Model-Driven Spreadsheets

Setembro de 2013

This work is funded by the ERDF through the Programme COMPETE and by the Portuguese Government through FCT - Foundation for Science and Technology, within projects: FCOMP-01-0124-FEDER-010048, FCOMP-01-0124-FEDER-020532, FCOMP-01-0124-FEDER-022701, and grant ref. BI2-2012_PTDC/EIA-CCO/108613/2008.

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Rui Alexandre Afonso Pereira

Querying for Model-Driven Spreadsheets

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de
Professor Doutor João Saraiva
Professor Doutor Orlando Belo

Setembro de 2013

DECLARAÇÃO

Nome

Rui Alexandre Afonso Pereira

Endereço electrónico: rui.pereira@di.uminho.pt Telefone: 927 713 007 / __________

Número do Bilhete de Identidade: 14030134

Título dissertação ☐ / tese ☐

Querying for Model-Driven Spreadsheets

Orientador(es):

João Alexandre Saraiva

Orlando Manuel Belo Ano de conclusão: 2013

Designação do Mestrado ou do Ramo de Conhecimento do Doutoramento:

Mestrado em Eng. Informática

Universidade do Minho, 22 / 10 / 2013

Assinatura: [illegible]

Acknowledgements

To begin, I want to thank both of my supervisors, Prof. João Saraiva and Prof. Orlando Belo, due to their knowledge, dedication, experience, professionalism and ability to easily communicate with me, led me on an amazing path during this thesis, which helped me greatly. And especially to Prof. João Saraiva for motivating me tremendously and pushing me further than I expected I could go, giving me great opportunities to take advantage of.

I would also like to further thank Jácome Cunha, who was like another supervisor to me, always being available when I needed him, helping me very much during this thesis, even if he had no obligation to, and putting up with me annoying him constantly. I would also like to thank João Paulo Fernandes for the helpful and insightful comments during the first stage of my thesis.

To my laboratory buddies, Claudio, Jorge, Tiago, and both Pedros, who put up with my antics on a regular basis and kept me in check, and also for the great and unforgettable moments passed in Romania.

To my friends, from the first year of university up until now, who put up with me, and always helped and supported me along the way. All the good and unforgettable moments, to the idiotic things we pulled on each other, ultimately not allowing me to go insane this year! I would also like to give a special mention to Bernardo, Paulo, and Rui, who followed me closely during the completion of my thesis, giving me good pointers and helpful comments.

And finally, I would like to thank my parents for their endless support in all my decisions in life, and for opening the doors to allow me to get where I am now. Even with the countless struggles and problems, you always believed in me, and believed in where I could get in life. Thank you so much.

Abstract

Querying for Model-Driven Spreadsheets

Spreadsheets are used for a diverse number of objectives, that range from simple applications to complete information systems. In all of these cases, they are frequently used as data repositories that can grow tremendously in size, and as the amount of the data grows, the frustration and challenge to withdraw information out of them also grows.

This Thesis project focuses on the problem of spreadsheet querying. Specifically, the objective is to meticulously and carefully study competing query languages, and proposing our very own expressive and composable query language to be used in spreadsheets, where intuitive queries can be defined. This approach builds on a model-driven spreadsheet development environment, and queries are expressed referencing ClassSheet model entities instead of the actual data. Furthermore, this language shall be integrated into the MDSheet framework, taking into account evolution mechanisms, auto-generation of models for query results, and shall rely on Google's QUERY function for spreadsheets.

Resumo

Interrogação de Folhas de Cálculo Dirigidas por Modelos

As folhas de cálculo são utilizadas para diversos fins, desde aplicações simples até sistemas de informação completos. Entre todos estes casos, são frequentemente utilizadas para armazenar grandes volumes de dados, sendo que, à medida que o repositório cresce, a frustação e o desafio de recolher informação também aumenta. O projeto desta dissertação foca-se no problema da consulta e interrogação de folhas de cálculo. Especificamente, o objetivo é estudar de forma cuidada e meticulosa diversas linguagens de interrogação existentes, e propôr a nossa própria linguagem para ser utilizada em folhas de cálculo, que se caracteriza por ser uma linguagem expressiva, que possibilita a composição de interrogações e a definição das mesmas de forma intuitiva.

A abordagem a utilizar passa pela utilização de folhas de cálculo dirigidas por modelos, sendo as interrogações expressas através de entidades do modelo ClassSheet em vez de dados em concreto. Além disto, a linguagem desenvolvida será integrada no framework MDSheet, considerando diversos mecanismos de evolução, geração automática de modelos para os resultados de uma interrogação, e será baseada na função QUERY desenvolvida pela Google para a interrogação de folhas de cálculo.

Contents

Acronyms

GQF	Google QUERY Function
MDE	Model-Driven Engineering
MDQL	Model-Driven Query Language
OLAP	Online Analytical Processing
RDBMS	Relational Database Management System
SQL	Structured Query Language
UML	Unified Modeling Language

List of Figures

List of Tables

Listings

1 Introduction

1.1 Motivation

Nowadays, spreadsheets can be considered the most popular programming system around, particularly in the field of business applications. With their availability on any computing device (PC, smart-phone, etc.) and in the cloud, visual simplicity, low learning curve for new users, and flexibility when it comes to what can be written in a spreadsheet, the amount of users per year increases drastically. Although spreadsheets begin as a simple, single-user software artifact, they may evolve into a large and complex data-centric software [Chambers and Scaffidi, 2010].

But while estimates of tens of millions of business workers create hundreds of millions of spreadsheets per year [Engels and Erwig, 2005], advances in this domain fall off compared to advances made in other programming languages. Also with the exception of a few, not much software engineering work has been made on spreadsheets. For example, Ireson-Paine created an object-oriented textual language to specify spreadsheet models, dubbed *Model Master* [Ireson-Paine, 1997]. Following Ireson-Paine's work, came *ViTSL* [Abraham et al., 2005], which eventually set the foundation for the creation of *ClassSheets* [Engels and Erwig, 2005], a high-level object-oriented model for spreadsheets. Even with this work, since spreadsheets are used in many cases as a way to store information, much like the traditional databases systems such as Oracle Database[1], MySQL[2], and SQL Server[3], manipulating a large amount of data in a traditional matrix structure becomes an arduous task. This issue arises in spreadsheets, unlike the traditional database systems, due to one huge flaw: the absence of a data query language.

Attempts have been made to try to replicate traditional querying systems, such as MS-Query (Microsoft) and the QUERY function (Google), but in doing so, many restrictions regarding how data must be stored and organized are imposed. Also, much like a database administrator or analyst would look at the relational model of a database to construct quer-

[1]Oracle Database: `http://www.oracle.com/index.html`
[2]MySQL: `http://www.mysql.com/`
[3]SQL Server: `http://www.microsoft.com/sqlserver/`

ies, and not the data itself, we too would benefit from a model reflecting our spreadsheet data to easily construct our spreadsheet queries. Additionally with the growing usage of Model-Driven spreadsheets, which have been proven to improve productivity with empirical studies [Beckwith et al., 2011a], also lacking ways to query the data, querying in this environment is becoming much sought after.

1.2 Research Questions

Three important questions arose during my Thesis work, relative to the design and implementation of a Model-Driven Query Language, and users' productivity using it:

1. *Would it be possible to create a query language for use on spreadsheet models which can be translated into Google's QUERY function language?*
2. *If using Google's QUERY function, what extra work and implementation must be done to correctly query a model-driven spreadsheet?*
3. *How efficient and productive is querying Model-Driven spreadsheets, when compared to directly querying spreadsheet data (e.g. MS-Query or Google's QUERY function)?*

By the time I conclude my Thesis, I plan to easily answer all of these questions, acknowledging that to answer the last question, an empirical study must be executed.

1.3 Structure of the Thesis

This Thesis is organized as follows:

Section 2 - Spreadsheet Engineering - contains the State of the Art, with information on previous spreadsheet engineering work and two tools created to query spreadsheets.

Section 3 - Model-Driven Spreadsheet Engineering - contains examples of spreadsheet models along with the description of ClassSheet models. This section also presents spreadsheet evolution techniques using ClassSheet models.

Section 4 - Querying Model-Driven Spreadsheets - presents how we envision a simple SQL-like query language where users can easily construct queries right in their spreadsheet environment using ClassSheet models. This is explained using an example of a possible real-life scenario.

Section 5 - MDQL: Design and Implementation - contains information regarding the design and implementation of the Query Language, and the process from creating the language, to retrieval of data from the Google QUERY function.

Section 6 - QuerySheet - describes and showcases the prototype tool developed, dubbed QuerySheet, which implements all the techniques presented in Section 5

Section 8 - Conclusion - concludes this Thesis with comments on the work done, results, and future work, along with answers to our Research Questions.

2 Spreadsheet Engineering

Spreadsheets can be considered the most popular programming environment around, widely used by non-professional programmers, particularly in the field of business applications. This can be due to the immense flexibility, learnability, portability, and simplicity for the end-users, allowing them to easily jump right into this powerful tool.

But this liberty and elbowroom, however, does come with the high price of inducing end-users to accidentally and unknowingly fill their spreadsheets with errors, which could lead up to thousands of dollars in damage such as the Colorado Student Loan Program incident, where a type error caused an operating fund to be understated by $36,131 [US Department of Education, 2003]. Various studies also show that up to 90% of spreadsheets contain errors [Panko, 2008]. Huge company losses and social problems caused by erroneous spreadsheets are documented in the Spreadsheets Horror Stories web-page[4].

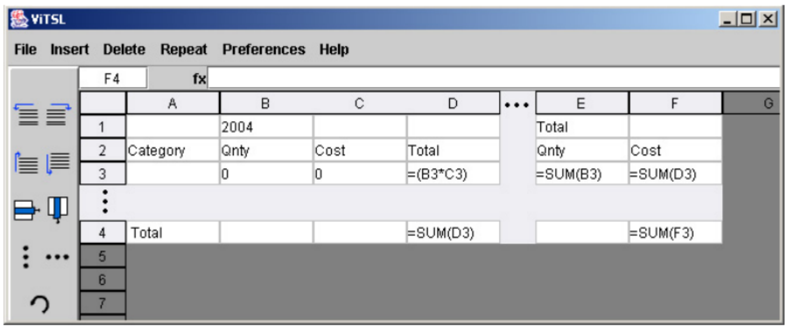
To overcome these constant problems in spreadsheets, the spreadsheet community decided to apply Model-Driven Engineering methodologies to spreadsheets. Model-Driven Engineering is a development methodology in Software Development that uses and exploits domain models, or abstract representations of a piece of software [Schmidt, 2006]. By using this methodology, engineers can impose constraints and detect and prevent errors in the early stages of software development by performing model checking.

Even then, spreadsheet modeling had already been proposed by various researchers, such as the early work by Ireson-Paine who, as early as 1997, introduced a compiler and object-oriented textual language to specify spreadsheets called *Model Master* [Ireson-Paine, 1997].

Later in 2005, Abraham, Erwig, Kollmansberger, and Seifert familiarized us with a visual specification language, quite similar to spreadsheets called ViTSL [Abraham et al., 2005], as shown in 1 (taken from [Abraham et al., 2005]).

Soon after, Engels and Erwig introduced ClassSheets as a high-level and object-oriented formalism to specify the business logic of spreadsheets [Bals et al., 2007]; allowing end-users to express object structures within their actual spreadsheets, using Unified Modeling Language, or UML, fundamentals. This gave end-users the possibility of characterizing

[4]http://www.eusprig.org/horror-stories.htm



	A	B	C	D	E	F
1		2004			Total	
2	Category	Qnty	Cost	Total	Qnty	Cost
3		0	0	=(B3*C3)	=SUM(B3)	=SUM(D3)
4	Total			=SUM(D3)		=SUM(F3)

Figure 1: *ViTSL editor*

their spreadsheets, giving the latter names, column labels, value constraints, and expansion direction (i.e. horizontally or vertically). Indeed, empirical studies show that ClassSheet-driven spreadsheet development does improve the productivity of spreadsheet programming [Beckwith et al., 2011b] [Mendes, 2012]. Without models such as these, an end-user's experience can be negatively affected when facing the challenges of understanding their requirements as well as making decisions about design, and other activities [Ko et al., 2011] which can be done in spreadsheets.

For these reasons and others (including further research based on ClassSheets such as MDSheet [Cunha et al., 2012d]), I will be using ClassSheets for the work done during my Thesis, taking advantage of the specification of object structures in spreadsheets.

To further take advantage of models/diagrams, research has been made to automatically infer ClassSheet models or other class diagrams from spreadsheets [Cunha et al., 2010] [Hermans et al., 2010], allowing the end-users to easily obtain model representations for their spreadsheets. Eventually, Cunha et al. began to work on allowing one to edit a spreadsheet instance based on the relational model inferred [Cunha et al., 2012e] [Cunha et al., 2009], assisting the user in editing the spreadsheet, offering features such as auto-completion, guarded deletion and controlled insertion, to help avoid redundancy, loss, or corruption of data during the edit actions.

But even up till later works such as Hermans et al.'s Gyro [Hermans et al., 2010], and Cunha's HaExcel framework derived from his work [Cunha, 2011], there was no progress made on the subject of questioning the actual spreadsheet data, or in other words, a

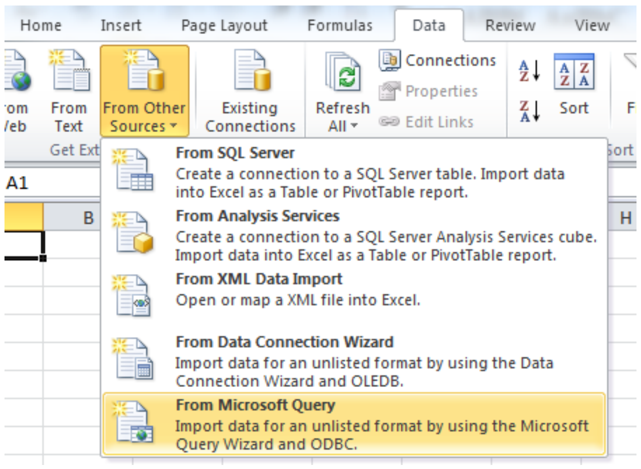
Query Language for Model-Driven Spreadsheets, originating the foundation and reason of my Thesis.

2.1 Spreadsheets and Queries

There have been attempts to query standard spreadsheets, using some form of SQL, or Structured Query Language, a special-purpose language for accessing and managing data in relational database management systems (RDBMS). While SQL is specifically for RDBMS, the general idea behind it can be applied to spreadsheets in a tabular format, allowing the user to specify what attributes he/she would like to know, and apply his or her filters to the same query he/she would like to invoke. The following section will present two of the principal tools used to try to mimic the general idea of SQL for RDBMS, but this time for spreadsheets, each with their own benefits and disadvantages.

2.1.1 MS-Query Tool

One of these principal tools is Microsoft's Query tool, or MS-Query. MS-Query is a database query interfaced used by Microsoft Word and Excel, a utility which imports databases, text files, OLAP cubes and other spreadsheet representations (such as csv) into Excel. Figure 2 shows how one would invoke MS-Query from Excel.



From SQL Server
Create a connection to a SQL Server table. Import data into Excel as a Table or PivotTable report.

From Analysis Services
Create a connection to a SQL Server Analysis Services cube. Import data into Excel as a Table or PivotTable report.

From XML Data Import
Open or map a XML file into Excel.

From Data Connection Wizard
Import data for an unlisted format by using the Data Connection Wizard and OLEDB.

From Microsoft Query
Import data for an unlisted format by using the Microsoft Query Wizard and ODBC.

Figure 2: *Invoking MS-Query*

After selecting the data source, a query builder wizard appears to begin constructing a query, starting with the table and column selection as shown in the following image. As one may notice, MS-Query uses a visual query building approach, which helps users not experienced in constructing/writing SQL statements.

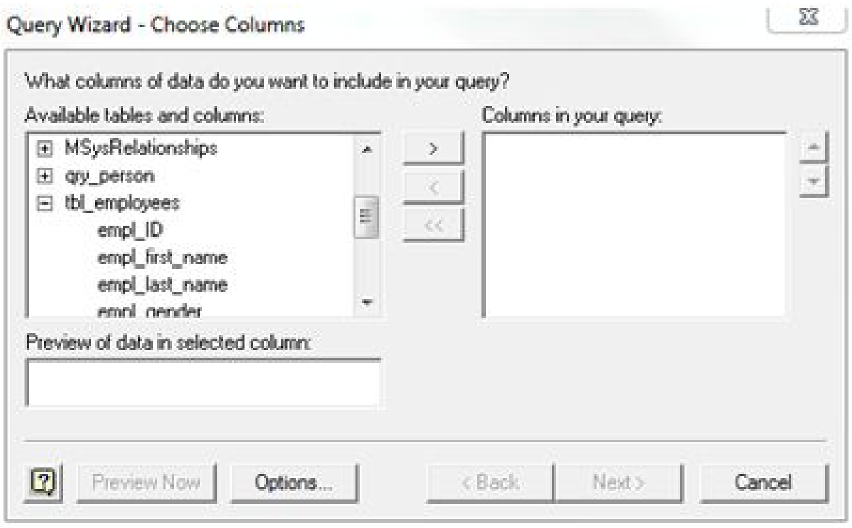

Query Wizard - Choose Columns
What columns of data do you want to include in your query?
Available tables and columns:
MSysRelationships
qry_person
tbl_employees
empl_ID
empl_first_name
empl_last_name
empl_gender
Columns in your query:
Preview of data in selected column:
Preview Now
Options...
< Back
Next >
Cancel

Figure 3: *MS-Query Column window*

After selecting the table and columns (Figure 3), the user is presented a filter data window, where he/she can include filters for each of the columns previously selected, or in other words, construct the `WHERE` clause in SQL, shown in Figure 4.

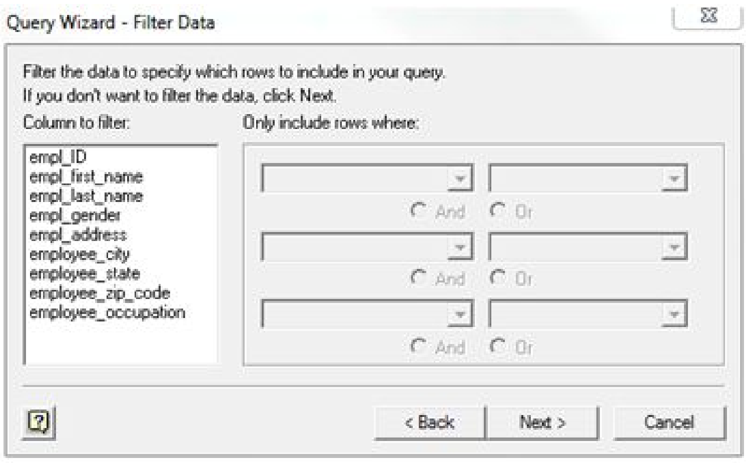
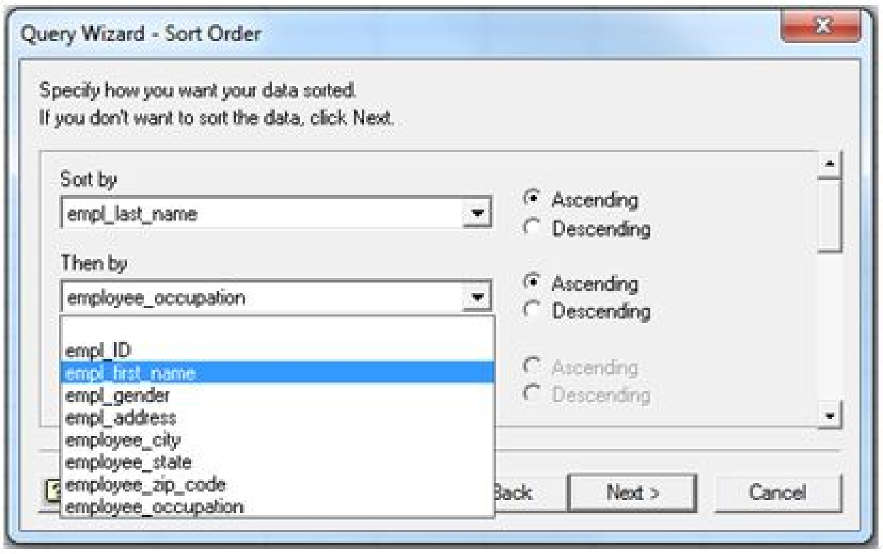


Figure 4: *MS-Query Filter Data window*



Query Wizard - Sort Order

Specify how you want your data sorted.
If you don't want to sort the data, click Next.

Sort by: empl_last_name [✓ Ascending] Descending

Then by: employee_occupation [✓ Ascending] Descending

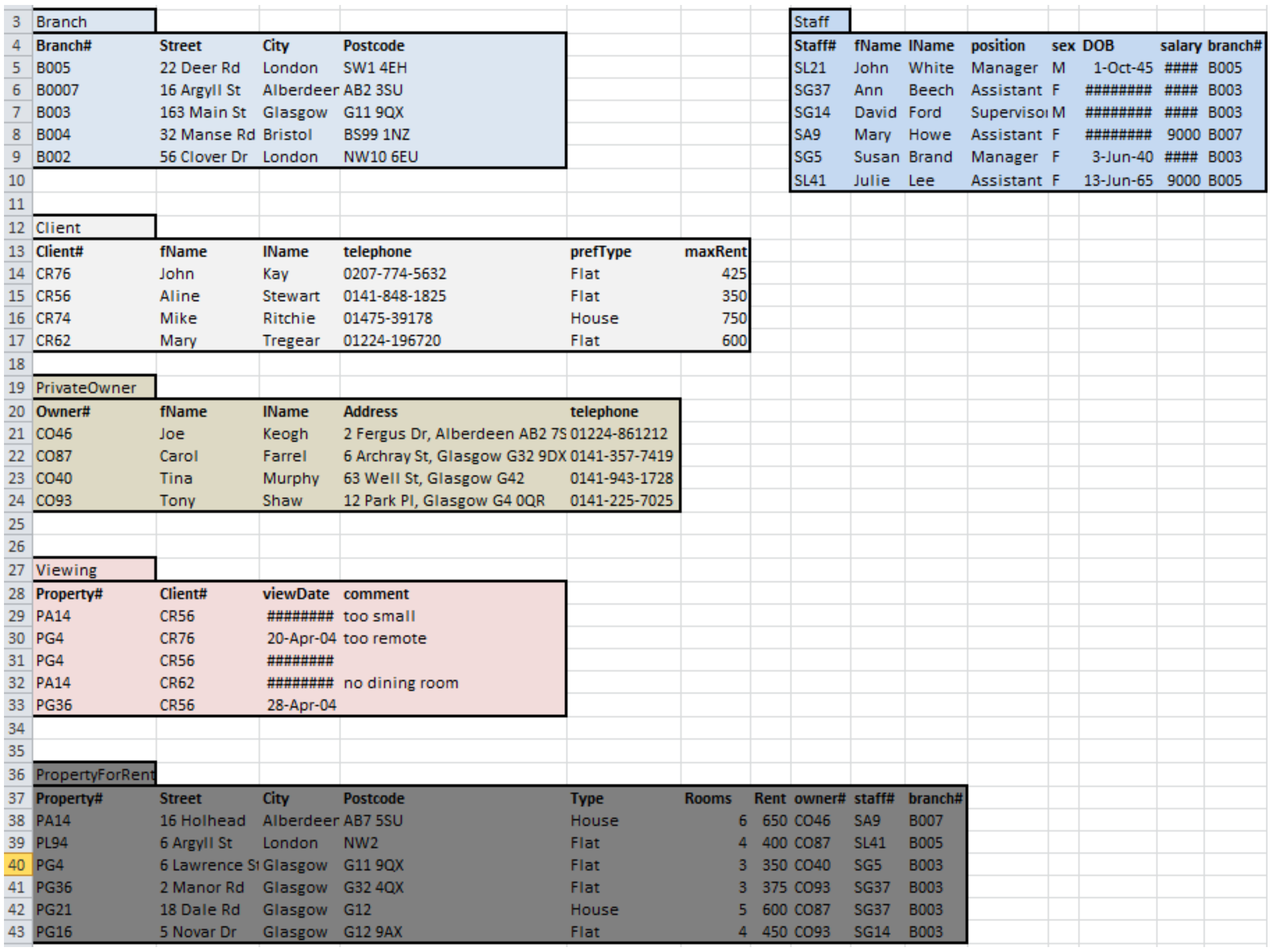
empl_ID
empl_first_name
empl_gender
empl_address
employee_city
employee_state
employee_zip_code
employee_occupation

Figure 5: *MS-Query Sort Order Window*

To finish the query building, you may choose, if needed, the `Sort Order` of the query (Figure 5), and after if you wish to return the data to an Excel spreadsheet or view the data in MS-Query.

While MS-Query is mostly used to analyze OLAP cubes, represent a database or database table in Excel, or show the results of a SQL query done on a database, it can be used to query data from another spreadsheet, placing the data into an intermediate database table to be able to apply the query, and represent the findings, but with some restrictions.

To be able to query your spreadsheet data, the data itself must be in a table format, with the headers having the column names. In most cases, users tend to use their spreadsheet for more than one entity in a single worksheet, not joining all the information into one single table.



Branch

Branch#	Street	City	Postcode
B005	22 Deer Rd	London	SW1 4EH
B0007	16 Argyll St	Alberdeer	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

Staff

Staff#	fName	lName	position	sex	DOB	salary	branch#
SL21	John	White	Manager	M	1-Oct-45	####	B005
SG37	Ann	Beech	Assistant	F	########	####	B003
SG14	David	Ford	Supervisoı	M	########	####	B003
SA9	Mary	Howe	Assistant	F	########	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	####	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

Client

Client#	fName	lName	telephone	prefType	maxRent
CR76	John	Kay	0207-774-5632	Flat	425
CR56	Aline	Stewart	0141-848-1825	Flat	350
CR74	Mike	Ritchie	01475-39178	House	750
CR62	Mary	Tregear	01224-196720	Flat	600

PrivateOwner

Owner#	fName	lName	Address	telephone
CO46	Joe	Keogh	2 Fergus Dr, Alberdeen AB2 7S	01224-861212
CO87	Carol	Farrel	6 Archray St, Glasgow G32 9DX	0141-357-7419
CO40	Tina	Murphy	63 Well St, Glasgow G42	0141-943-1728
CO93	Tony	Shaw	12 Park Pl, Glasgow G4 0QR	0141-225-7025

Viewing

Property#	Client#	viewDate	comment
PA14	CR56	########	too small
PG4	CR76	20-Apr-04	too remote
PG4	CR56	########	
PA14	CR62	########	no dining room
PG36	CR56	28-Apr-04	

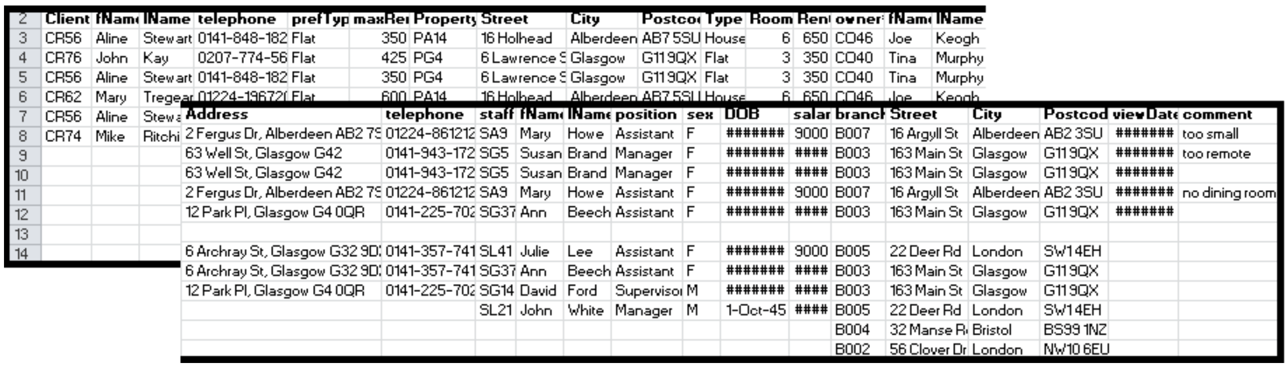
PropertyForRent

Property#	Street	City	Postcode	Type	Rooms	Rent	owner#	staff#	branch#
PA14	16 Holhead	Alberdeer	AB7 5SU	House	6	650	CO46	SA9	B007
PL94	6 Argyll St	London	NW2	Flat	4	400	CO87	SL41	B005
PG4	6 Lawrence St	Glasgow	G11 9QX	Flat	3	350	CO40	SG5	B003
PG36	2 Manor Rd	Glasgow	G32 4QX	Flat	3	375	CO93	SG37	B003
PG21	18 Dale Rd	Glasgow	G12	House	5	600	CO87	SG37	B003
PG16	5 Novar Dr	Glasgow	G12 9AX	Flat	4	450	CO93	SG14	B003

Figure 6: *Property Renting Example*

We can see how the data is not in one single table, nor are the headers represented in a single row (Figure 6).

Also, the worksheet contains various entities, each with their own attributes and relationships between each other. Notice how the **Staff** entity contains a **branch#** code, which relates it to the **Branch** entity, making the data normalized in a sense, prohibiting the complete freedom to represent the spreadsheet data how a user wishes.



2	Client	fNam	lName	telephone	prefTyp	maxRe	Property	Street	City	Postco	Type	Room	Rent	owner	fNam	lName
3	CR56	Aline	Stewart	0141-848-182	Flat	350	PA14	16 Holhead	Alberdeen	AB7 5SU	House	6	650	CO46	Joe	Keogh
4	CR76	John	Kay	0207-774-56	Flat	425	PG4	6 Lawrence S	Glasgow	G11 9QX	Flat	3	350	CO40	Tina	Murphy
5	CR56	Aline	Stewart	0141-848-182	Flat	350	PG4	6 Lawrence S	Glasgow	G11 9QX	Flat	3	350	CO40	Tina	Murphy
6	CR62	Mary	Tregear	01224-196720	Flat	600	PA14	16 Holhead	Alberdeen	AB7 5SU	House	6	650	CO46	Joe	Keogh
7	CR56	Aline	Stewa													
8	CR74	Mike	Ritchi													
9																
10																
11																
12																
13																
14																

Address	telephone	staff	fNam	lName	position	sex	DOB	salar	branch	Street	City	Postcod	viewDate	comment
2 Fergus Dr, Alberdeen AB2 7S	01224-861212	SA9	Mary	Howe	Assistant	F	#######	9000	B007	16 Argyll St	Alberdeen	AB2 3SU	#######	too small
63 Well St, Glasgow G42	0141-943-172	SG5	Susan	Brand	Manager	F	#######	####	B003	163 Main St	Glasgow	G11 9QX	#######	too remote
63 Well St, Glasgow G42	0141-943-172	SG5	Susan	Brand	Manager	F	#######	####	B003	163 Main St	Glasgow	G11 9QX	#######	
2 Fergus Dr, Alberdeen AB2 7S	01224-861212	SA9	Mary	Howe	Assistant	F	#######	9000	B007	16 Argyll St	Alberdeen	AB2 3SU	#######	no dining room
12 Park Pl, Glasgow G4 0QR	0141-225-702	SG37	Ann	Beech	Assistant	F	#######	####	B003	163 Main St	Glasgow	G11 9QX	#######	
6 Archray St, Glasgow G32 9D	0141-357-741	SL41	Julie	Lee	Assistant	F	#######	9000	B005	22 Deer Rd	London	SW1 4EH		
6 Archray St, Glasgow G32 9D	0141-357-741	SG37	Ann	Beech	Assistant	F	#######	####	B003	163 Main St	Glasgow	G11 9QX		
12 Park Pl, Glasgow G4 0QR	0141-225-702	SG14	David	Ford	Supervisor	M	#######	####	B003	163 Main St	Glasgow	G11 9QX		
		SL21	John	White	Manager	M	1-Oct-45	####	B005	22 Deer Rd	London	SW1 4EH		
									B004	32 Manse R	Bristol	BS99 1NZ		
									B002	56 Clover Dr	London	NW10 6EU		

Figure 7: *Property Renting Example (Denormalized)*

Figure 7 shows the necessary representation of our data, from the spreadsheet example in Figure 6, in a completely denormalized state, and having the headers of each attribute explicitly represented in a single row, so we may be able to query the data using the MS-Query tool.

As you may notice, the representation of the data in this way is much harder for someone to read, manage, and analyze, and with a real-life spreadsheet data source, the number of columns might reach the hundreds.

Along with the previous mentioned problems, a user may also not expand his information horizontally, but only vertically, to conform to the table format needed to query, allowing even much less freedom to represent their data.

2.1.2 Google QUERY Function

The other tool is Google's QUERY function which allows users, using a SQL-like syntax, to perform a query over an array of values, for example their Google Docs spreadsheets, where the function is built-in.

It is a two part query, consisting of a **Range** input, to state the range of cells for example **A1:B6**. The second part consists of the actual Query String, using the subset of the SQL language, using column letters instead of column names. The `QUERY's` input also assumes the first zero or more rows as headers, and each column of the input can only hold values of the following types:

- **Boolean** - Boolean literals either *True* or *False*
- **Number** - Numeric literals specified in decimal notation
- **String** - String literals enclosed in either single or double quotes
- **Date** - Using keyword *date* followed by a string (ex: date "2013-05-02")
- **Datetime** - A date and a time, using keyword *datetime* followed by a string (ex:date time '2013-05-02 11:31:34.123')
- **Timeofday** - Using keyword *timeofday* followed by a string in literal fromat HH:mm:ss (ex: timeofday '11:31:45')

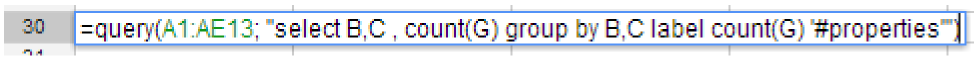
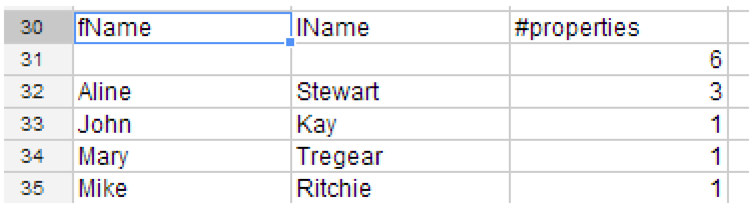

30 =query(A1:AE13; "select B,C , count(G) group by B,C label count(G) '#properties'")

Figure 8: *Google QUERY function used in the Property Renting Example (Denormalized)*

The user, using the `Google QUERY` function, shown in Figure 8, can actually write his query in the same worksheet as his data. This allows on-the-spot results (Figure 9), with no pre-configurations needed. The query engine is also very potent, as with any `Google` made query engine.

While being a powerful query function created by `Google`, it still has its flaws. The `QUERY` function shares the same problems as MS-Query in regards to the data representation. Much like MS-Query, to run the Query function, the data needs to be represented with a single header row, without relationships between entities, in other words, also denormalized (as shown in Figure 7). Along with the difficulty of managing the data in such a way, the



30	fName	lName	#properties
31			6
32	Aline	Stewart	3
33	John	Kay	1
34	Mary	Tregear	1
35	Mike	Ritchie	1

Figure 9: *Google QUERY function results from the Property Renting Example (Denormalized)*

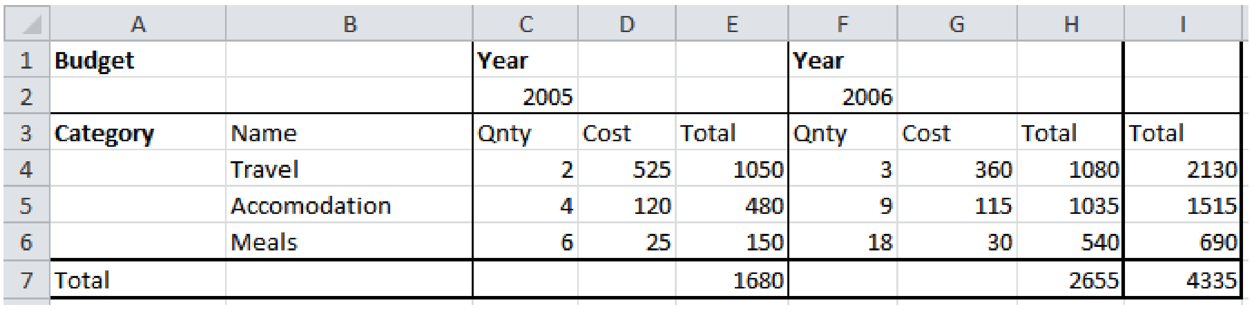
QUERY function has another flaw, the actual SQL-like query. Instead of writing the query using column names, one must use the column letters (as shown in Figure 8) to write his or her query. Even with the small sized example we have been using, column letters and not names can get confusing, counter-intuitive, and almost impossible to understand what the query is supposed to do, without having the data sheet alongside you. Even then, if we introduced a spreadsheet with over one-hundred columns, we would need to spend quite some significant time. Moreover, Google queries do not fully and truly support evolution since they do not adapt/evolve when the spreadsheet data evolves. That is to say, by adding a column to the spreadsheet we may turn a query invalid or incorrect because data changed places in the spreadsheet.

Even though each has their own advantages, such as the query builder wizard of MS-Query, or the extremely fast Google query engine for the QUERY function, improvements can be made, mainly in making the actual SQL-like language the QUERY function uses more user-friendly and understandable, and allowing users to express their spreadsheet data with much more freedom, without having to conform to the format needed by these two spreadsheet query alternatives.

So while attempts of querying standard spreadsheets have been made, there hasn't been work done on querying data specifically for Model-Driven spreadsheets, an environment proven to facilitate and improve productivity for users. Having such a functionality added to a Model-Driven environment would further improve it, allowing users to finally, easily, and efficiently query their data.

3 Model-Driven Spreadsheet Engineering

Model-Driven Engineering (MDE) is a solution to the handling of complex and evolving software system [Bézivin, 2005], something which could and has been applied to spreadsheets, making model-driven spreadsheets possible. As mentioned in the previous section (Section 2), there has been work done on creating a bridge between real-world representations and spreadsheet data, in other words, spreadsheet models. The following subsections will further explain models in spreadsheets, more specifically ClassSheets, along with how evolution affects these models, and introduce a model-driven spreadsheet environment, named MDSheet [Cunha et al., 2012d] [Mendes, 2012], which was used during this thesis.



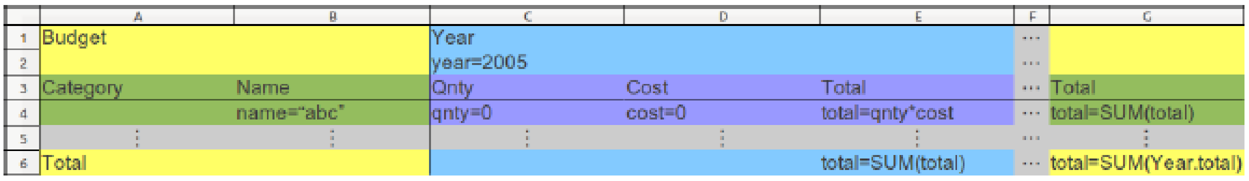
	A	B	C	D	E	F	G	H	I
1	**Budget**		**Year**			**Year**			
2			2005			2006			
3	**Category**	Name	Qnty	Cost	Total	Qnty	Cost	Total	Total
4		Travel	2	525	1050	3	360	1080	2130
5		Accomodation	4	120	480	9	115	1035	1515
6		Meals	6	25	150	18	30	540	690
7	Total				1680			2655	4335

Figure 10: *Spreadsheet data for a Budget example*

Above in Figure 10 we have a spreadsheet used to store information about the budget of a company. This spreadsheet contains information about the Category of budget use (such as Travel or Accommodation) and the Year. The relationship between these two gives information on the Quantity, the Cost and the Total Costs. Spreadsheets use layouts to structure its data, making it easier to understand and perform computations on it. In this case, each new year adds three new columns in the spreadsheet. This, the spreadsheet data grows horizontally. New types of expenses in Category can also be added, by adding new rows to the spreadsheet data, growing the spreadsheet vertically. This spreadsheet also has information on the sum of total costs per Year and per Category. Also, for each new Year or Category, the formulas for the Total values need to be updated, or manually written. This spreadsheet will be used as a running example for the following subsections to describe ClassSheet models, model evolution, and using MDSheet.

3.1 Models in Spreadsheets

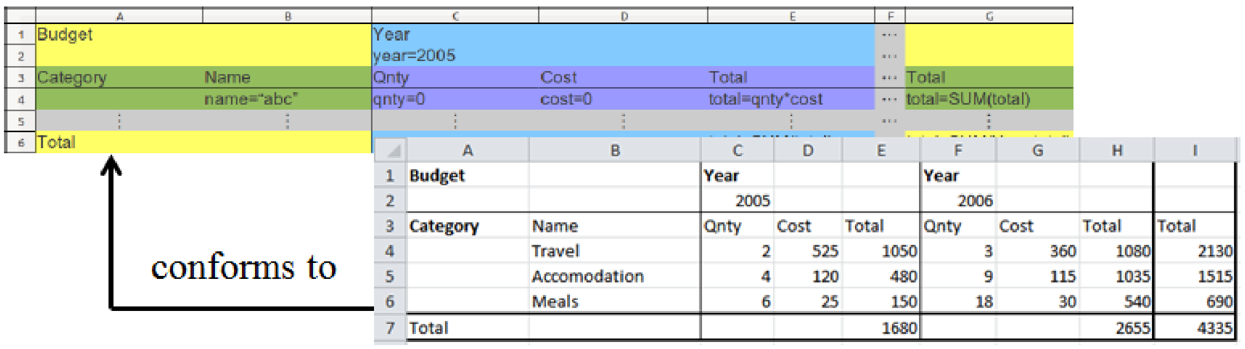
Overtime, our example spreadsheet can grow tremendously, becoming much more complex than what is currently shown. As a result of spreadsheets being able to evolve into complex software systems, Engels and Erwig introduced ClassSheets [Engels and Erwig, 2005] as a way to express business logic spreadsheet data as a model formalism. ClassSheets are a high-level and object-oriented formalism using the notion of classes and attributes. This formalism offers a model-driven software development approach to spreadsheets. Therefore, we can define the business logic of the spreadsheet in a concise and abstract formalism. This results in users being able to understand, evolve and maintain complex spreadsheets by just analyzing the (ClassSheet) models, avoiding the need to look at large and complex data.



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

Figure 11: *ClassSheet model for a Budget example*

To showcase ClassSheets, we have a ClassSheet model for the Budget example shown in Figure 10. In this ClassSheet model (Figure 11 (taken from [Mendes, 2012])), a **Budget** has a **Category** and **Year** class, expanding vertically and horizontally, respectively. The joining of these gives us a **Quantity**, a **Cost** and **Total** of a **Category** in a given **Year**, each with its own default value. The **Total** in column G gives us the total of each **Category** and the **Total** in column A gives us the total of each **Year**.



	A	B	C	D	E	F	G
1	Budget		Year			···	
2			year=2005			···	
3	Category	Name	Qnty	Cost	Total	···	Total
4		name="abc"	qnty=0	cost=0	total=qnty*cost	···	total=SUM(total)
5	⋮	⋮	⋮	⋮	⋮	···	⋮
6	Total						

	A	B	C	D	E	F	G	H	I
1	**Budget**		**Year**			**Year**			
2			2005			2006			
3	**Category**	Name	Qnty	Cost	Total	Qnty	Cost	Total	Total
4		Travel	2	525	1050	3	360	1080	2130
5		Accomodation	4	120	480	9	115	1035	1515
6		Meals	6	25	150	18	30	540	690
7	Total				1680			2655	4335

Figure 12: *Spreadsheet model and example in conformity*

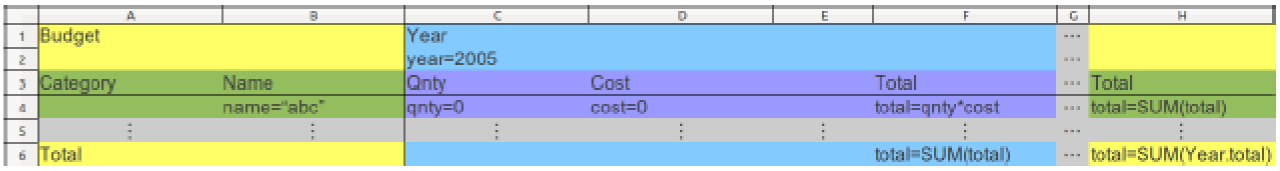
This ClassSheet model specifies the business logic of the budget spreadsheet data. In model-driven engineering we would say that the spreadsheet data (Figure 10) *conforms to* the model (Figure 11), as shown in Figure 12.

3.2 Model-Driven Spreadsheet Evolution

Software evolution [Mens and Demeyer, 2008] is a term which defines the process of changing an existing software system or program, due to needs, rules, and other factors, is updated, or in other words evolves, to continue to be useful in its environment. Factors such as new requirements which emerged while using the software, business environment changes, bugs and errors, new and better performing computers and equipment, or even simply just wanting something changed for no reason.

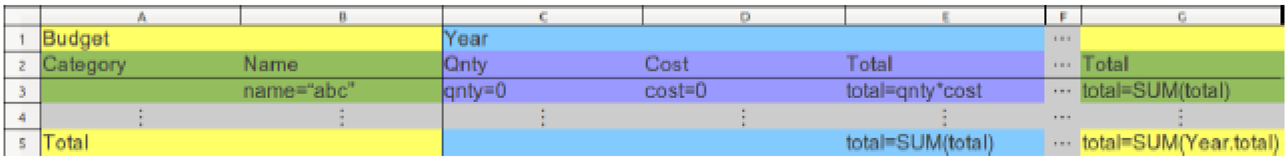
Spreadsheets over time tend to evolve and change too, just like software programs and systems. Users may need to add new columns they previously did not have due to new laws, new business requirements, new information, or because the previous spreadsheet was incorrectly constructed. Not only can it evolve due to new columns, but new data, or rows, can be inserted, columns removed, or even the whole spreadsheet can be changed to represent the information previously contained, in a different way.

For example, using the original model from Figure 11, we can add a new column after column D resulting in Figure 13 (taken from [Mendes, 2012]). We may also decide to remove a row, in this case row 2, from the original model, giving us Figure 14 (taken from [Mendes, 2012]).



	A	B	C	D	E	F	G	H
1	Budget		Year				···	
2			year=2005				···	
3	Category	Name	Qnty	Cost		Total	···	Total
4		name="abc"	qnty=0	cost=0		total=qnty*cost	···	total=SUM(total)
5	⋮	⋮	⋮	⋮	⋮	⋮	···	⋮
6	Total					total=SUM(total)	···	total=SUM(Year.total)

Figure 13: *Budget Model after adding a new column*



	A	B	C	D	E	F	G
1	Budget		Year			···	
2	Category	Name	Qnty	Cost	Total	···	Total
3		name="abc"	qnty=0	cost=0	total=qnty*cost	···	total=SUM(total)
4	⋮	⋮	⋮	⋮	⋮	···	⋮
5	Total				total=SUM(total)	···	total=SUM(Year.total)

Figure 14: *Budget Model after removing a row*

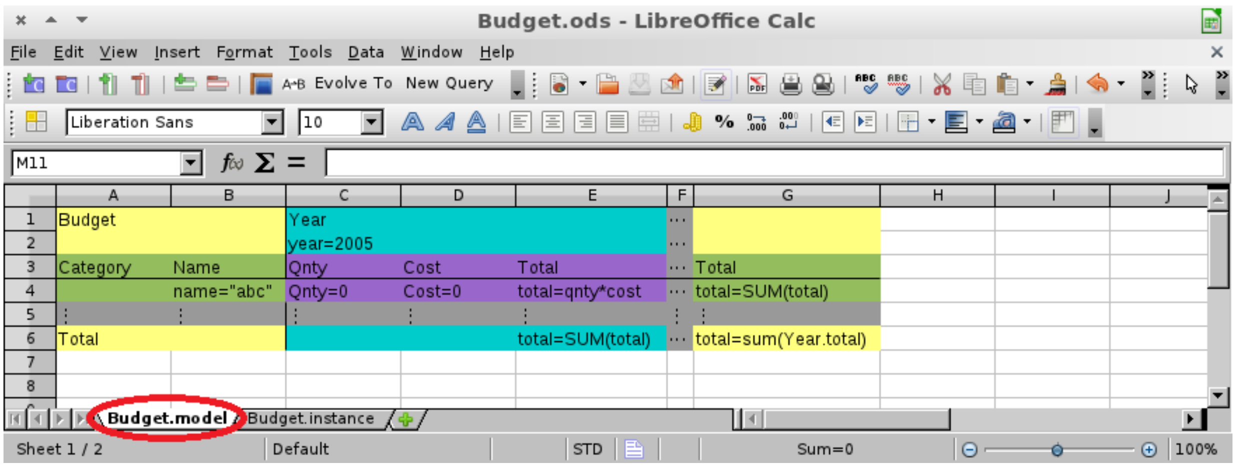
Many more operations can be made on the model to allow it to evolve to the user's specifications and need, conforming the data to the model. The same can be done to the data, which in turn would affect the model due to bidirectional transformation [Cunha et al., 2012c].

3.3 MDSheet

MDSheet [Cunha et al., 2012d] is a framework and model-driven spreadsheet environment, developed by Cunha et al., which permits the previous two model-driven spreadsheet engineering points (models and evolution of spreadsheets), while also allowing embedding of model and instance, model inference and bidirectional transformations between models and instances, all with a simple OpenOffice plugin.

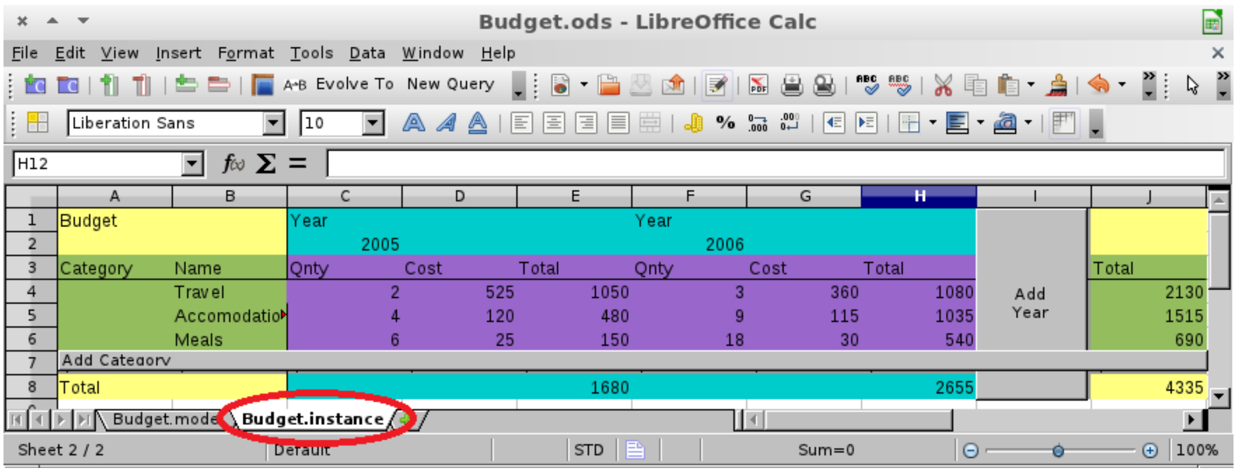
3.3.1 Embedding and Bidirectionality

MDSheet builds upon a technique to allow embedding of spreadsheet models and instances in the same working spreadsheet and environment [Cunha et al., 2011], as shown in Figure 15 and Figure 16 for the model and data worksheet respectively. This embedding also allows model-driven spreadsheet evolution (as explained in Section 3.2) in the same environment, allowing a user to modify the model in one worksheet, the instance is updated accordingly, and vice-versa. This model and instance embedding has also shown promising results and suggest productivity gains are possible using this model-driven spreadsheet development environment [Cunha et al., 2013a].



	A	B	C	D	E	F	G	H	I	J
1	Budget		Year			…				
2			year=2005			…				
3	Category	Name	Qnty	Cost	Total	…	Total			
4		name="abc"	Qnty=0	Cost=0	total=qnty*cost	…	total=SUM(total)			
5	⋮	⋮	⋮	⋮	⋮	⋮	⋮			
6	Total				total=SUM(total)	…	total=sum(Year.total)			
7										
8										

Figure 15: *Model on the first worksheet of a spreadsheet*



	A	B	C	D	E	F	G	H	I	J
1	Budget		Year			Year				
2			2005			2006				
3	Category	Name	Qnty	Cost	Total	Qnty	Cost	Total		Total
4		Travel	2	525	1050	3	360	1080	Add Year	2130
5		Accomodatio	4	120	480	9	115	1035		1515
6		Meals	6	25	150	18	30	540		690
7	Add Category									
8	Total				1680			2655		4335

Figure 16: *Data on the second worksheet of a spreadsheet*

This embedding supports evolution by allowing model changes to be reflected upon the instances, but the reverse is also possible because of a bidirectional transformation system which MDSheet has, allowing the instance to be changed and the model reflects those changes using the theoretical foundations from [Cunha et al., 2012c]. This bidirectionality extends the previous features, now allowing end-users to change either model or instance, all in the same spreadsheet (due to embedding).

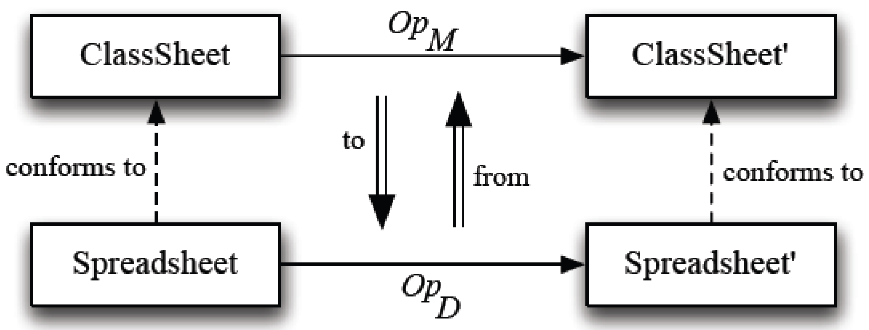


Figure 17: *Bidirectional evolution diagram*

These model transformations are related to a sequence of transformations on the data, which ensures that a valid model transformation is equal to a valid data transformation and guarantees that evolved data instance maintains conformity with the model. To allow the opposite to happen, a similar technique is used, as shown in Figure 17 (taken from [Mendes, 2012]), once again guaranteeing consistency between model and instance.

3.3.2 Architecture

When MDSheet was being designed, modularity was kept in mind, to not only be able to work on various systems, but also allow new features and functionality to be extended in this system.

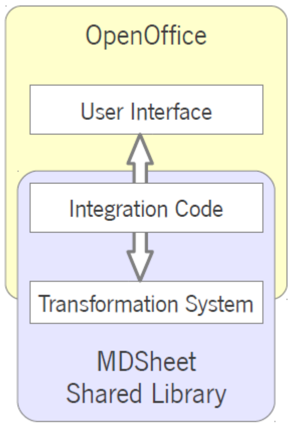


Figure 18: *MDSheet Architecture*

As we can see in Figure 18 (taken from [Mendes, 2012]), the architecture is divided into three main components: the User Interface, the Integration Code, and the Transformation System.

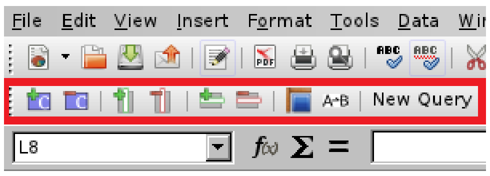


Figure 19: *MDSheet Button Interface*

The User Interface component represents the graphical representation, in the form of buttons in OpenOffice, of the ClassSheet operations which a user can do, such as creating new ClassSheet classes, adding new columns/rows to the classes, and other operations, as we can see in Figure 19. Along with the buttons, are the windows where users can specify the expansion direction of the classes, and the naming of the class.

One of the most important components of the MDSheet architecture is the Integration Code. It is here where the connection between the OpenOffice components (of BASIC, C/C++) and the Haskell framework (where the core of this system is implemented in) is made. In this level, the Foreign Function Interface is used to close the gap between Haskell and C/C++ code, making possible the conversion of Haskell data structures to C/C++'s data structures.

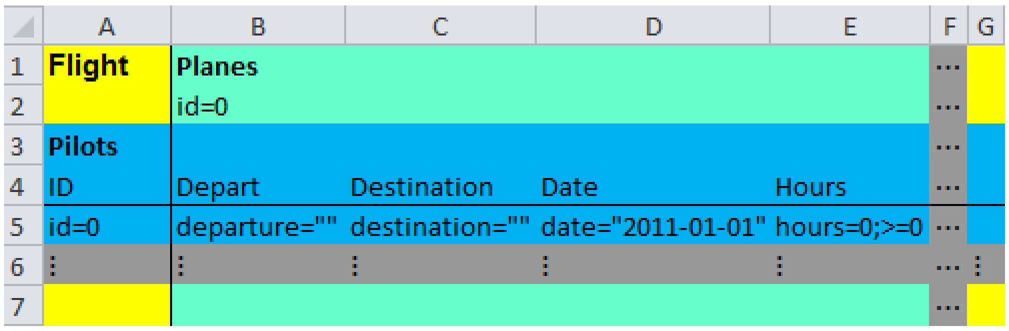
The third component is the Transformation System, which consists of the functions used to manipulate ClassSheet models and spreadsheet data, including the functions to perform the operations already mentioned (adding a new ClassSheet model, adding new attributes to a model, etc), along with the bidirectional transformation system explained in the previous Subsection 3.2, and shown with Figure 17.

The modularity of the MDSheet framework allowed me to integrate my thesis work, which allowed me to take advantage of ClassSheet model specifications, and the OpenOffice graphical interface.

4 Querying Model-Driven Spreadsheets

What I envision is a simple SQL-like query language where users can easily construct queries right in their spreadsheet environment, without the need of complicated configurations, or extra programs other than a simple add-on. This approach would build upon a Model-Driven spreadsheet development environment, where the queries would be expressed referencing entities in ClassSheet models instead of the actual data, allowing the user to not have to worry about the arrangement of the spreadsheet's data, but only what information is present. This would allow spreadsheet evolution to occur on the data or the arrangement of entities within a spreadsheet model, without invalidating previously constructed queries, as long as the entities continue to exist in the model. To exemplify what I envision, a practical example of a real-life scenario will be presented.

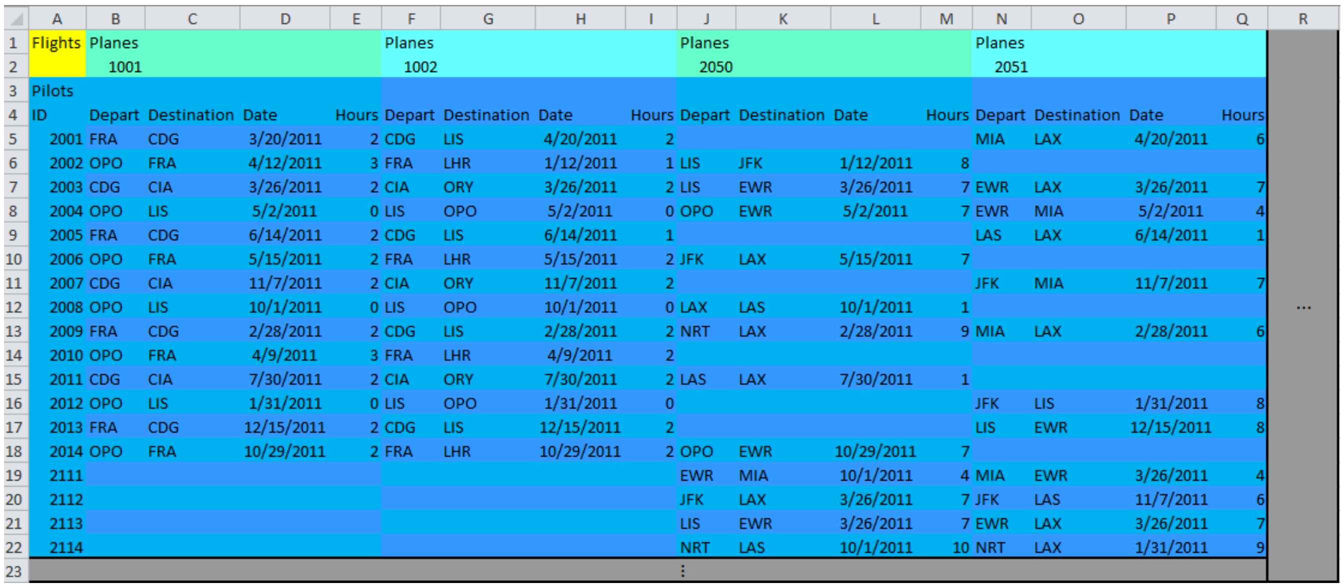
4.1 Querying Model-Driven Spreadsheets: An Example



	A	B	C	D	E	F	G
1	**Flight**	**Planes**				...	
2		id=0				...	
3	**Pilots**					...	
4	ID	Depart	Destination	Date	Hours	...	
5	id=0	departure=""	destination=""	date="2011-01-01"	hours=0;>=0	...	
6	⋮	⋮	⋮	⋮	⋮	...	⋮
7						...	

Figure 20: *ClassSheet Model for a flight spreadsheet*

Figure 20 is an example of a ClassSheet model for a fictional flight spreadsheet. As the reader can see, the **Flight** class is composed of both a **Pilot** and a **Plane** class, each with its own identification code, or **ID**, while expanding vertically and horizontally respectively. The joining of the **Pilot** and **Plane** classes gives us four distinct attributes: **Depart**, **Destination**, **Date**, and **Hours**, each with its own default value.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	Flights	Planes				Planes				Planes				Planes				
2		1001				1002				2050				2051				
3	Pilots																	
4	ID	Depart	Destination	Date	Hours	Depart	Destination	Date	Hours	Depart	Destination	Date	Hours	Depart	Destination	Date	Hours	
5	2001	FRA	CDG	3/20/2011	2	CDG	LIS	4/20/2011	2					MIA	LAX	4/20/2011	6	
6	2002	OPO	FRA	4/12/2011	3	FRA	LHR	1/12/2011	1	LIS	JFK	1/12/2011	8					
7	2003	CDG	CIA	3/26/2011	2	CIA	ORY	3/26/2011	2	LIS	EWR	3/26/2011	7	EWR	LAX	3/26/2011	7	
8	2004	OPO	LIS	5/2/2011	0	LIS	OPO	5/2/2011	0	OPO	EWR	5/2/2011	7	EWR	MIA	5/2/2011	4	
9	2005	FRA	CDG	6/14/2011	2	CDG	LIS	6/14/2011	1					LAS	LAX	6/14/2011	1	
10	2006	OPO	FRA	5/15/2011	2	FRA	LHR	5/15/2011	2	JFK	LAX	5/15/2011	7					
11	2007	CDG	CIA	11/7/2011	2	CIA	ORY	11/7/2011	2					JFK	MIA	11/7/2011	7	
12	2008	OPO	LIS	10/1/2011	0	LIS	OPO	10/1/2011	0	LAX	LAS	10/1/2011	1					...
13	2009	FRA	CDG	2/28/2011	2	CDG	LIS	2/28/2011	2	NRT	LAX	2/28/2011	9	MIA	LAX	2/28/2011	6	
14	2010	OPO	FRA	4/9/2011	3	FRA	LHR	4/9/2011	2									
15	2011	CDG	CIA	7/30/2011	2	CIA	ORY	7/30/2011	2	LAS	LAX	7/30/2011	1					
16	2012	OPO	LIS	1/31/2011	0	LIS	OPO	1/31/2011	0					JFK	LIS	1/31/2011	8	
17	2013	FRA	CDG	12/15/2011	2	CDG	LIS	12/15/2011	2					LIS	EWR	12/15/2011	8	
18	2014	OPO	FRA	10/29/2011	2	FRA	LHR	10/29/2011	2	OPO	EWR	10/29/2011	7					
19	2111									EWR	MIA	10/1/2011	4	MIA	EWR	3/26/2011	4	
20	2112									JFK	LAX	3/26/2011	7	JFK	LAS	11/7/2011	6	
21	2113									LIS	EWR	3/26/2011	7	EWR	LAX	3/26/2011	7	
22	2114									NRT	LAS	10/1/2011	10	NRT	LAX	1/31/2011	9	
23										⋮								

Figure 21: *Flight spreadsheet data*

Above (Figure 21) is a flight spreadsheet in conformance with our previously mentioned ClassSheet from Figure 20, which we will be using as our example. This data spreadsheet represents flight data for four **Planes** and eighteen **Pilots**.

Now let us imagine a flight manager, John, who needs to know which **Pilots** and **Plane**s made their way to Lisbon Airport, and since he is in good spirit, wants to know how many hours each of those **Pilots** has flown, to possibly give them a raise.

So John needs to answer the following questions:

Query1: Which Pilots and Planes have had their destination as LIS?

Query2: How many hours has each of those Pilots flown?

With only needing to look at the ClassSheet Model, if he does not remember or know the structure of the spreadsheet data, John would be able to write following simple SQL-like query:

Listing 1: *Model-driven query for Query1*

```
SELECT Planes.*, Pilots.*
FROM Flights
WHERE Destination = 'LIS'
```

Listing 2: *Model-driven query for Query2*

```
SELECT Pilots.*, sum(Hours)
FROM Flights
WHERE Destination = 'LIS'
GROUP BY Pilots.*
LABEL sum(Hours) 'Total Hours'
```

And have the following results:

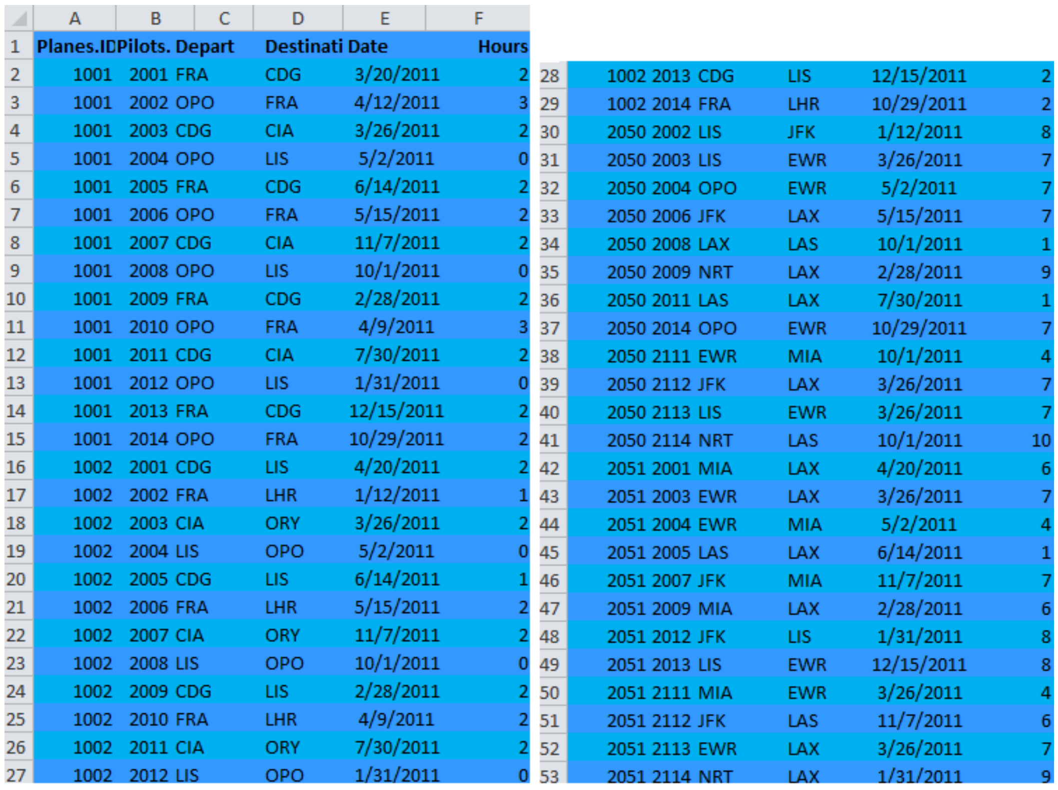
Table 1: *Results for Query1*

Planes.ID	Pilots.ID
1001	2004
1001	2008
1001	2012
1002	2001
1002	2005
1002	2009
1002	2013
2051	2012

Table 2: *Results for Query2*

Pilots.ID	Total Hours
2001	2
2004	0
2005	1
2008	0
2009	2
2012	8
2013	2

Now if he were to use an alternative approach, such as the `Google Query Function` (GQF) to answer his questions, the task would be much more difficult. First off, the spreadsheet data would have to be in a denormalized state, so either the manager manually re-writes his spreadsheet data temporarily, or he re-writes the spreadsheet permanently, possibly making the data much harder to read, manage, update, and analyze for him and any other user, just to be able to run the GQF.



	A	B	C	D	E	F
1	Planes.ID	Pilots.	Depart	Destinati	Date	Hours
2	1001	2001	FRA	CDG	3/20/2011	2
3	1001	2002	OPO	FRA	4/12/2011	3
4	1001	2003	CDG	CIA	3/26/2011	2
5	1001	2004	OPO	LIS	5/2/2011	0
6	1001	2005	FRA	CDG	6/14/2011	2
7	1001	2006	OPO	FRA	5/15/2011	2
8	1001	2007	CDG	CIA	11/7/2011	2
9	1001	2008	OPO	LIS	10/1/2011	0
10	1001	2009	FRA	CDG	2/28/2011	2
11	1001	2010	OPO	FRA	4/9/2011	3
12	1001	2011	CDG	CIA	7/30/2011	2
13	1001	2012	OPO	LIS	1/31/2011	0
14	1001	2013	FRA	CDG	12/15/2011	2
15	1001	2014	OPO	FRA	10/29/2011	2
16	1002	2001	CDG	LIS	4/20/2011	2
17	1002	2002	FRA	LHR	1/12/2011	1
18	1002	2003	CIA	ORY	3/26/2011	2
19	1002	2004	LIS	OPO	5/2/2011	0
20	1002	2005	CDG	LIS	6/14/2011	1
21	1002	2006	FRA	LHR	5/15/2011	2
22	1002	2007	CIA	ORY	11/7/2011	2
23	1002	2008	LIS	OPO	10/1/2011	0
24	1002	2009	CDG	LIS	2/28/2011	2
25	1002	2010	FRA	LHR	4/9/2011	2
26	1002	2011	CIA	ORY	7/30/2011	2
27	1002	2012	LIS	OPO	1/31/2011	0
28	1002	2013	CDG	LIS	12/15/2011	2
29	1002	2014	FRA	LHR	10/29/2011	2
30	2050	2002	LIS	JFK	1/12/2011	8
31	2050	2003	LIS	EWR	3/26/2011	7
32	2050	2004	OPO	EWR	5/2/2011	7
33	2050	2006	JFK	LAX	5/15/2011	7
34	2050	2008	LAX	LAS	10/1/2011	1
35	2050	2009	NRT	LAX	2/28/2011	9
36	2050	2011	LAS	LAX	7/30/2011	1
37	2050	2014	OPO	EWR	10/29/2011	7
38	2050	2111	EWR	MIA	10/1/2011	4
39	2050	2112	JFK	LAX	3/26/2011	7
40	2050	2113	LIS	EWR	3/26/2011	7
41	2050	2114	NRT	LAS	10/1/2011	10
42	2051	2001	MIA	LAX	4/20/2011	6
43	2051	2003	EWR	LAX	3/26/2011	7
44	2051	2004	EWR	MIA	5/2/2011	4
45	2051	2005	LAS	LAX	6/14/2011	1
46	2051	2007	JFK	MIA	11/7/2011	7
47	2051	2009	MIA	LAX	2/28/2011	6
48	2051	2012	JFK	LIS	1/31/2011	8
49	2051	2013	LIS	EWR	12/15/2011	8
50	2051	2111	MIA	EWR	3/26/2011	4
51	2051	2112	JFK	LAS	11/7/2011	6
52	2051	2113	EWR	LAX	3/26/2011	7
53	2051	2114	NRT	LAX	1/31/2011	9

Figure 22: *Flight spreadsheet data in denormalized state*

So after John re-writes his data to the required state, as seen in Figure 22, he can finally begin writing his questions in an SQL-like query. But here's the catch, the query writing isn't as simple as stating what Attributes he wants to use, or what models he wishes to question, he needs to state what area in his spreadsheet has the data, and which Columns, not Attributes, he will use.

The previous two questions the manager wanted answered, Query1 and Query2, would be written in the following way:

Listing 3: *Google QUERY function for Query1*

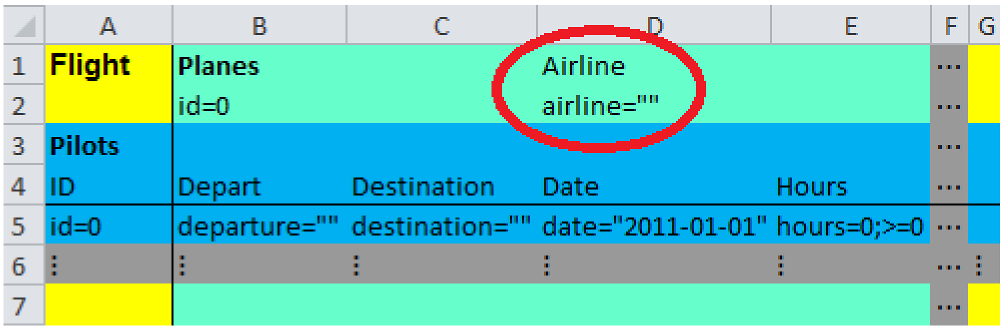
```
=query(A1:F53;"SELECT A, B WHERE D = 'LIS'")
```

Listing 4: *Google QUERY function for Query2*

```
=query(A1:F53;"SELECT B, SUM(F) WHERE D = 'LIS'
   GROUP BY B LABEL SUM(F) 'Total Hours'")
```

As shown, the first part of the function needs the spreadsheet data area, in our manager's case, **A1:F53**, which is all of his data, including the headers. The second part of the function needs the actual query, and as you may notice, it uses column letters to specify the attributes, not their names.

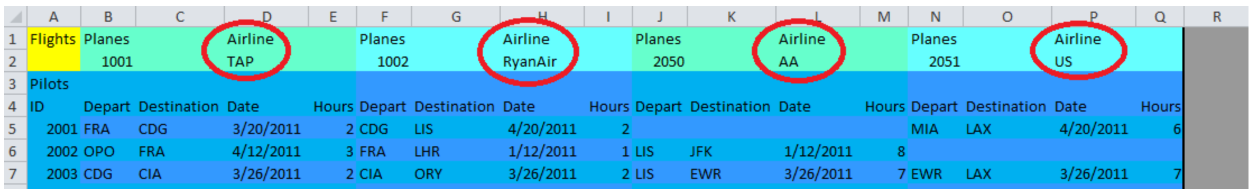
These results are identical to the results previously returned from Query1 and Query2, as they should be.



	A	B	C	D	E	F	G
1	Flight	Planes		Airline		...	
2		id=0		airline=""		...	
3	Pilots					...	
4	ID	Depart	Destination	Date	Hours	...	
5	id=0	departure=""	destination=""	date="2011-01-01"	hours=0;>=0	...	
6	⋮	⋮	⋮	⋮	⋮	...	⋮
7						...	

Figure 23: *ClassSheet Model for a flight spreadsheet after Airline name attribute inserted*

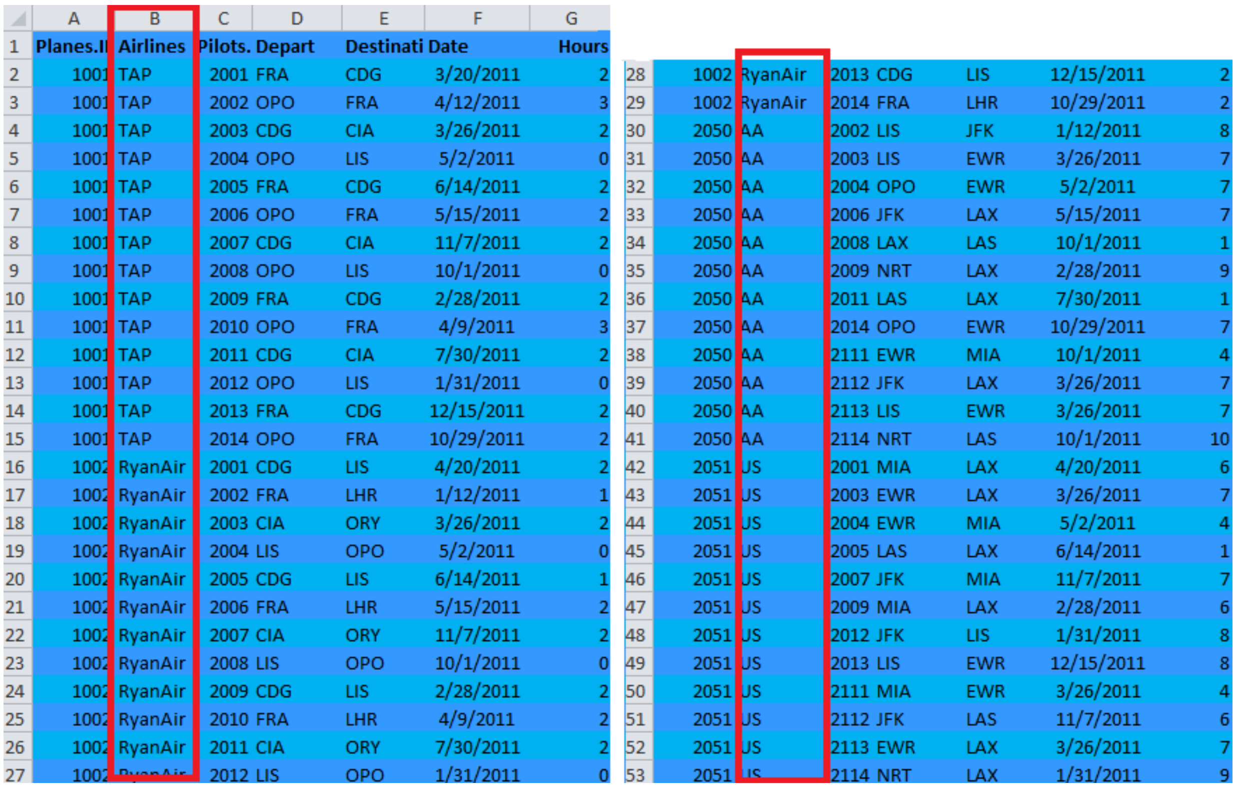
But John just realized he forgot to add the Planes' **Airline** name! John promptly goes on to update the spreadsheets to reflect this. Using a model-driven environment, John just needs to add the new attribute to the ClassSheet model, as shown in Figure 23, where he added to column D/row 2 the **Airline** attribute. And as expected, the spreadsheet data in conformance also updates accordingly and all John needs to do is write the Airline names (Figure 24), no worries and no problems occur due to this evolution, fast and simple.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	Flights	Planes		Airline		Planes		Airline		Planes		Airline		Planes		Airline		
2		1001		TAP		1002		RyanAir		2050		AA		2051		US		
3	Pilots																	
4	ID	Depart	Destination	Date	Hours	Depart	Destination	Date	Hours	Depart	Destination	Date	Hours	Depart	Destination	Date	Hours	
5	2001	FRA	CDG	3/20/2011	2	CDG	LIS	4/20/2011	2					MIA	LAX	4/20/2011	6	
6	2002	OPO	FRA	4/12/2011	3	FRA	LHR	1/12/2011	1	LIS	JFK	1/12/2011	8					
7	2003	CDG	CIA	3/26/2011	2	CIA	ORY	3/26/2011	2	LIS	EWR	3/26/2011	7	EWR	LAX	3/26/2011	7	

Figure 24: *Flight spreadsheet data after Airline name attribute inserted*

The two queries are untouched, and correctly function as John would hope. But what if he used the GQF instead? How would he go about on making the spreadsheet change to reflect the **Airline** names?



	A	B	C	D	E	F	G
1	Planes.II	Airlines	Pilots.	Depart	Destinati	Date	Hours
2	1001	TAP	2001	FRA	CDG	3/20/2011	2
3	1001	TAP	2002	OPO	FRA	4/12/2011	3
4	1001	TAP	2003	CDG	CIA	3/26/2011	2
5	1001	TAP	2004	OPO	LIS	5/2/2011	0
6	1001	TAP	2005	FRA	CDG	6/14/2011	2
7	1001	TAP	2006	OPO	FRA	5/15/2011	2
8	1001	TAP	2007	CDG	CIA	11/7/2011	2
9	1001	TAP	2008	OPO	LIS	10/1/2011	0
10	1001	TAP	2009	FRA	CDG	2/28/2011	2
11	1001	TAP	2010	OPO	FRA	4/9/2011	3
12	1001	TAP	2011	CDG	CIA	7/30/2011	2
13	1001	TAP	2012	OPO	LIS	1/31/2011	0
14	1001	TAP	2013	FRA	CDG	12/15/2011	2
15	1001	TAP	2014	OPO	FRA	10/29/2011	2
16	1002	RyanAir	2001	CDG	LIS	4/20/2011	2
17	1002	RyanAir	2002	FRA	LHR	1/12/2011	1
18	1002	RyanAir	2003	CIA	ORY	3/26/2011	2
19	1002	RyanAir	2004	LIS	OPO	5/2/2011	0
20	1002	RyanAir	2005	CDG	LIS	6/14/2011	1
21	1002	RyanAir	2006	FRA	LHR	5/15/2011	2
22	1002	RyanAir	2007	CIA	ORY	11/7/2011	2
23	1002	RyanAir	2008	LIS	OPO	10/1/2011	0
24	1002	RyanAir	2009	CDG	LIS	2/28/2011	2
25	1002	RyanAir	2010	FRA	LHR	4/9/2011	2
26	1002	RyanAir	2011	CIA	ORY	7/30/2011	2
27	1002	RyanAir	2012	LIS	OPO	1/31/2011	0
28	1002	RyanAir	2013	CDG	LIS	12/15/2011	2
29	1002	RyanAir	2014	FRA	LHR	10/29/2011	2
30	2050	AA	2002	LIS	JFK	1/12/2011	8
31	2050	AA	2003	LIS	EWR	3/26/2011	7
32	2050	AA	2004	OPO	EWR	5/2/2011	7
33	2050	AA	2006	JFK	LAX	5/15/2011	7
34	2050	AA	2008	LAX	LAS	10/1/2011	1
35	2050	AA	2009	NRT	LAX	2/28/2011	9
36	2050	AA	2011	LAS	LAX	7/30/2011	1
37	2050	AA	2014	OPO	EWR	10/29/2011	7
38	2050	AA	2111	EWR	MIA	10/1/2011	4
39	2050	AA	2112	JFK	LAX	3/26/2011	7
40	2050	AA	2113	LIS	EWR	3/26/2011	7
41	2050	AA	2114	NRT	LAS	10/1/2011	10
42	2051	US	2001	MIA	LAX	4/20/2011	6
43	2051	US	2003	EWR	LAX	3/26/2011	7
44	2051	US	2004	EWR	MIA	5/2/2011	4
45	2051	US	2005	LAS	LAX	6/14/2011	1
46	2051	US	2007	JFK	MIA	11/7/2011	7
47	2051	US	2009	MIA	LAX	2/28/2011	6
48	2051	US	2012	JFK	LIS	1/31/2011	8
49	2051	US	2013	LIS	EWR	12/15/2011	8
50	2051	US	2111	MIA	EWR	3/26/2011	4
51	2051	US	2112	JFK	LAS	11/7/2011	6
52	2051	US	2113	EWR	LAX	3/26/2011	7
53	2051	US	2114	NRT	LAX	1/31/2011	9

Figure 25: *Flight spreadsheet data in denormalized state after Airline name attribute inserted*

First, he would have to go to the denormalized data from Figure 22 and add the new column, and since he wants to group closely together the information about the **Planes**, so reading and writing the spreadsheet data won't become even more confusing, he decides to add the column between column A and column B, and then fill in the cells with the names, in this case 53 cells versus the 4 cells if using models. Finally he has the spreadsheet data reflecting the new change as shown in Figure 25.

But John isn't done yet, after a few days he noticed that his two Google queries were not working correctly! Possibly causing errors and monetary losses. He noticed that not only is the Range incorrect, since he added a new column, but the queries are also not the same ones as he previously wrote. The data range changed from A1:F53 to A1:G53, and in this case, each column letter has to be *shifted* over one letter after column A. Eventually John figured out the solution and re-wrote the two queries as follows:

Listing 5: *Rewritten Google QUERY function for Query1*

```
=query(A1:G53;"SELECT A, C WHERE E = 'LIS'")
```

Listing 6: *Rewritten Google QUERY function for Query2*

```
=query(A1:G53;"SELECT C, SUM(G) WHERE E = 'LIS'
  GROUP BY C LABEL SUM(G) 'Total Hours'")
```

The problem with the Google queries was that since the query string and range input is based completely off of the positioning of the data in the spreadsheet rather than referencing entities and attributes in a spreadsheet, when any new column or row is added, the range input needs to be updated, and quite possibly the query strings too.

Now, our example spreadsheet was pretty simple in terms of the amount of data, only having six columns, and fifty-two rows. But imagine the spreadsheets used by business workers and companies, with thousands and thousands of rows, and hundreds of columns. Our poor manager would end up getting lost while writing the query, "Was it column BD or BL that had the salary? Or was it AA?", or even while analyzing and updating the spreadsheet in the restrictive denormalized state, and just imagining the trouble the user would go through just to add new information to the spreadsheet while having to update the queries manually is enough to deter many from using the GQF.

So as one can see, our approach hopes to make querying spreadsheets more:

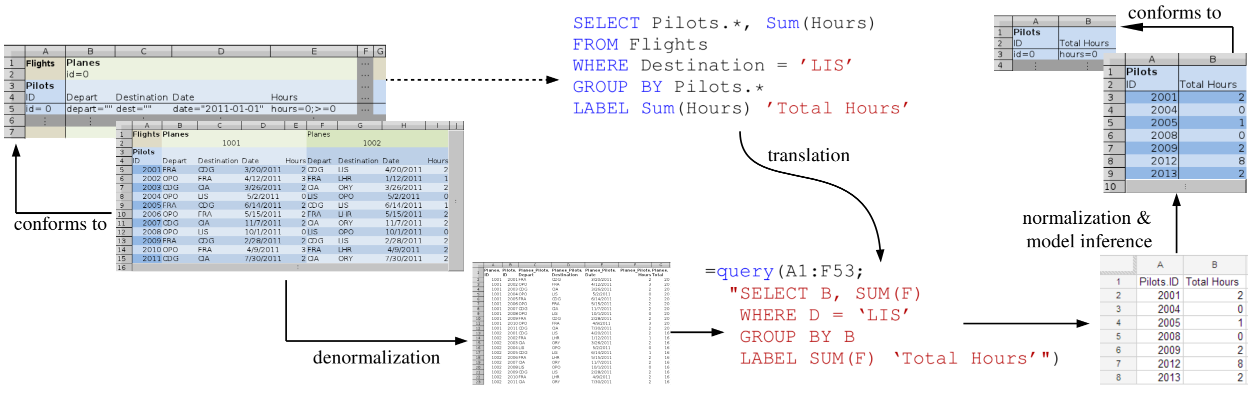
- *Humanized* - Now we can represent attributes and data areas (models) using human designated names, instead of column letters.

- *Understandable* - Now we can actually understand and easily read the queries, knowing exactly what they do
- *Robust* - Unless attributes in the query are removed or renamed, the queries can still correctly function even with spreadsheet data/model evolution
- *Productive* - No need to manually think through what spreadsheet area our data is in, or what column letter is a given attribute.

So while GQF does have its benefits such as using a Google made query engine to query the results, which anyone would have trouble trying to compete with, the end-user has a much more difficult time to get to the point where he/she can effortlessly view his/her results and manage his/her data.

5 MDQL: Design and Implementation

In this section we explain how the model-driven query language system we have created works. Figure 26 presents how the system functions. This system is implemented on top of MDSheet [Cunha et al., 2012d].



```
SELECT Pilots.*, Sum(Hours)
FROM Flights
WHERE Destination = 'LIS'
GROUP BY Pilots.*
LABEL Sum(Hours) 'Total Hours'
```

```
=query(A1:F53;
 "SELECT B, SUM(F)
  WHERE D = 'LIS'
  GROUP BY B
  LABEL SUM(F)  'Total Hours'")
```

Pilots ID	Total Hours
2001	2
2004	0
2005	1
2008	0
2009	2
2012	8
2013	2

Figure 26: *The Model-Driven query system*

This means that all the mechanisms to handle models and instances are already created and ready to use. This is the starting point: in the left part of the figure we show a spreadsheet instance and its corresponding model. The second required part is the query over the model/instance. This will be explained in detail in the next Subsection 5.1. The spreadsheet instance is then denormalized, as we will explain in Subsection 5.2, and the query over the model is translated into a Google query, as explained in Subsection 5.3. The Google query and the denormalized data are sent to Google and the result received is shown in the bottom-right part of the figure, described in Subsection 5.4. Finally, a new model is inferred so the result can be used as input to a new query, as explained in Subsection 5.5. This last step is necessary since we want the queries to be composable. This chapter concludes with a brief explanation of the integration done into MDSheet and OpenOffice.

5.1 Model-Driven Query Language

The Model-Driven Query Language (MDQL) is very similar to the standard SQL language, while also allowing some of the GQF's clauses such as LIMIT and LABEL. To create the MDQL, we used advanced langauge engineering techniques from [Lämmel et al., 2006], and wrote the grammar/parser using ANTLR[5], a computer-based language recognition parser generator that uses LL(*) parsing, taking as input a grammar that specifies a language and generates as output source code for a recognizer for that language (in this case Java), while also allowing the generation of lexers, parsers, tree parsers, and combined lexer-parsers.

We then chose to use Tom[6], a stable software environment for defining transformations, manipulating tree structures and providing pattern matching to inspect objects and retrieve values. Additionally, since we used Tom along with Java, we were provided a generator of efficient object oriented tree based data-structures, named Gom, and a strategy language that could be used to control transformations, which helped us greatly when translating our query language to Google's.

With these tools, I was able to use advanced compiler techniques such as:

- generalized top-down parser (ANTLR) [Parr and Quong, 1995]
- generalized abstract trees (Tom) [Moreau et al., 2003]
- generalized tree traversals in the form of strategic programming [Visser and Saraiva, 2004]

Both the MDQL grammar and tree based data-structures (Gom) can be found in Appendix A and B respectively. Its syntax is shown in Listing 7. As we can see, instead of selecting column letters in the SELECT clause, the user can select the ClassSheet attributes he/she wishes to query, while also allowing him/her to further specify, as to avoid any naming conflicts which may occur, alternative ways of naming the attribute such as:

- stating its name - (**Destination**)
- stating the attribute along with its classes' name (**Pilots.ID**)
- stating both classes (**(Pilots,Planes).Destination** or **(Planes,Pilots).Destination**)
- stating all the attributes in a given class (**Planes.***)

[5]ANTLR: `http://www.antlr.org/`
[6]Tom: `http://tom.loria.fr/`

Listing 7: *Part of the model-driven query language syntax*

```
SELECT [DISTINCT] (* | attr₁, ...,Agg(attrX), ...)
 [FROM ClassSheet1, [JOIN ClassSheet2], ...]
 [WHERE conditions]
 [GROUP BY attr₁, ...]
 [ORDER BY attr₁ [ASC|DESC], ...]
 [LIMIT numRow]
 [LABEL attr₁ 'new_attr₁', ...]
 [WITH HISTOGRAM]
attr ::= attribute
     | Class.*
     | (Class1, Class2).*
     | Class.attribute
     | (Class1, Class2).attribute
```

The MDQL also has a FROM clause, very reminiscent of the standard SQL FROM clause where the user chooses what tables he/she wishes to query, which allows the user to choose which ClassSheet model(s) to use for the query, in cases where more than one ClassSheet is present in a worksheet. Also note that as in SQL we allow JOIN operations between two ClassSheets, nonexistent in the GQF. We also have the LIMIT clause to limit the amount of results returned by a given number, and LABEL to rename attributes to a given name, both originating from GQF clauses. The WHERE, GROUP BY, and ORDER BY clause works the same as in SQL, applying filters such as where an attribute is equal to a given name (ex. Pilot.Name = 'Bob'), grouping values to apply an aggregation function, and ordering by a given attribute either ascendant (ASC) or descendant (DESC), all three respectively. Finally, the DISTINCT clause was also implemented, also nonexistent in the GQF, to remove duplicated results which may occur, and WITH HISTOGRAM is used to state if the user wishes the results produce a histogram chart to visually show the results.

Since this language is very similar to SQL, it allows users who already know basic SQL to simply jump into query writing in this system, avoiding the need to learn a new language, allowing us to adapt the most used query language instead of creating one, while also allowing queries to be more elegant, concise, robust and understandable for spreadsheets, along with being easy to learn since the SQL-language is often described as ”English-like” because many of its statements read like English [Melton, 1998].

5.2 Denormalization of Spreadsheet Data

As previously mentioned several times, to be able to use the GQF, the data must be in a single matrix format, with the headers present in the first row. In consequence of this restriction, for a user to be able to write all the queries possible with his data, every bit of data from the spreadsheet has to be written in this single matrix structure. To do so, the data has to be in a redundant state, combining the data from multiple ClassSheet classes together, reminiscent of a JOIN between tables in databases, duplicating the data in such a way that every bit of information, represented in the data, is able to be read. In other words, we have to denormalize our spreadsheet data [Shin and Sanders, 2006].

To correctly do this, we must first obtain all the necessary and critical information from the ClassSheet models, and their attributes/data. To begin, we obtain this information from the MDSheet framework (atleast in this context), such as which ClassSheet classes exist, their names, their expansion direction (horizontally, vertically, both, or none at all) and most importantly the attributes in each class. So, if we look back on Figure 21 (which now will be our running example for this sub-section) we would have:

- **Class** = Planes ; **Expansion** = Horizontal ; **Attributes** = {Id}
- **Class** = Pilots ; **Expansion** = Vertical ; **Attributes** = {Id}
- **Class** = Planes_Pilots ; **Expansion** = Both ; **Attributes** = {departure, destination, date, hours}

Now that the classes and their attributes are organized together, the actual data for each attribute can now be obtained. Using a function in MDSheet which has an input of a class and attribute name, and returns the data values for that attribute, we can now also add the data to each attribute in the classes. So for example, doing so for only the first attribute in each class (shown above) we would have the following data sets:

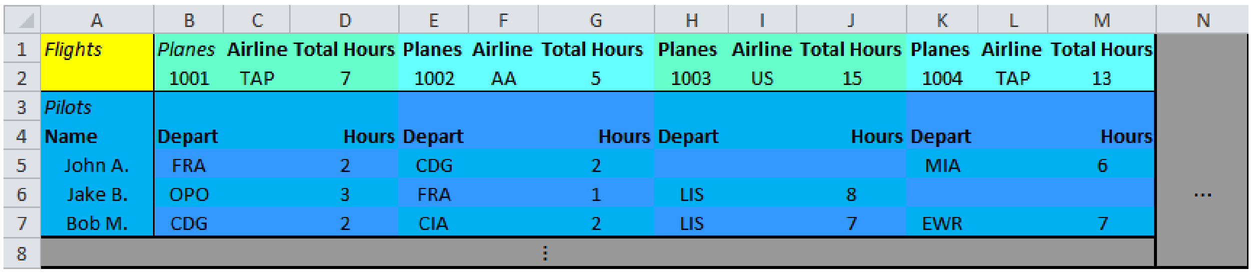
- **Class** = Planes ; **Attribute**[id] = {1001, 1002, 2050, 2051}
- **Class** = Pilots ; **Attribute**[id] = {2001, 2002, 2003, ..., 2112, 2113, 2114}
- **Class** = Planes_Pilots ; **Attribute**[departure] = {FRA, OPO, CDG, OPO, FRA, OPO, ..., EWR, LAS, LAX, LAX}

What is not known to the user, is that each piece of attribute data also has information regarding what Horizontal and Vertical (expanding) instance it belongs to. To illustrate what this means, take for example the data from cells **B5;C5;D5;E5** (Figure 20), each one would have their horizontal instance (Planes) as 0 and vertical instance (Pilots) as 0, this means they have a relationship with the first (counting from 0 onward) instance of the Plane and Pilot, ID = 1001 and ID = 2001 respectively. Another example, if we look at cells **J12;K12;L12;M12**, they would have their horizontal instance as 2 and vertical instance as 7 respectively, signifying a relationship with (Planes) ID = 2050 and (Pilots) ID = 2008. This information regarding the instances they belong to is extremely important, since this can be used to correctly and carefully group the information from other classes in their rightful spot, almost as if these were the foreign keys in a database, guaranteeing consistent information.

Finally, we join the classes together, and form the matrix like structure, having all the attributes in the first row, and below each attribute the correct data. This way, each row represents one piece of information from the original spreadsheet data, now in its denormalized state.

5.2.1 Denormalization of Spreadsheet Data: Querying Problems

As stated before, we denormalize the data into a single table, but what was not stated is that the *problems* caused by denormalized data querying, such as *derived data* and *denormalized attribute aggregation* are automatically dealt with. In fact, these are well-known and well-studied problems in the database realm [Maier, 1983]. To show and explain some of the possible problems, a modified and smaller sample flight spreadsheet will be used, as shown in Figure 27.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	*Flights*	*Planes*	Airline	Total Hours	Planes	Airline	Total Hours	Planes	Airline	Total Hours	Planes	Airline	Total Hours	
2		1001	TAP	7	1002	AA	5	1003	US	15	1004	TAP	13	
3	*Pilots*													
4	Name	Depart		Hours	Depart		Hours	Depart		Hours	Depart		Hours	
5	John A.	FRA		2	CDG		2				MIA		6	
6	Jake B.	OPO		3	FRA		1	LIS		8				...
7	Bob M.	CDG		2	CIA		2	LIS		7	EWR		7	
8						⋮								

Figure 27: *Modified spreadsheet flight example to show some possible query problems*

Using Figure 20 for our base ClassSheet model, we changed the Pilot ID attribute to a Name attribute, added Airline (name) and Total Hours to the Planes class, and removed all but Departure city and Hours flown from the Planes_Pilots class. Table 3 shows how Figure 27 would be denormalized, giving us 6 column attributes, and 10 rows of information.

Table 3: *Modified spreadsheet denormalized*

ID	Airline	Total Hours	Name	Departure	Hours
1001	TAP	7	John A.	FRA	2
1001	TAP	7	Jake B.	OPO	3
1001	TAP	7	Bob M.	CDG	2
1002	AA	5	John A.	CDG	2
1002	AA	5	Jake B.	FRA	1
1002	AA	5	Bob M.	CIA	2
1003	US	15	Jake B.	LIS	8
1003	US	15	Bob M.	LIS	7
1004	TAP	13	John A.	MIA	6
1004	TAP	13	Bob M.	EWR	7

One of the problems which may occur would be that of querying one of the single expanding classes (non bi-direction where Expansion = Both), such as Pilots or Planes with this example, and using an aggregation. This is due to there being repeated information when the data is denormalized, the same problem would occur in the database realm if one were to join two tables, and query only one with an aggregation. To exemplify, lets say we want to answer these two questions:

- How many hours has each Airline flown?
- How many hours have all the planes flown?

These two queries could be answered respectively with:

Listing 8: *How many hours has each Airline flown?*

```
SELECT Airline, sum('total hours') FROM Flights GROUP BY Airline
```

Listing 9: *How many hours have all the planes flown?*

```
SELECT sum('total hours') FROM Flights
```

The correct responses should be:

Airline	sum(total hours)
TAP	20
AA	5
US	15

sum(total hours)
40

But what would be returned is:

Airline	sum(total hours)
TAP	47
AA	15
US	30

sum(total hours)
92

This is because the information is repeated, and when we run the aggregation on the **Total Hours** column, it just does its job and doesn't consider redundant or repeated information. To solve this, when parsing the query and this problematic case is detected, a

Table 4: *Modified spreadsheet denormalized with column controllers*

ID	Airline	Total Hours	Name	Departure	Hours	Planes.CC
1001	TAP	7	John A.	FRA	2	1
1001	TAP	7	Jake B.	OPO	3	0
1001	TAP	7	Bob M.	CDG	2	0
1002	AA	5	John A.	CDG	2	1
1002	AA	5	Jake B.	FRA	1	0
1002	AA	5	Bob M.	CIA	2	0
1003	US	2	Jake B.	LIS	8	1
1003	US	2	Bob M.	LIS	7	0
1004	TAP	17	John A.	MIA	6	1
1004	TAP	17	Bob M.	EWR	7	0

new column is added for the class in question, designated as a *column controller attribute*, where the attribute for each row is either 1 or 0. If the first instance of a class is detected, the attribute is placed as 1, else it is 0. To exemplify this, take a look at Table 4.

With this, we can now isolate the values, and by just automatically adding to the user's query *"WHERE Planes.CC = 1"*, we can correctly obtain the desired results.

Another problem which may occur with querying denormalized data, in this context of a model-driven environment, is with *derived data*. By this I mean any attribute which is calculated or derived from other data, in other words a spreadsheet formula. Using the same example, if we look closely, **Total Hours** can also be written with a spreadsheet formula on the business level, or ClassSheet model, instead of being a simple value. If we took that as our example, total hours could be written as **=sum(hours)** in the ClassSheet model. What this may bring, is a small level of ambiguity when writing a query aggregating that data, which is completely avoidable with correct query building and basic understanding of the model written. But just for explanation purposes, lets imagine the user wants to know:

- How many hours each Pilot has flown?
- How many hours were flown out of each city?

The correct way to write these two queries would be:

Listing 10: *How many hours has each Airline flown?*

```
SELECT name, sum(hours) FROM Flights GROUP BY name
```

Listing 11: *How many hours have all the planes flown?*

```
SELECT departure, sum(hours) FROM Flights GROUP BY departure
```

But what may happen is instead of using **hours** the user may use **total hours** since he/she believes it may be the same, considering how one is derived and dependent on the other. What happens when this case is detected, the query will be allowed to be executed, because there may be cases in which such a query is correct, and return a warning to the user stating it *might* not be exactly what he is trying to question. Initially, I did want to implement some sort of *intelligent* deduction to the queries, by this I mean allowing the parser to notice this particular problem and automatically change **sum(total hours)** to **sum(hours)** after analyzing the attribute and noticing how it depends on the hours attribute. But the moment the formula depends on more than one attribute, or is much more complicated, such a system would become considerably harder to develop, and in the end, it would constrain the user (since it would *force* the attribute change) and limit what he can or cannot query.

5.3 Translation to Google Query

The main reason we chose not to develop a new querying engine, but re-utilize the QUERY function's querying engine, is because we do not want to try to compete with Google in terms of performance and speed where Google has shown dominance in when developing querying engines. To properly run the GQF, our model-driven queries must adhere to the *Visualization API Query Language* [Google, 2013a], specified by Google. So, for our model-driven queries to function correctly, a translator was made to transform the model-driven queries written by a user to their exact equivalents for the GQF. To do so, we used the Gom tree based data-structures provided by Tom, and took advantage of the strategy language to control transformations and pattern matching, to translate and inspect the query respectively.

The translator automatically calculates the *range* from the ClassSheet models selected, in the FROM clause for example, by using a lookup function to find what is the new range of data after the denormalization process. It also substitutes the attribute names to their corresponding column letters in the denormalized data, without the user having to do so. After parsing the user's query, and verifying that each attribute chosen by the user exists, and has absolutely no conflicts, such as any ambiguous attribute names due to the attribute name repeating in more than one ClassSheet (which may be solved by adding the class name beforehand as shown in 5.1), we apply another lookup function on each attribute, and calculate the column letter corresponding to each attribute.

Now having both the denormalized data and translated model-driven query ready, we can send the spreadsheet data to Google Spreadsheets, run the GQF and afterwards retrieve the results for the user to view in an OpenOffice worksheet.

5.4 Google Spreadsheets

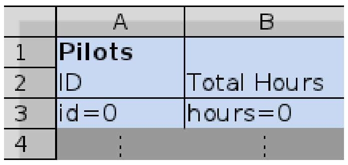
To be able to send the spreadsheet data to Google Spreadsheets and run the GQF, I turned to the *Google Spreadsheets API version 3.0* [Google, 2013b], an API which enables developers to be able to create applications that can read, write and modify the data in Google Spreadsheets. It allows us to manage the worksheets in a Google spreadsheet, manage cells in a worksheet by position, and also allows us to create spreadsheets, worksheets, insert and delete data, and retrieve a single worksheet or a spreadsheet, along with authorizing requests and authentication.

So before we acquire the query results, we begin by creating a temporary worksheet which will be filled with the denormalized data, followed by creating a second temporary worksheet where the query function string is sent to. The second the query function is inserted into a cell, it calculates the results, and now that second worksheet contains the query results. Finally, the results are retrieved, the temporary worksheets wiped and an inference technique is ran before presenting it to the user.

5.5 ClassSheet Inference

Originally, we did not envision nor plan on having the queried results return to the user with an associated model and the data conformed to it. But during the months that followed, we decided it would be beneficial to not only the user to continue working with the ClassSheet models, since the original data would be in the form of a ClassSheet model, but it would also allow us to make the queries composable, that is, to allow the output of a model-driven query as the input of another model-driven query. Without a model for the output of the GQF's result, this would not be possible.

Previous work in this field introduced a technique to automatically infer a ClassSheet model from spreadsheet data [Cunha et al., 2010]. Thus, applying this technique on the results obtained from the GQF, we can now infer the correct ClassSheet model and have it alongside the queried results. For example, applying the inference technique to the results from Query 2 presented in Table 2, we would obtain the ClassSheet model shown in Figure 28, and now present the user the results alongside its model.



	A	B
1	**Pilots**	
2	ID	Total Hours
3	id=0	hours=0
4	⋮	⋮

Figure 28: *Model automatically inferred from the spreadsheet data shown in Table 2*

5.6 Integration with MDSheet and OpenOffice

Along with the core work which was presented in the beginning of this chapter, further work was also done to link and integrate it to the MDSheet framework and OpenOffice.

The MDSheet framework uses more than one programming language, mainly C++ for its core, BASIC for the OpenOffice macros and interface, and Haskell for manipulating the spreadsheet data and ClassSheet models. Considering how the ClassSheet's data and information regarding the classes was needed such as the name, expansion direction, which attributes and classes it contained and the instances the data belonged to, some extra Haskell functions needed to be created to retrieve this extremely important and vital information for the denormalization process to be correctly done.

Along with Haskell, since I was integrating the work into MDSheet, a chunk of C++ code was needed to be written, mainly to use the functions and operations from the rest of the framework, to call upon the Haskell functions, to obtain the query from OpenOffice and the ClassSheet information from Haskell to send off to be parsed/translated, and denormalized respectively. But since Java was used for the core of my work, I would need to somehow bridge C++ code with Java code. To do so, I used the *Java Native Interface*, or JNI [7].

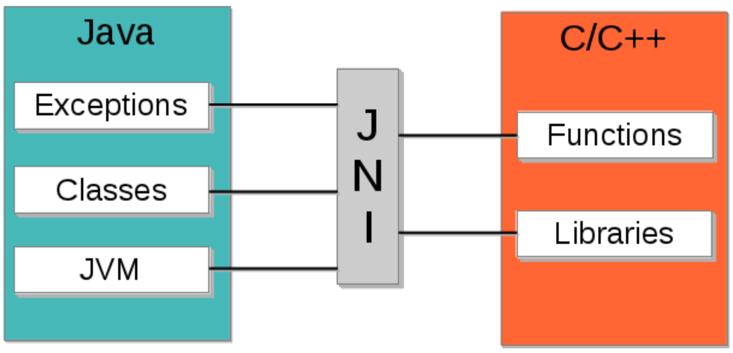


Figure 29: *JNI connecting C/C++ with Java*

JNI [Oracle, 2013] is a programming interface for writing Java native methods and embedding the Java Virtual Machine (JVM) into native applications such as C or C++. Using JNI now made it possible to convert the C++ primitive types into Java classes and types, to invoke Java methods and to convert Java classes and types back into C++ primitives.

To finalize, some BASIC code was written in OpenOffice to create a button to launch a window for the user to write his/her query along with Execute and Cancel buttons, and the necessary code to send this information to C++ to execute.

[7]JNI: `http://docs.oracle.com/javase/6/docs/technotes/guides/jni/`

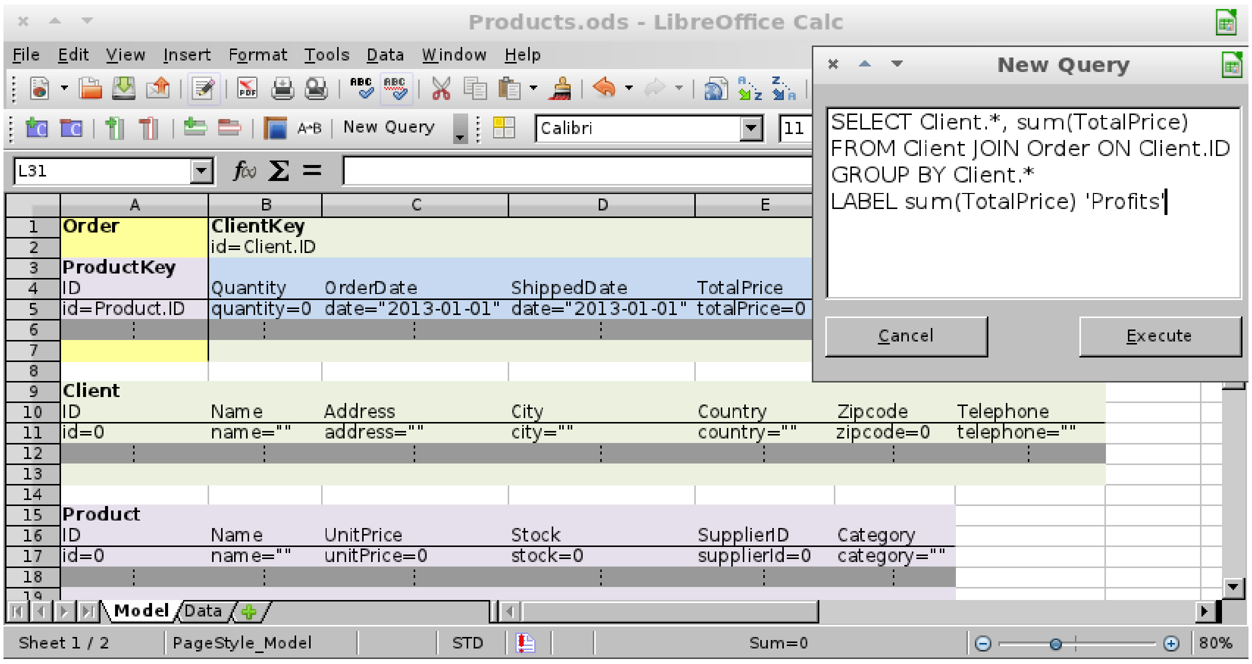
6 QuerySheet

The model-driven query language and the techniques proposed in the previous section (5) and in [Cunha et al., 2013b] were the building blocks used to join it all into one single tool, integrated in MDSheet and OpenOffice, named *QuerySheet* [Belo et al., 2013].

To demonstrate *QuerySheet*, we will be using the ClassSheet model shown on the left in Figure 30 to describe the business model, and as expected its data, is in conformance with the model.

Suppose that we would like to know the following:

- How much have we profited from each client?
- How much have we profited from USA clients (with its histogram)?



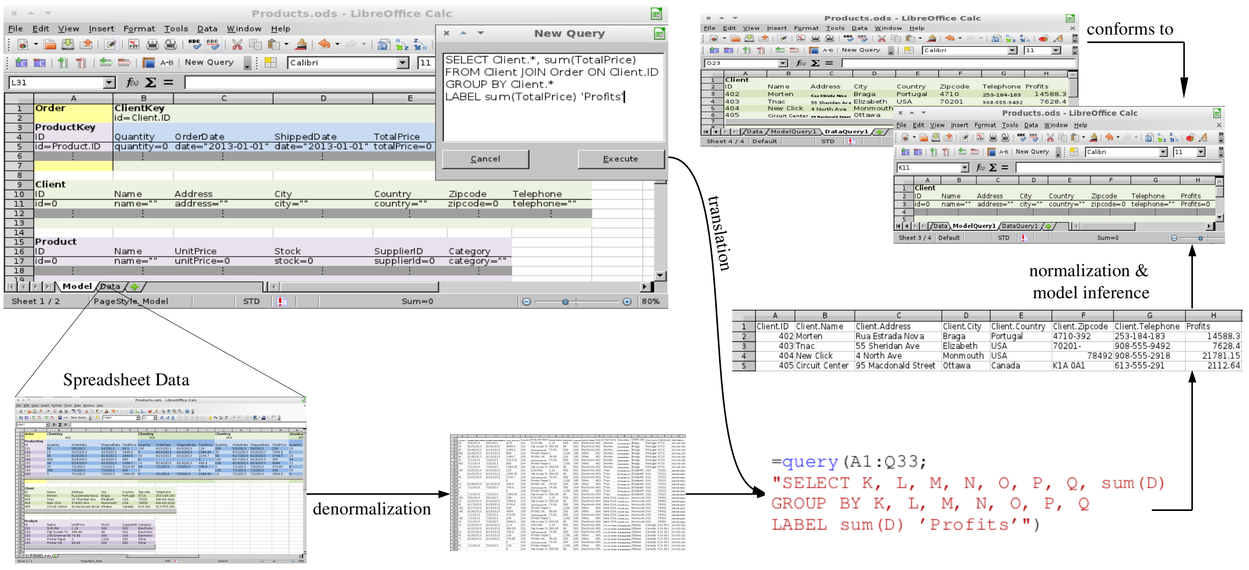
```
SELECT Client.*, sum(TotalPrice)
FROM Client JOIN Order ON Client.ID
GROUP BY Client.*
LABEL sum(TotalPrice) 'Profits'
```

	A	B	C	D	E		
1	**Order**	**ClientKey**					
2		id=Client.ID					
3	**ProductKey**						
4	ID	Quantity	OrderDate	ShippedDate	TotalPrice		
5	id=Product.ID	quantity=0	date="2013-01-01"	date="2013-01-01"	totalPrice=0		
6	⋮	⋮	⋮	⋮			
7							
8							
9	**Client**						
10	ID	Name	Address	City	Country	Zipcode	Telephone
11	id=0	name=""	address=""	city=""	country=""	zipcode=0	telephone=""
12	⋮	⋮	⋮	⋮	⋮	⋮	⋮
13							
14							
15	**Product**						
16	ID	Name	UnitPrice	Stock	SupplierID	Category	
17	id=0	name=""	unitPrice=0	stock=0	supplierId=0	category=""	
18	⋮	⋮	⋮	⋮	⋮	⋮	

Figure 30: *A model-driven spreadsheet representing orders, clients and products*

In *QuerySheet*, we can express the query based on the ClassSheet model. The tool provides a *New Query* button, which opens a text box to allow the user to define a query. As we can see in Figure 30 on the top right, we have the query for our first question, and as expected, the query looks very much like SQL, using the same keywords and syntactic structure. Moreover, we now use the ClassSheet entities to identify the attributes to be queried.

When executing the query, *QuerySheet* passes through all the phases explained in Section 5 and shown in Figure 26, while also generating the result as a ClassSheet-driven spreadsheet. As a matter of fact, two new worksheets are added to the original spreadsheet: one containing the spreadsheet data that results from the query (DataQuery1), and the other contains the ClassSheet model (ModelQuery1), shown in Figure 32. This whole process is depicted in Figure 31.



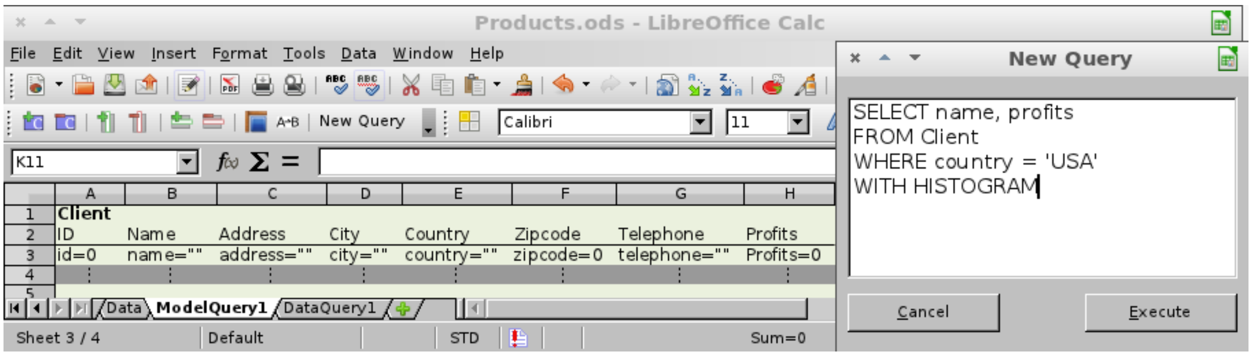
```
SELECT Client.*, sum(TotalPrice)
FROM Client JOIN Order ON Client.ID
GROUP BY Client.*
LABEL sum(TotalPrice) 'Profits'
```

	A	B	C	D	E	F	G	H
1	Client.ID	Client.Name	Client.Address	Client.City	Client.Country	Client.Zipcode	Client.Telephone	Profits
2	402	Morten	Rua Estrada Nova	Braga	Portugal	4710-392	253-184-183	14588.3
3	403	Tnac	55 Sheridan Ave	Elizabeth	USA	70201-	908-555-9492	7628.4
4	404	New Click	4 North Ave	Monmouth	USA	78492	908-555-2918	21781.15
5	405	Circuit Center	95 Macdonald Street	Ottawa	Canada	K1A 0A1	613-555-291	2112.64

```
=query(A1:Q33;
"SELECT K, L, M, N, O, P, Q, sum(D)
GROUP BY K, L, M, N, O, P, Q
LABEL sum(D) 'Profits'")
```

Figure 31: *The architecture of* QuerySheet

Now, since we generate a new model for the results, we may write composable queries. We can take advantage of this, and write our second query based on the first query's results! Once again we click on the *New Query* button and write our second query, shown in the top right of Figure 32.

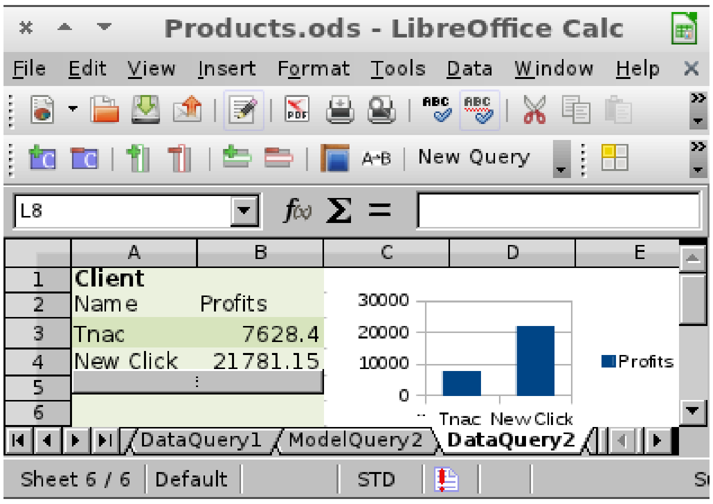


	A	B	C	D	E	F	G	H
1	Client							
2	ID	Name	Address	City	Country	Zipcode	Telephone	Profits
3	id=0	name=""	address=""	city=""	country=""	zipcode=0	telephone=""	Profits=0

```
SELECT name, profits
FROM Client
WHERE country = 'USA'
WITH HISTOGRAM
```

Figure 32: *The model-driven spreadsheet produced after executing the first query*

Yet again, once executed, *QuerySheet* passes through all the phases and generates another ClassSheet-driven spreadsheet, one containing the spreadsheet data with a histogram (DataQuery2), and the other containing the ClassSheet model (ModelQuery2). The results are shown in Figure 33.



	A	B
1	Client	
2	Name	Profits
3	Tnac	7628.4
4	New Click	21781.15

Figure 33: *The spreadsheet (data) result after executing the second query*

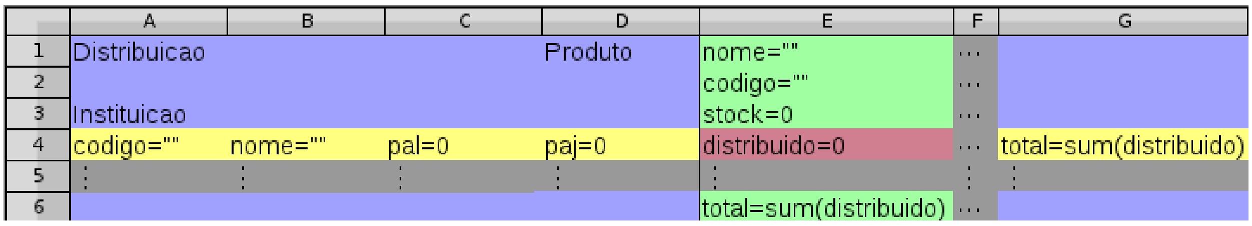
7 Empirical Evaluation

To validate my thesis work, a preliminary study was planned and executed, to obtain results of end-user's experiences, productivity, and feedback.

For this preliminary study, we used seven students, all with basic or minimal knowledge of SQL, who are studying informatics/computer sciences, ranging from licentiate to PhD students. For this study, they were taught how to use Google's QUERY function and the QuerySheet system, followed by practicing with a series of exercises using both systems. When the users were comfortable with each system, the actual study was performed.

In the actual study, a real-life spreadsheet was used, which we obtained, with permission to use, from the local food bank in our hometown of Braga, Portugal. We then explained to the students how the information was represented, and how to properly read the spreadsheet, in this case, information regarding distributions of basic products and institutions. This specific 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 (since we wanted to study the end-user's interaction with the two different systems, and already knew that denormalizing over 1000 lines of information would take a long time), and also prepared the spreadsheet model and conformed instance in the MDSheet environment. Since we can not show the actual spreadsheet due to revealing private information, the spreadsheet model (the same one used in the study) is presented below in Figure 34.



	A	B	C	D	E	F	G
1	Distribuicao			Produto	nome=""	⋯	
2					codigo=""	⋯	
3	Instituicao				stock=0	⋯	
4	codigo=""	nome=""	pal=0	paj=0	distribuido=0	⋯	total=sum(distribuido)
5	⋮	⋮	⋮	⋮	⋮	⋮	⋮
6					total=sum(distribuido)	⋯	

Figure 34: *A model-driven spreadsheet representing institutions, products, and distributions, used in the empirical evaluation*

As we can see, in the model, and hence the actual spreadsheet, the **Distribuição** (Distribution) class is composed of a **Instituição** (Institution) class and a **Produto** (Product) class. The **Instituição** class has its **Código** (Institution's Code), **Nome** (Intitution's Name), **PAL** (units used for lunch and snacks) and **PAJ** (units used for lunch and dinner). The **Produto** class has a **Nome** (Product's Name), a **Código** (Product's Code), and **Stock** which represents the amount of that specific product they have in stock. The relationship between both classes gives us information on the quantity **Distribuido** (Distributed) of a specific **Produto** to a specific **Instituição**. Along with that, both **Produto** and **Instituição** have a **Total** attribute which represents how many units were distributed of each and to each respectively.

For the study, a series of four questions were asked to the students, regarding the information present in the distributions spreadsheet. For each question, they would answer it using Google's QUERY function, and the QuerySheet system, alternating between starting with one then the other (the starting system would also alternate between students, so one would begin alternating starting with QuerySheet, and another would begin alternating starting with Google's QUERY function). The students were asked to write down the time after carefully reading each question, and the time after the queries were executed with no errors (the correctness of the queries and results were analyzed afterwards), repeating for each system, so they would read the question, write down initial time, write down concluding time, and repeat starting with reading the question once again.

Along with writing down the time, after each question, and having answered it using both systems, the students were asked to choose which system they felt was more:

- Intuitive
- Faster (to write the queries)
- Easier (to write the queries)
- Understandable (being able to explain and understand the written queries)

After finishing answering the questions, the students answered which system they preferred and why, and what advantages/disadvantage existed between the systems.

The results were gathered, and analyzed. Below is a graph, detailing the information gathered from the study.

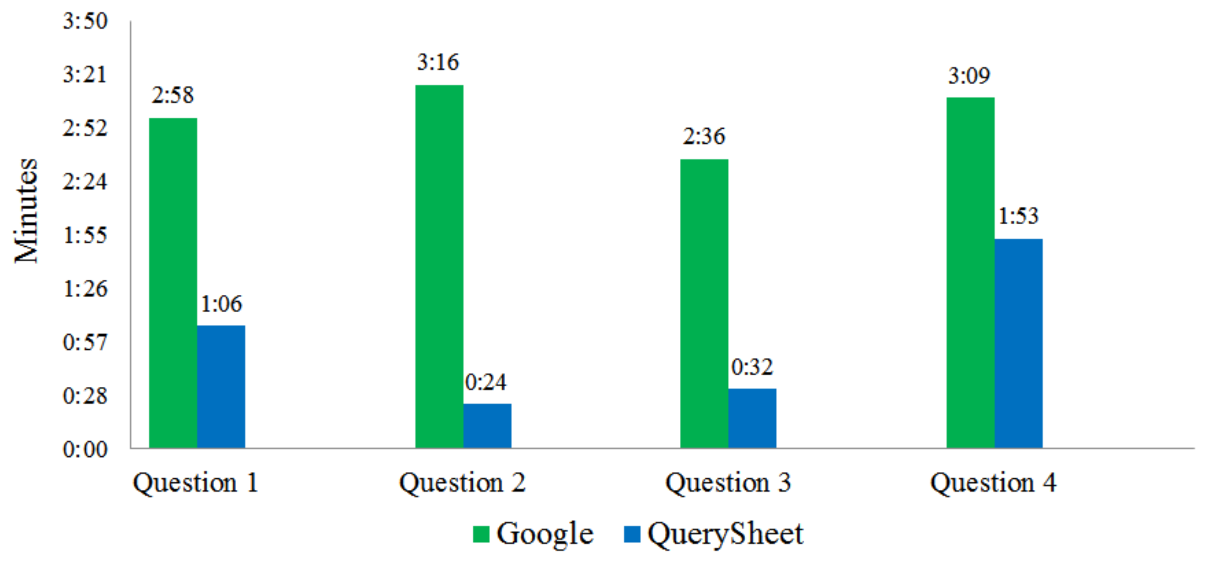


Figure 35: *A chart detailing the information gathered from the empirical evaluation*

The left side (Y-Axis) represents the 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 QuerySheet system.

As we can see, users using the QuerySheet system spent significantly less time to write the queries to answer the questions, ranging from as much as 90% less to 40% less, averaging out to 68% faster! Regarding the system they felt was more Intuitive, Faster, Easier, and Understandable, almost all chose the QuerySheet system. We also analyzed the results and queries written, and in the cases where the queries/results were incorrect, almost all were with the Google QUERY function system, ranging from incorrect column letters chosen, to incorrect ranges. Furthermore, the written questions at the end also gave us positive feedback. Users stated that using the QuerySheet system was much easier to write the query, being able to look at the model to understand the logic behind the information, and not having to deal with calculating the ranges, or worry about positing of information, while being easier to understand what is being written and in turn was more intuitive.

With the user feedback, we were also able to understand what is still needed in QuerySheet, such as having a way to store the previous queries for future use. Along with the direct user feedback, we also realized that a basic knowledge of SQL is needed, as expected, to be able to correctly answer the questions. Users who incorrectly wrote queries in the QuerySheet system always incorrectly wrote them in Google's QUERY function, due to bad query construction. One of the comments received was to have an interface to build the query visually and not descriptively written, something we already believed would be helpful and needed for a user not used to SQL writing, and further justifies the future work on building a Query Wizard like interface.

Obviously though, this was only a preliminary study, as preliminary studies are used to fine-tune the questions/exercises so that a more elaborated study is conducted with more end-users, as advocated/explained in [Wohlin et al., 2012], a book about Empirical Studies. Thus, a larger study is planned to be conducted, using students who have above basic SQL knowledge. Since this was only a preliminary study, basically being a first stepping stone to validating the system, we can adjust what we felt is needed for the study, and can use a much larger group, to further validate if the results can be repeated (averaging 68%). If the results are repeated, for better or for worse, we can have a much better understanding of a user's experience with the QuerySheet system, and compare it to Google's QUERY function.

8 Conclusion

Throughout my Thesis investigation and work, I tried to develop a framework as closely to what was envisioned (shown in Section 4 - Querying Model-Driven Spreadsheets) as possible, initially ignoring what challenges and technical difficulties might of occurred to implement such a query system for model-driven spreadsheets, focusing primarily on how expressive, friendly, readable, and intuitive the queries would be to the users. This focus on the user's experience with the query system made it so that some technical challenges were faced during development, such as integrating the query system framework into *MDSheet* while having to use JNI, programming what was necessary to denormalize ClassSheet models, and even using the *Google Spreadsheet API*.

But what blossomed out of this focus and approach was *QuerySheet* (explained in detail in Section 6 - QuerySheet), which even surpassed our original vision of a model-driven query system, allowing histogram charts to be generated using the query language, permitting queries to be written on models from previous query results, and even automatically dealing with various problems which frequently occur due to querying denormalized data (some examples shown in Subsection 5.2.1 - Denormalization of Spreadsheet Data: Querying Problems). With this, the four main points we hoped to originally achieve when imagining such a query system for model-driven spreadsheets are finally fulfilled.

Our query system now allows querying spreadsheets to be more:

- **Humanized**
- **Understandable**
- **Robust**
- **Productive**

8.1 Research Questions Answered

In the beginning of my Thesis (Subsection 1.2 - Research Questions) I presented three important questions that appeared during my Thesis work, which were relative to the design and implementation of a MDQL, and users' productivity using such a system. Now with the conclusion of my Thesis, I can answer these questions.

Q1 *Would it be possible to create a query language for use on spreadsheet models which can be translated into Google's QUERY function language?*

A1 Yes, using the tools from Tom, a software environment for defining transformations and providing pattern matching, the Gom data-structures, and a lookup function which upon receiving the model-driven attribute full-name, would return the column letter position where that information is present in the denormalized data sheet, it was possible to effortlessly translate the model-driven queries to Google's QUERY function language,

Q2 *If using Google's QUERY function, what extra work and implementation must be done to correctly query a model-driven spreadsheet?*

A2 Without considering the extra work put into automatically resolving the problems which would appear from querying denormalized data (since this was not necessarily needed to correctly use the Google QUERY function), the extra work and implementation needed to query a model-driven spreadsheet is divided into two parts: first is the very important denormalization of the model-driven queries into the singular matrix like structure, with all the attributes represented (always in the column header), and all the possible information from the models present, and second was the actual sending of the (denormalized) data and the now translated Google QUERY to Google Spreadsheets to run the query and then retrieve the results. This second part was done using the *Google Spreadsheets API version 3.0.*

Q3 *How efficient and productive is querying Model-Driven spreadsheets, when compared to directly querying spreadsheet data (e.g. MS-Query or Google's QUERY function)?*

A3 After running a preliminary empirical evaluation, we calculated that users spent roughly 68% less time to answer a series of questions, on a real-life spreadsheet, when compared to using Google's QUERY function. We also received feedback from the users who stated that QuerySheet was much more intuitive, faster to write, easier to write, and more understandable, in almost all the cases!

8.2 Results

Along with this dissertation, my thesis work gave fruit to three publications (Authors in alphabetical order):

- **Querying Model-Driven Spreadsheets**, Jácome Cunha, João Paulo Fernandes, Jorge Mendes, Rui Pereira, and João Saraiva. In the proceedings of the *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'13)*, San Jose, CA, USA, September 15–19, 2013. IEEE Computer Society.
  Conference in Information Systems, CORE ranking Class A, indexed by SCOPUS, acceptance rate 33%

- **QuerySheet: A Bidirectional Query Environment for Model-Driven Spreadsheets**, Orlando Belo, Jácome Cunha, João Fernandes, Jorge Mendes, Rui Pereira, and João Saraiva. In proceedings of the *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2013)*, San Jose, CA, USA. IEEE Computer Society, September 2013.
  Conference in Information Systems, CORE ranking Class A, indexed by SCOPUS, acceptance rate 33%

- **Spreadsheet Querying**, Rui Pereira. In proceedings of the 5^{th} *International Summer school on Domain Specific Languages (DSL'2013)*, Cluj-Napoca, Romania, 2013. (to appear)

- **Graphical Querying of Model-Driven Spreadsheets**, Jácome Cunha, João Paulo Fernandes, Rui Pereira, and João Saraiva. In proceedings of the 16^{th} *International Conference on Human-Computer Interaction (HCI2014)*, Heraklion, Crete, Greece, June 2014. (submitted extended abstract)

During the 14th-19th of September, I also went to San Jose to present my work at the VL/HCC'2013 conference and went to the Domain Specific Languages 2013 summer school, in Cluj-Napoca, Romania, to present this same work in the PhD workshop.

8.3 Future Work

Even with the good results and responses in regards to the work already completed, some interesting and important parts can be picked up for future work and research, namely:

- *Incremental Denormalization* - Currently, each time a user executes a query, the data is denormalized on-the-spot. A possible way to improve this is to have it so that this full on denormalization is done only once in the beginning, and further changes to data and/or models are changed incrementally, either during the changes, or in the next query execution.

- *Syntax Highlighting* - A common feature in existent query systems, and almost any programming language IDE, is syntax highlighting. Currently, the text window in which the user writes the query has no syntax highlighting, due to certain limitations in OpenOffice and what it provides for interface building. A possible way around this, would be to use a different design interface such as Java's SWING to design the interface window, and provide syntax highlighting.

- *Smell Detecting* - One interesting way to take advantage of a query language for spreadsheets, is to use it for detecting smells in spreadsheets, such as incorporating previous work such as *SmellSheet Detective: A Tool for Detecting Bad Smells in Spreadsheets* [Cunha et al., 2012b], which utilizes an initial catalog for spreadsheet smells [Cunha et al., 2012a], much like Fowler's bad smell catalog for code [Fowler, 1999]. With this, it would be possible and desirable to use these tools for easy access by users, allowing on-the-spot results of their spreadsheets. A user would be able to check for a specific bad smell which may exist in his or her spreadsheet, before having to handle critical data, possibly avoiding severe losses before even beginning to touch the data. An example could be a smell query, which the user may write on-the-spot, specifying what smells, if not all, does he want to check for in a certain spreadsheet, worksheet, or any variation, while having returned to him visually, the results found, possibly indicating exactly what cells may have or bring about problems, utilizing a color range to state the severity of the smell, from low to high.

- *Query Builder Wizard* - One of the nice things about MS-Query, is it provided users with a query building wizard for extremely inexperienced users, to easily build the query without manually writing it verbatim. Borrowing this idea, we could make it so it would present all the attributes and ClassSheet models in a drop down list, and other features such as the ones present in Subsection 2.1.1 - MS-Query Tool.

- *Query Synchronization with Original Data* - An extremely interesting topic which can bring in another big level of functionality to the framework, would be synchronization with the query results and original data. By this I mean, allowing a user to for example update the information of one of his/her employees from a previous query result (ex. "All employees who have worked here over 10 years), and in turn this update would reflect upon the original data which the results came from. Acting almost as if the results were a View Table on the original spreadsheet data, possibly using techniques from [Bohannon et al., 2006] regarding ways to solve the update-view problems.

References

[Abraham et al., 2005] Abraham, R., Erwig, M., Kollmansberger, S., and Seifert, E. (2005). Visual specifications of correct spreadsheets. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, VLHCC '05, pages 189–196, Washington, DC, USA. IEEE Computer Society.

[Bals et al., 2007] Bals, J.-C., Christ, F., Engels, G., and Erwig, M. (2007). Classsheets - model-based, object-oriented design of spreadsheet applications. In *Proceedings of the TOOLS Europe Conference (TOOLS 2007), Zürich (Swiss)*, volume 6, pages 383–398. Journal of Object Technology.

[Beckwith et al., 2011a] Beckwith, L., Cunha, J., ao Paulo Fernandes, J., and Saraiva, J. (2011a). End-users productivity in model-based spreadsheets: An empirical study. In *Proceedings of the Third International Symposium on End-User Development*, IS-EUD '11, pages 282–288.

[Beckwith et al., 2011b] Beckwith, L., Cunha, J., Fernandes, J. P., and Saraiva, J. (2011b). An empirical study on end-users productivity using model-based spreadsheets. *CoRR*, abs/1112.4190.

[Belo et al., 2013] Belo, O., Cunha, J., Fernandes, J. a. P., Mendes, J., Pereira, R., and Saraiva, J. a. (2013). Querysheet: A bidirectional query environment for model-driven spreadsheets. In *Proceedings of the 2013 IEEE Symposium on Visual Languages and Human-Centric Computing*, VLHCC '13, San Jose, CA, USA. IEEE Computer Society.

[Bézivin, 2005] Bézivin, J. (2005). Model driven engineering: An emerging technical space. In [Lämmel et al., 2006], pages 36–64.

[Bohannon et al., 2006] Bohannon, A., Vaughan, J. A., and Pierce, B. C. (2006). Relational lenses: A language for updateable views. In *Principles of Database Systems (PODS)*. Extended version available as University of Pennsylvania technical report MS-CIS-05-27.

[Chambers and Scaffidi, 2010] Chambers, C. and Scaffidi, C. (2010). Struggling to excel: A field study of challenges faced by spreadsheet users. In Hundhausen, C. D., Pietriga, E., Díaz, P., and Rosson, M. B., editors, *VL/HCC*, pages 187–194. IEEE.

[Cunha, 2011] Cunha, J. (2011). *Model-based Spreadsheet Engineering*. PhD thesis, University of Minho.

[Cunha et al., 2010] Cunha, J., Erwig, M., and Saraiva, J. (2010). Automatically inferring classsheet models from spreadsheets. In *IEEE Symp. on Visual Languages and Human-Centric Computing*, pages 93–100. IEEE CS.

[Cunha et al., 2012a] Cunha, J., Fernandes, J. a. P., Ribeiro, H., and Saraiva, J. a. (2012a). Towards a catalog of spreadsheet smells. In *Proceedings of the 12th international conference on Computational Science and Its Applications - Volume Part IV*, ICCSA'12, pages 202–216, Berlin, Heidelberg. Springer-Verlag.

[Cunha et al., 2012b] Cunha, J., Fernandes, J. P., Martins, P., Mendes, J., and Saraiva, J. (2012b). Smellsheet detective: A tool for detecting bad smells in spreadsheets. In Erwig, M., Stapleton, G., and Costagliola, G., editors, *VL/HCC*, pages 243–244. IEEE.

[Cunha et al., 2012c] Cunha, J., Fernandes, J. P., Mendes, J., Pacheco, H., and Saraiva, J. (2012c). Bidirectional transformation of model-driven spreadsheets. In *ICMT '12*, volume 7307 of *LNCS*, pages 105–120. Springer.

[Cunha et al., 2012d] Cunha, J., Fernandes, J. P., Mendes, J., and Saraiva, J. (2012d). MDSheet: A framework for model-driven spreadsheet engineering. In *Proc. of the 34rd Int. Conf. on Software Engineering*, pages 1412–1415. ACM.

[Cunha et al., 2013a] Cunha, J., Fernandes, J. P., Mendes, J., Visser, J., and Saraiva, J. (2013a). Embedding, evolution and validation of spreadsheet models in spreadsheet systems. *ACM Transactions on Software Engineering and Methodology*. submitted.

[Cunha et al., 2013b] Cunha, J., Mendes, J., Fernandes, J. P., Pereira, R., and Saraiva, J. (2013b). Querying model-driven spreadsheets. In *IEEE Symp. on Visual Lang. and Human-Centric Comp.*, San Jose, CA, USA. IEEE CS.

[Cunha et al., 2011] Cunha, J., Mendes, J., Fernandes, J. P., and Saraiva, J. (2011). Embedding and evolution of spreadsheet models in spreadsheet systems. In *IEEE Symp. on Visual Lang. and Human-Centric Comp.*, pages 186–201. IEEE.

[Cunha et al., 2012e] Cunha, J., Saraiva, J., and Visser, J. (2012e). Model-based programming environments for spreadsheets. In *Programming Languages*, pages 117–133. Springer.

[Cunha et al., 2009] Cunha, J., Saraiva, J. a., and Visser, J. (2009). Discovery-based edit assistance for spreadsheets. In *Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, VLHCC '09, pages 233–237, Washington, DC, USA. IEEE Computer Society.

[Engels and Erwig, 2005] Engels, G. and Erwig, M. (2005). Classsheets: automatic generation of spreadsheet applications from object-oriented specifications. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 124–133, New York, NY, USA. ACM.

[Fowler, 1999] Fowler, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Google, 2013a] Google (2013a). Google query function. `http://goo.gl/p0FKW`. [Accessed on March 2013].

[Google, 2013b] Google (2013b). Google spreadsheet api. `https://developers.google.com/google-apps/spreadsheets`. [Accessed on March 2013].

[Hermans et al., 2010] Hermans, F., Pinzger, M., and van Deursen, A. (2010). Automatically extracting class diagrams from spreadsheets. In *Proceedings of the 24th European* *conference on Object-oriented programming*, ECOOP'10, pages 52–75, Berlin, Heidelberg. Springer-Verlag.

[Ireson-Paine, 1997] Ireson-Paine, J. (1997). Model master: an object-oriented spreadsheet front-end. *Computer-Aided Learning using Technology in Economies and Business Education*.

[Ko et al., 2011] Ko, A. J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., et al. (2011). The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)*, 43(3):21.

[Lämmel et al., 2006] Lämmel, R., Saraiva, J., and Visser, J., editors (2006). *Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers*, volume 4143 of *Lecture Notes in Computer Science*. Springer.

[Maier, 1983] Maier, D. (1983). *The Theory of Relational Databases*. Computer Science Press.

[US Department of Education, 2003] US Department of Education (2003). Audit of the colorado student loan program's establishment and use of federal and operating funds for the federal family education loan program. technical report.

[Melton, 1998] Melton, J. (1998). Database language sql. In Bernus, P., Mertins, K., and Schmidt, G., editors, *Handbook on Architectures of Information Systems*, International Handbooks on Information Systems, pages 103–128. Springer Berlin Heidelberg.

[Mendes, 2012] Mendes, J. (2012). Evolution of model-driven spreadsheets. Master's thesis, University of Minho.

[Mens and Demeyer, 2008] Mens, T. and Demeyer, S., editors (2008). *Software Evolution*. Springer.

[Moreau et al., 2003] Moreau, P.-E., Ringeissen, C., and Vittek, M. (2003). A pattern matching compiler for multiple target languages. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 61–76, Berlin, Heidelberg. Springer-Verlag.

[Oracle, 2013] Oracle (2013). Java native interface. `http://docs.oracle.com/javase/6/docs/technotes/guides/jni`. [Accessed on April 2013].

[Panko, 2008] Panko, R. R. (2008). Spreadsheet errors: What we know. what we think we can do. *CoRR*, abs/0802.3457.

[Parr and Quong, 1995] Parr, T. J. and Quong, R. W. (1995). Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810.

[Schmidt, 2006] Schmidt, D. C. (2006). Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31.

[Shin and Sanders, 2006] Shin, S. K. and Sanders, G. L. (2006). Denormalization strategies for data retrieval from data warehouses. *Decis. Support Syst.*, 42(1):267–282.

[Visser and Saraiva, 2004] Visser, J. and Saraiva, J. (2004). Tutorial on strategic programming across programming paradigms. In *8th Brazilian Symposium on Programming Languages, SBLP*.

[Wohlin et al., 2012] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., and Regnell, B. (2012). *Experimentation in Software Engineering*. Springer.

Appendices

A: ANTLR Grammar

```
grammar Mdql;


options {
  output=AST;
  ASTLabelType=Tree;
  tokenVocab=MdqlTokens;
}

@header { package mdql; }
@lexer::header { package mdql; }

query
        : select from where groupby orderby limit labels ->^(Query
           select from where groupby orderby limit labels histogram
           )
        ;

select
        : SELECT distinct ALL ->^(SelectAll distinct)
        | SELECT distinct clauses ->^(Select distinct clauses)
        ;

distinct
        : DISTINCT ->^(Distinct)
        | ->^(NoDistinct)
        ;

clauses
        : attributes (',' attributes)* ->^(Clauses attributes+)
        ;

attributes
        : standardAttr ->^(StandardAttr standardAttr)
        | aggregationAttr ->^(AggregationAttr aggregationAttr)
        | allClass ->^(AllClass allClass)
        ;

allClass
        : mdclass '.' ALL ->^(CAll mdclass)
        | '(' (m1=mdclass) ',' (m2=mdclass) ')' '.' ALL ->^(BCAll
           $m1 $m2)
        ;


standardAttr
        : attr ->^(DefaultAttribute attr)
        | mdclass '.' attr ->^(CAttribute mdclass attr)
        | '(' (m1=mdclass) ',' (m2=mdclass) ')' '.' attr ->^(
           BCAttribute $m1 $m2 attr)
        ;

attr
    : LETTERS ->^(LETTERS)
        ;

mdclass
        : LETTERS ->^(LETTERS)
        ;

aggregationAttr
        : 'avg(' standardAttr ')' ->^(Avg standardAttr)
        | 'count(' standardAttr ')' ->^(Count standardAttr)
        | 'max(' standardAttr ')' ->^(Max standardAttr)
        | 'min(' standardAttr ')' ->^(Min standardAttr)
        | 'sum(' standardAttr ')' ->^(Sum standardAttr)
        ;

from
        : FROM (models=mdmodels) ->^(From $models)
        | ->^(NoFrom)
        ;

where
        : WHERE filters ->^(Where filters)
        | ->^(NoWhere)
        ;

filters
        : filter (filter)* ->^(Filters filter+)
        ;

filter
        : standardAttr LOGIC NUMBER logicAO ->^(FilterNum
           standardAttr LOGIC NUMBER logicAO)
        | standardAttr LOGIC LETTERS logicAO ->^(FilterStr
           standardAttr LOGIC LETTERS logicAO)
        | standardAttr LOGIC 'date' LETTERS logicAO ->^(FilterDate
           standardAttr LOGIC LETTERS logicAO)
        ;

logicAO
        : AND ->^(And)
        | OR ->^(Or)
        | ->^(NoLogicAO)
        ;

mdmodels
        : mdmodel (',' mdmodel)* ->^(Models mdmodel+)
        ;

mdmodel
        : LETTERS ->^(Model LETTERS)
        ;

columnsG
        : attributes (',' attributes)* ->^(Columns ^(ColumnG
          attributes)+)
        ;

columnsO
        : attributes (',' attributes)* ->^(Columns ^(ColumnO
          attributes)+)
        ;

groupby
        : GROUPBY columnsG ->^(GroupBy columnsG)
        | ->^(NoGroupBy)
        ;


orderby
        : ORDERBY columnsO orderType ->^(OrderBy columnsO orderType
          )
        | ->^(NoOrderBy)
        ;

orderType
        : ASC ->^(Asc)
        | DESC  ->^(Desc)
        | ->^(NoOrderType)
        ;

limit
        : LIMIT NUMBER ->^(Limit NUMBER)
        | ->^(NoLimit)
        ;

labels
        : LABEL label (',' label)* ->^(Labels label+)
        | ->^(NoLabels)
        ;

label
        : attributes labelRename ->^(Label attributes labelRename)
        ;

labelRename
        : LETTERS ->^(LETTERS)
        ;

histogram
        : HISTOGRAM ->^(WithHistogram)
        | ->^(NoHistogram)
        ;

DISTINCT : 'DISTINCT' | 'distinct' ;
LOGIC    : '=' | '>=' | '<=' | '!=' | '>' | '<' ;
AND : 'AND' | 'and' ;
OR : 'OR' | 'or' ;
WHERE    : 'WHERE' | 'where' ;
ASC      : 'ASC' | 'asc' ;
DESC     : 'DESC' | 'desc' ;
ORDERBY : 'ORDER BY' | 'order by' ;
LABEL    : 'LABEL' | 'label' ;
LIMIT    : 'LIMIT' | 'limit' ;
GROUPBY : 'GROUP BY' | 'group by' ;
ALL      : '*' ;
SELECT   : 'SELECT' | 'select' ;
FROM     : 'FROM' | 'from' ;
HISTOGRAM : 'WITH HISTOGRAM' | 'with histogram' ;
NUMBER   : (DIGIT)+ ;
LETTERS : (LETTER)+
        | '\'' (LETTER2)+ '\''
        ;
STRING   : '"'((LETTER2) | (DIGIT))+'"';
WHITESPACE : ( '\t' | ' ' | '\r' | '\n'| '\u000C' )+    { $channel
   = HIDDEN; } ;

fragment DIGIT   : '0'..'9' ;
fragment LETTER : 'a'..'z' | 'A'..'Z';
fragment LETTER2 : 'a'..'z' | 'A'..'Z' | ' ' ;
```

B: Gom Data Structures

```
module Mdql
imports String int

abstract syntax

QUERY = Query(s:SELECT, f:FROM, w:WHERE, g:GROUPBY, o:ORDERBY,
   li:LIMIT, la:LABELS, h:HISTOGRAM)

SELECT = Select(d:DISTINCT, c:CLAUSES)
                | SelectAll(d:DISTINCT)

DISTINCT = Distinct()
                | NoDistinct()

CLAUSES = Clauses(ATTRIBUTE*)

ATTRIBUTE = StandardAttr(s:STANDARD)
                | AggregationAttr(a:AGGREGATION)
                | AllClass(ac:ALLCLASS)

ALLCLASS = CAll(class1:String)
                | BCAll(class1:String,class2:String)

STANDARD = DefaultAttribute(name:String)
                | CAttribute(class1:String, name:String)
                | BCAttribute(class1:String, class2:String,
                   name:String)

AGGREGATION = Avg(s:STANDARD)
                        | Count(s:STANDARD)
                        | Max(s:STANDARD)
                        | Min(s:STANDARD)
                        | Sum(s:STANDARD)



FROM = From(models:MODELS)
                | NoFrom()

WHERE = Where(filters:FILTERS)
                | NoWhere()

FILTERS = Filters(FILTER*)
```

```
FILTER = FilterNum(standard:STANDARD, logic:String, num:int,
   l:LOGICAO)
                | FilterStr(standard:STANDARD, logic:String,
                   str:String, l:LOGICAO)
                | FilterDate(standard:STANDARD, logic:String,
                   str:String, l:LOGICAO)

LOGICAO = And()
                | Or()
                | NoLogicAO()

MODELS = Models(MODEL*)

MODEL = Model(name:String)

GROUPBY = GroupBy(cols:COLUMNS)
                | NoGroupBy()

COLUMNS = Columns(COLUMN*)

COLUMN = ColumnG(a:ATTRIBUTE)
                | ColumnO(a:ATTRIBUTE)

ORDERBY = OrderBy(cols:COLUMNS, type:ORDERTYPE)
                | NoOrderBy()

ORDERTYPE = Asc()
                | Desc()
                | NoOrderType()

LIMIT = Limit(i:int)
                | NoLimit()

LABELS = Labels(LABEL*)
                | NoLabels()

LABEL = Label(a:ATTRIBUTE, rename:String)

HISTOGRAM = WithHistogram()
                | NoHistogram()
```