
**André Brás Simões**

Licenciado em Engenharia Informática

# Expressiveness Improvements of *OutSystems* DSL Query Primitives

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientadores : Hugo Lourenço, Senior Software Engineer, OutSystems
Hugo Torres Vieira, Assistant Professor, Universidade de Lisboa
João Costa Seco, Assistant Professor, Universidade Nova de Lisboa

Júri:

| | |
|---|---|
| Presidente: | Prof. Pedro Duarte de Medeiros |
| Arguente: | Prof. André Leal Santos |
| Vogal: | Prof. João Costa Seco |

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

**Março, 2013**

**Expressiveness Improvements of *OutSystems* DSL Query Primitives**

Copyright © André Brás Simões, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa

A Faculdade de Ciências e Tecnologia e a Universidade Nova de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objectivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

*To my father Aurélio,*
*my mother São,*
*my sisters Vera and Ângela,*
*my brother David,*
*my nephew Filipe*
*and my love Tânia.*
*I owe you more than I can ever say.*

# Acknowledgements

*"The future belongs to those who believe in the beauty of their dreams."*

— Eleanor Roosevelt

The dissertation that I present here was the result of five years of dedication, hard work and effort to conquer a goal, and specially a dream. Dream that I was not able to achieve without the contribute of many people.

So, I would like to express my heartfelt gratitude to my advisors Hugo Vieira and Hugo Lourenço for the guidance, support, quality and because they always put me in the right track. I present them my thanks for the dedicated hours and several headaches that my work caused them. But without you would it would not have been possible.

I am equally grateful to Professor João Costa Seco and António Melo for their contribute as well as to *Faculdade de Ciências e Tecnologia* from *Universidade Nova de Lisboa* for the financial support and for the articulation between *OutSystems* that allowed me to develop the study that is the basis of this thesis.

I also would like to thank to Professor Vitor Teodoro, for reading the thesis and providing precious feedback.

Thanks to Lúcio Ferrão and to his critical thought, to Hélio Dolores, José Caldeira, and other wonderful R & D team elements that received me so well and acquainted me about *OutSystems Agile Platform* as well as other new technologies for me.

And, because, five years were made not only of study but also with academic life and friendship, I want to thank to Miguel Alves and Nuno Grade ("*Os três da vida airada*") and to Stefan Alves, Pedro Almeida, Sérgio Silva, Miguel Pinheiro, Sérgio Casca and, more recently, Tiago Almeida. Also, I would like to say that I am very grateful to have know you, Kinga and Heidi, during our Erasmus experience.

A huge thank you to São Brás and Aurélio Simões for being the best parents that I could ever wish. For being a model and a life example to achieve. I want to thank them for giving me wings to fly, for being always there to catch me if I need, for their constant love and immeasurable sacrifice.

I also acknowledge to my sisters and brother Vera, Ângela and David Simões. We are part of a whole and I have got in you all the dreams from the entire world. I would not forget my pesky little nephew, Filipe, and my brothers-in-law Flávio Ferreira and Joel Cruz. To my uncle Luis for being more than a uncle, for being a friend – thank you.

Last but not least, I want to express my eternal debt towards Tânia, my precious love, for her enduring love, for believing in me long after I would lost belief in myself, and for sharing my wish to reach the goal of completing this task but caring enough to love me even if I never achieved it.

All errors and limitations remaining in this thesis are mine alone.

# Abstract

In the ever more competitive market, companies are forced to reduce their operational costs and innovate. In order to do that, some companies successfully adopted new approaches, some of them using domain specific languages (DSL), building their entire system and all the respective layers in less time and more focused in their business. Frequently, the application business layer interacts with the data layer through SQL queries, in order to obtain or modify data. There are some products in the market that try to make life easier for developers, allowing them to get the data using the features of visual query builders, also available in standard SQL. However, it is not expectable that every possible query can be written through these visual query builders, which leads us to the following questions "*What should and what can easily be supported by visual query builders?*". These questions are relevant in order to help improving the experience of developers and save them time.

This work aims to study and analyse techniques that help detecting patterns in structured data and, afterwards, propose a suitable way to view and manage the visualization of the occurrence of such detected patterns. In order to help identify the most frequent patterns and thus contribute to solve the above questions, with this conjunction of topics we expect to provide a way to improve the experience of understanding a large amount of data in a particular context. Once understood some patterns that could be present in the data and their importance, we are ready to propose a new model in the context of *OutSystems Agile Platform*$^{TM}$, in terms of their visual query builder, aiming to increase its value, improve its expressiveness and offer a powerful visual way to build queries.

**Keywords:** Database query languages, Visual query builders, Structure mining

# Resumo

No mercado actual, cada vez mais competitivo, as empresas sentem-se forçadas a reduzir os seus custos operacionais e a inovar. Para tal, algumas destas empresas têm optado com sucesso por novas abordagens, algumas optando por linguagens de domínio específico, construindo todo o seu sistema e todas as respectivas camadas em menos tempo e mais focadas aos seus negócios. Frequentemente, a camada de lógica da aplicação interage com a camada de dados através de consultas SQL, de forma a obter ou modificar esses dados. Existem produtos no mercado que procuram facilitar a vida dos programadores permitindo-lhes obter a informação das base de dados através de "construtores visuais de consultas"usando propriedades que também se encontram disponíveis no SQL comum. Contudo, nem tudo é possível fazer através destes construtores visuais, surgindo assim as questões "*O que deverá e o que poderá ser suportado por estes construtores visuais de consultas?*". Estas duas questões são importantes na medida que visam ajudar a melhorar a experiência dos programadores e poupar-lhes tempo.

Com este trabalho pretende-se estudar e analisar técnicas que visem ajudar na detecção de padrões em dados estruturados. Posto isto, apresentar uma forma adequada para visualizar e gerir a visualização da ocorrência dos padrões detectados. De modo a ajudar a identificar os padrões mais frequentes e então contribuir para responder às anteriores questões, conjugando estes tópicos referidos espera-se apresentar uma forma que melhore a experiência de compreender um determinado grande conjunto de dados inseridos num contexto em particular. Uma vez percebidos alguns dos padrões que possam estar presentes nos dados e as respectivas importâncias, encontram-se reunidas as condições necessárias para propor o novo modelo no contexto da *OutSystems Agile Platform*$^{TM}$, relativamente ao seu construtor visual de consultas, com o objectivo de aumentar o seu valor, melhorar a sua expressividade e oferecer uma poderosa forma visual para construir consultas.

**Palavras-chave:** Linguagens de consulta para bases de dados, Construtores visuais de

consultas, Análise de estruturas

# Contents

# List of Figures

# List of Tables

# Listings

# 1

# Introduction

In the ever more competitive market, caused by the present economy, companies are forced to reduce their operational costs and innovate, trying to increase their annual profits. Among many solutions, companies try to take into account the new applications needed in the near future and the best ways to getting them. It is important never forget some relevant points, such as:

- Time to market;
- High productivity;
- Easy integration;
- Technical fit and flexibility;
- Final price.

As the creation of the software is a very abstract process, it would be great to know exactly what we are obtaining up front, but unfortunately a software project cannot be predicted like that.

In the last years, some new methodologies appeared some more effective than others: among them Agile Software, the software development paradigm related to the work presented in this dissertation. Agile Software emerged to face some of the aspects listed above, in such a way that we are dealing with:

- *Implemented requirements can be seen fast* - it lets developers the possibility to see what was implemented and show it to the business owner, in a short time;

- *Fast Delivery* - allowing the companies to have their customized products delivered on-time and on-budget, faster and more cost effectively, rather than with other alternatives;

- *Acceptance to changes* - Agile processes accept the reality of change without rigid specifications. Although there are domains where requirements cannot change, the great majority of projects have changing requirements;

- *Constant communication* - communication between developers and the business owner is essential. The business owner can follow the development of specifications and provide feedback that will be considered during constant iterations aiming at improving the product.

Taking advantage of such characteristics of Agile Software and with the same problems to solve, *OutSystems Agile Platform*™ [Out12] (see Chapter 2) appeared in the market.

Since design is as important as functionality, *Agile Platform*™ provides tools allowing to create modern and easy to use Enterprise Applications.

*Predictable Projects* is another characteristic associated with the *Agile Platform*™. The two main risks on a project involve going over budget and not delivering what the business really needs. However, they are overcome by doing fixed-cost projects that count on business client participation to ensure that the application is built according to what is really needed. Moreover, it is *Ready to Grow* in order to adapt the product to the new company and market needs.

## 1.1 Motivation

Software companies attempt not to forget the usability and the market trends in their products, trying to develop them with some topics in mind, such as nice interfaces, available features and maintainability.

Some development decisions are taken on the very beginning of the product development. After some years, these decisions may have to be reviewed in order to adapt the new needs of users to the product itself. In this context, it is essential not to forget the new impact these changes and needs imply, which requires careful study and a focused analysis.

One of the main goals of *OutSystems Agile Platform*™ is to ease the development of Enterprise Applications, and once known that for inexperienced users it is simpler to use its visual query builder instead of the common SQL. Our motivation is to improve the expressiveness of this visual query builder, i.e. allow to write more queries with it, and there will be less reasons to use common SQL.

We want to guaranty that all the changes on the current model are the ones that will bring more benefits to *OutSystems* users, i.e. the ones that are really reflecting the needs of the users of *Agile Platform*™. There are some different ways to accomplish this final

goal – our approach will be performed through an analysis over a large set of queries with the help of a tool. This set of queries is the result from the last **2 years** of work in the context of *OutSystems*, involving **10 clients**, totalizing **27,000 distinct queries**. Then, the new model proposal regarding the visual query builder of *Agile Platform*™ will take into account this analysis performed over the data since the analysis aims to identify the main features that should be supported by the model. Nevertheless, it is important to refer that this abstract model may help in other contexts, since the identified essential features are expected to be general enough to fit in other contexts whenever visual query models are of use.

Next, we depict the work description of this thesis, going deep into this context.

## 1.2 Work Description

This work is carried out in the context of the company *OutSystems* and is part of a process of improving *Agile Platform*™ expressiveness regarding the way it interacts with databases.

Considering that the graphical Domain Specific Language (DSL)[1] of *OutSystems* is idealized for creation and maintenance of *web* and *mobile web* applications, its primitives that allow the user to query/search data are of the highest importance. On this moment, there are two relevant features in the language to query/search data:

- **Simple Queries** - with advantages mainly around usability and validation, mostly due to the visual builder and the TrueChange™;
- **Advanced Queries** - more powerful but not so controlled, since they allow the user to write the queries SQL code [SKS10].

*OutSystems* offered the possibility to carry out this master thesis work with the challenge of proposing a new design with the advantages of *Simple Queries* and the expressiveness closer to *Advanced Queries*. With this in mind, we aim to allow a developer to use more often the *Simple Queries* during all the development stages avoiding the usage of *Advanced Queries*, as well as writing SQL code. Since *Simple Queries* have a validation system integrated, as much as they are used, more will be the validation support provided to users.

To understand the source of lack of expressiveness on *Simple Query* model we need to analyse the dataset of queries extracted from the applications of *OutSystems* clients, that will support our decisions. At first sight, as depicted in Figure 1.1, we can see that the dataset is mainly composed by *Select* statements, which will be the focus of our analysis since the selection of data is the purpose of *Simple Queries*.

However, these patterns are too simple and nowadays development teams need to understand other more complex patterns such as the use of *Distinct* keyword, use of

[1] A Domain Specific Language is a programming language or executable specification language designed to express solutions to problems from a particular domain.

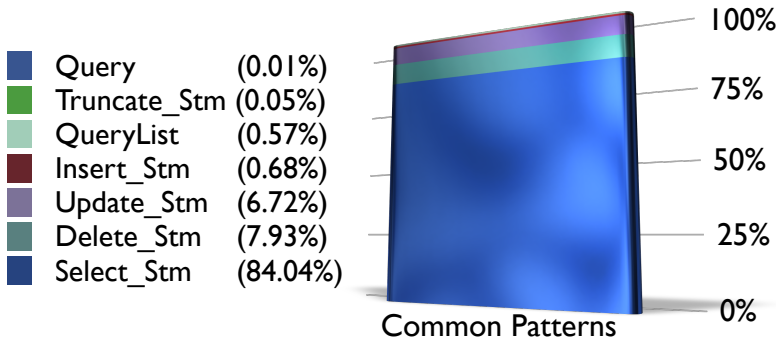


Figure 1.1: Chart with occurrence frequency of each common pattern

*Complex Join* conditions or even the *Group By* clause. These patterns are reflected in the structure of a query and can be dependents between them. This lead to a thorough analysis over all the dataset of queries extracted focusing on their structure, trying to point out what are the most frequent patterns on the dataset that are missing on *Simple Queries*.

The analysis is divided in several stages, from the creation of a tool to explore the dataset using XPath expression (since a query is well structured as an XML tree [HM04] and a XPath expression represents a pattern), to a clustering algorithm applied on the dataset that grouped the queries depending on their structure leading us to the creation of a method to analyse each one of the clusters.

The mentioned method is composed by three different kinds of analysis: a visual analysis based on a colored graph that represents the elements stated on the cluster (each node from the graph is colored depending on its frequency inside that cluster); an analysis on the histogram of SQL terms created during the clustering (here we can perceive not only the frequency of a term on that specified cluster but also on the global dataset of queries); last but not least, an analysis over the queries composing that cluster (e.g. it can help to distinguish if an *And* operator is used on a *Join* condition or on the *Where* clause, when the visual analysis creates this doubt).

Combining these three distinct analysis we are able to understand and discover some interesting patterns and, in the majority of the cases, to quantify the occurrence frequency of such patterns (see Chapter 4). Listing 1.1 displays an example of a query that shows out one of the identified frequent patterns during the visual analysis, exposing one of the sources for the lack of expressiveness on *Simple Queries*.

Listing 1.1: SQL example with a pattern not supported in the current *Simple Query* model

```
1 SELECT {User}.FirstName, '' , {User}.LastName,
2 FROM {User}
```

Currently, if we would like to apply this query from Listing 1.1, we would need to perform it using an *Advanced Query* since *Simple Queries* have some limitations. The limitation visible in this particular case is the fact that it is not possible to define a specific set of columns to be selected, or even to assign a particular value to a field (in the case of the example, an empty string). This case usually occurs when a developer needs to get some data from entities to send to be consumed by a *Web Service*, since the *Web Service* imposes the structure of the data to consume, which in the context of *OutSystems* implies to use an *Advanced Query*, since *Simple Queries* do not allow to select only some columns from an entity (all the columns need to be selected) nor allow to assign values for a particular field.

After identifying the set of most frequent patterns, referred in the model context as features, we propose a new *Simple Query* model extending the current one with these set of features. Defining an order of implementation of these features becomes necessary, once in the software businesses / real markets although predictable it is impossible to precise how many features can be implemented in a specific period of time. This constraint is due to several variables that can change over the time such as potential unexpected problems that may arise, the need to focus human resources in other tasks, among others.

Once the order of implementation is defined, we have the challenge of starting to propose a new model of *Simple Queries* with the specific pattern reflected on Listing 1.1, and we discover that this feature could reduce the use of *Advanced Queries* in almost 18%. Then, after discussing with the R & D team and iterating several times our proposal to extend this feature, we come across an interface proposal for *Simple Queries* as shown in Figure 1.2.

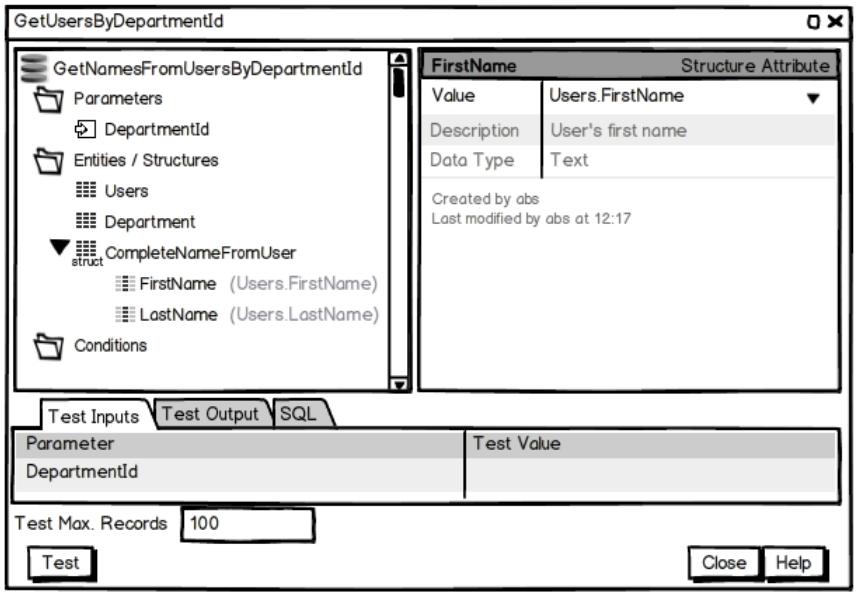


Figure 1.2: Mockup - extending *Simple Queries* with the first new feature

This interface partially increases the expressiveness of *Simple Queries* from *Service Studio* since it refers to the addition of only one feature, however we are looking for even more expressiveness which guide us to propose also the extension of model with the other features from the set of most frequent features defined.

## 1.3 Contributions

When we intend to drive a study, we need to consider the knowledge and applicability which may arise from that, i.e. the contributions of the study. Regardless that our main contribution is directly related with our primary goal of identify the key features of a new visual query model, we have also done contributions in the field of analysis and presentation of results. Next, we will present all the contributions that we have identified.

**Develop an automatic analysis process** to extract patterns from a large set of queries exploiting some of the studied techniques, adapting these techniques to our context, and provide a proof of concept prototype tool that realizes the process. The input of the tool is a specific grammar file (with rules similar to standard SQL) and a set of queries that obey the rules of this grammar.

**Present the results** of the automatic process. These results are composed by a list of the patterns detected out of the set of queries and the due frequency of each pattern. It is important to visualise these results to the development process as *OutSystems* designers may use the tool to take more informed decisions on the sort of queries that should be supported by the *OutSystems Agile Platform*™ in the future.

**Identify the key features of a new Simple Query model** in the context of *OutSystems Agile Platform*™ based on the most frequent query patterns singled out by the automatic process, and on the feasibility of supporting the development of such query patterns using simple intuitive interactions. As everything behind this is somehow SQL, we expect that the identified key features at the basis of the model will be general enough to fit other contexts whenever visual query models are of use.

## 1.4 Outline

In this Chapter we introduced the motivation of our work, including the respective involved context, followed by our work description with some hints for the proposed solution as well as the contributions we have identified.

In Chapter 2 we give an overview of *OutSystems Agile Platform*™ as well as some relevant elements from the visual language that it uses. These elements are related with the query primitives that allow *Agile Platform*™ to interact with the data layer. One of

these primitives will be the focus of our main goal, which is to identify the key features that should extend this primitive in order to improve its expressiveness.

Afterwards, in Chapter 3 we display the work carried out to help us taking supported decisions in terms of features to extend the model and some technical details of our tool that will allow us to look for specific patterns in a dataset. This dataset will appear with the help of *OutSystems* R & D team that provided thousands of applications from their clients, allowing to build the dataset of distinct queries to be analysed.

Our new model proposal based on the results from the analysis is described in Chapter 4. This propose consists in stating some new features that would extend the current *Simple Query* model from *OutSystems Agile Platform*$^{TM}$, and also propose the interface that would be able to support each feature in the *Platform*.

Thereupon, in Chapter 5 we depart from a medium-fidelity prototype and present a high-fidelity prototype submitted to usability tests. Then, we present the scenario created on the usability tests coupled with feedback from testers, as well as a list of the top issues extracted from the feedback.

Finally, in Chapter 6 we take some conclusions from our work. Besides, we present several proposes to be addressed as future work fitting in the same topic.

# 2

# *OutSystems Agile Platform™*

In this chapter we present *OutSystems* flagship product, *Agile Platform™*, as well as some important aspects from it once that the thesis is inserted in the context of this product. During the explanation, we also present in detail the existing query primitives that are at the core of this thesis work.

## 2.1 *Agile Platform™ & Service Studio*

*OutSystems Agile Platform™* is composed by several heterogeneous parts that contribute to integrate all the deployment and evolution cycle of web applications; focusing on a specific component called *Service Studio* since it is there that are available the query primitives to communicate with the data layer.

In *OutSystems* a web application project is known as an *eSpace* and the format used to save *eSpaces* to disk is the OML that stands for *OutSystems Markup Language* (represented by files .oml). *Service Studio* is the *OutSystems* Integration Development Environment (IDE) that allows editing an *eSpace* as well as publishing it to a development environment, to be tested and analysed, or publishing it to a production environment. Figure 2.1 shows the development environment of *Service Studio*.

## 2.2 Visual Programming Language

A Domain Specific Language (DSL) [MHS05, vDKV00] is a programming language or executable specification language designed to express solutions to problems of a particular domain, through appropriate notations and abstractions.

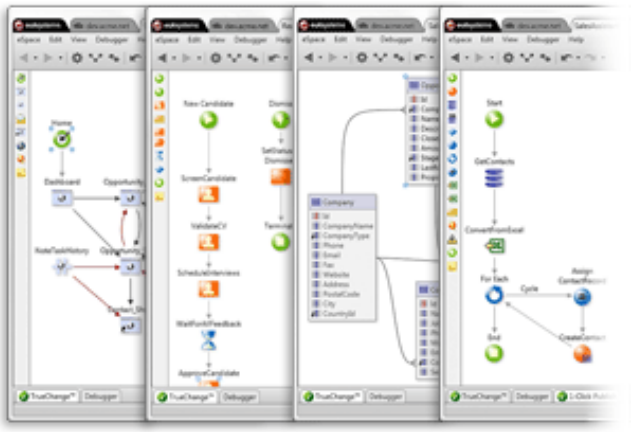

Figure 2.1: Development environment of *Service Studio*

*Service Studio* implements a graphical DSL designed to create web applications with a high-level of abstraction. The constructions provided ease the interaction with the data layer, the manipulation of data, and the definition of user interfaces. There are four main high-level elements of the language, *Web Screens* and *Web Blocks* which define the interface of an application, *Action Flows* that define the business logic needed to manipulate data, and *Entities* that define the data model.

*Web Screen* and *Web Blocks* are elements of the *eSpace* used to produce HTML pages where the user can interact through links, buttons, and forms.

*Action Flows* are designed using a collection of elements that graphically represent the behaviour of an action. The elements available are assignments, queries, conditional expressions, and loop statements. Figure 2.2 shows an example of an *Action Flow* which retrieves to a *Web Screen* the list of Users stored in the database as well as their computed average salary.


Figure 2.2: Example of an *Action Flow*

Hereupon, we present the language constructions provided in *Service Studio* from *Agile Platform*™ that will be the focus of our work although there are much more constructions:

- **Simple Query** Allows the developer to visually build an executable database query (*GetUsers* node from Figure 2.2).
- **Advanced Query** Also allows the interaction with the data layer. However, instead of a visual query builder, the developer writes its own query (*GetAvgSalary* node from Figure 2.2). Both query primitives are explained in detail on the next section.

For more detailed information about the language constructions and *Service Studio* please refer to Chapter *Designing Actions* in [Hel12].

### 2.2.1 Query primitives

As we have stated, currently, there are two ways in *Agile Platform*™ to interact with the data layer that we now explain further. Following the concept of its graphical DSL, *OutSystems* provided its product with *Simple Query*, an easy, intuitive and visual feature to produce queries selecting data. The *Simple Queries* have advantages mainly in what concerns usability and validation, mostly due to the visual query builder and the TrueChange™[1].

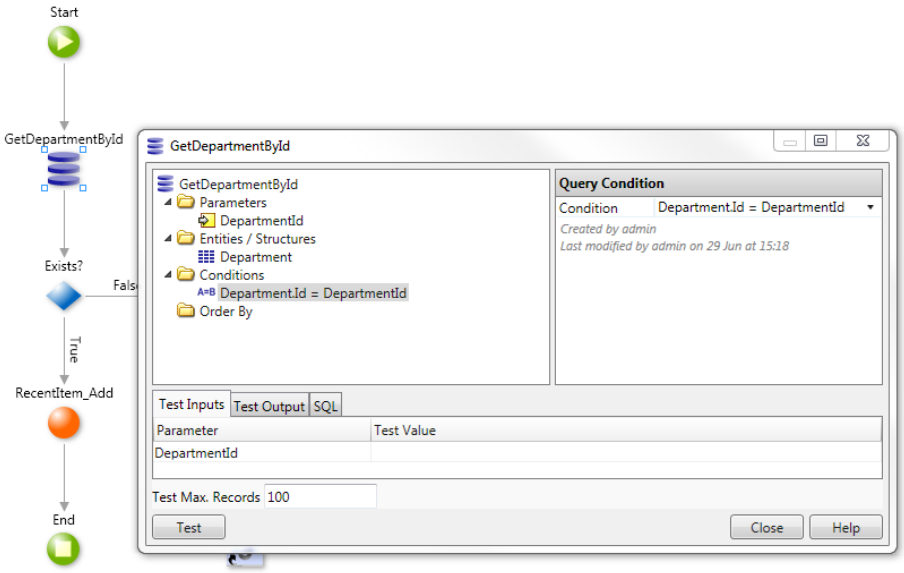


Figure 2.3: Example of a *Simple Query*

Figure 2.3 shows an example of a *Simple Query* that queries the data model for a specific department with a given *Id*. Note the left side of the *Simple Query* window where

[1] TrueChange™ is an available feature that is automatically executed while the developer is designing the application. It is responsible to check for errors and possible warnings in order to validate the *eSpace*, just allowing the developer to Run or Publish the eSpace after it is valid.

the structure of the query is defined, like input parameters, involved entities, conditions to limit the result, and sorting.

However, we can identify some limitations on *Simple Queries*, i.e. actions that are not possible to do using the current *Simple Query* model, such as:

- Definition of columns from the output of the query is not supported;
- Use of aggregate functions is also not supported (as well as *Group by* clause), excluding **Count** function;
- Arrange the execution order of joins;
- Use complex join conditions, composed by more than one condition for example.

Another way to interact with data layer is through an *Advanced Query*. This query primitive allows the developers to write their own queries with all the needed expressiveness, using standard SQL (roughly). The difference to the standard SQL is mainly around the curly brackets surrounding each entity, in order to ease the identification of entities from the query by *Agile Platform*™. The query built on example from Figure 2.3 can also be expressed as an *Advance Query* as we can see on Figure 2.4. Note the folder *Output Structure*, it restricts the SQL query written since the selected columns on the query need to be consistent with the attributes from the entity or structure that is selected on the *Output Structure* folder.

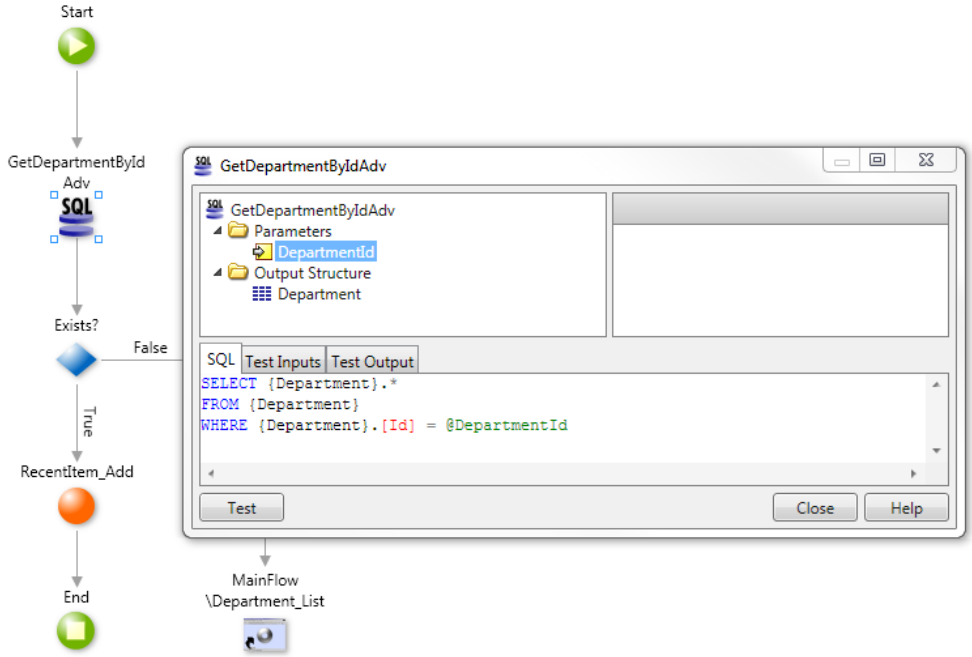


Figure 2.4: Example of an *Advanced Query*

It is possible to state that every imaginable SQL query can be done as *Advanced Query*, however not so easily comparing with *Simple Queries* and since it demands knowledge

about SQL. Furthermore, it is not possible to have validation system so strict as in *Simple Queries* which can lead users to produce queries with errors not trackable on the fly.

## 2.3 Discussion

In this Chapter we have presented an overview over *OutSystems* product, *OutSystems Agile Platform*™, as well as some important aspects of this product. We focused on its main component, *Service Studio*, that is the IDE that allows developers to build applications composed by different modules.

Afterwards, we have displayed some important facts of *OutSystems DSL*, visual language used on *Service Studio*. This language has several kinds of nodes but we lay emphasis on the query primitives since they are the core of our development so as to improve the expressiveness of *Agile Platform*™, more specifically on its way of interacting with the data layer.

In the next Chapter, we introduce the analysis process that we need to carry out in order to understand what patterns are reflected in the dataset of queries from our study.

# 3

# Analysis

This chapter starts to present a pre-analysis stage called Extraction where we explain how to obtain the dataset of queries through the extraction of distinct queries from several applications from different *OutSystems* clients. The remaining sections focus on the detailed analysis over the dataset and the respective obtained results, where we start with an advanced SQL term histogram that is not powerful and structured enough for our purpose and lead us to the SQL parser.

After developing the parser, we use it to help us searching for patterns via a tool created for that purpose. The parser allow us also to search for specific patterns previously identified and mentioned by *OutSystems* Research and Development (R&D) team as well as other interesting patterns. However, we want to go deeper trying to find unidentified patterns and we propose to follow a clustering algorithm from [LCMY04] that allows to group queries from the dataset regarding their structure. Once the queries are grouped, we are ready to analyse some clusters through a visual analysis and we can also present the respective results.

Moreover, we reveal a possible way to analyse the initial global dataset via a chart tool and we point out some aspects that can be extracted from the charts. Finally, we present some previous studies related with the use of SQL in industrial applications.

## 3.1 Pre-analysis - Extraction

To support building a model based on the percentage of queries that have certain pattern we need to have a dataset. Our test set is obtained as an outcome of the process illustrated in Figure 3.1.

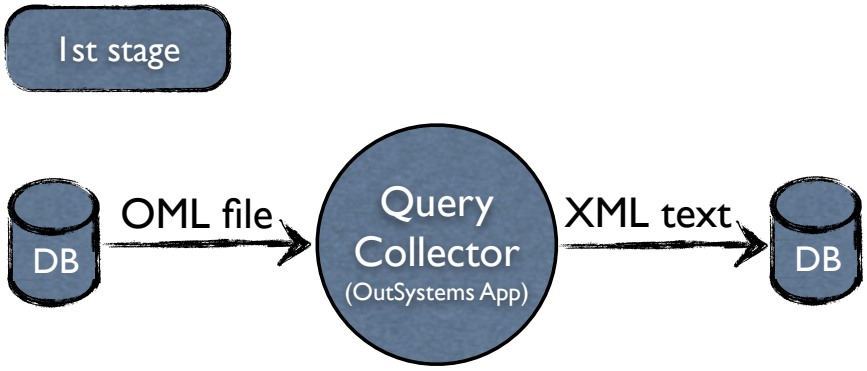


Figure 3.1: Extraction process - 1st stage

In the first stage (Figure 3.1) we create an *OutSystems* Application called *Query Collector* that given an OML file, and using an extension previously provided by *OutSystems* R & D team, stores in the database the XML text, that results as output of the OML file conversion. This XML is not more than all the elements needed in the corresponding Application, from data model entities to all the actions that are presented in the application, through all the queries used.

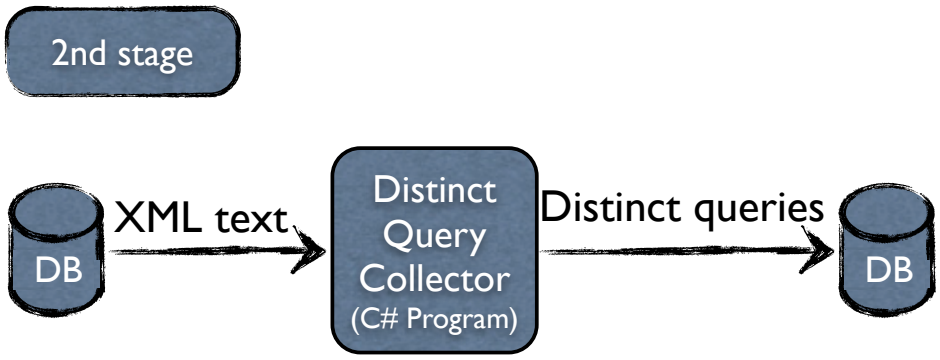


Figure 3.2: Extraction process - 2nd stage

In the second stage of the process (Figure 3.2), we are dealing with each XML text correspondent to an eSpace, and we create a C# program called *Distinct Query Collector* that treats the XML text and obtains just the relevant nodes, i.e. the nodes corresponding to *Advanced Queries*. After having all the nodes, we look at each one (composed by query name, query key, SQL code, parameters, etc.) and if we have already this query in the database we ignore it and go to the next one. To distinguish the queries we use the criteria illustrated in Figure 3.3.

Essentially, we compare the query with all the queries belonging to the same *eSpace* that have the same key. If the SQL code is not the same we have interest in storing this new query because some change happened over time.

Then, after all this previous process occurs, we obtain a filled table with all the queries from our dataset.

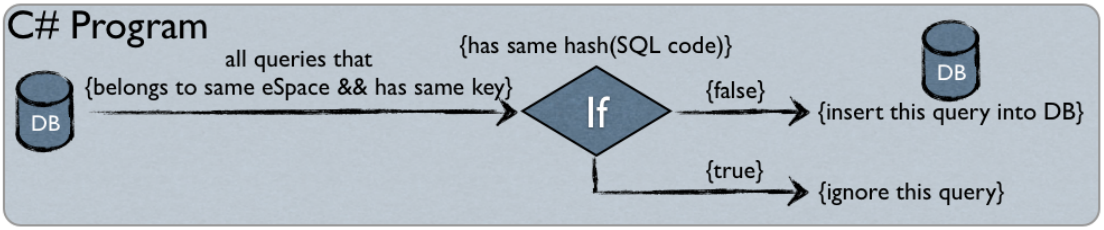


Figure 3.3: Getting distinct queries

## 3.2 Advanced SQL Term Histogram

This analysis addresses the occurrence of terms and involves a C#-based program to produce a term histogram. This term histogram is important to have an idea about the frequency of a specific term. We count both total occurrences and total queries with at least one occurrence of the term. These terms can be single words or composed by more than one word, such as *Group By*, *Order By*, any kind of *Joins*, among others.

With this histogram it is possible to get values closer to the real ones, since we can catch some words that using the method **contains()** from a string we could miss. Figure 3.4 shows the obtained histogram to some terms that we consider important (interesting/more frequent).

However, the figure shown are not enough and as precise as we need, since we cannot determine the number of occurrences of some specific pattern, e.g. two *Inner Joins* in a row in the same query, or the frequency of queries with *inner queries/subqueries* with precision. The query from Listing 3.1[1] has a pattern that can not be distinguished and detected using this approach in order to determine the number of occurrences of that pattern.

Listing 3.1: SQL example with a pattern that cannot be detected using term histograms

```
1 SELECT * FROM {User}
2 Inner Join {EntityA} on ({User}.EntityAId = {EntityA}.Id)
3 Inner Join {EntityB} on ({User}.EntityBId = {EntityB}.Id)
4 WHERE {User}.Id = 10
```

One of the patterns present in the query is the use of two successive *Inner Joins*. To discover that, we need to follow a different approach from the term histogram, therefore we chose to use a Parser that we go on to explain with detail in the following section.

## 3.3 SQL Parser

In this section we introduce the SQL Parser that we use on the third stage of the extraction process.

[1]Note that the queries presented have some differences from the common SQL (using a variant of SQL), mainly around the definition of tables since these tables are surrounded by curly brackets. It happens to help the compiler to easier recognize the tables.

| Total queries in the test-set | 100% | 27439 | |
|---|---|---|---|
| **Term** | | **Number of queries with this term** | **Total occurences** |
| from | **92%** | 25169 | 40030 |
| where | **89%** | 24462 | 35997 |
| select | **86%** | 23600 | 37947 |
| and | **61%** | 16841 | 83344 |
| on | **51%** | 13872 | 56511 |
| order by | **38%** | 10353 | 10826 |
| inner join | **37%** | 10055 | 30353 |
| and ( | **32%** | 8681 | 37379 |
| or | **31%** | 8543 | 41130 |
| select ( | **22%** | 5980 | 7498 |
| where ( | **21%** | 5770 | 7352 |
| left join | **18%** | 4963 | 14901 |
| count | **17%** | 4587 | 5849 |
| like | **16%** | 4292 | 11945 |
| in | **15%** | 4201 | 6287 |
| on ( | **14%** | 3890 | 11730 |
| distinct | **14%** | 3882 | 4670 |
| in ( | **14%** | 3775 | 5625 |
| as | **14%** | 3714 | 21039 |
| null | **13%** | 3653 | 7923 |
| not | **12%** | 3316 | 4803 |
| is | **11%** | 3041 | 6032 |
| or ( | **11%** | 2898 | 9304 |
| from ( | **10%** | 2704 | 5020 |
| end | **9%** | 2503 | 6893 |
| then | **9%** | 2494 | 9231 |
| case | **9%** | 2491 | 6896 |
| else | **9%** | 2398 | 6087 |
| join | **8%** | 2320 | 9375 |
| group by | **8%** | 2251 | 2972 |
| delete | **8%** | 2213 | 3148 |
| sum | **8%** | 2115 | 5728 |
| case when | **7%** | 2033 | 5097 |
| set | **7%** | 2012 | 2117 |
| update | **7%** | 1970 | 2123 |
| \| \| | **6%** | 1685 | 8453 |
| a | **5%** | 1280 | 1646 |
| union | **5%** | 1249 | 2290 |
| exists | **5%** | 1247 | 1892 |
| exists ( | **4%** | 1211 | 1765 |
| id | **4%** | 1117 | 2066 |
| desc | **4%** | 1067 | 1458 |
| when | **4%** | 1000 | 4141 |
| asc | **4%** | 963 | 1291 |
| join ( | **3%** | 902 | 1463 |
| rownum | **3%** | 879 | 936 |
| top | **3%** | 857 | 1616 |
| max | **3%** | 813 | 1464 |

Figure 3.4: Advanced SQL Term Histogram

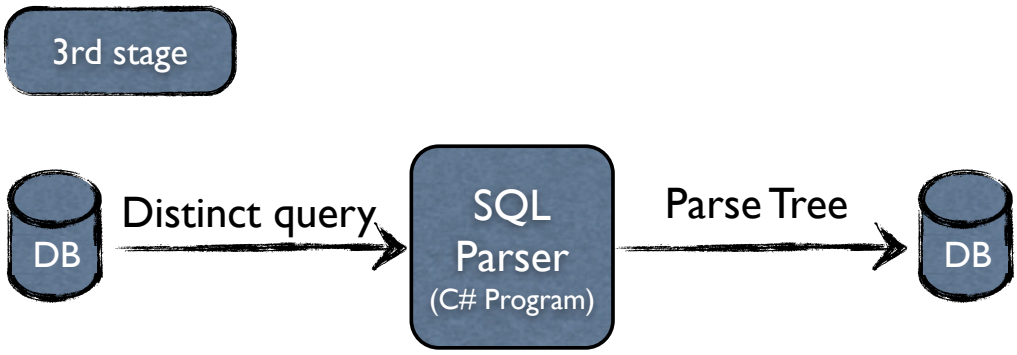


Figure 3.5: Extraction process - 3rd stage

In the third stage exposed in Figure 3.5, we start getting the distinct queries from the database and parsing them using the SQL Parser. The result of the parse process is a parse tree, and to get a better performance during the analysis in real time we need to choose between storing it in the database or save it in memory during the analysis execution. Since we were dealing with much better results in terms of time response if we have the tree stored in the database, reducing from minutes to seconds (between 10 and 20 seconds), this was our choice (also referred in Section 3.4).

Now, regarding the parser *per se*, we start with the SQL Parser (LALR - LookAhead LR parser) used in *Service Studio* that uses a generic Gold Parser Engine (that it is explained better below). The goal is try to find a way to build a parse tree, more precisely using XML as the final output of this parse.

The Gold Parser is a free parsing system, and it can be used by anyone to develop their own programming languages, scripting languages and interpreters. Day after day, it tries to be a development tool that can be used with various programming languages and on multiple platforms [Coo12]. It is composed by three logical components, the Builder, the Engine and a Compiled Grammar Table file (.cgt) definition which acts as an intermediary between the Builder and the Engine (Figure 3.6) [Wik12].


Figure 3.6: Gold Flow

In the Gold website we have some Gold Parser Engines available to a numerous programming languages that can be freely used. So, after taking a look at all available Engines and according to our needs, we find Calitha C# Gold Parser Engine. It is an engine that "can be used to parse text and construct a parse tree that can be traversed in an object oriented manner" [Cal09]. It is possible to integrate it with some tool, using GOLD Parser Builder (downloadable from the Gold Parser website) to construct a grammar following

their syntax and after that, create a compiled grammar table file (.cgt)[2]. Afterwards, such file can be loaded into the Calitha Gold Parser Engine and tested with some input that follows the specified grammar [Cal09].

The primary output obtained from the parser, not doing any change, is a nonterminal token. Each nonterminal token contains several properties such as the rule that caused the reduction, the symbol that it represents and, maybe the most important, an array of tokens that can be terminals or nonterminals too, representing the tokens that are reduced. This last property mentioned is crucial afterwards to build the XML structure that we intend to each query.

So, at first sight, we would like to have a root node directly related with the nonterminal token returned as parser output. After that, we can add their children and grandchildren depending if they are terminal or nonterminal tokens, recursively.

According to our needs, we dump the nonterminal token from output into an XML tree. As example, in Listing 3.3[3] we present the XML tree generated after using our program to parse the query from Listing 3.2.

Listing 3.2: SQL example to test the generation of a parse tree

```
SELECT * FROM {User} WHERE id > 5
```

Listing 3.3: XML parse tree resulted from a *Select* statement

```
<Select_Stm>
  <Terminal symbol="SELECT" value="SELECT">
  </Terminal>
  <Restriction>
  </Restriction>
  <Column_Source>
    <Terminal symbol="*" value="*">
    </Terminal>
  </Column_Source>
  <Into_Clause>
  </Into_Clause>
  <From_Clause>
    <Terminal symbol="FROM" value="FROM">
    </Terminal>
    <Table_Alias>
      <Terminal symbol="VirtualTable" value="{User}">
      </Terminal>
      <ExpandInline>
      </ExpandInline>
    </Table_Alias>
  </From_Clause>
  <Where_Clause>
```

[2]Compiled grammar table file is platform and programming language independent, allowing it to be loaded by any Engine implementation, and after that it is possible to use its information.

[3]Note that each time we parse a query to show as example, or even as case study, we use a grammar from *OutSystems* and related to the set of queries in study

```
      <Terminal symbol="WHERE" value="WHERE">
      </Terminal>
      <Pred_Exp>
        <Simple_Id>
          <Terminal symbol="SimpleId" value="id">
          </Terminal>
        </Simple_Id>
        <Terminal symbol=">" value=">">
        </Terminal>
        <Value>
          <Terminal symbol="IntegerLiteral" value="5">
          </Terminal>
        </Value>
      </Pred_Exp>
    </Where_Clause>
    <Group_Clause>
    </Group_Clause>
    <Having_Clause>
    </Having_Clause>
    <Order_Clause>
    </Order_Clause>
  </Select_Stm>
```

At this moment, since we are able to parse the dataset of queries and get structured data representing the query, we can start understanding and questioning our dataset using a tool that we have created to look for patterns. We go on to explain some details about this search for patterns in the following section.

## 3.4 Searching for Patterns using XPath

As we previously shown our dataset of queries is structured as XML trees, thus we can use XPath to do searches over XML trees since it is a powerful and advanced way to look for specific nodes or sequence of nodes in a XML tree. Then, we need to adapt our tool by adding a search feature that accepts an XPath expression, and after that looks over all the dataset of queries for XML trees that match such expression. Note that a sequence of XML nodes is considered a pattern, which means that a XPath expression can also represent a pattern. In the end, we obtain not only the set of queries matching with such expression, but also the percentage of queries (occurrence frequency) from the dataset that contains such pattern.

From the beginning, we knew about the possibility to face some problems and the need to take into account the performance of the operation of searching on the complete dataset of queries. And, as expected, after trying to analyse all the queries at runtime (parse them and apply the XPath), we get a problem. Performance improvements are mandatory since we are getting a timeout during the HTTP Request due to iterate all the list of queries, around 27,000.

So, the next step is to identify if we can and where we can improve this process of parse the query and analyse its XML tree against the XPath.

Then, we conclude, once again as expected, that the most of time of all the process execution is spent to parse the query and return the XML tree (getting around 80 seconds to parse 7,000 queries, from the total of 81 seconds to parse and test the XPath of the same set).

Therefore, we change the database to store the XML parse tree, so the parsing only has to be done once for each query, distinguishing if it has some syntax error or not. When it has a syntax error, instead of storing an XML parse tree, we store "*Syntax error*", as it is useful when we want to collect just the ones syntactically correct.

The result of this development option is an interesting ≈15 seconds of response time (which means that this time there is no HTTP Request timeout), executing the XPath expression and analysing it against all the XML parse trees from the database.

Since XPath is not the only query language that currently exists, we present in the next section a brief overview over different query languages that could allow to perform searches looking for patterns on the dataset (all related with XML since we have the queries presented as XML trees). Besides, we also introduce some information about SQL since it can also be considered as a query language, and it is directly related with our dataset.

### 3.4.1 Query Languages

**SQL** [SKS10] is a programming language designed for managing data in relational databases and has several parts such as Data-definition language (DDL), providing commands to define and modify relation schemas and to delete relations; Data-Manipulation Language(DML), the reason for SQL to be inserted in this section, that includes a query language based on both relational algebra and the tuple relational calculus (offering operations to insert, delete and modify rows from the database); View definition; Transaction control; Embedded and dynamic SQL, in order to be able to integrate it with other programming languages; Integrity, defining constraints that data from the database have to satisfy; and Authorization, specifying access rights to relation and views.

**XPath** [HM04, Sim02] has the main goal to provide a common syntax and semantics for functionality shared between XSL Transformations and XPointer. Furthermore, through the writing of expressions it allows to identify parts of XML documents, e.g. some particular element that we are looking for. These expressions allow to apply conditions in order to filter the result as detailed as desired, looking for a sequence of elements with a specific attribute value for example.

**XQuery** [Wal07] arose with the expansion of the XML databases and data stored in structured XML documents. Furthermore, it appears to address some issues that XPath can

not deal with such as to allow to use functions and recursion, join XML nodes in the result, structure the result set of some executed query, providing expression for iteration, for binding variables to intermediate results, for ordering, among others.

**TQL** [CGA+02, CG04] is another query language that can be used to query semistructured data, such as XML files. It emerged aiming to combine the expression of types, constraints, and queries in just one language, and to use this union of features for optimization and error-checking purposes. TQL queries are more "declarative" than queries in comparable languages, making some of them much easier to express and allowing the adoption of better optimization techniques. Although the authors conclude that the expression of queries which involve recursion, negation, or universal quantification, keeps in TQL a clear declarative nature (against other languages that are forced to adopt a more operational approach), there is still one important issue that could be addressed in the future, the fact that TQL is based on an unordered nested multisets data model.

**Relation with our approach** Our dataset is composed by queries written using a syntax closer to SQL. XPath or XQuery could be interesting possibilities to be used in the tool to find the specific patterns that we are looking for and as is explained in previous section. TQL could be also a possible approach to follow to query the dataset for patterns, however what it has to offer is much more that what we need since it could be used to check properties, extract tags that satisfy a property, query a document to get some specific element, among others. Hereupon, the best solution could be a simplest language in terms of what we can achieve with it, in order to turn the tool easier to use.

In the next section we navigate to the specific domain *OutSystems* where we have the need to look for specific patterns that are known due to, for example, development choices taken on an early phase of implementation of the product, *OutSystems Agile Platform*™.

## 3.5 Specific Domain *OutSystems*

In the specific domain of *OutSystems*, they have some concerns and interest about specific patterns that are reported by some customers that could be useful on *Simple Queries* or some particular details reported by *OutSystems* developers during the development stage or even when reviewing some past development choices. Then, in the following sections we present the common patterns from the dataset as well as two specific cases from *OutSystems* explaining the path that we should follow to get the answer to them. All the presented examples are related with the entities from Figure 3.7 and their respective

data is depicted in Figure 3.8. This example was created to simplify the way to understand the patterns that we are looking for, and also because we cannot explicitly present data from *OutSystems* due to privacy policies.

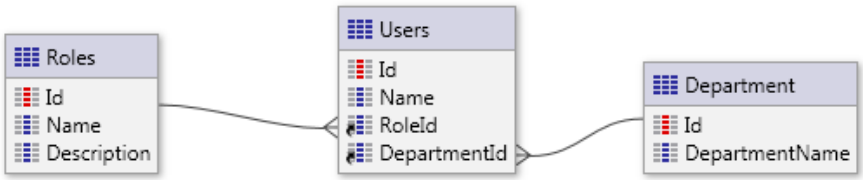


Figure 3.7: Entities Diagram used as example in the next presented cases

**Roles**

| Id (PK) | Name | Description |
|---|---|---|
| 1 | Teacher | Should give pratical and theoretical lessons |
| 2 | President | Manages all directors |
| 3 | Director | Manages all teachers from a specific department |

**Users**

| Id (PK) | Name | RoleId (FK) | DepartmentId (FK) |
|---|---|---|---|
| 1 | Serge | 1 | null |
| 2 | JCC | 1 | 1 |
| 3 | LC | 2 | 1 |
| 4 | LNG | null | 3 |

**Departments**

| Id (PK) | Name |
|---|---|
| 1 | Informatic |
| 2 | Mathematics |
| 3 | Physic |

Figure 3.8: Data from each entity used in the example

### 3.5.1 Common patterns

Since we can use XPath to analyse all the trees generated from our dataset of queries, it is possible to check what are the most simple patterns there. After finding a pattern, we can check other patterns excluding the ones that we have found, and if we do this pattern by pattern, we isolate them and we can present the frequency for each one.

In Figure 3.9 is shown how we have done this process of getting all the common patterns contained in the dataset. We started looking for all the queries contained in the dataset excluding the ones with *Select_Stm* as root element, and we got a set of queries that are a fragment of the dataset. Then, we looked inside this fragment and we found queries with *Update_Stm* as root element, which lead us to another search over the complete dataset using the tool, and excluding this time queries with *Select_Stm* and *Update_Stm* as root elements obtaining a new set of queries. And so on, until we perform a search looking for queries excluding all the different root elements found until then and we get an empty set.

Once we already have each one of the root elements (common patterns) that compose our dataset, we can analyse the occurrence of each one using in our tool a simple XPath query started with "/" and followed by the name of the element, e.g. **/Select_Stm**. Then, Figure 3.10 depicts the corresponding occurrence frequency.

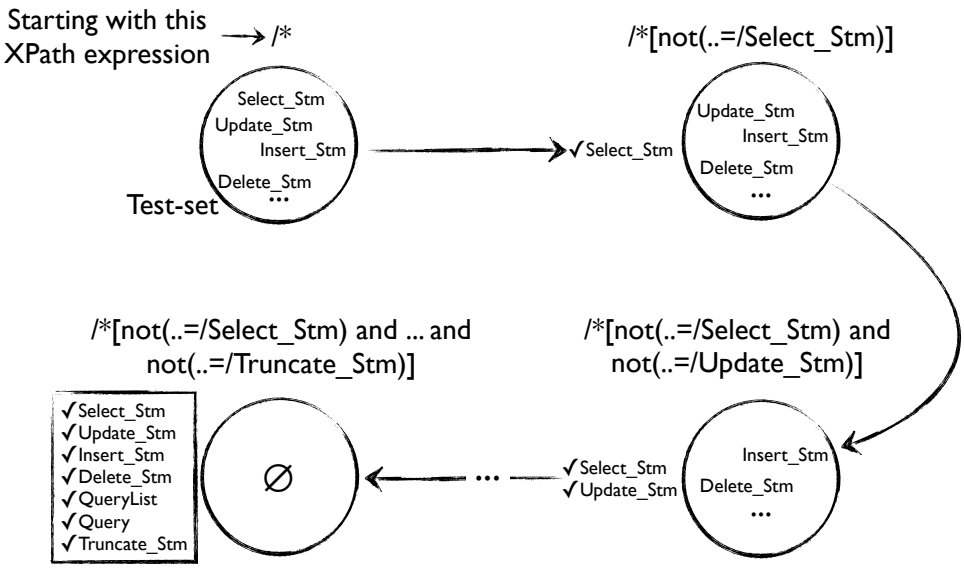


Figure 3.9: Discovering the different roots from our dataset

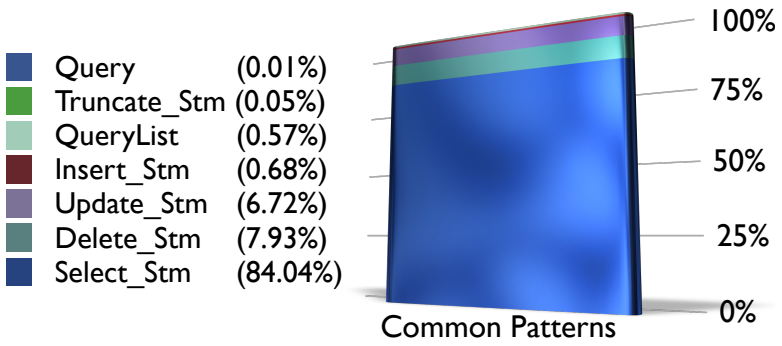


Figure 3.10: Chart with occurrence frequency of each common pattern

In Figure 3.10 is shown that *Select* statements are the most used during the writing of SQL code on *Advanced Queries* in *Service Studio* with around 84% from 25,081 queries without *Syntax error*. Other queries with a meaningful frequency are the *Delete* and *Update* statements, with 7.9% and 6.8% respectively. The other shown patterns together have less than 2% of occurrences, which means that they are really uncommon to use.

### 3.5.2 First case study - Complex Joins

The first situation that the R & D team exposes is related with a specific kind of join, since in the *Simple Queries* if we try to change the default join condition introducing another condition(s) or even with just one condition that is not an equality (or if it is an equality, it is not a $Entity.Column = Entity2.Column$ equality), *Service Studio* automatically changes it from the chosen Join to an *implicit join* where the previous condition is changed to a condition in the *Where* Clause (this is the concept of an implicit join, where the tables on the query are separated by commas and joined through the conditions presented in the *Where* Clause). So, it introduces some constraints on the *Simple Queries*, particularly on *Outer Joins* with more than one condition (on *Inner Joins* it is not a problem since, in *Service Studio*, *Inner Joins* can be easily represented as *Implicit Joins*).

We can use the query from Listing 3.4 as example to this case, that should return all the users with Teacher's Role (RoleId = 1) and all the other kind of Roles (just the roles, not the users with those roles).

Listing 3.4: Example of query using a *Complex Join*

```
1 SELECT {Users}.*, {Roles}.* FROM {Roles}
2 left outer join {Users}
3 on ({Users}.[roleId] = {Roles}.[Id] and {Roles}.[Id] = 1)
```

So, if we are trying to present it on *Service Studio*, we start building a query with a *Left Join*, Figure 3.11. After that, we try to alter the condition from the *Outer Join*, and, as it is possible to see in Figure 3.12, *Service Studio* changes from *Join* to *Condition*, and also that the SQL generated is different from the desired, using the *Implicit Join* as explained before.

To accomplish what was intended, the user is forced to use the *Advanced Query* as Figure 3.13 depicts.

Therefore, this particular case generates the question "***What is the occurrence frequency of a query with On clauses from an outer join that are composed by more than one comparison or have just one comparison but that it is not a equality between values from two columns of distinct Entities?***"

The answer to this question it is not simple and needs to be further studied, and since we already can execute XPath queries to our dataset of XML trees, we need to understand how to migrate the previous formulated question to an XPath expression. And, nothing better to study this practical case that starting with an example. The XML parse tree from Listing 3.5 was generated after parse the query example in Listing 3.4.

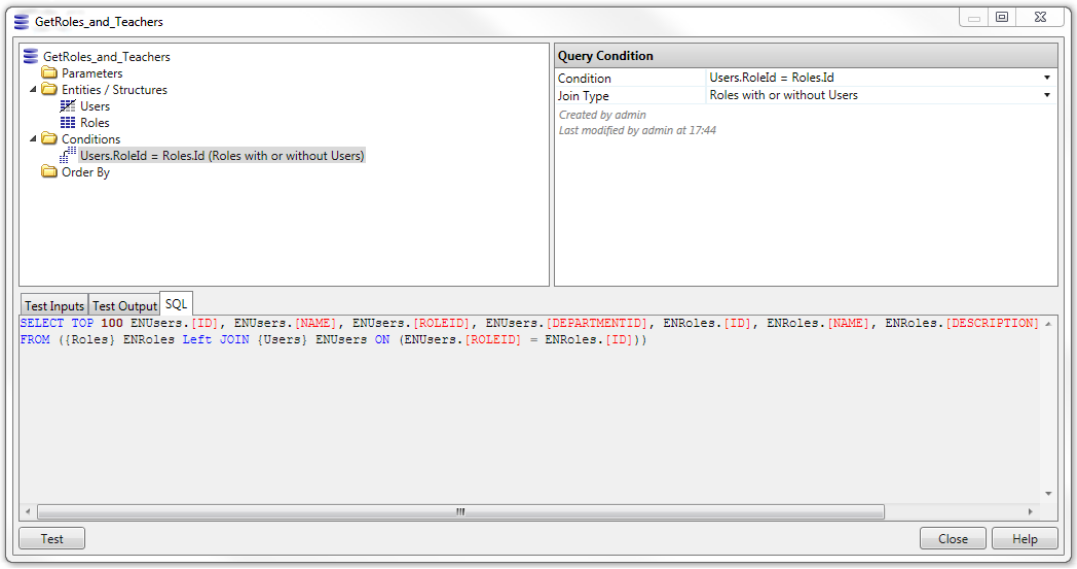


Figure 3.11: *Simple Query* using *Outer Join* with one condition in *On* Clause

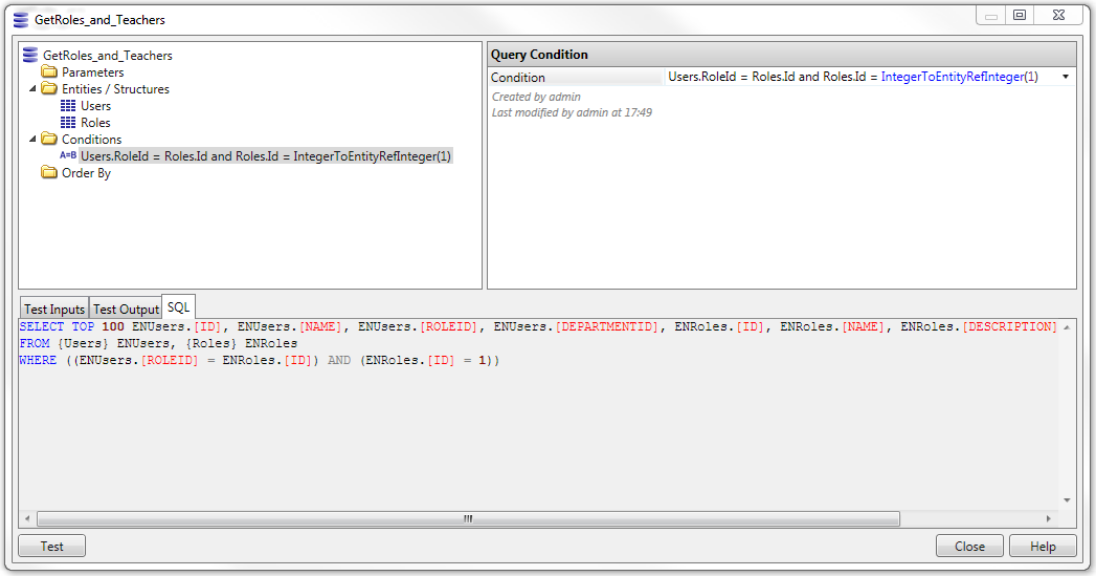


Figure 3.12: *Simple Query* using *Outer Join* changed to Implicit Join by *Service Studio*

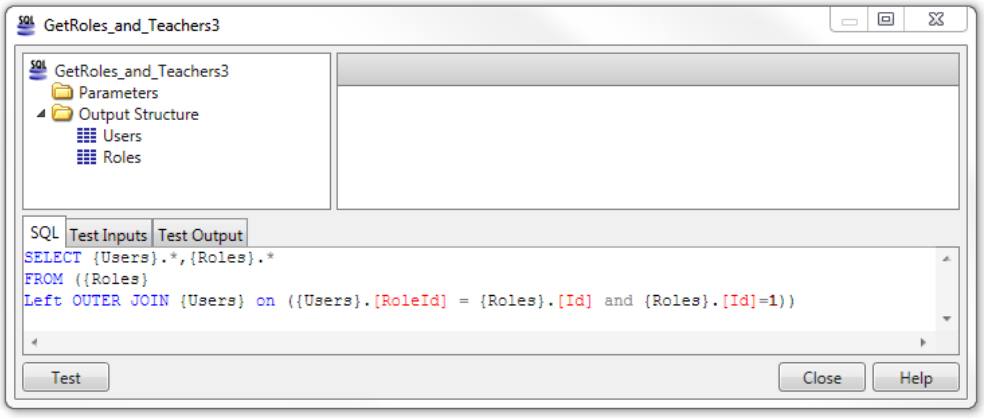


Figure 3.13: *Advanced Query* using *Outer Join* with multiple conditions on *On* Clause

Listing 3.5: XML parse tree from query using *On* clause

```
<Select_Stm>
  <Terminal symbol="SELECT" value="SELECT"/>
  <Restriction/>
  <Column_Source>
    <Terminal symbol="*" value="*"/>
  </Column_Source>
  <Into_Clause/>
  <From_Clause>
    <Terminal symbol="FROM" value="FROM"/>
    <Simple_Join>
      <Table_Alias>
        <Terminal symbol="VirtualTable" value="{USER}"/>
        <ExpandInline/>
      </Table_Alias>
      <Join_Type>
        <Terminal symbol="LEFT" value="left"/>
        <Outer>
          <Terminal symbol="OUTER" value="outer"/>
        </Outer>
        <Join_Hint/>
        <Terminal symbol="JOIN" value="join"/>
      </Join_Type>
      <Table_Alias>
        <Terminal symbol="VirtualTable" value="{ROLE}"/>
        <ExpandInline/>
      </Table_Alias>
      <Terminal symbol="ON" value="on"/>
      <Value>
        <Terminal symbol="(" value="("/>
        <And_Exp>
          <Pred_Exp>
            <Id>
              <Terminal symbol="ComposedId" value="{USER}.[roleId]"/>
            </Id>
```

```
35              <Terminal symbol="=" value="="/>
36              <Id>
37                <Terminal symbol="ComposedId" value="{ROLE}.[id]"/>
38              </Id>
39            </Pred_Exp>
40            <Terminal symbol="AND" value="and"/>
41            <Pred_Exp>
42              <Id>
43                <Terminal symbol="ComposedId" value="{USER}.[col3]"/>
44              </Id>
45              <Terminal symbol="=" value="="/>
46              <Id>o
47                <Terminal symbol="ComposedId" value="{ROLE}.[col3]"/>
48              </Id>
49            </Pred_Exp>
50          </And_Exp>
51          <Terminal symbol=")" value=")"/>
52        </Value>
53      </Simple_Join>
54    </From_Clause>
55    <Where_Clause/>
56    <Group_Clause/>
57    <Having_Clause/>
58    <Order_Clause/>
59  </Select_Stm>
```

As we can see inside <**Simple_Join**>, we have always a couple of nodes and, particularly, an <**Outer**> node inside <**Join_Type**> if we are dealing with an (*Left*, *Right* or *Full*) [*Outer*] *Join*, the presence of this <**Outer**> node is enough to understand that the query has a *Left*, *Right* or *Full Outer Join* or a *Left*, *Right* or *Full Join*, regardless if it is or is not child (this happens due to the rules from the used grammar). So, one of the conditions to have in mind is the presence of a <**Simple_Join**> as parent of a <**Join_Type**> and, in its turn, being this last node parent of an <**Outer**> node.

Another important point that should be kept in mind is the presence or not of parentheses after the *On* clause, therefore we should be looking for trees with an <**And_Exp**> element optionally preceded by a Value node (if we have this node it means that we are facing a *On* clause surrounded by parentheses), inside <**Simple_Join**> element.

Last but not least, the second half of the question related with the occurrence of queries with an *On* clause with just a comparison that is not a equality. To confirm that, we need to access the value from the symbol attribute of the <**Terminal**> child of <**Pred_Exp**>, as this value represents the comparison operator used in the comparison expression, and it should be different from "=" or in the case that it is equals to '=', there should be also an element <**Value**> that is child of <**Pred_Exp**>. If the element <**Value**> is there it means that the equality is between a column from an entity (majority of the cases) and a parameter or an integer value (which is different from the equality between two columns of different entities).

After the previous analysis, it is possible to define the XPath expression from Listing 3.6 that retrieves the answer to what we are looking for.

Listing 3.6: XPath expression to filter of queries with a *Complex Join*

```
/Select_Stm/From_Clause//Simple_Join
  [Join_Type/Outer and
   ((And_Exp or Value/And_Exp) or (Or_Exp or Value/Or_Exp) or
    (Pred_Exp/Terminal/@symbol!='=' or Value/Pred_Exp/Terminal/@symbol!='=') or
    (Pred_Exp/Terminal/@symbol='=' and Pred_Exp/Value) or
    (Value/Pred_Exp/Terminal/@symbol='=' and Value/Pred_Exp/Value))]
```

Figure 3.14 presents the obtained result from this XPath expression in terms of relative frequency. About 1,525 queries follow this specific pattern which is a meaningful amount, 6.09% from the total queries of the dataset without *syntax error* (25,081), and 7.24% from the total *Select* queries (21,077) .

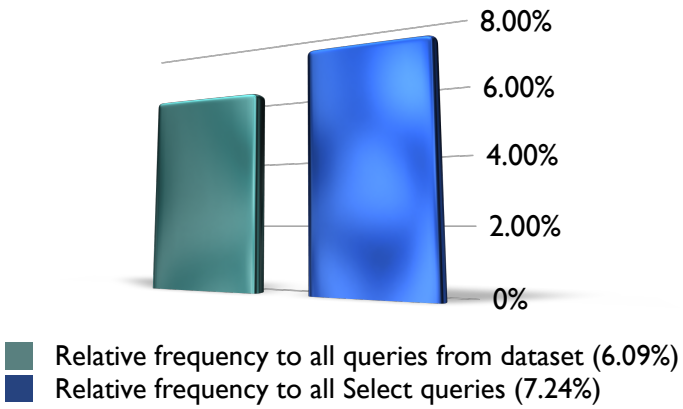


Figure 3.14: Chart with occurrence frequency of pattern *Complex Join*

### 3.5.3 Second case study - specific use of Outer Join followed by Inner Join

Other situation that cannot be done simply using the *Simple Queries* on *Service Studio* is to define the order that we want the joins to occur as we show on the next figures.

If we would like to do the query from Listing 3.7 using the Simple Query feature available, it would seem to be easy. However, in this feature, the order that we see in the section ***Conditions*** (see Figure 3.15 and compare with the generated SQL, *Join* with Role in the conditions come first however it is the last join in the SQL) from the window does not matter since every time that the query is executed in *Service Studio* the *Inner Joins* have priority over other kind of joins.

Listing 3.7: Example of query using *Outer Joins* and *Inner Joins*

```
SELECT * FROM {USER}
right outer join {ROLE} on ({USER}.[roleId] = {ROLE}.[id])
inner join {DEPARTMENT} on ({USER}.[depId] = {DEPARTMENT}.[id])
```

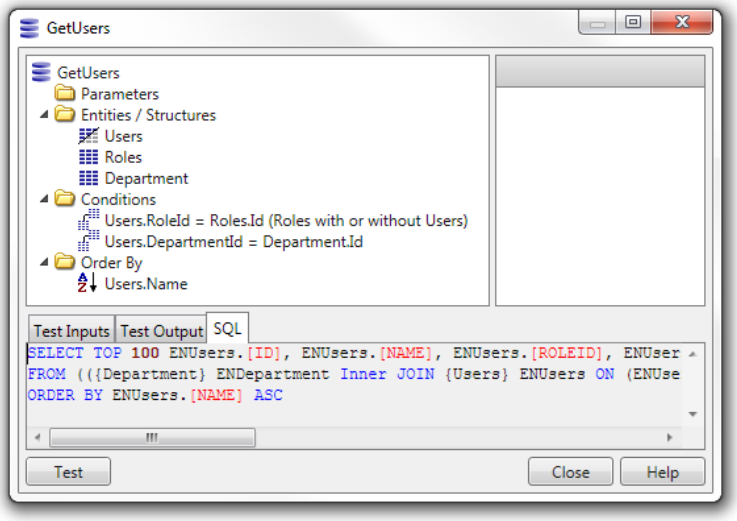


Figure 3.15: *Simple Query* from *Service Studio*

Instead of the intended SQL code (from Listing 3.7), the *Simple Query* in Figure 3.15 generates the SQL from Listing 3.8.

Listing 3.8: SQL generated by the *Simple Query* produced

```
1 SELECT * FROM {USER}
2 inner join {DEPARTMENT} on ({USER}.[depId] = {DEPARTMENT}.[id])
3 right outer join {ROLE} on ({USER}.[roleId] = {ROLE}.[id])
```

Actually, the only way to do this is through the use of an *Advanced Query*, expressing there the exact code from Listing 3.7, as we do on Figure 3.16.

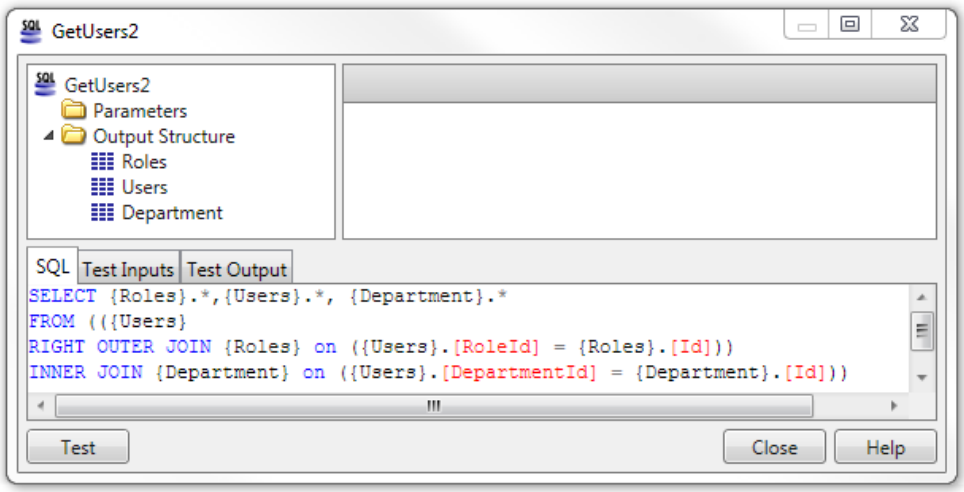


Figure 3.16: *Advanced Query* to force the *Joins* order

In some particular cases (e.g. presence of *null* values), the result can obviously be different since we have a query with a *Right Outer Join* and the left table from this join is used in a subsequent *Inner Join* (Figure 3.17).


Figure 3.17: Screen showing the result of *Simple Query* and *Advanced Query*

Then, another emerging question appears, "***What is the occurrence frequency of a query with Left (Right) Joins succeeded by Inner Joins where the Right (Left) Join involved entity is also used in the On clause of the Inner Joins, in such a way that the joins' order will matter during query processing?***".

To look for the correct XPath expression to this particular case we focus in the example from Listing 3.7. Below follows the XML tree resulted from the parse of that SQL query.

Listing 3.9: XML parse tree from query using *Outer Joins* and *Inner Joins*

```
<Select_Stm>
  <Terminal symbol="SELECT" value="SELECT"/>
  <Restriction/>
  <Column_Source>
    <Terminal symbol="*" value="*"/>
  </Column_Source>
  <Into_Clause/>
  <From_Clause>
    <Terminal symbol="FROM" value="FROM"/>
    <Simple_Join>
      <Simple_Join>
        <Table_Alias>
          <Terminal symbol="VirtualTable" value="{USER}"/>
          <ExpandInline/>
        </Table_Alias>
        <Join_Type>
          <Terminal symbol="RIGHT" value="right"/>
          <Outer>
            <Terminal symbol="OUTER" value="outer"/>
          </Outer>
          <Join_Hint/>
          <Terminal symbol="JOIN" value="join"/>
        </Join_Type>
        <Table_Alias>
```

```
25            <Terminal symbol="VirtualTable" value="{ROLE}"/>
26            <ExpandInline/>
27          </Table_Alias>
28          <Terminal symbol="ON" value="on"/>
29          <Value>
30            <Terminal symbol="(" value="("/>
31            <Pred_Exp>
32              <Id>
33                <Terminal symbol="ComposedId" value="{USER}.[roleId]"/>
34              </Id>
35              <Terminal symbol="=" value="="/>
36              <Id>
37                <Terminal symbol="ComposedId" value="{ROLE}.[id]"/>
38              </Id>
39            </Pred_Exp>
40            <Terminal symbol=")" value=")"/>
41          </Value>
42        </Simple_Join>
43        <Join_Type>
44          <Terminal symbol="INNER" value="inner"/>
45          <Terminal symbol="JOIN" value="join"/>
46        </Join_Type>
47        <Table_Alias>
48          <Terminal symbol="VirtualTable" value="{DEPARTMENT}"/>
49          <ExpandInline/>
50        </Table_Alias>
51        <Terminal symbol="ON" value="on"/>
52        <Value>
53          <Terminal symbol="(" value="("/>
54          <Pred_Exp>
55            <Id>
56              <Terminal symbol="ComposedId" value="{USER}.[depId]"/>
57            </Id>
58            <Terminal symbol="=" value="="/>
59            <Id>
60              <Terminal symbol="ComposedId" value="{DEPARTMENT}.[id]"/>
61            </Id>
62          </Pred_Exp>
63          <Terminal symbol=")" value=")"/>
64        </Value>
65      </Simple_Join>
66    </From_Clause>
67    <Where_Clause/>
68    <Group_Clause/>
69    <Having_Clause/>
70    <Order_Clause/>
71  </Select_Stm>
```

In this case, it is not possible to apply directly XPath to get an answer at this moment since we need to get some attribute values from particular nodes and use them to

compare with other attribute values from different nodes. This means that something additional is needed on the tool such as variables to assign the values, for example.

The closer XPath expression that is possible to build is the one that we present in Listing 3.10, which retrieves all the queries that have *Outer Joins* succeeded by *Inner Joins*.

Listing 3.10: XPath expression to filter particular queries using *Outer* and *Inner Joins*

```
1 /Select_Stm//Simple_Join
2   [Simple_Join/Join_Type/Outer and Join_Type/Terminal/@symbol="INNER"]
```

The sub-set that we intend to get is embedded in the set of queries obtained with the previous expression. Thus, a possible next approach is to understand which tables are involved in the *Outer Join* and check if they are used in the succeeded *Inner Joins*.

### Discussion on the case studies

As shown, the dataset is mainly composed by *Select* statements, and it is the base of our case studies. The case studies and the subsequent analysis aim to improve the expressiveness of the visual query builder, and it only regards the selection of data.

In terms of the question that came from the first case study and taking into account the current version of our tool, we can get the answer about the occurrence frequency of queries with *Complex Joins*, a specified kind of *On* clause condition from an *Outer Join*. This *Complex Join* pattern present a relevant frequency on the dataset, then it is a potential target to integrate the list of features that will extend the new model of the visual query builder.

However, the XPath feature provided in our tool is not enough to answer the second case study since it does not allow us to identify some aspects occurring after a specific element. This creates an opportunity for future work since the tool could be improved, however it is need to find a suitable solution to overcome XPath limitations and to find a proper answer.

These case studies were presented by *OutSystems* regarding decisions taken a few years ago, in a early phase of the development of the *Agile Platform*™ product. At the time these decisions were taken, they were good approaches according to the market and user needs. However, at this moment, R & D team starts questioning if such decisions still make sense, if they are or are not limiting too much the expressiveness of *Simple Queries* and if such decisions need to be reviewed.

## 3.6 Clustering Phase

In this section we explain in detail the clustering phase. To this phase we have defined as our main goal to automatically discover new patterns.

In order to properly cluster our data, we studied several algorithms and decided initially to implement a simplified variant of the algorithm described in [LCMY04]. The

authors describe and show it as efficient and scalable, which is exactly what we need in our context since we are dealing with a large dataset.

### 3.6.1 Implementation of the Clustering Algorithm

[LCMY04] presents an efficient and scalable algorithm for clustering XML documents by structure. It combines a *similarity* metric with a clustering algorithm, partitioning a large collection of XML documents into groups according to their structural characteristics.

The *similarity* between XML documents can be defined using various concepts depending on how the documents are seen. Since XML documents can often be modeled as node-labeled trees, one option is to use tree distance to measure their similarity.

To better understand the algorithm presented in [LCMY04] there is some need to know relevant concepts. The first concept involves to understand what is a graph and what are the elements that compose it since as we will explain in the next sections, the XML tree representing each query will be converted to a graph. Note that a graph is a representation of a set of nodes where some pairs of nodes are connected by edges.

The second concept involves to understand how the distance between XML documents/graphs can be measured and it is explained in the following section.

#### 3.6.1.1 Similarity between XML Documents

**Definition** For two XML documents $C_1$ and $C_2$, the distance between them is defined by

$$\text{dist}(C_1, C_2) = 1 - \frac{|\text{sg}(C_1) \cap \text{sg}(C_2)|}{\max\{|\text{sg}(C_1)|, |\text{sg}(C_2)|\}}$$

where $\text{sg}(C_1)$ and $\text{sg}(C_2)$ is the representation in graph of the XML documents $C_1$ and $C_2$, respectively, $|\text{sg}(C_i)|$ is the number of edges in $\text{sg}(C_i), i \in \{1, 2\}$ and $\text{sg}(C_1) \cap \text{sg}(C_2)$ is the set of common edges of $\text{sg}(C_1)$ and $\text{sg}(C_2)$.

If the number of common element-subelement relationships between $C_1$ and $C_2$ is large, the distance between the graphs will be small, and vice versa. In Figures 3.18 and 3.19, we have the graphs of three different XML documents from our context. If we use the metric from definition stated above, we would have $\text{dist}(\{query_2\}, \{query_3\}) = 0.15$ and $\text{dist}(\{query_1\}, \{query_2\}) = \text{dist}(\{query_1\}, \{query_3\}) = 0.64$. A clustering algorithm based only on the distance between documents (and depending on the similarity threshold given as input, in this example case greater or equal than 0.15) would merge $query_2$ and $query_3$, and leave $query_1$ alone. With this example is shown that the metric is effective in separating documents that have a different structure.

#### 3.6.1.2 Framework for Clustering XML Documents

The purpose of the algorithm is to cluster XML files based on their structure. It can be done after summarizing their structure in s-graphs and using the metric defined in the previous section. This approach is implemented in two steps:

```
<Select_Stm>
  <Terminal symbol="SELECT" value="SELECT" />
  <Column_Source>
    <Terminal symbol="*"  value="*" />
  </Column_Source>
  <From_Clause>
    <Terminal symbol="FROM" value="FROM" />
    <Table_Alias>
      <Terminal symbol="VirtualTable" value="{User}" />
    </ Table_Alias>
  </ From_Clause>
  <Where_Clause/>
  <Order_Clause>
    <Terminal symbol = "ORDER" /> <Terminal symbol = "BY" />
    <Order_List>
      <Simple_Id><Terminal /></ Simple_Id>
      <Order_Type/>
    </ Order_List>
  </Order_Clause>
</ Select  Stm>
```

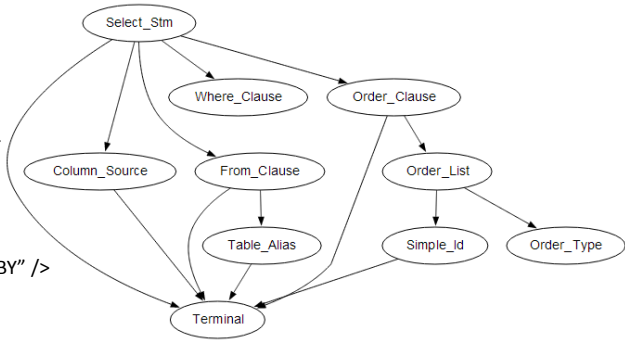


Figure 3.18: Simplified XML and S-graph from $query_1$

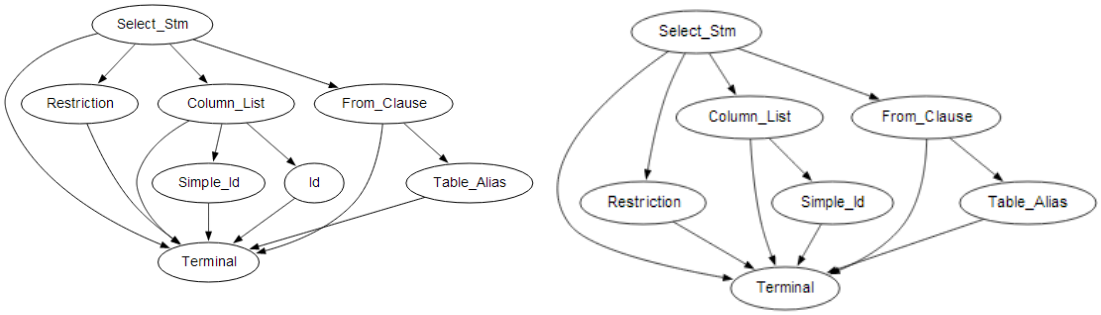


Figure 3.19: S-graph from $query_2$ and $query_3$

- Step 1. Extract and encode structural information: In this step, the documents are scanned, their s-graphs computed and encoded in a data structure.

- Step 2. Perform clustering on the structural information: In this step is applied a suitable clustering algorithm on the encoded information to generate clusters.

#### Extract and encode structural information

In this first step that can be called *pre-clustering*, we are visiting each element of an XML document, creating the representative vertex in a new graph. When we have visited all the elements of the XML, we completed the correspondent graph.

In this graph, also known as structured-graph (s-graph), the edges between vertices represent that the elements are related in the XML with a parent-child relationship ([From parent vertex] -> [To child vertex]). Once the graph is built, we are ready to store it in a data structure and process the next XML document until we have all the collection processed. Furthermore, we associate each graph to an *Id* from the query that generated the XML document responsible for that graph. After process all the collection, we are ready to go to the next step.

#### Perform clustering on the structural information: S-Grace Algorithm

In order to perform clustering on our structural information, we follow one of the studied algorithms, S-Grace Algorithm from [LCMY04], explaining what changes we do during its implementation to better fit in our context.

At the beginning, we start to compute the distance between all graphs, and store them as neighbours or not. So as to understand if they are neighbours or not, it is used a metric that help us to define such relationship. In this case, and as proposed on [LCMY04], we use the definition presented in the previous Section 3.6.1.1 that allow us to calculate the distance between graphs. After compute this distance, we just need to compare it with a similarity threshold ($\theta$) defined by us before start the execution of the algorithm (possible to tune), and if the distance between two graphs is less or equal than the threshold it means both graphs are neighbours/similar.

Afterwards, we need to deal with clusters, however we have to create them first. Initially, each graph represents a cluster, which means that if we have $N$ graphs we start the algorithm with $N$ clusters. Each cluster has some properties that characterize it such a graph, a local heap that we will explain its purpose later on, and, indirectly, a list of all query *Ids* linked to that cluster.

After determining all distances between documents, it is time to fill a new structure called *Link*. This structure is responsible for store the number of common neighbours between a pair of clusters.

Then, we fill the local heap from each cluster. Local heap refers to a max heap of a cluster, and it stores all the existent relationships between it and other clusters. The

value that defines the storage order on the heap is the number of common neighbours computed on the previous stage and that is stored on *Link* structure.

As soon as all the local heaps are filled, all the conditions are satisfied in order to start building the global heap. Global heap refers to a max heap and is there that are stored the number of common neighbours between two clusters and one of these clusters. This global heap defines the merge order of clusters, since in our case we are looking for clusters that have a high number of common neighbours in order to group them in a new cluster. This resulting cluster is defined by the union of each one of the graphs that represented the clusters to be merged, as well as the union of their local heaps.

Once all the previous data structures are filled, the algorithm starts iterating in order to reduce the number of clusters, through a merging process. The pairs of clusters with higher linkage between them start to be merged, until there are no more clusters with any linkage or the algorithm reaches $K$ clusters, and here is mainly where this variant differs from S-Grace algorithm since they propose a different condition to stop the merging process starting once again the algorithm depending also on other parameters. The $K$ value mentioned is also given as input parameter of this clustering algorithm alongside $\theta$.

All the previous explanation can be translated to the pseudocode presented on Listing 3.11.

Listing 3.11: Pseudocode of Clustering Algorithm - variant of S-Grace

```
1  /*Input D: our dataset of XML documents*/
2  /*Input θ: similarity threshold */
3  /*Input K: a control parameter for the number of clusters */
4
5
6  ListOfSGraphs = preClustering(D);
7  Distance = computeDistance(ListOfSGraphs);
8  Link = computeLink(Distance, ListOfSGraphs, θ);
9
10 LocalHeaps = initialize();
11 for each s ∈ ListOfSGraphs do {
12   LocalHeaps[s] = buildLocalHeap(Link,s);
13 }
14
15 GlobalHeap = initialize();
16 GlobalHeap = buildGlobalHeap(Link,s);
17
18 while GlobalHeap.isNotEmpty() and GlobalHeap.size > K do {
19   u = GlobalHeap.extractMax();
20   v = LocalHeap[u].getMax();
21   GlobalHeap.delete(v);
22
23   w = mergeClusters(u,v);
24   for each x ∈ LocalHeap[u] ∪ LocalHeap[v] do {
25     Link[x,w] = Link[x,u] + Link.get[x,v];
26
```

```
27      LocalHeap[x].delete(u);
28      LocalHeap[x].delete(v);
29      LocalHeap[x].insert(w,commonNeighbours(x,w));
30      LocalHeap[x].insert(x,commonNeighbours(x,w));
31
32      GlobalHeap.update(x,LocalHeap[w]);
33    }
34  }
```

### 3.6.2 Execution of the Clustering Algorithm

Due to physical restrictions we need to take into account the memory consumption when filling the data structures, e.g. by releasing data structures as soon as we no longer need them.

Regarding the execution times, they are not as good as expected at first sight since in [LCMY04] they present their worst preprocessing time as around 1,360 seconds against ours 14 seconds. On the other hand, we spend much more time on clustering: they present times of around 1 hour and 15 minutes, against ours almost 3 hours of execution. However, we can try to point out a plausible explanation to justify such gap. We notice the number of s-graphs that they are dealing with in their worst-case scenario is around 7,700 against ours 21,000.

To understand the algorithm results when the execution is finished, we decide to dump all the final clusters into files. Each file is related to a cluster and contains its s-graph, a list of queries grouped inside that cluster, and the total number of queries to analysis effects. This total number of queries *per* cluster can help us to build a histogram with the distribution of queries *per* cluster. The purpose of this histogram is to allow us to analyse algorithm results with specific input parameters. In the end, we are trying to get an interesting distribution over all clusters that allow us to start the analysis looking for patterns. From our point of view, an interesting distribution is characterized by a uniform distribution of queries *per* clusters trying to avoid clusters with few queries. In the case of such thing is not possible, we try to minimize their differences on distribution.

The first problem that we notice in the obtained distributions is that the first 3 clusters have an astonishing difference in terms of number of queries, since the first cluster has around 10,000 queries against 415 queries on the second bigger cluster and 400 queries on the third one (see Figure 3.23(a), at this point we are just testing the algorithm with 12,500 queries and different types of queries, not only *Select* statements). We will try to prevent this algorithm behaviour, seeking for closer number of queries between them and better distributions, and focusing only on the set of *Select* statements (21,077 queries).

#### 3.6.2.1 First change - Modifications on the XML tree

To accomplish better distributions, there are only few factors that we are able to manage. First of all, we look to the XML structure, since after parse the queries we have

stored the XML as it was and there are some nonterminal elements that can be removed once they are without any child in the tree. Without dealing with this problem we could have queries in the same cluster that should not be. For example, if one query is using the *Where* clause with just one condition and another is not even using it, they could be inserted in the same cluster depending on the similarity threshold since both XML representations have an element *Where Clause*. Figures 3.20 and 3.21 depict two different graphs converted from the same XML document (i.e. same query), however Figure 3.20 represents the graph before the modification on the XML and Figure 3.21 represents the graph after the modification on the XML.

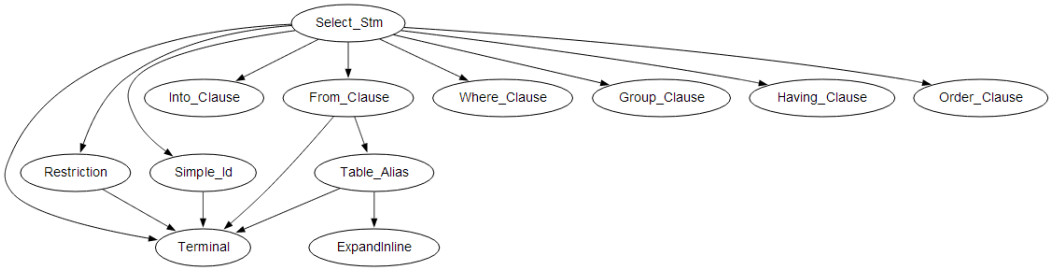


Figure 3.20: Graph from query before performing changes on the XML structure

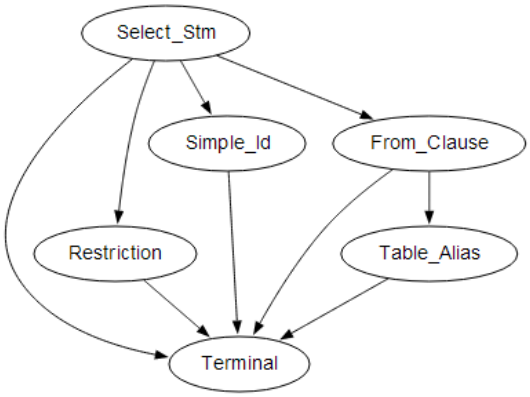


Figure 3.21: Graph from query after performing changes on the XML structure

#### 3.6.2.2 Second change - Modifications on the XML tree

Another factor that can be managed is once again related with the XML structure, more precisely, the terminal elements. At this moment, these elements are considered a problem in terms of efficiency during the clustering, since they are not specific enough when presented in the s-graph not allowing to understand what kind of query elements they really represent. The problem is that all the terminal elements are mapped into a vertex called *Terminal*, which means that, at some point, all vertices from s-graphs will point to a *Terminal* vertex, what can be seen as an ambiguity. The solution is to rename each terminal element, starting to get the value from *@symbol* attribute of the XML terminal

element. As an example, if we have initially an element:

$$< \textbf{Terminal } @symbol = "Distinct" \ ... \ / >$$

After this modification we interpret it as the following element:

$$< \textbf{Distinct } ... \ / >$$

and create the respective s-graph vertex named *Distinct*. In Figure 3.22 is depicted the final graph obtained after perform all the changes mentioned above on the initial XML structure, note that a generic query is readable on this new graph representation, to this particular graph we have a query similar to the one from Listing 3.12.

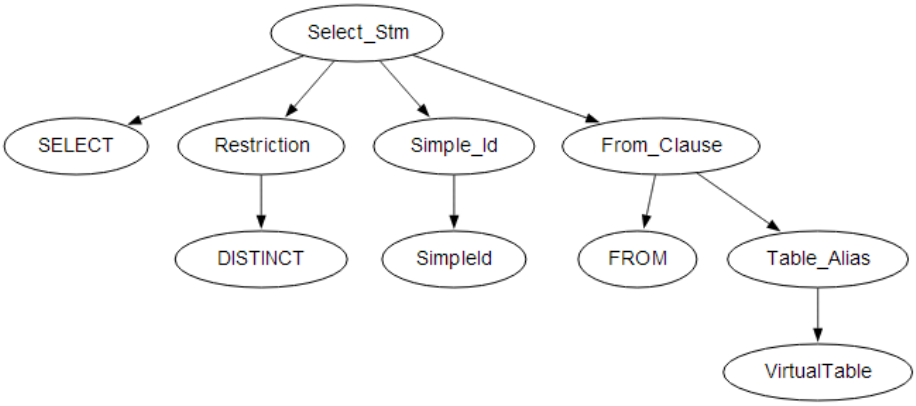


Figure 3.22: Graph from query after performing new changes on the XML structure

Listing 3.12: Query representing the graph from Figure 3.22

```
1 SELECT DISTINCT {Entity}.[Column] FROM {Entity}
```

#### 3.6.2.3 Third change - Tune of Similarity Threshold

Our last move to get better distributions is related with the similarity threshold $\theta$, since it is an input parameter that can easily be changed. This parameter can vary from 0 to 1, if the value is closer to 0 it means that we are squeezing the comparison between documents, and the documents need to have much more in common in structural terms to be considered neighbours comparing when using values closer to 1. At the beginning, we use $\theta$ as 0.25, and we start reducing this value in order to try to distribute the queries over all clusters. We stop reducing on 0.10 with a much better distribution. In Figure 3.23(b) is shown the best distribution that we obtain after all the mentioned changes.

As we can see, the clustering algorithm returns the dataset of queries distributed over a total of 4,095 clusters, with the similarity threshold $\theta = 0.10$ and all the adjustments that we have made on the XML.

| Cluster | Percentage of queries |
|---|---|
| C1 | 80.89% |
| C2 | 3.32% |
| C3 | 3.19% |
| C4 | 2.77% |
| C5 | 0.84% |
| C6 | 0.77% |
| C7 | 0.70% |
| C8 | 0.45% |
| C9 | 0.22% |
| C10 | 0.22% |
| ... | |
| C297 | 0.01% |
| C298 | 0.01% |
| C299 | 0.01% |

(a) Initial distribution - before performing changes

| Cluster | Percentage of queries |
|---|---|
| C1 | 10.33% |
| C2 | 3.28% |
| C3 | 2.24% |
| C4 | 2.01% |
| C5 | 1.44% |
| C6 | 1.34% |
| C7 | 1.10% |
| C8 | 1.02% |
| C9 | 1.00% |
| C10 | 0.91% |
| C11 | 0.86% |
| C12 | 0.82% |
| C13 | 0.78% |
| C14 | 0.71% |
| ... | |
| C100 | <0.01% |
| C101 | <0.01% |
| C102 | <0.01% |
| ... | |
| C4093 | ≈0.00% |
| C4094 | ≈0.00% |
| C4095 | ≈0.00% |

(b) Final distribution - after performing changes

Figure 3.23: Clustering algorithm - distribution of queries *per* cluster

The biggest cluster is now covering more than 10% of the entire dataset. If we focus the analysis on the first 10 clusters, and we will, we are considering a coverage of 25% which is a significant value. In terms of cost/benefit relationship, the analysis of the biggest 10 clusters is enough since to analyse a bigger percentage of coverage we would have to analyse much more clusters guiding us to possible time issues, e.g. to get almost 50% of coverage we would need to analyse all the first 100 clusters.

Thus, we focus our analysis on these first 10 clusters and we need to specify a way to understand the patterns contained in each one of this clusters (characteristics that identify the cluster), which lead us to the visual analysis that we present in the next section.

### 3.6.3 Visual Analysis

As previously mentioned, a cluster is represented – albeit not exclusively – by an s-graph, that is the union of the s-graphs of each query linked to that cluster. Therefore, it is possible to quantify the number of occurrences of each vertex in the context of one cluster, which lead us to a treatment that we made to the s-graph that represent a cluster. We verify the occurrence of each vertex and, depending on that, we assign it a color. To better understand this procedure, Figure 3.24 shows the s-graph from one of the first 10 clusters.

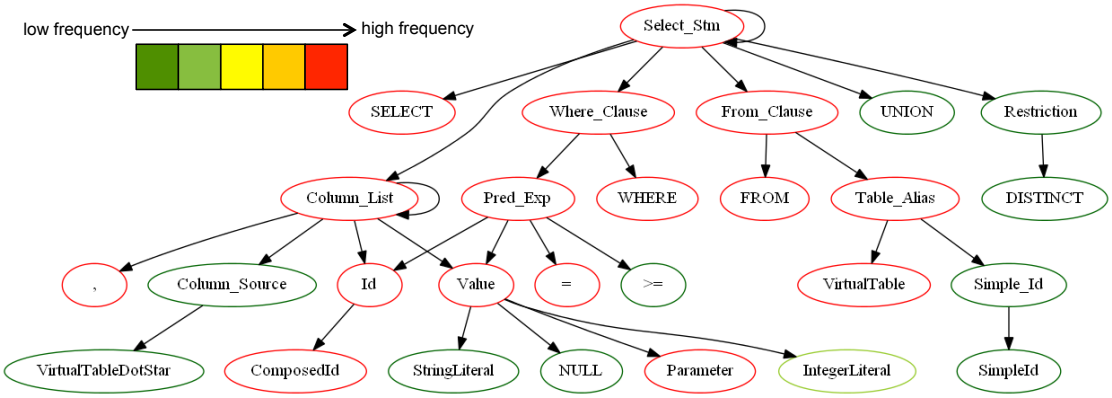


Figure 3.24: Example of a colored graph from a cluster

In this particular case from Figure 3.24, it is possible to conclude that this cluster is mainly characterized by:

- **List of columns** selected from the involved entity, confirmed by the hot node $< Column\_List >$
- All the selected data comes from **only one entity**, which means there are no joins on all the queries of that cluster. It can be corroborated with the non existence of any $< Simple\_Join >$ node on the graph.
- **Only one condition** on the *Where* clause, since there is no nodes representing the $< AND >$ or $< OR >$ operator.

If we think about these characteristics and try to decode them to SQL code, we can point out Listing 3.13 as a query that defines the cluster.

Listing 3.13: SQL structure from visual analysis of Cluster example

```
1 SELECT [Col1],[Col2],...,[ColN] FROM {Table}
2 WHERE [Col] = Value
```

However, the majority graphs are composed by much more vertices, which can difficult the distinction between the relevant vertices and the others. So, we decide to build also an s-graph to each cluster that represent only the *hot nodes* - red vertices - so it is easier to perceive the characteristics of that cluster. Figure 3.25 illustrates the s-graph from the previous example, although only with the *hot nodes*, and as it is possible to see the principal information that were the basis of our previous analysis regarding the characteristics of the cluster still remaining there. We define this kind of s-graphs as *hot nodes graphs*.

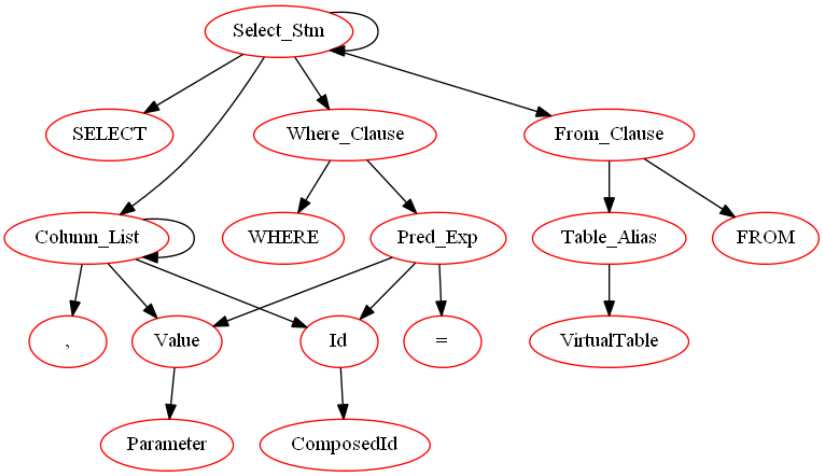


Figure 3.25: Example of a *Hot nodes graph* from a cluster

Beyond the visual analysis, it is possible to complement the cluster analysis by inspecting the set of queries that are linked to a cluster, and also to the histogram of terms that we built in the end of the clustering execution. The histogram of terms is interesting since it allows us to see when a specific term is relevant or not, not only in the context of a cluster but also on the entire dataset. Therefore, both analyses are helpful since they can help to confirm characteristics found during visual analysis but also help to find new ones.

The next section presents the results from visual analysis, always supported by query analysis inside each cluster and also by the advanced SQL terms histogram.

### 3.6.4 Results

In the previous section we explained how visual analysis works, and now we present the analysis results for the first 10 clusters. Note that the clusters are named according their

position in the histogram of distribution of queries *per* cluster, which means the biggest cluster is referred as *Cluster 1*, second biggest cluster is referred as *Cluster 2*, and so on.

For each Cluster we start to present its *hot nodes* graph since the majority of the *colored* graphs are oversized, although when it is possible we present the *colored* graphs. After that, the graph is followed by its visual analysis and we finish with an example of a generic query belonging to that Cluster.

**Cluster 1 (10.3%)**

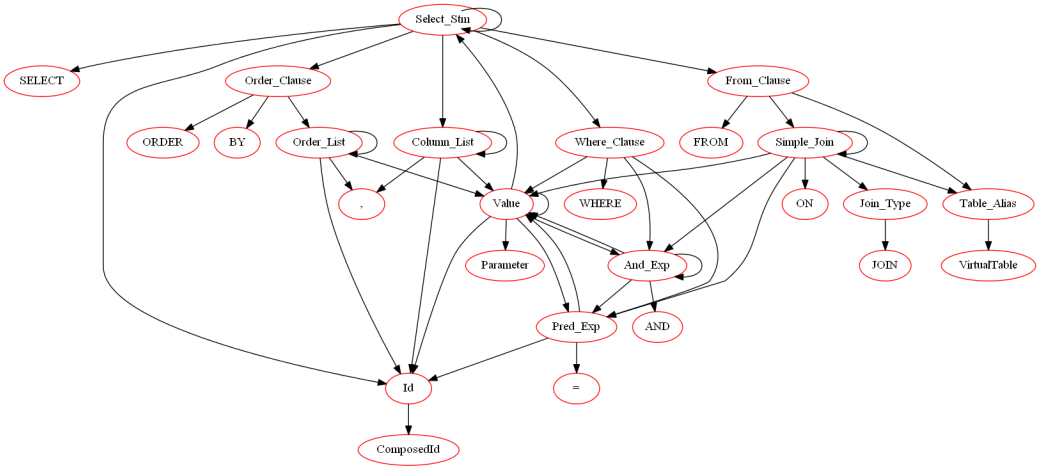


Figure 3.26: *Hot nodes graph* from *Cluster 1*

As shown in Figure 3.26, *Cluster 1* is defined by **select different columns** from the involved entities (*hot-node Column_List*). In terms of entities, it is noticeable that there are **at least one join** on each query from the cluster (*hot-node Simple_Join*). In the *Where* clause there is at least one ***And*** operator (*hot-node And_Exp*), which means that it has always more than one condition. Furthermore, the results use to be **ordered by** only one column (*hot-node Order_Clause* that reflects the existence of Order clause, and after analyse the set of queries from the cluster we can see that the order is, in the majority, by only one column). Listing 3.14 shows the possible generic structure of a query from *Cluster 1*.

Listing 3.14: SQL structure from visual analysis of *Cluster 1*

```
1 SELECT [Col1],[Col2],...,[ColN] FROM {Table} JOIN {Table2} ON ( JoinCond )
2 WHERE Condition AND Condition2 ...
3 ORDER BY [Column]
```

**Cluster 2 (3.3%)**

As Figure 3.27 depicts, *Cluster 2* is also defined by **select different columns** from the involved entities (*hot-node Column_List*). In terms of entities, is noticeable that the results are coming from at least **two entities** since there are a minimum of one Join *per* query (*hot-nodes Simple_Join* and *Join_Type*). After take a look to the complete colored graph we

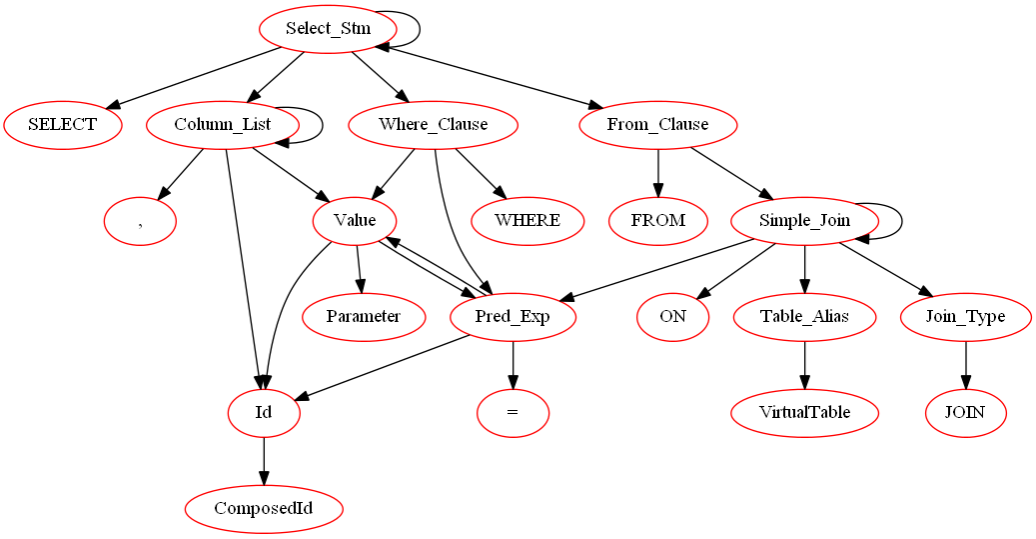


Figure 3.27: *Hot nodes graph* from *Cluster 2*

can also conclude that the referred join is mainly an Inner Join, although the node Inner is not an *hot-node* it is highly frequent. Moreover, in the *Where* clause there is no ***And*** operator, in other words it has **one**, and only one, **condition** (there is not any *hot-node And_Exp*). Listing 3.15 shows a possible generic structure of a query from *Cluster 2*.

Listing 3.15: SQL structure from visual analysis of *Cluster 2*

```
1 SELECT [Col1],[Col2],...,[ColN] FROM {Table} INNER JOIN {Table2} ON ( JoinCond )
2 WHERE Condition
```

### Cluster 3 (2.2%)

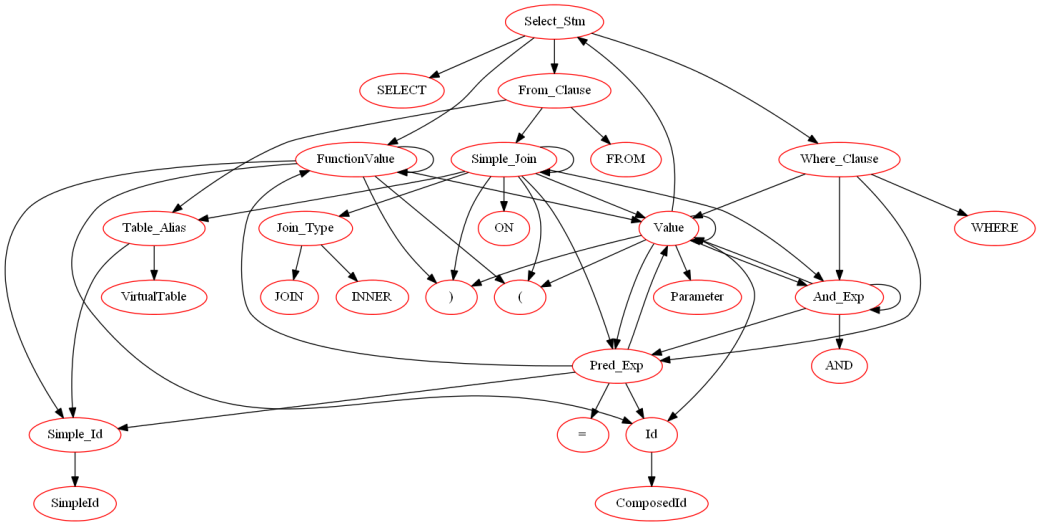


Figure 3.28: *Hot nodes graph* from *Cluster 3*

As presented in Figure 3.28, *Cluster 3* is distinguished by the use of a **function** on the selection (*Select_Stm* node is connected to *hot-node FunctionValue*). This function can be Count() (occurs in 70% of the cases), Sum() (in 29% of the cases), Avg(), Min(), or Max(). There are always two entities involved on each query with an **Inner Join** (*hot-nodes Simple_Join* and *INNER*). The *Where* clause has at least **two conditions** (*hot-node And_Exp*). Listing 3.16 shows a possible structure of a query from *Cluster 3*.

Listing 3.16: SQL structure from visual analysis of *Cluster 3*

```
SELECT function(...) FROM {Table} INNER JOIN {Table2} ON ( JoinCond )
WHERE Condition AND Condition2
```

**Cluster 4 (2.0%)**

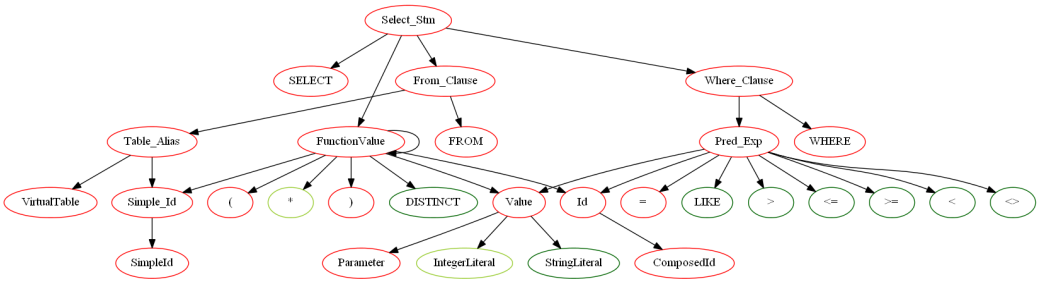


Figure 3.29: *Colored graph* from *Cluster 4*

After analyse Figure 3.29 it is possible to assert that *Cluster 4* is also characterized by a **function** on the selection, once again these function can be Count(), Sum(), Avg(), Min(), or Max() (*hot-node FunctionValue*). Furthermore, there is only **one entity** on the *From clause* (*From_Clause* node connected to *hot-node Table_Alias*) and **one condition** on the *Where* clause (there is not any *And_Exp* node). Listing 3.17 shows the structure of a query from *Cluster 4*.

Listing 3.17: SQL structure from visual analysis of *Cluster 4*

```
SELECT function(...) FROM {Table}
WHERE Condition
```

**Cluster 5 (1.4%)**

As Figure 3.30 shows, *Cluster 5* has the **same properties that Cluster 4** (*hot-node* FunctionValue and no Joins), although the *Where* clause is composed by at least **two conditions** linked with an ***And*** operator (*Where_Clause* node connected to *hot-node And_Exp*). Listing 3.18 shows a possible structure of a query from *Cluster 5*.

Listing 3.18: SQL structure from visual analysis of *Cluster 5*

```
SELECT function(...) FROM {Table}
WHERE Condition AND Condition2
```

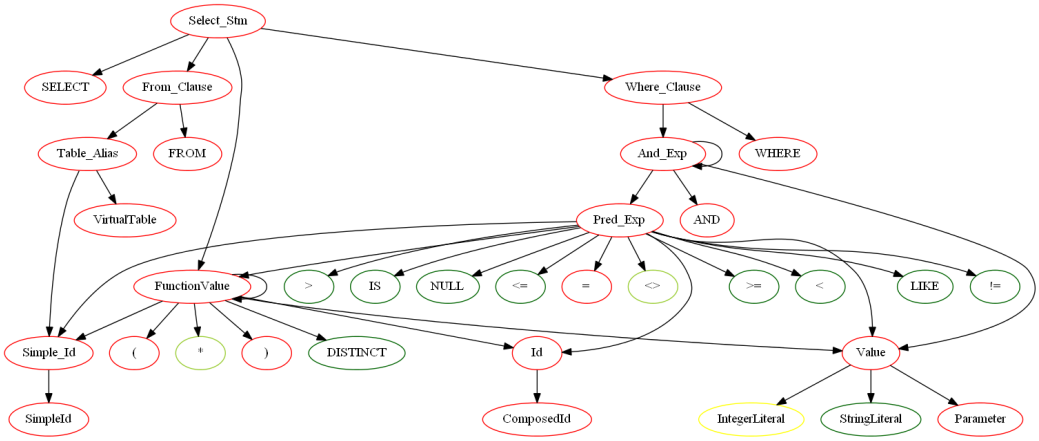


Figure 3.30: *Colored graph* from *Cluster 5*

**Cluster 6 (1.3%)**

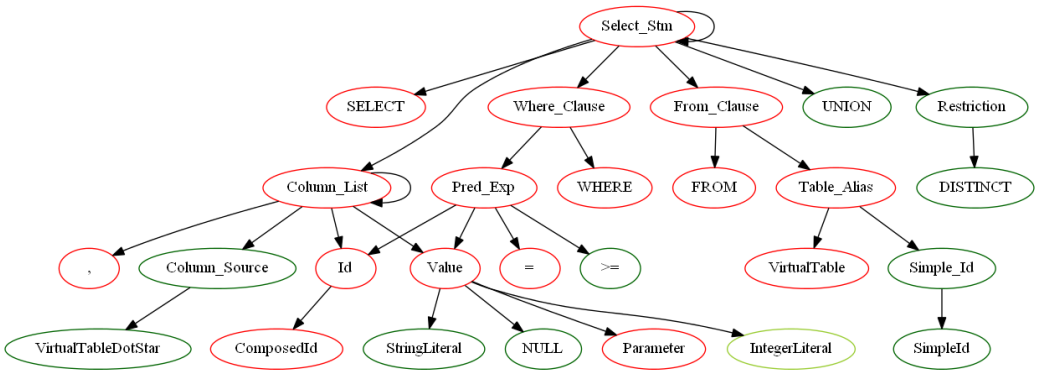


Figure 3.31: *Colored graph* from *Cluster 6*

As Figure 3.31 presents, one of the properties from *Cluster 6* is the **selection of different columns** from the involved entities (*hot-node Column_List*). In terms of entities, it is noticeable that the results are coming only from **one entity** (*From_Clause* node connected to *hot-node Table_Alias*). Moreover, in the *Where* clause there is just **one condition** (there is not any *And_Exp* node). In Listing 3.19 is shown a possible generic structure of a query from *Cluster 6*.

Listing 3.19: SQL structure from visual analysis of *Cluster 6*

```
1 SELECT [Col1],[Col2],...,[ColN] FROM {Table}
2 WHERE Condition
```

**Cluster 7 (1.1%)**

*Cluster 7* is also characterized by **select different columns** from the involved entities. Through visual analysis of the completed graph with all vertices and the histogram of terms for this cluster, it was also possible to note that the *Distinct* clause is very frequently

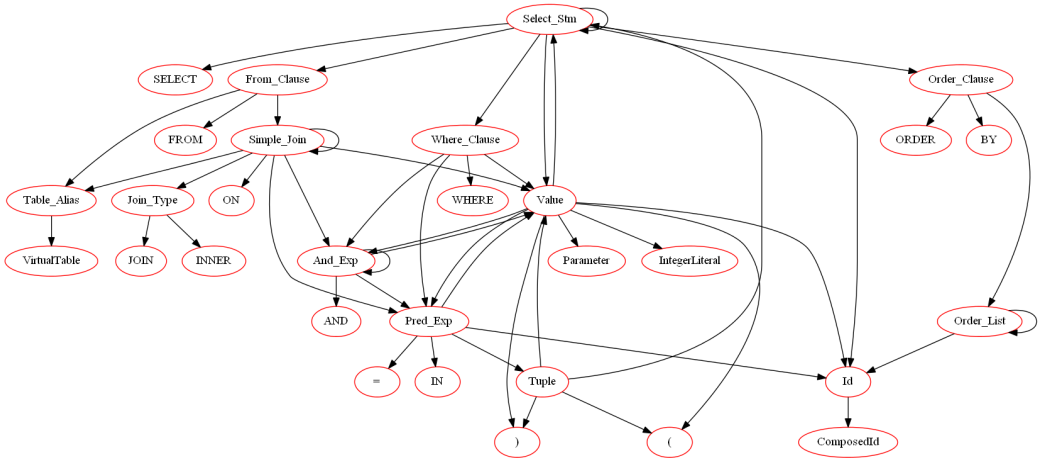


Figure 3.32: *Hot nodes graph* from *Cluster 7*

used, not only on this cluster but also in all the dataset, with an occurrence higher than 17%. As depicted in Figure 3.32, in terms of entities, there is always at least one ***Inner Join*** on the *From clause* (*hot-node INNER* and *JOIN*). The *Where* clause has **several conditions**, where always one of them is checking if a specific column value is ***In*** a list of values returned from a sub-query (*hot-node IN* along with analysis on the set of queries that belongs to this cluster). To complete, all the queries from this cluster are ordering the data to retrieve by one column (*hot-node Order_List* connected to *Id*). Listing 3.20 shows a possible generic structure of a query from *Cluster 7*.

Listing 3.20: SQL structure from visual analysis of *Cluster 7*

```
SELECT Distinct [Col1],[Col2],...,[ColN]
FROM {Table} INNER JOIN {Table2} ON ( JoinCond )
WHERE Condition AND ... AND [ColA] IN (SELECT ...)
ORDER BY [ColB]
```

### Cluster 8 (1.0%)

This cluster brings some interesting and different patterns, once all the set of queries that composed it follows the syntax from Listing 3.21, that is the proper syntax from *Oracle SQL Server* to limit the **maximum number of records** to be retrieved (which can currently be done on *Agile Platform*$^{TM}$ using *Simple Queries*). The selection is managed in order to select a specific **list of columns**, and according Figure 3.33 there are only **one involved entity** (*hot-node From_Clause* connected to *Table_Alias*). Query conditions from *Where* clause are mainly linked with ***And*** operator, however it is also noticed the presence of ***Or*** operator (*hot-node Or_Exp*) but always with both conditions **surrounded by parenthesis**.

Listing 3.21: SQL structure from visual analysis of *Cluster 8*

```
SELECT * FROM (
```

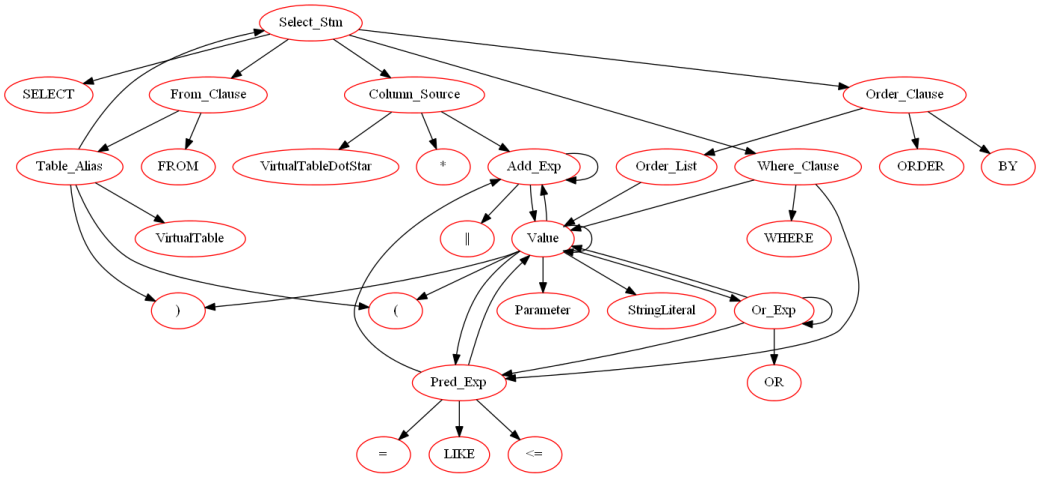


Figure 3.33: *Hot nodes graph* from *Cluster 8*

```
  SELECT [Col1],[Col2],...,[ColN] FROM {Table}
  WHERE (Condition AND condition2) AND ... AND
      (ConditionM OR ConditionN)
  ORDER BY [ColB])
WHERE ROWNUM = @MaxRecords
```

### Cluster 9 (1.0%)

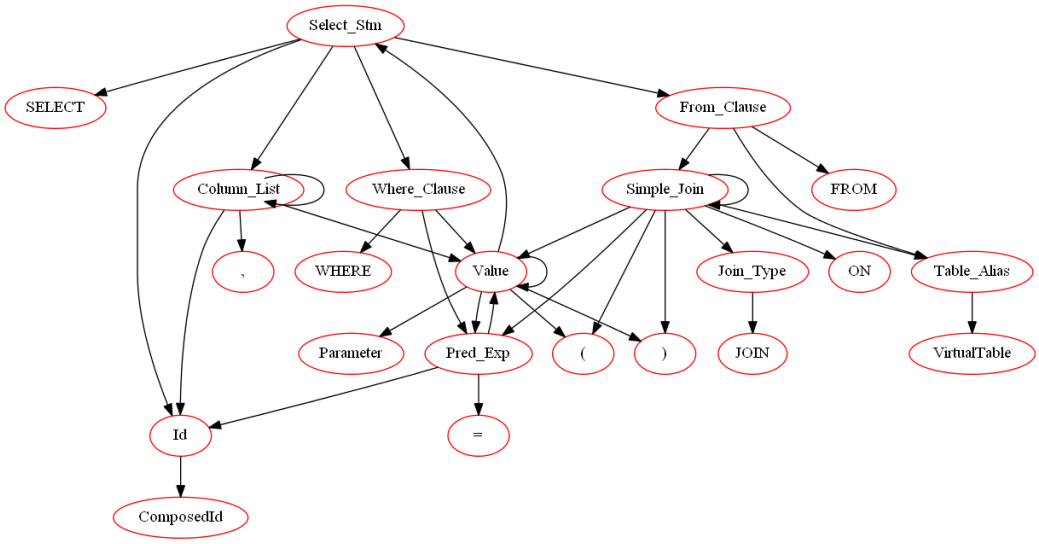


Figure 3.34: *Hot nodes graph* from *Cluster 9*

Figure 3.34 shows that majority of queries are **selecting a list of columns** from different entities (*hot-node Column_List*), which lead us to **at least one Join** *per* query (*hot-nodes Join_Type* and *Join*). With the analysis of the set of queries composing this Cluster, there is one characteristic which stands out and can distinguish it from the others, once a relevant frequency of its queries (almost 55%) are using a *Complex Join* (see Section 3.5.2). In

terms of type of Joins, we can not mention a specific one since the occurrence frequency is distributed between *Inner* and *Outer Joins*. The *Where* clause is composed by only **one condition** (there is not any *hot-node And_Exp*). Listing 3.22 shows the syntax of a query from this cluster.

Listing 3.22: SQL structure from visual analysis of *Cluster 9*

```
SELECT [Col1],[Col2],...,[ColN]
FROM {Table} JOIN {Table2} ON (JoinCond AND JoinCond2)
WHERE Condition
```

**Cluster 10 (0.9%)**

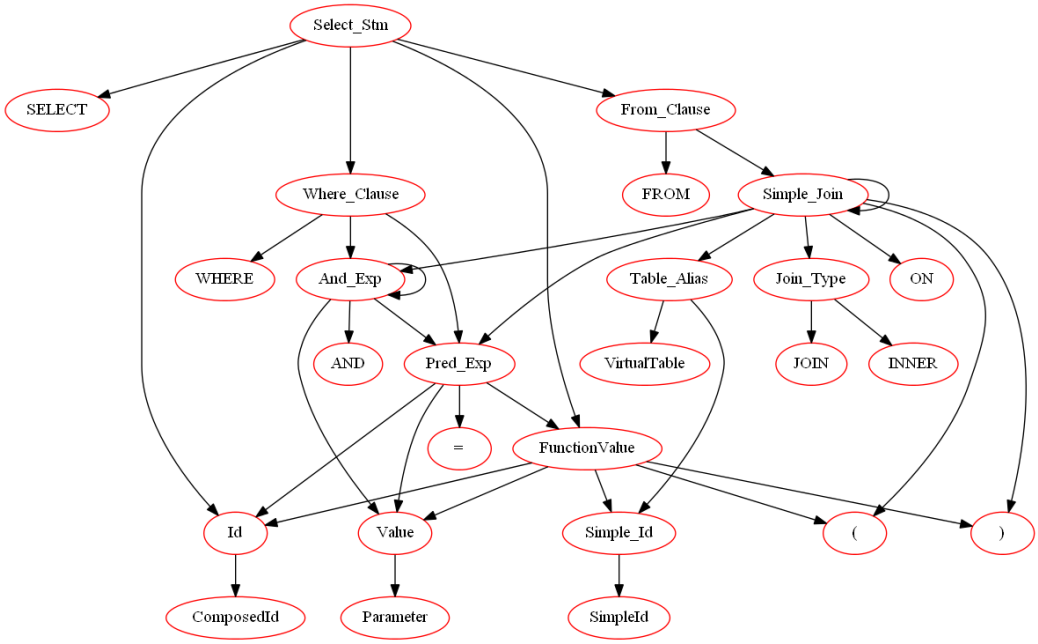


Figure 3.35: *Hot nodes graph* from *Cluster 10*

Once again, as shown in Figure 3.35, we have a cluster where prevails the use of an **aggregate function** such as Count(), Sum(), Avg(), Min(), or Max() (*Select_Stm* node is connected to *hot-node FunctionValue*). ***Inner Joins*** present high frequency in terms of occurrence on the set of queries from this cluster (*hot-node INNER* and *JOIN*). Regarding to query conditions from *Where* clause, it is noteworthy that these **conditions** are always **at least two equalities** (*hot-node '='*) linked with an *AND* (*hot-node And_Exp*). Listing 3.23 shows the syntax of a query from *Cluster 10*.

Listing 3.23: SQL structure from visual analysis of *Cluster 10*

```
SELECT function(...)
FROM {Table} INNER JOIN {Table2} ON ( JoinCond )
WHERE EqualityCondition and EqualityCondition2 ...
```

**In summary** In the Appendix Chapter is presented Table A.1, it shows all the characteristics that we saw over all these previous clusters. The sequence of such characteristics can be considered the patterns that we were looking for.

#### Discussion on the Visual Analysis Results

Regarding the features still not being supported by *Agile Platform™*, it seems reasonable to lay emphasis on the possibility to manage columns to be selected and the ability to use the aggregate function *Sum()*.

Select specific columns as output from the query would not be a problem to *Simple Queries* if the data retrieved would be used in the context of a *Web Screen*, since the platform optimizes such queries in order to select only what will be used on the screen. However, it is a problem if the target to consumption of the retrieved data is a *Web Service*, and if its exposed actions need to receive a list of records (or even a single record) from certain type, usually, from a specific *Structure Reference* or *Entity Reference* that is also exposed by the *Web Service*.

It is interesting to see that some queries from the analysed clusters could actually be written as *Simple Queries*, which can lead to the question *"Why were these queries created as **Advanced Queries** instead of using the visual query builder from **Simple Queries**?"*. Although it is not possible to have 100% sure about a particular answer, we can already try to discuss some ideas on it. The first idea that arises is related with the fact that *OutSystems Agile Platform™* did not support some functionalities that would be needed in the query such as the *Count()* function that just started to be supported after Version 4.2, or even for the simple fact that the developer was not used to the *Simple Query* builder and continued preferred to write queries using SQL.

Coming out of context of *OutSystems* but still referring to visual query builders, we can say exactly the same. If a query is specified with the writing method but however it could have been done using the visual query builder, it can be due to the fact that the query builder is a novelty for developers and they do not want to lose time seeing that new environment and will do it afterwards, or because it is easier for them to do it with the writing approach. Regarding the features that should be supported by these visual query builders at a first sight on the clusters, we can also refer to the freedom to choose what developers want to select on the query (*Selection of columns*), as well as the support for *Aggregate functions*.

Besides the presented clustering algorithm there were other approaches using different algorithms or techniques over structured data that could be followed. Next section refers to some of these related topics.

### Sequential Pattern Mining for Structure-Based XML Document Classification

[GMT06] presents a supervised classification technique for XML documents which relies only on its structure, to show the relevance of using only structured information in order

to detect different "structural families of documents". Each XML document is seen as an ordered labeled tree, represented only by its tags. Their method is composed by three steps, each one represented by its number in the figure 3.36.

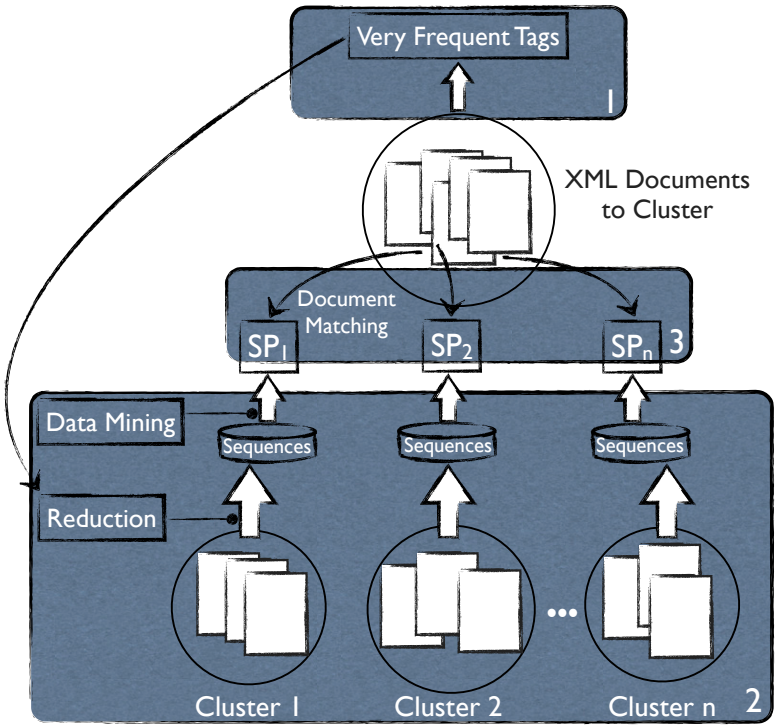


Figure 3.36: Overview of the clustering method

The first step consists in performing a clean over the XMLs in which the frequent tags embedded in the collection are stored. The main goal is to remove some of the most frequent tags since some of them can be considered irrelevant since they do not help separating the documents.

The second is to carry out a data mining step on each cluster from the collection, intending to transform each one of the XML documents belonging to the cluster in a sequence. After this, a mapping operation is carried in order to define which tag exists at some point and in which level of the tree it is located. At the same time this mapping operation occurs, the frequent tags extracted from the previous step (1) are removed. In order to conclude this step, it is necessary to perform another data mining move over the set of sequences produced to extract the sequential patterns. For each cluster $C_i$, they are provided with the set of frequent sequences collected ($SP_i$) that characterizes $C_i$.

Finally the step number 3, where they match each document of the collection with each cluster which is characterized by a set of frequent structural subsequences. The matching technique consists in computing the average matching between the test document and the set of sequential patterns which describes a cluster.

They prove the efficiency of their approach with experiments on a set of XML documents with data describing movies from IMDB database, following the process from

Figure 3.36 and computing the score between each cluster in order to understand the similarity between them. Furthermore, they want to improve the method to better accept certain types of XML documents with similar frequent patterns.

**Relation with our approach** The extraction process described in this work is related to our approach. First of all, each time we parse a query from our dataset we obtain a structured result (see Section 3.3). Then, we can translate it in runtime to a sequence with the nodes properly mapped as they describe in [GMT06]. After that, it is possible to perform an operation to extract sequential patterns from our dataset. Finally, we could follow the process applying the matching techniques to find queries related with the extracted patterns.

Although web usage mining is distant from our work, there are some similarities in what concerns the different stages from the analysis process. In the next section we describe these similarities.

### Web Usage Mining

Web usage mining is one of the categories of web mining, it allows the collection of Web access information for Web pages, taking into account information from server logs such as users history, in order to understand user behaviour and web structure and better serve the needs of Web-based applications.

Web usage mining is divided in three different phases. In [SCDT00], each one of the phases are described as presented below:

- *Preprocessing* - necessary for the purpose of pattern discovery to convert all the information of the usage, content, and structure in several data sources available into the data abstractions.

- *Pattern discovery* - several methods and algorithms like statistics, data mining, machine learning and pattern recognition could be applied to identify user patterns. This phase is divided in 6 sub-stages, where we would like to highlight some of them such as *Statistical Analysis*, *Clustering* and *Sequential Patterns*.

- *Pattern analysis* - this phase targets to filter out uninteresting rules or patterns from the set found during the previous phase. Thereafter, it aims to understand, visualize and provide interpretation to these interesting patterns. Visualization techniques could be used to help during the analysis and highlight overall patterns or trends in the data.

[SCDT00] tried to provide a survey about the fast growing area of the Web Usage mining, once the interest in analyse Web usage data is growing as well to better understand the Web usage and apply the knowledge to better serve users.

**Relation with our approach** In terms of process, the three steps described before could be a good starting point in order to find interesting patterns. The middle phase *Pattern discovery* has the part of *Statistical Analysis* that we will have in mind too, the *Clustering* part can be related with the first grouping that we have in Section 3.5.1. Finally, the *Sequential Pattern* part where we can attempt to find patterns in our the dataset.

### 3.6.5 Discussion

In this Clustering Phase section we gave an overview on the clustering algorithm that we studied and implemented, as well as we presented the results obtained using the clustering algorithm. Then, we needed to understand these results and that is the reason why we propose a new method of analysis, the visual analysis through colored graphs.

After explaining with detail how this visual analysis works, we present the analysis for the first 10 biggest clusters focusing on the patterns presented in each one. We work around these first 10 clusters since they cover 25% of the queries, which can be considered relevant to be the target of the analysis.

In the end, we also presented some interesting work related with this subject, clustering, and we also tried to connect it to our approach. In the next section will be presented another interesting subject related with visualization of datasets to understand them.

## 3.7 Visualization Manager

### 3.7.1 Visualization Tool

The visual understanding of the data that we will work with during the analysis is also an important aspect that should not be forgotten since it can help to understand in a somewhat more general way that data.

Thereby, after searching for different ways to visually understand the data, we found a tool that would allow it through parallel graphics, called parallel-sets. This tool was majority created by Robert Kosara and Caroline Ziemkiewicz [Eag12a] and leverages a Javascript library D3.js [Dav12, Bos12], and it is an open source program, which means that the program code is freely available [Eag12b]. Since we have access to the code, we were able to debug the tool and understand why some errors where occurring when we where trying to visualize our data, and then we fixed them.

Hereupon, to be able to visualize the charts related with our dataset, we need to define what will be the dimensions that should be in the chart as well as the attributes that compose each one of the dimensions. In order to better understand the concept of dimension and attribute of dimension we can present a particular example.

To analyse the accident occurred on the Titanic we can define dimensions such as Sex, Class, Age, Survivor. Each one of the previous dimensions would be composed by attributes, which must be mutually exclusive between them on the same dimension. The dimension Sex would be composed by attributes Male and Female, dimension Class

would be composed by First, Second, Third and Crew. On the other hand, Age would be comprised by Adult and Child, and the last dimension, Survived would be only comprised by the two attributes Yes and No. Figure 3.37 depicts the complete parallel-set of this particular example.

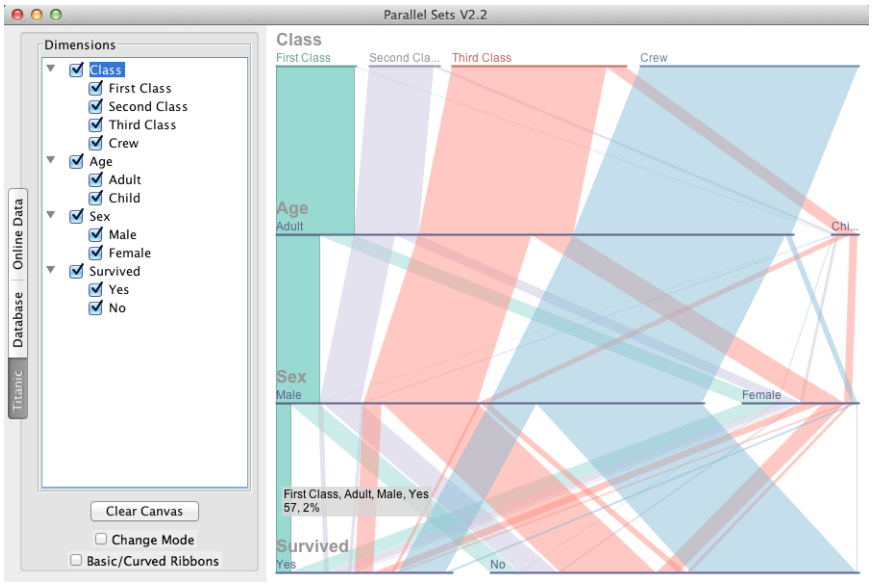


Figure 3.37: Screenshot from the tool running the example of Titanic dataset

Note that all the dimensions and attributes are draggable which will ease the action of question the graphic, since it is easy to do it moving the attribute of each dimension to the left and then we just need to put the pointer over the connection that we are looking for the answer. In the case of Figure 3.37, we are asking "*How many men (Adult and Male) survived that had a 1st Class ticket?*" and we see that the answer is 57 people, or we can also understand the percentage, that in this case is 2%.

Once understood the concepts of dimensions and attributes with the previous example we are ready to present how we have decided, together with R & D team, decompose our dataset.

We have decided to analyse only the *Select* statements in order to understand what should be the simplest patterns to have on the Visualization Manager that would allow to later understand the dataset and explore it. Then, we concluded that the patterns should be organized with the dimensions and attributes as we show below.

- Use of Select statements with **selection** of: just **columns**, any **aggregate** or **built-in functions**, arithmetic expressions with operators (such as addition, subtraction, multiplication operator or division **operators**), Inner **Selects**, **CASE** expressions
- Use of **Expand Inline** parameters : Yes, No
- Use of **DISTINCT** clause: Yes, No
- **Number of tables** involved on the From clause: 1, 2-7, 8+

- If there is any **Inner Join**, **Outer Join** or **Full Outer Join**: each dimension with attributes Yes and No
- Use of **subqueries** on the FROM clause: Yes, No
- Use of **IN**, **OR**, **NOT** or **NULL** on the WHERE clause: each one with attributes Yes and No
- If there is a **HAVING** on the select statement: Yes, No
- The query is **Ordered by**: Column, Using an Aggregate function or simply Not ordered
- If there is any kind of **restrictions** in terms of the **number of records** to retrieve: Yes or No

Since the dimensions and their attributes are already defined, we need to adapt our tool to export a file that can be read by the tool in order to be possible to import our dataset. Figure 3.38 presents a simplified parallel-set from our dataset of queries, since there are only three dimensions selected to be on the chart. Such thing is possible due to this tool that allow us to check what dimensions and attributes we would like to consider for the current chart, using the tree from the left side panel *Dimensions*.

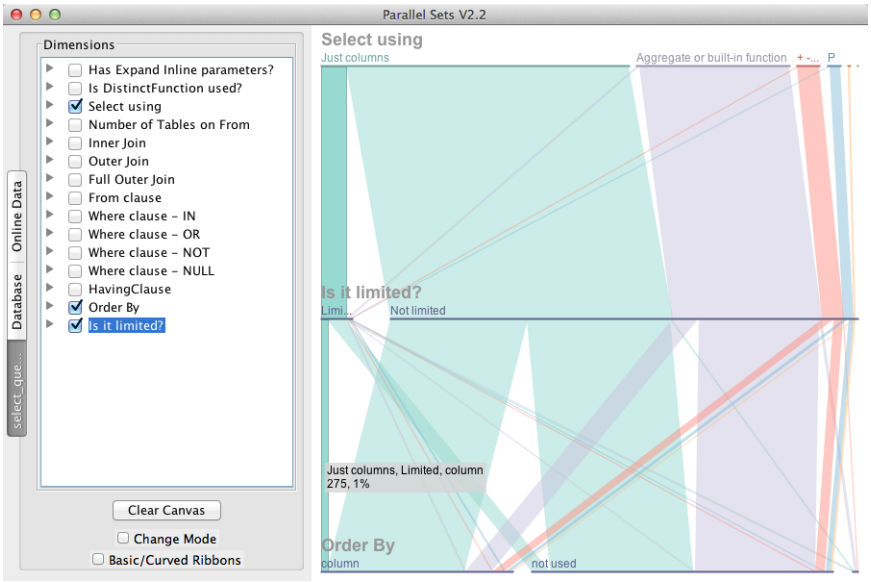


Figure 3.38: Screenshot from the tool running our dataset of queries

Follows some results that we can get through a deep exploration of the data using parallel-sets and selecting also other dimensions:

- Around 61% of the *Advanced Queries* (Select statements in this case) are not concerned with the **order** of the result, whereas 38% of Select queries are ordered by columns and the last 1% ordered using an aggregate function

- In terms of **number of tables** on From clause, 41% have just one table, 51% have from 2 to 7 tables, and the last 8% are the Selects with 8 or more tables involved
- 30% use **Aggregate functions** on the selection
- 30% of Select queries use **OR** operator for disjunction of conditions from the **Where** clause
- 23% of **Select** statements use at least one *Expand Inline* parameter
- 13% use an **IN** on the **Where** clause
- 9% of Select statements use a subquery on their **From** clause
- Less than 1% of these queries are using a **Having** clause in order to filter the result applying conditions using aggregate functions. Which match with the conclusions of the study presented on Section 3.8 that the **Having** clause started to no longer be used.

Although this tool can be a great help to have an initial idea about what is present in the dataset, it does not give the freedom that we would like to have to explore the data, once all the dimensions and attributes need to be predefined.

In the next section, we present some other works related with the visualization of data trying to relate them with our work.

### 3.7.2 Statistical Graphics

As it was previously described, visualization techniques could be used to help during the analysis and to highlight some relevant aspects from a specific data. For this reason we decided to investigate it further and we present some conclusions, explaining how this techniques could be applied in our approach.

In [Tuf01] they present the principles of Graphical Excellence that we list below.

- Graphical Excellence is the well-designed presentation of interesting data - a matter of ***substance***, of ***statistics***, and of ***design***;
- It consists of complex ideas communicated with clarity, precision and efficiency, telling the truth about the data, which means it should be easy to understand and visualize;
- It should provide to the viewer the greatest number of ideas in the shortest time with the least ink in the smallest space;
- And, last but not least, it is nearly always multivariate.

Figure 3.39 (taken from [Tuf01]) is one of the best show cases of Graphical Excellence, and it shows the evolution of successive losses in men of the French Army in the Russian

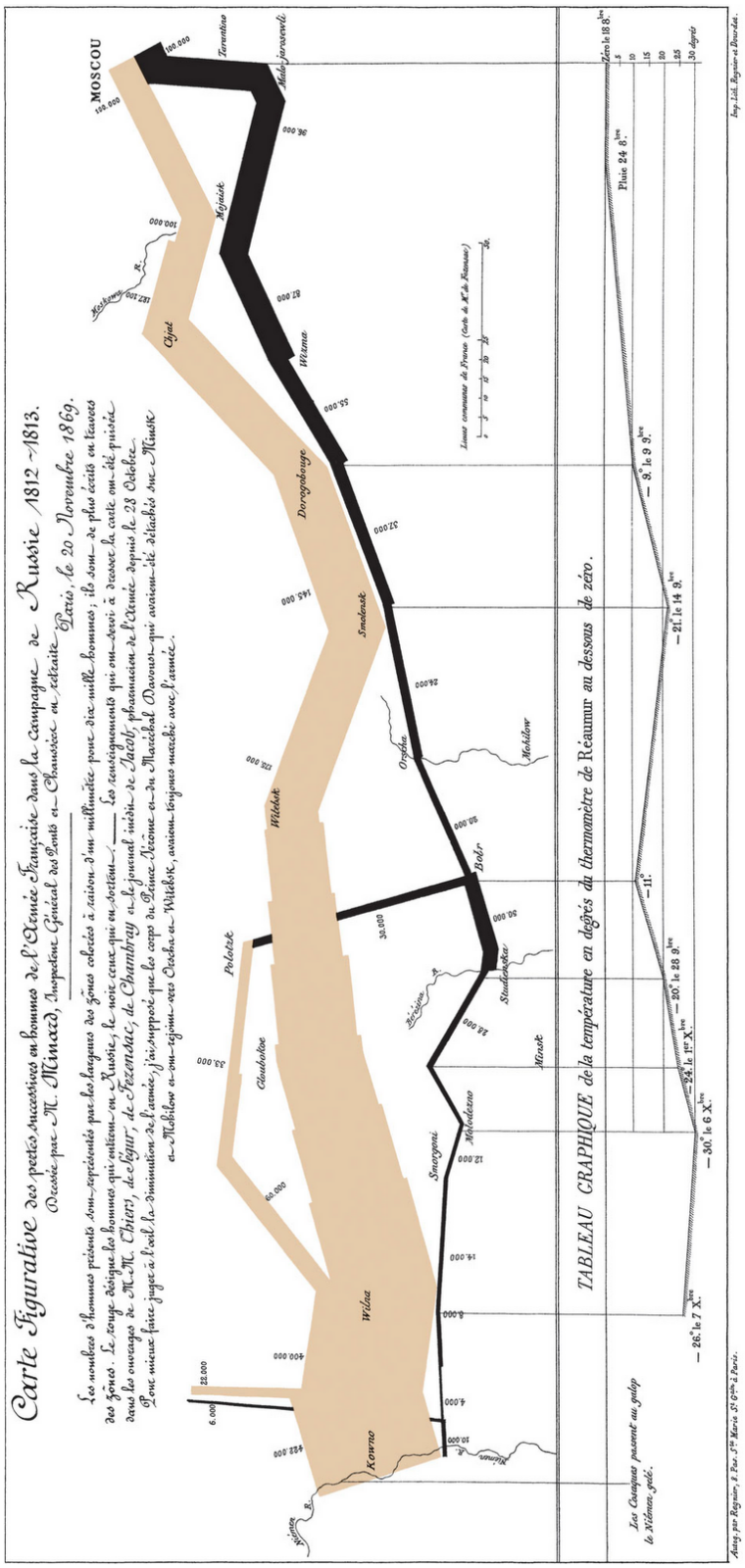


Figure 3.39: Minard's Figurative Chart of Napoleon's 1812 campaign

Campaign (1812-1813). We can see that the army starts the campaign with 422,000 men, and reached Moscow with 100,000 (due to the fact that it was sacked and deserted). During the march back to Poland, caused by bad weather conditions the army starts to lose more men and arrived back to Poland with only 10,000 men remaining.

Thus, this chart from Figure 3.39 shows time, motion and, at the same time, tells a story with a lots of information that allows the "reader" to easily understand all the facts involving the static data.

Based on it and another *Sankey Diagrams*[4] [Phi07], and also trying to solve the problem visualization of flow in the web that works with dynamic data, Google launched a new feature to Google Analytics, the Flow Visualization (Figure 3.40) presented during the event [Goo11]. This feature allows the Webmaster to better understand how people are moving around website's pages.

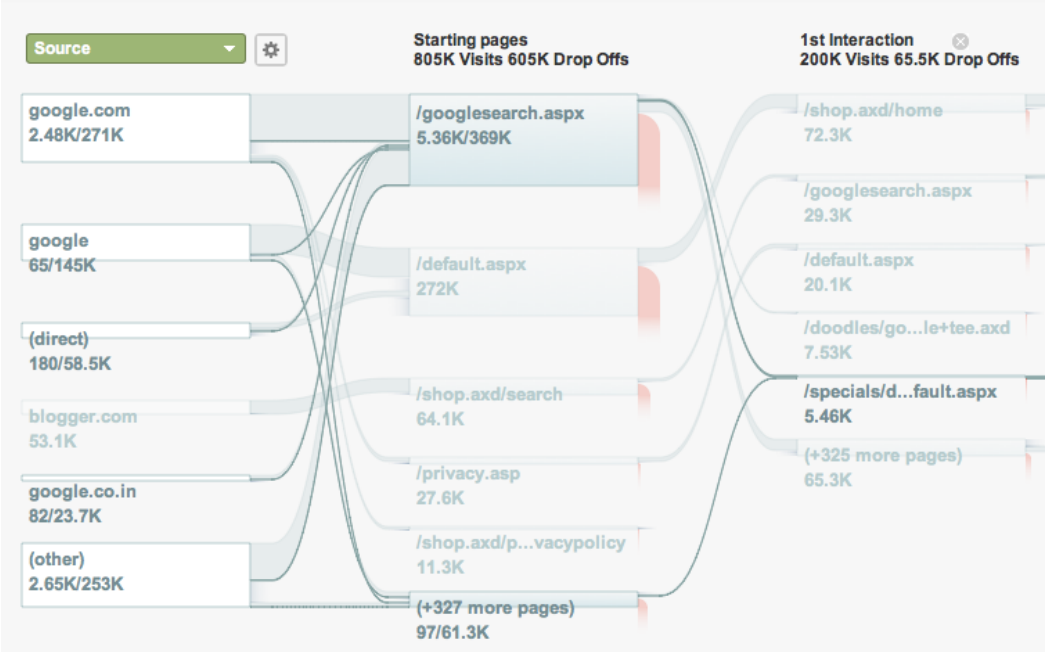


Figure 3.40: Flow visualization feature from Google Analytics

Since we are talking about visualizing lots of data it could not be an easy task to get an answer to the previous problem, in order to allow someone to cosily visualize the web flow. Figure 3.40 illustrates the Visitors Flow view from Google Analytics that provides a graphical representation of visitors flow through the site by traffic source (or any other dimensions) so it is possible to see their journey, as well as where they decided to drop off.

**Relation with our approach** This section shows interesting ways to easily understand the flow and great amounts of data. We could build our tool on this *Sankey Diagrams* to

[4]Sankey diagrams are a specific type of flow diagrams, in which the width of the arrows is shown proportionally to the flow quantity.

understand how the queries are presented in some set and easily understand and manage the visualization of the existing patterns. Imagining the example from Figure 3.41 that we start with a root node Select statement (20.1K occurrences) that is composed by sub-nodes (children) such as From (9K occurrences from 20.1K), and this sub-node once again with children nodes such as Outer Join (6K occurrences from 9K) and Inner Join (3K occurrences from 9K).


Figure 3.41: Example of representation of *Select* statement visualization

However, after we trying to visualize a dataset using *Sankey Diagrams* we discover that it is not so understandable, which means it is not the best way to visualise and understand a dataset. Then, we decide to look for other ways and we find the parallel-sets, that are presented on the previous section.

At this moment, we finished the presentation of all the analysis process and, in the next section, we can present some studies carried out about the usage of SQL which is somehow related with our work. We also describe the differences between their goals and ours.

## 3.8 Use of SQL in Industrial Applications

Some studies have been carried out with the aim of further understanding the use of SQL within industries. [LCW93] present a study with the purpose of gaining a broad view of the use and acceptance of SQL in Singapore's IT industry. With this study it was possible to develop a profile of the major SQL users and applications, depending on the job title of the developers, the application areas of SQL, the complexity of SQL queries formed and the use of particular clauses to **Order** or **Group**. As a conclusion, they found that a very small number of queries were nested more than 3 levels against a quarter with just 1 level and another quarter of the queries with 2 or 3 levels. In terms of relations, half the queries have just 1 or 2 relations, about a quarter involve 3 or 4 relations and the remain queries with more than 4 relations. Regarding to operations such as Intersect, Union and Minus, they discovered that around 25% of the queries use the **Intersect**, about 25% use the **Union**, and as expected, Minus operation was the less used. The **Group** and **Order By** were also target of study with a significant presence (nearly half the queries each one). Furthermore, they detected that related to aggregate functions (Sum, Count, Max, Min, Avg), the most commonly used were **Sum** with around 40% and **Count** in almost 38% of the queries.

Other study [Pö95], similar to the previous one, was developed some years later. It was an empirical study to investigate the use of SQL in commercial applications of three large Austrian companies. Based on 38,000 statements, they analysed the practical meaning of the Data Manipulation Language - part of SQL language constructs. They conclude that the frequency of use of some particular statements depends on the environment, e.g. modify statements are more used in development environment in comparison with a production environment. The use of Joins is not expected since in the production environments they avoid to use **Joins**, from their point of view due to unfamiliarity with the use of joins and uncertain about their performance. Moreover they found the use of **Order By** clause is one of the most important requirements of commercial applications and the most frequently used comparison operator was the equal-operator. Finally, the aggregate functions that were also aim of study and with a relevant presence, mainly the use of **Count** and **Sum** functions.

**Relation with our approach** This approach fits in our work because it analyses the use of SQL in the Business World. As we can see, some years ago, they just worried about the use of clauses such as *Order by*, *Group by*, Aggregate functions, among others, and also about the complexity of written SQL in order to distribute the developers along different profiles. Even if some of these points still be interesting to study about their use and frequency, at the moment the concerns changed, and in this particular case, we are dealing with the attempt to reduce written SQL improving the use and experience of visual query builders, whether to make the developers life easier or to do not have the need of developers with so much skills. Consequently, we are mainly interested in what the visual query builders do not provide and what is frequently written in SQL as we analysed in Section 3.5.

## 3.9 Discussion

In this Chapter we explained how we extract all the queries to create a dataset to support our analysis, and some interesting *OutSystems* case studies.

In a first stage, we were ready to carry out the analysis starting with a term histogram that allows to understand, albeit limited, the frequency of some terms in queries. Since this term histogram was not a solution to the problem, we showed an adapted parser responsible to parse a program that follows some grammar, in the presented case we are parsing queries with a syntax closer to SQL (variant of SQL) and using the proper grammars provided by *OutSystems*.

Afterwards, we focused our discussion in the *OutSystems* context in order to try to find an answer for the provided case studies. The case study from Section 3.5.2 has a definitive answer, however, the second case still needs to be studied since it cannot be correctly answered using the current specific pattern finder feature (using XPath) from our tool.

In the next Chapter we identify the most frequent patterns and present a proposal for a novel *Simple Query* model.

# 4

# Model Proposal

In this chapter we present an overview of the features we identified as possible additions to the current *Simple Query* model, justifying our choices with the results of the analysis presented in the previous chapters. Moreover, we will add a propose in terms of the order of implementation of the identified features as well as a propose of how to integrate such features with *Agile Platform*$^{TM}$.

The order of implementation is of the highest importance since in the business domains there are schedules to comply and goals to achieve, which can create some technical issues in terms of what can be done in the remaining time. Thus, some goals need to be split into several small goals ordered by priority (cost-benefit relationship), e.g. improve the expressiveness of *Simple Query* model which can be split into several parts, each one regarding the extension of the model with one feature and its priority considers the gains offered versus costs of the feature for that business and product.

## 4.1 Most Frequent Identified Patterns

After the analyses from the previous chapters, we are facing different patterns that result in a list of key features getting to the target of the new model proposal. In the following sections we present these features as well as a scenario where they can occur and the justification for extend the model with them regarding previous analyses.

### 4.1.1 Selection of Columns

#### What characterizes it?

The developer should be able to manage the output of a query. Instead of what happens currently, where just entities have meaningful values assigned, it should be also possible to select a structure as query output and define which value should be assigned to each field/attribute from that structure.

#### When does it occur?

A common use case where such feature would be worthwhile and would bring some extra value to *Simple Queries* is precisely when a developer wants to send some data to be consumed by a Web Service over an exposed action. The problem is that the action needs to receive a specific type of records, usually, a structure with some specific fields.

Listing 4.1 shows some examples of queries following the pattern identified in this section. The first query refers to the selection of just two columns from an entity *Users* instead of all its columns. Another case where this pattern can occur is shown on the second query, the results to retrieve should have five attributes *per* row however we just can fill two of them with the First and Last Name from *Users*. Thus, the developers need to assign default values in the other fields as it is possible to see with ", NULL and 1.

Listing 4.1: Examples of SQL queries *Selecting* specific values to retrieve

```
1 SELECT {Users}.[FirstName],{Users}.[LastName]
2 FROM {Users}
3
4 SELECT '',{Users}.[FirstName],{Users}.[LastName],NULL,1
5 FROM {Users} WHERE [Col] = Value
```

#### Why should the model be extended with it?

The decision of extend *Simple Queries* with this feature is supported by cluster analysis from Section 3.6.5, query analysis inside clusters, as well as by collected feedback from different scenarios.

### 4.1.2 *IN* Operator

#### What characterizes it?

*In* operator allows specifying multiple values in the query condition, it introduces the possibility to specify a list of values and check if another value belongs to that list.

**When does it occur?**

**Scenario:** The University administrator wants to list all the Users that have one of the roles, assuming that the Role Ids are known *a priori* by the Administrator (looking for RoleId 1, 2 or 3).

Listing 4.2: Example of SQL query using *IN* operator

```
SELECT {Users}.* FROM {Users}
WHERE {Users}.[RoleId] IN ('1','2','3')
```

**Why should the model be extended with it?**

This pattern was detected in different clusters and proven that it occurs in almost 16% of queries from all the dataset of *Select* statements.

### 4.1.3 Complex Joins

**What characterizes it?**

In our context a *Complex Join* occurs when the Join condition is not just a simple equality between two columns or is a condition composed by at least one **AND** or one **OR** operator. This particular type of joins was the subject from our case study presented in Section 3.5.2.

**When does it occur?**

**Scenario:** A member from the University staff wants to get all the Users from a specific Department joining them also by a particular RoleId

Listing 4.3: Example of SQL query using *Complex join*

```
SELECT {Users}.* FROM {Users}
LEFT OUTER JOIN {Department}
  ON ({Users}.[DepartmentId] = {Department}.[Id] AND {Users}.[RoleId] = 1)
```

**Why should the model be extended with it?**

*Complex joins* were defined as a key feature during the first case study presented in Section 3.5.2, after being cleared that it occurs with significant frequency.

### 4.1.4 *Distinct* Values

**What characterizes it?**

Occasionally, duplicated values can appear within the retrieved data. This could not be a problem, however sometimes the developer will need to retrieve just the distinct values and, currently, to do that using *Agile Platform*™ the developers would need an *Advanced Query*.

**When does it occur?**

**Scenario:** The University staff would like to know which departments from the institution have users from Lisbon. The result should be the list of distinct department names.

Listing 4.4: Example of SQL query selecting *Distinct* values

```
SELECT DISTINCT {Department}.[Name]
FROM {Users} INNER JOIN {Department}
  ON ({Users}.[DepartmentId] = {Department}.[Id])
WHERE {Users}.[City] = 'Lisbon'
```

**Why should the model be extended with it?**

During cluster analysis, we discovered that a specific cluster used it with high frequency, and such behaviour led us to detect that around 17% of the dataset was also using *Distinct* keyword. For this reason it is considered a key feature in our model proposal.

### 4.1.5 Aggregate Functions

**What characterizes it?**

Aggregate functions are characterized by performing a calculation on a set of values and return a single value. All aggregate functions ignore null values except Count().

In this context, we focus in specific functions such as Count(), Sum(), Avg(), Max() and Min() functions. Note that *Group By* is handled separately as you will see on Section 4.1.7.

**When does it occur?**

**Scenario:** An University accountant wants to create a report to the University secretary. One of the details that he wants to attach to the report is the average of salaries of professors (RoleId = 1).

Listing 4.5: Example of SQL query from scenario using *Aggregate functions*

```
SELECT Avg({User}.[Salary])
FROM {Users} WHERE {User}.[RoleId] = 1
```

**Why should the model be extended with it?**

Despite the fact that Count() function is already supported by *Simple Queries*, its use is highly restricted since it only allows to know the number of rows that would be retrieved by a query. Furthermore, it is useful to have available other functions such as Sum(), Avg(), Min() or even Max().

Such a conclusion is entailed after visual cluster analysis, backed up by *Advanced SQL Term Histogram*. Histogram shows that Count term is presented in almost 20% of the *Select* statements, against 9% from Sum term and almost 5% of Max, Min and Avg terms.

### 4.1.6 *Append* Literals

**What characterizes it?**

It allows appending values from different columns retrieving just one column or even adding a specific literal to some column.

**When does it occur?**

**Scenario:** The employee of the secretary wants to do the list of all the Users from the University ordered first by their LastName and after that by their First Name.

The results should be retrieved as one field only (full name), thus the employee should append first name with last name and separate them with a comma.

Listing 4.6: Example of SQL query using *Append literals*

```
SELECT {Users}.[LastName] || ", " || {Users}.[FirstName]
FROM {Users}
ORDER BY {Users}.[LastName], {Users}.[FirstName]
```

**Why should the model be extended with it?**

The decision to support this feature in the new model is taken according terms histograms from several clusters where the operator occurs with some frequency, and regarding all dataset where it occurs in almost 7% of the queries.

### 4.1.7 *Group By* Columns

**What characterizes it?**

The *Group By* statement can be used in conjunction with the aggregate functions to group the result-set by one or more columns. However, if there are no aggregate functions in the query it can produce the same results as the *Distinct* operator.

**When does it occur?**

**Scenario:** The University accountant wants to build a report containing the average salaries grouped by role.

Listing 4.7: Example of SQL query using *Group By*

```
SELECT {Role}.[Name], Avg({Users}.[Salary])
FROM {Users} INNER JOIN {Role}
  ON ({Users}.[RoleId] = {Role}.[Id])
GROUP BY {Role}.[Name]
```

**Why should the model be extended with it?**

The need of this feature can be justified through its presence in almost 9% of the queries from dataset. It brings much more freedom to developers, e.g. it eases the creation of reports. However, note that its use is much more valued when combined with aggregate functions, which can create some concerns about possible dependencies in terms of what needs to come first.

## 4.2 Defining an Order of Implementation

Each feature has its own impact on the entire solution. Thus, the model proposal should follow an incremental process, i.e. we propose the model idealizing and designing feature by feature in order to better understand what is the value added by such feature.

Also it is important to understand if there are any dependencies between the identified features, and that is what we present in the next section.

### 4.2.1 Dependencies

Analysing the set of identified features, it is reasonable to say that *Distinct*, *Append literals* and *Aggregate functions* will need to be used in the context of certain columns selected, which represent that these three features depend on the *Selection of columns*. *Group By* feature is not really depending on *Aggregate functions*, however in our opinion and according developers it makes much more sense to use simultaneously with *Aggregate functions*, that it is the reason why it is considered as a dependency.

In relation to *In* operator with a list of values, it is independent from other features, however it creates a dependency to the implementation of *In* operator with a sub-query.

Regarding *Complex Joins*, it is the only feature that is totally independent from the others, which means that it can be implemented without concern about other features. All the previous identified dependencies are shown in Figure 4.1.

In the meantime, we discuss this topic with R & D team and we come to some conclusions. All features involving sub-queries have high costs in terms of visual changes to the current *Simple Query* model and also on the implementation time needed to have it ready to insertion on the product. Even not proposing new features that depend on sub-queries, we decide to study their impact in terms of coverage and then we can answer to "*What would be the coverage of queries the new model would cover with these features?*".

### 4.2.2 Heuristic

After realizing what are the features to be extended in the new model and their dependencies, it is time to define an order of implementation. This order definition becomes necessary, once in the software businesses / real markets although predictable it is impossible to precise how many features can be implemented in a specific period of time. This constraint is due to several variables that can change over the time such as potential

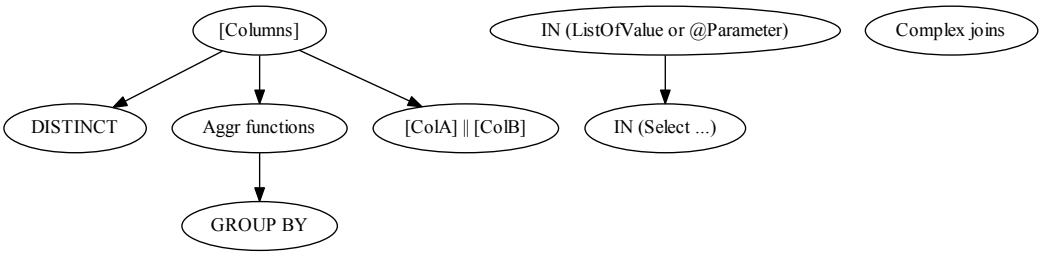


Figure 4.1: Dependencies between identified features

unexpected problems that may arise, the need to focus human resources in other tasks, among others.

To accomplish an acceptable order of implementation we need to define a strategy which involves creating an heuristic that allows us to find a plausible solution to order the implementation process. This heuristic is based on extend the model always according the feature that will bring a higher immediate gain in terms of coverage. In other words, if we have to choose between a *feature A* that gives a total of 17% of immediate coverage, and a *feature B* that gives just 2% of coverage, we choose to implement first *feature A*. Thus, it is an incremental process once we extend the current *Simple Query* model with a new feature on each iteration.

To verify what is the coverage offered by a specific *feature A*, we have a grammar that supports all the implemented functionalities until then and we add it new rules in order to start also supporting the new *feature A*. Thereby, we are able to parse all *Select* queries from the dataset that were considered *Advanced Queries*. The parse action allows us to verify the percentage of queries that is valid against the built grammar. This percentage of queries without any kind of syntax problems is considered the new coverage of queries following that new intermediate grammar. It is noteworthy that every time we add rules to a grammar it represents a new intermediate grammar, which means that we will have several intermediate grammars until we have a final grammar supporting all the identified features since we will do several iterations. Figure 4.2 illustrates how the proposal process works.

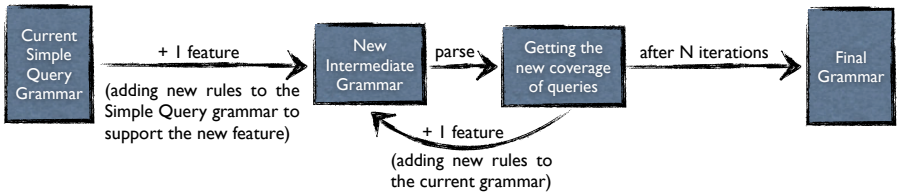


Figure 4.2: Model proposal process

### 4.2.3 Order of Implementation

This section defines the path to follow to integrate all the key features in a new model. According the identified dependencies from Figure 4.1, there are only three features that

are completely independent:

- Selection of columns
- In operator with list of values
- Complex joins

Since the above features are free of dependencies, one of them is our start point in terms of first feature to be implemented. According the heuristic defined, after create all the three grammars that support all the functionalities of *Simple Queries* plus each one of the features, we test them against the dataset of *Advanced Queries*. The results are shown in Figure 4.3

Note that, since we discovered on the clustering analysis that there are several queries on the dataset that have patterns that are already supported on *Simple Queries*, we decide to quantify them. To do that, we buid a grammar with all the rules supported by *Simple Queries* and then we parse all the dataset of *Advanced Queries* to analyse how many of them are successfully parsed. We find out that 3,725 queries are considered *Simple Queries*, which means that our dataset of *Advanced Queries* is reduced to 17,352 queries $(21,077 - 3,725)$ from now on, and all mentions to coverage will be related with this reduced dataset.

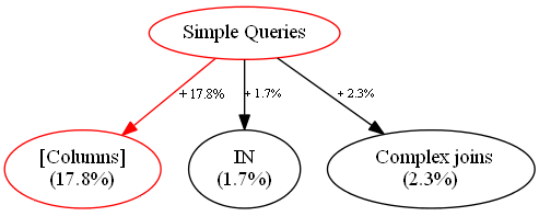


Figure 4.3: Heuristic - extending *Simple Queries* with the first new feature

Figure 4.3 depicts the astonishing increase of 17.8% in terms of coverage with the feature of *Selection of columns*, against 1.7% with *In* feature and 2.3% with the addition of *Complex Joins*. Thus, the feature that offers greater immediate gains and that we propose to be implemented first is the *Selection of columns*.

Once the first new feature to add is already defined, we can do another iteration using the defined heuristic without forgetting the dependencies. Figure 4.4 shows the gains obtained to each one of the possible features to this iteration.

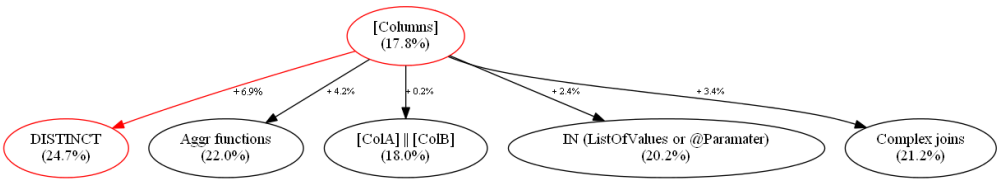


Figure 4.4: Heuristic - extending *Simple Queries* with the second new feature

This time the gains are smaller than in the first iteration and the feature that stands out is the *Distinct*. Starting to support queries with *Distinct* values (after introducing the possibility to manage the selection of columns) brings us gains of 6.9%, against 4.2% from adding support to *Aggregate functions*, 3.4% from *Complex Joins*, 2.4% from *IN* operator with list of values or parameters, and 0.2% from *Append of literals*. Thus, the feature that offers greater immediate gains and that we propose to be the second feature being implemented is the *Distinct values*.

Since the second new feature to extend the model of *Simple Queries* is defined, we can go to the next iteration using the defined heuristic and taking into account once again the dependencies defined in Section 4.2.1. Figure 4.5 shows the gains obtained to each one of the possible features to this iteration.

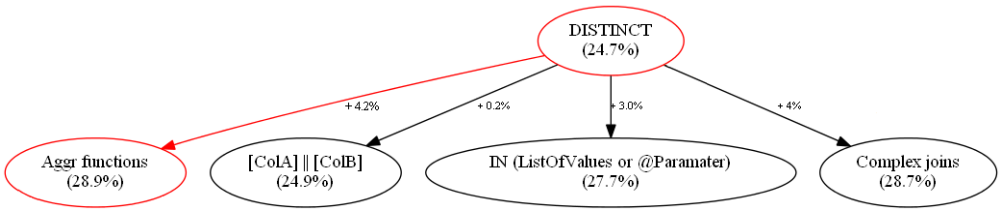


Figure 4.5: Heuristic - extending *Simple Queries* with the third new feature

Following the heuristic, after extending the model with *Selection of columns* and *Distinct values* features, what brings more immediate gains are the *Aggregate functions* with 4.2%. The other three features that could be added on this iteration are the *Append of literals* with 0.2% of gains, *In* operator with 3.0% and, last but not least, *Complex joins* with worthy gains of 4% which means that could bring also high gains. Thus, the third feature to implement is the *Aggregate functions*.

Another iteration of the heuristic is done and the third new feature to extend is defined, then we can continue iterating the heuristic. This iteration allows us to choose between four different features as shown in Figure 4.6, and the next feature extending the model is the *Complex Joins*, offering more 4.2% of coverage.

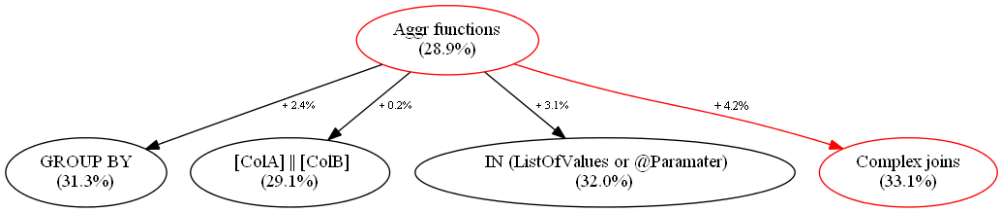


Figure 4.6: Heuristic - extending *Simple Queries* with the fourth new feature

After define the fourth new feature to extend the model we can perform a new iteration of the heuristic. There are three features remaining and, according the iteration results that are presented in Figure 4.7, it is the *In operator* the next feature to be proposed

since it presents immediate gains of 3.5%. *Group By clause* would allow an increase of 2.6% in terms of coverage, and *Append literals* brings a very small percentage of 0.6%.

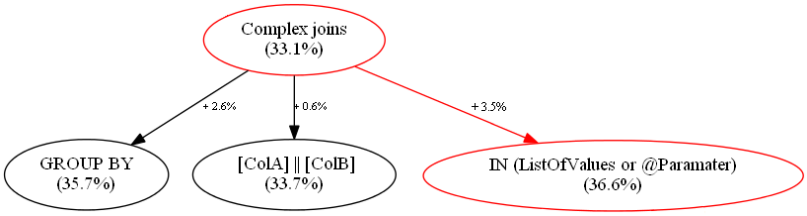


Figure 4.7: Heuristic - extending *Simple Queries* with the fifth new feature

At the moment, the intermediate grammar is composed by all the features supported by *Simple Queries*, as well as *Selection of columns*, *Distinct values*, *Aggregate functions*, *Complex joins* and *In operator*. There are only two features remaining to have the final model complete and these features are *Group By clause* and *Append Literals*.

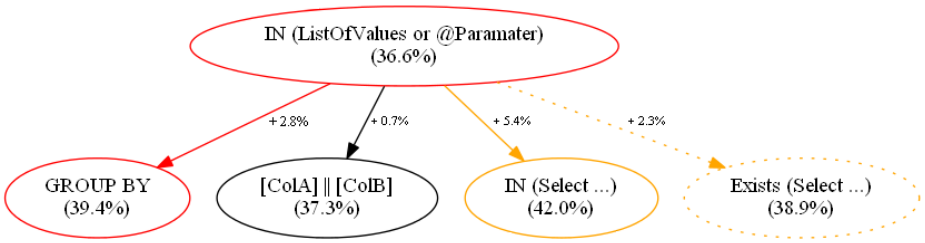


Figure 4.8: Heuristic - extending *Simple Queries* with the sixth new feature

As illustrated in Figure 4.8, the next feature to implement is the *Group By* offering gains of 2.8%. However, if we did not consider the implementation costs of features with sub-queries, of course the natural path to follow would be the implementation of *IN operator* with sub-queries since it would increase the coverage in 5.4%.

Another worthy note is related with the *Exist* operator (that offers 2.3% of gains) since in our context the queries that are using it can also be expressed using the *In* operator, which lead us to a conclusion. If we would propose any features involving sub-queries, we could present a proposal regarding only the *IN* operator and consider as gains of coverage at least the 7.7% offered by both features (5.4% + 2.3%).

Until now we have proposed the order of integration for the first six features in the current model of *Simple Queries*, and there is only one iteration left to finish the execution of the heuristic method defined. The feature that is missing is *Append literals*, and after analyse Figure 4.9, it is possible to see that this last feature only brings gains of less then 1%.

As Figure 4.10 depicts, it is not only now that the coverage gains offered by *Append literals* feature are less than 1%, which means that its 7% of presence that we refer in Section 4.1.6 are not only depending on the features that we proposed but also on other

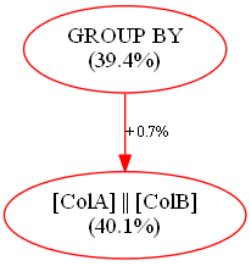


Figure 4.9: Heuristic - extending *Simple Queries* with the seventh new feature

features that still not being supported in the current model proposed. Thus, this means that this feature will not be proposed since its gains are meaningless.

After identifying the list of key features that should support the new model for a visual query builder, as well as identifying the respective order of implementation, we see that these seven features together increase the expressiveness of the current *Simple Query* model, since about 40% of the *Advanced Queries* can now be specified as *Simple Queries*. Although these 40% might seem a small percentage, it presents a really good relationship cost/benefit. At this moment, increasing this value with additionally 10% (reaching the overall percentage of 50%) would have the same cost as the current 40%. This means there is a large difference between the cost of implementing features that cover 40%, and features that cover 50%, since it costs twice as much.

In the next section we present our model proposal integrating each one of the identified key features in the context of visual integration with the product *Agile Platform*$^{TM}$, more specifically in its component *Service Studio*, as well as how works the conversion from a query in the visual query builder to *SQL* code. The proposal follows the order defined above.

## 4.3 Extending the Model

The new target model will be composed by all the functionalities available on *Simple Queries* plus the pointed key features. Between these two models we will have the intermediate models.

### 4.3.1 Selection of Columns

The *Selection of output* is one of the most important features since it gives a huge increase in terms of number of queries that can be done using the visual query builder from *Simple Queries*.

Regarding our previous experience with *Agile Platform*$^{TM}$, the first idea emerging to integrate the *Selection of Columns* to be the output of a query is through adding a new folder to the query tree from *Simple Query* (Figure 4.11). Thus, with the visual analysis of the query tree is clear and easy to understand how the mapping from *Simple Query* to SQL generated is done:

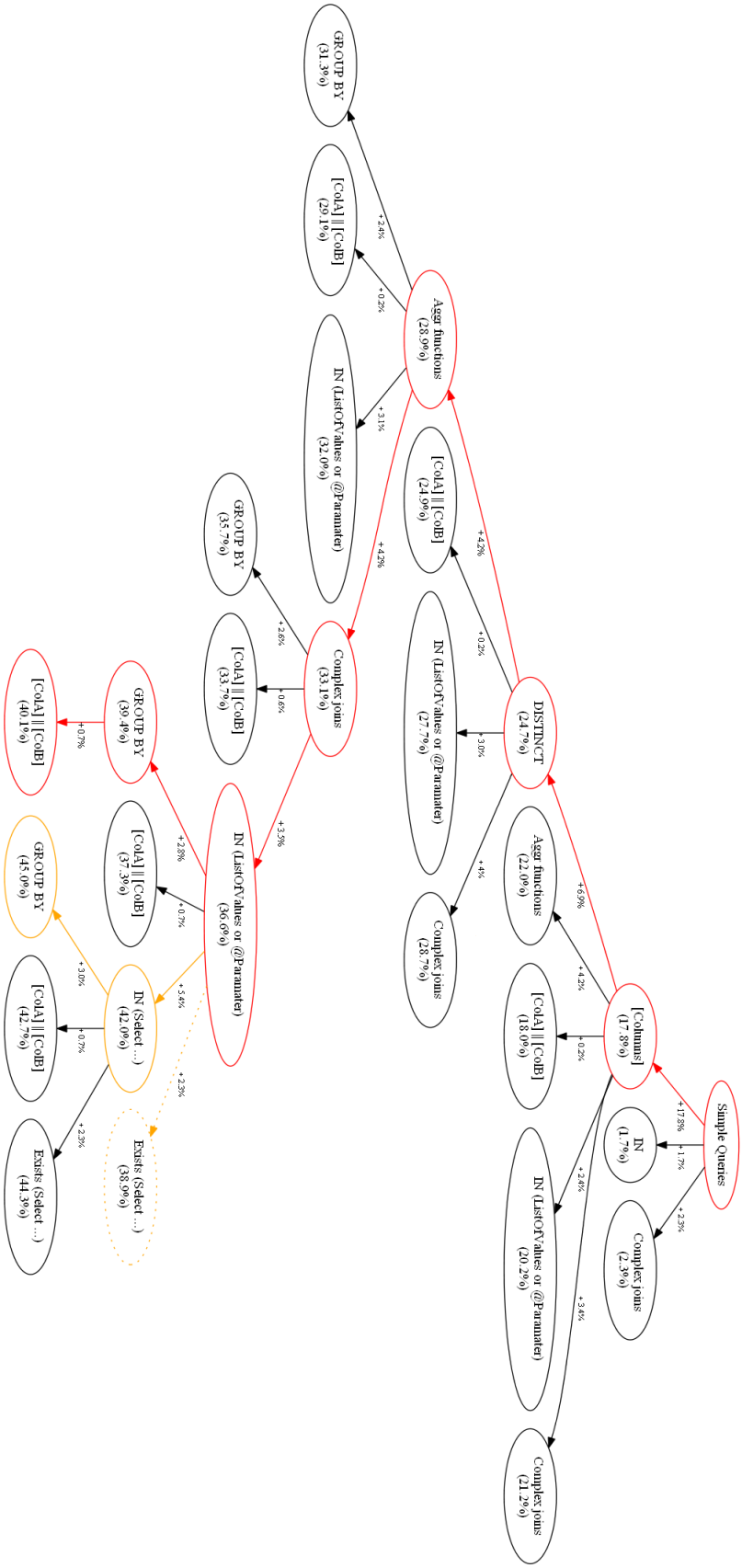


Figure 4.10: Heuristic - tree defining the complete path to follow during the implementation

- current folder "Parameters" remains there
- new folder called "Output Entities / Structures" allows to manipulate what it is intended to select
- current folder "Entities / Structures" remains there, possibly renamed to "Source Entities / Structures"
- current folder Conditions remains there with the same purpose, manage the conditions whether being used as join conditions or as filters

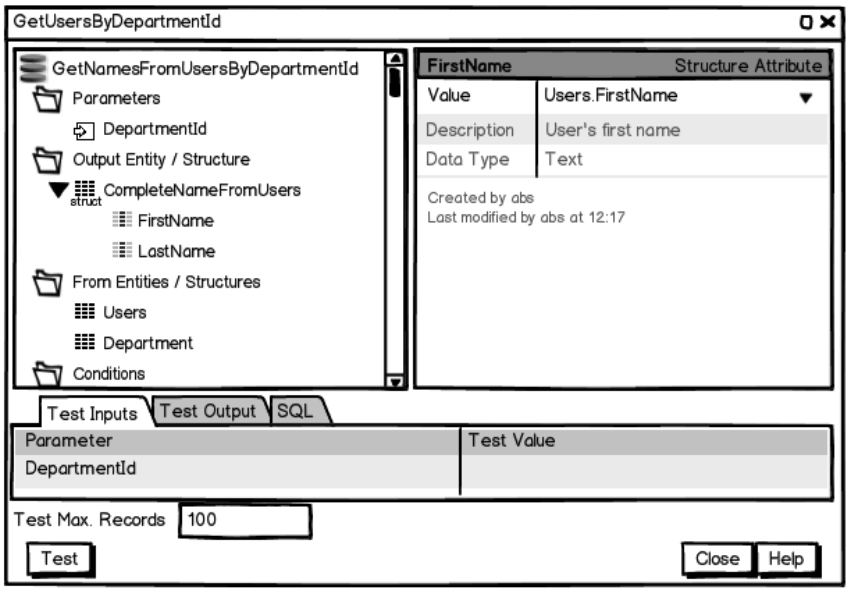


Figure 4.11: Mockup - extending *Simple Queries* with the first new feature (first approach)

Nevertheless, there are some *cons* on this approach, the first is related with the fact that entities inside the folder "Output Entities / Structures" are intrinsically replicated since, by design, all entities from folder "Entities / Structures" are selected in the query, i.e. in the new *Simple Query* context, they inherently belong to the folder "Output Entities / Structures".

Thus, it is necessary to discuss and try to find a better approach, not only to address this gap presented by the previous approach but also to try something closer to the current query builder. So we come to the conclusion that, since the output folder might bring duplicated objects to the tree, it might be possible to propose something without it.

Then, this new approach that we propose keeps the current structure of the query tree, with folders "Parameter", "Entity / Structure", "Conditions" and "Order By". However, there is a small change in a specific type of elements from the query, the elements that we are talking about are *Structures* from "Entity / Structures" folder. Until this moment, it is possible to add *Structures* to a *Simple Query*, although such *Structures* are not able to manage in terms of the values assigned to each one of its attributes (the assigned value

is the default value for the specific type of each attribute). What we propose is the possibility to define a value to each one of the attributes, referring to them in the context of *Simple Queries* as *Output Attributes*. The valid syntax for the value assigned to an *Output Attribute* is given in Figure 4.12.

$$
\begin{array}{lcll}
\textit{Output Attribute} & ::= & \textbf{null} & \\
 & | & \textbf{int} & \text{(Integer literal)} \\
 & | & \textbf{real} & \text{(Real literal)} \\
 & | & \textbf{string} & \text{(String literal)} \\
 & | & \textit{Table.Column} &
\end{array}
$$

Figure 4.12: *Output Attribute* Syntax

Note that when *Table.Column* appears in the syntax, it refers to an identifier from an entity of the database (*Table*) and other identifier from an attribute of this entity (*Column*).

Regardless the chosen approach, both allow the user of *Agile Platform*™ to expand a structure belonging to the query tree of a *Simple Query* and see the *Output Attributes* composing that structure. Thereafter, each one of these attributes is clickable and its properties are shown in the right panel of the *Simple Queries*. The only property that can be changed is the *Value* to assign to that attribute.

After defining all the specifications for the second approach, we are able to show in Figure 4.13 how this propose can fit in the *Agile Platform*™. In the figure we can see that the developer has defined that the value for the attribute *CompleNameFromUser.FirstName* is obtained from the *Users* entity using the expression "Users.FirstName", something similar is done to define the value for the attribute *CompleteNameFromUser.LastName* with the expression "Users.LastName".

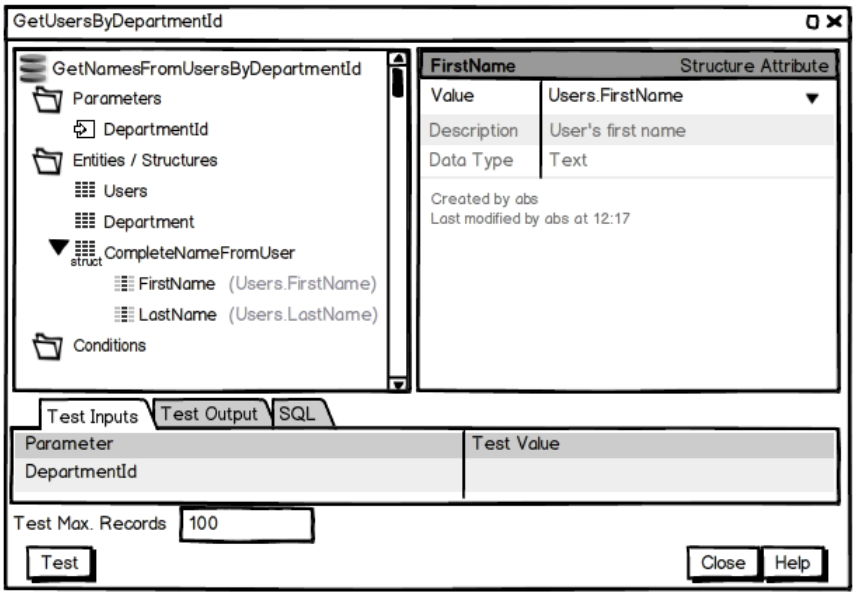


Figure 4.13: Mockup - extending *Simple Queries* with the first new feature (final approach)

Note that the value assigned to each one of the *Output Attributes* appears after their names, and it is simple to justify why this decision was taken, since the query tree should auto-express its behaviour. If the value of some attribute is not yet defined, it is only shown the name of the attribute in the tree.

#### Cast from List of Several Records to List of Records

Moreover, we can not forget the original purpose of this feature, it usually is used to select data from the database and send it to be consumed by a *Web Service*. Due to this fact, it is need to apply some changes on the behaviour of *Agile Platform*™ and then, from now on if a list is composed by a tuple of records, e.g. (A, B, C) but the *Web Service* just need to consume a record list of (B) such cast should be done intrinsically and this operation should be error free. Currently, if we do this action from the example we get an Invalid Data Type Error on TrueChange™ with the message "*The same 'Record' data type required. Expected 'B' instead of 'A,B,C'*".

### 4.3.2 *Distinct* Values

The support to *Distinct* values in *Simple Queries* can lead users of *Agile Platform*™ to avoid use *Advanced Queries* in specific cases. This particular feature can easily be seen as a query property, since we want a query to return a list of distinct values, which means that this *Distinct values* property can be coupled together with the other query properties.

Currently, there is a specific panel to show and manage all the properties of a selected object. What we propose is when a query is selected, the properties of it are shown as usual, however there is a new property in the list of properties from the panel. As we previously referred, this new property is called *Distinct values*, and can take the boolean values Yes or No. By default, it is reasonable to say that the property is No since all the entities have always an *Id* which means that, if all its fields are selected by the query, it is enough to guaranty the distinct values on the results.

Besides the addition of the property mentioned above, we need to allow the users to define if an *Entity* involved on the query, i.e. inside folder "Entities / Structure", will or will not belong to its Output (will be selected or not). Once, as already referred, the entities have always an *Id* and if we force them to belong to the Output then it is impossible to affect the results even after changing the *Distinct* property since, whatever its value, the results will be always distinct.

It should also exist different icons to distinguish when an *Entity* is belonging to the *Output* and when it is not belonging. "*Why?*" Because it should be possible and easy look to the query tree and understand what will be the behaviour of the query. After present the characteristics of the second feature to extend the model, we show in Figure 4.14 how would be its integration.

At a certain moment, we have to make some decisions once we are facing a standoff. This standoff is related with the fact that if the *Distinct values* property should or should

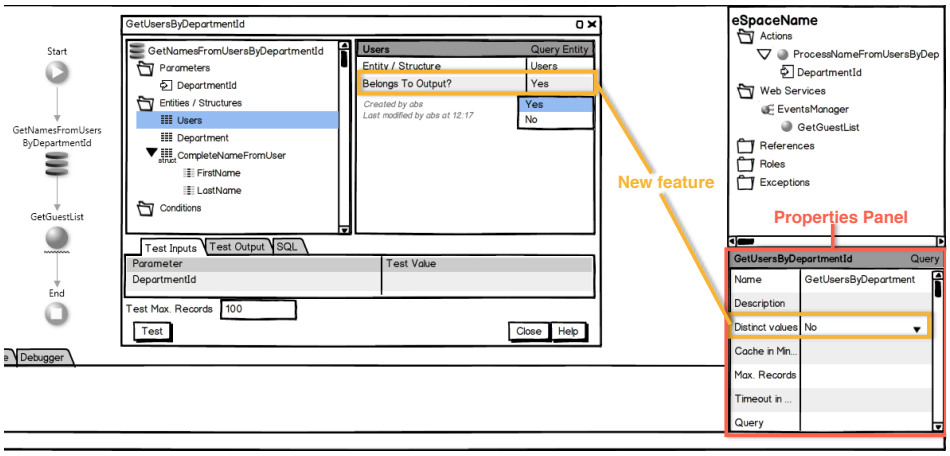


Figure 4.14: Mockup - extending *Simple Queries* with the second new feature

not be always enable to change its value from No to Yes, and vice-versa. The first propose is that it should be disabled until a *Structure* be added to the query and all the *Entities* from the query be defined as not belonging to the Output. However, this can create some problems, since the user can miss the property transition from disable to enable, which can be considered as an interface problem. That is the reason why we decide to let that query property always enable and inject a warning/info using *TrueChange*$^{TM}$ of *Service Studio* in the case of that property be set to Yes and there is at least one *Entity* selected as belonging to the Output.

Hereupon, in the next section it is defined the proposal that supports *Aggregate functions* since it is the third new feature to implement.

### 4.3.3 Aggregate Functions

*Aggregate functions* can give precious help when a developer is facing the creation of reports, since it is a common scenario in the current market and businesses. Such reports can allow to understand how were the current movements and actions of a company in a specific range of time. Sometimes they can be weekly reports, other times monthly, quarterly or even annually. Regardless the subject, aggregate functions can support these reports whether for show the total revenue achieved by a specific employee or a particular branch office in some month, or whether for show the annual expenses in terms of office material for distinct offices.

To start supporting this feature, the syntax for the value assigned to an *Output Attribute* needs to be reviewed. The new syntax is given in Figure 4.15.

After specifying the syntax that validates an expression of an *Output Attribute*, it is easy to idealize that there is no big changes in terms of interface, since what really needs change is the syntax that is supported by the expression of an *Output Attribute*. Thus, in Figure 4.16 is presented a mockup of a query tree using an aggregate function.

| | | | |
|---|---|---|---|
| *Output Attribute* | ::= | **null** | |
| | \| | **int** | (Integer value) |
| | \| | **real** | (Real value) |
| | \| | **string** | (String literal) |
| | \| | **Count**$(*)$ | (Count function) |
| | \| | **Count**(**int**) | |
| | \| | **Count**(**string**) | |
| | \| | **Count**([**DISTINCT**] $Table.Column$) | |
| | \| | **Sum** ([**DISTINCT**] $Table.Column$) | (Sum function) |
| | \| | **Avg** ([**DISTINCT**] $Table.Column$) | (Average function) |
| | \| | **Max** ($Table.Column$) | (Max function) |
| | \| | **Min** ($Table.Column$) | (Min function) |
| | \| | $Table.Column$ | |

Figure 4.15: *Output Attribute* Syntax - with Aggregate functions

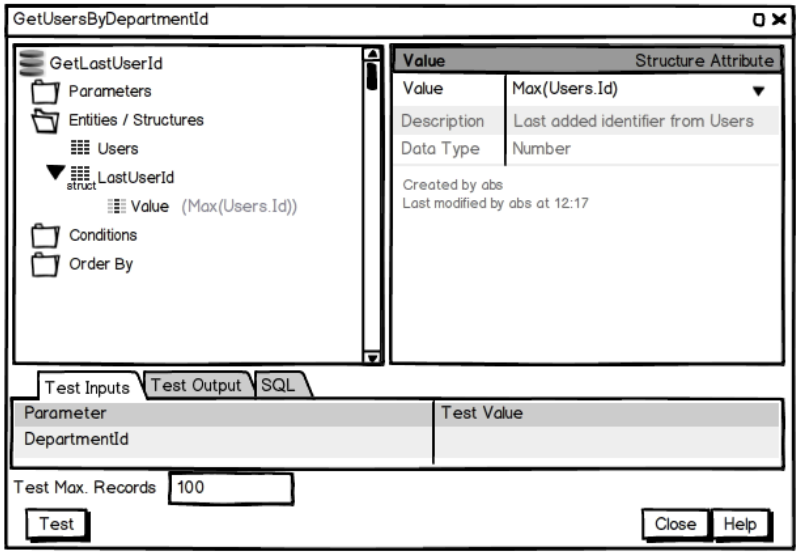


Figure 4.16: Mockup - extending *Simple Queries* with the third new feature

The only interface change that can be proposed in this stage is related with the expression editor of a Value from an *Output Attribute*. With this, we implicitly aware the developer to this new feature and it takes him to understand what are the boundaries of an *Output Attribute*. Figure 4.17 presents our proposal for the expression editor of an *Output Attribute*. It is possible to see the text area in the top where should be written the expression, the scope tree in the left bottom panel, and the right bottom panel is related with local properties.

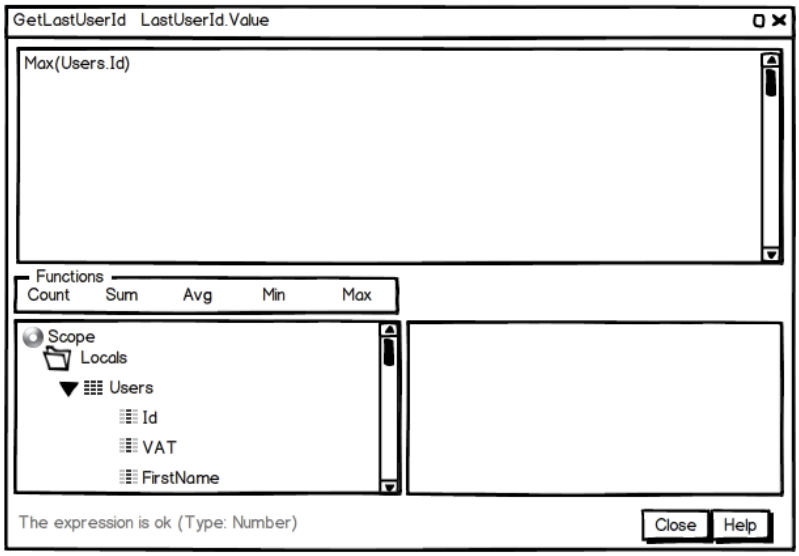


Figure 4.17: Mockup - extending *Simple Queries* with the third new feature (expression editor)

### 4.3.4 Complex Joins

*Complex joins* were one of ours case studies (see Section 3.5.2), regarding the fact that joins from *Simple Queries* need to follow a syntax $Table.Column = Table.Column$ as shown in Figure 4.18. We come into the conclusion that complex joins allow us to increase the coverage in terms of queries that can be done as *Simple Queries*.

$$
\begin{aligned}
Complex\ Join &::= \textbf{ON}\ Expression \\
Expression &::= Table.Column = Table.Column \\
&\quad\ \mid (Expression)
\end{aligned}
$$

Figure 4.18: *Join* Condition Syntax - current model

However, starting to support a new kind of joins can be challenging, once we have the duty of keep the usability of *Simple Queries*, as well as its simplicity. To do that, we should avoid putting all the responsibility to the developers' side and try to help them during the use of *Agile Platform*$^{TM}$. What we mean by this is that we should continue helping

developers over a detection and auto-conversion from a condition to a join condition, nevertheless we need to define the rules that would detect a *Join* condition.

As we previously mentioned, currently the rule that should be verified in order to consider a condition as a Join condition is an $Table.Column = Table.Column$ equality between two different entities, however we can try to explore this rule in order to deal with *Complex Joins* by relaxing the new rule and saying that if a condition contains at least a $Table.Column = Table.Column$ condition it is auto-classified as a Join condition. It is also important to notice that the columns from the left and right side of the condition should belong to different Entities, since these two Entities are important to define the Join Type property of a condition.

Join Type of a condition is the property that allows to distinguish if it is an *Inner Join*, a *Left Outer Join*,a *Right Outer Join* or a *Full Outer Join*. Figure 4.19 shows an example of a query where the join condition is more than a simple condition, since it is composed by a $Table.Column = Table.Column\ AND\ Table.Column = ....$

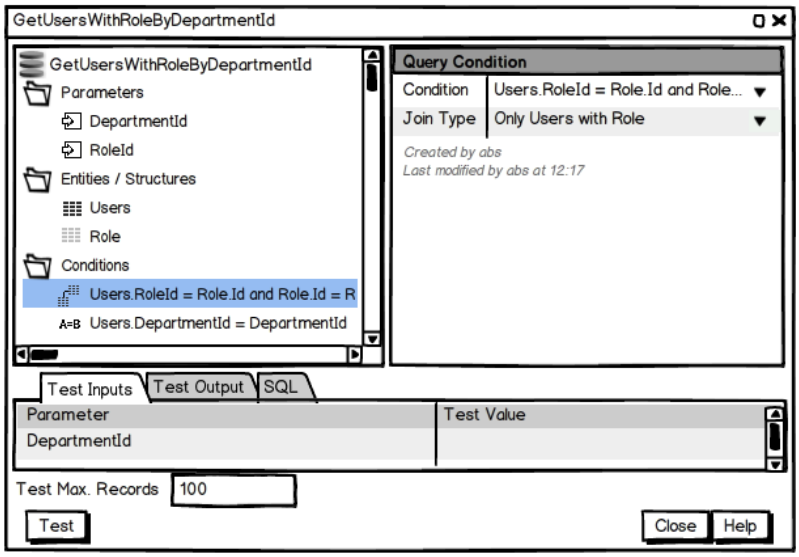


Figure 4.19: Mockup - extending *Simple Queries* with the fourth new feature (expression editor)

What we propose with the new specifications for *Join* conditions lead us to a new syntax shown in Figure 4.20. Furthermore, it is need to do some pos-verification in the condition to understand if there is at least one equality $Table.Column = Table.Column$ between two different tables in order to auto-classify it as a Join condition.

The specifications that we have defined above were based on the analysis regarding the *Join* condition patterns that we have made over the 4.2% of the new queries from the dataset that would start to be covered with the addition of *Complex Joins*, relaxing the *Join* condition, if they just need to follow the syntax from Figure 4.20. However we have the pos-verification to guaranty that they have at least a $Table.Column = Table.Column$ condition, thus we also check it in the different patterns from these 4.2%, and we come to

$$
\begin{array}{lll}
\textit{Complex Join} & ::= & \textbf{ON}\ \textit{Expression} \\
\\
\textit{Expression} & ::= & \textit{AndExpression}\ \textbf{OR}\ \textit{Expression} \\
 & | & \textit{AndExpression} \\
\\
\textit{AndExpression} & ::= & \textit{PredExpression}\ \textbf{AND}\ \textit{AndExpression} \\
 & | & \textit{PredExpression} \\
\\
\textit{PredExpression} & ::= & \textit{Value}\ \textbf{IS}\ [\textbf{NOT}]\ \textbf{NULL} \\
 & | & \textit{AritmeticExpression Operator AritmeticExpression} \\
 & | & \textit{AritmeticExpression} \\
\\
\textit{Operator} & ::= & = \mid <> \mid > \mid >= \mid < \mid <= \mid [\textbf{NOT}]\textbf{LIKE} \\
\\
\textit{AritmeticExpression} & ::= & \textit{AddExpression} + \textit{MultExpression} \\
 & | & \textit{AddExpression} - \textit{MultExpression} \\
 & | & \textit{MultExpression} \\
\\
\textit{MultExpression} & ::= & \textit{MultExpression} * \textit{SignalExpression} \\
 & | & \textit{MultExpression} / \textit{SignalExpression} \\
 & | & \textit{SignalExpression} \\
\\
\textit{SignalExpression} & ::= & -\textit{Value} \\
 & | & \textit{Value} \\
\\
\textit{Value} & ::= & \textit{Table.Column} \\
 & | & \textbf{null} \\
 & | & \textbf{int} \\
 & | & \textbf{real} \\
 & | & \textbf{string} \\
 & | & \textit{InputParameter} \\
 & | & \textit{Function}(\textit{Table.Column}) \\
 & | & (\textit{Expression})
\end{array}
$$

Figure 4.20: *Join* Condition Syntax - new model

the conclusion that 96% from this 4.2% refer to queries where the *Complex Join* conditions are always composed by, at least, a $Table.Column = Table.Column$ condition.

Moreover, 86% of the this 4.2% with complex joins refer to queries with *Join* condition involving just 2 entities, which help us to decide what will be the different Join Types. Then, for the new model we propose that if there are more than 2 entities in a complex join condition, the entities that are considered for the different Join Types are the first two entities of the condition $Table.Column = Table.Column$, e.g. if we have a *Join* condition such as $Table.Column = Table2.Column\ AND\ Table3.Column\ NOT\ LIKE''$, we can choose one of the four Join Types between $Table$ and $Table2$.

Out of curiosity, regarding the 4.2% of queries that follow the syntax shown in Figure 4.20, we also notice that:

- more than **47%** contains a condition with the syntax $Table.Column = Table.Column$ $AND\ Table.Column = @Parameter$
- almost **15%** contains a condition with the syntax $Table.Column = Table.Column$ $AND\ Table.Column = Table.Column$
- more than **12%** contains a condition with the syntax $Table.Column = Table.Column$ $AND\ Table.Column = 1$

**Review** Until this stage, we proposed how should be carried the extension of four new features and we also presented the reasons to do so. We have referred to *Selection of Columns*, *Distinct values*, *Aggregate Functions*, and *Complex Joins*, although regarding the list of most frequent identified patterns there stills missing three of them to propose, *IN operator*, *Group By clause* and *Append literals*. Adding all the features presented until now on the current model of *Simple Queries* allows us to specify 33.1% of the *Advanced Queries* from the dataset as *Simple Queries*.

### 4.3.5 *IN* (List of Values or @Parameter)

Since *In operator* was detected as a frequent pattern during the analysis of our dataset, it is included in the list of most frequent identified patterns. This operator can be split into two parts, primarily the *In* operator using a List of Values or a @Parameter, secondly the *In* operator using a sub-query. We decide that the second part is dependent on the first part as shown in Figure 4.1, mainly due to the costs related with features involving sub-queries, as we presented in Section 4.2.1.

The first thing to do is add a new operator *In* inside the group of other operators such as *like* in the query condition editor and, as an expression, this boolean operator has a left and a right part. The left part is related with the column that has a value, the right part should contain the list of values that will be checked against the value that comes from the left part, in order to understand if the left part is contained in the right part.

The question that emerges here is *"how should be this list of values presented?"*. At a first sight, it can be easy to have an idea, however it should be well argued. We could allow the developer to set the list in full way, like "('1','2','3',...)", however this approach could deviate from the current paradigm of lists in *Agile Platform*$^{TM}$. In *Agile Platform*$^{TM}$ a list is named *Record List* and has an associated *Record Definition* that can be an Entity or a Structure from the data model. What we propose is start supporting *Record Lists* also for *Paramaters*, just on the queries, and the *Record List* needs to be validated in order to check if it is a Structure composed by one, and only one, attribute of a basic type such as:

- Text;
- Number, Integer, Decimal;
- Identifier;
- Date, Date Time, Time;
- Phone Number, Email.

If this kind of parameter is supported in *Simple Queries*, we can create query conditions with *In operator*, where the left part is a column of an entity involved on the query and the right part is the specified parameter from the query.

The definition of the parameter that is used on the query condition is displayed on Figure 4.21. The scenario shown contains a query with a *ListOfValues* parameter. This parameter is defined as a *Record List* of *ActivityTypeIdentifiers* (structure composed by one attribute called Value of the type ActivityType Identifier).

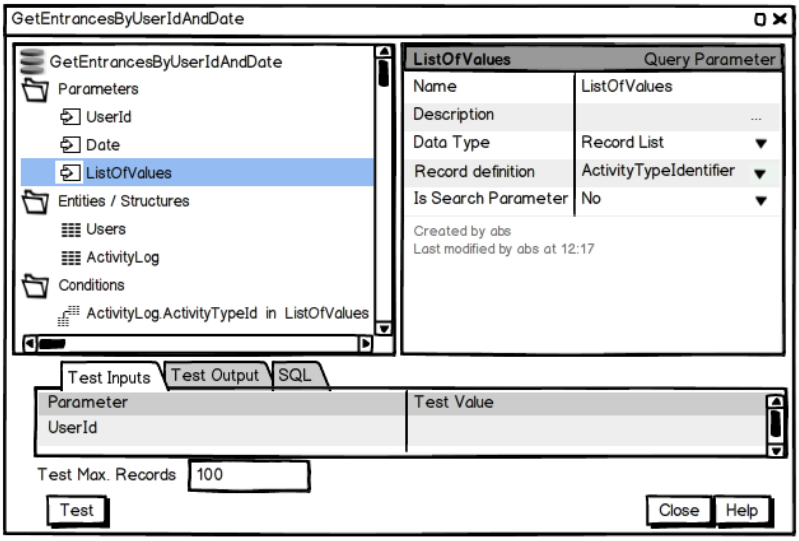


Figure 4.21: Mockup - extending *Simple Queries* with the fifth new feature (parameter definition)

This specification of allow only query *Parameters* as *Record List* of *Structures* with one, and only one, attribute of a basic type is a constraint that needs to be verified always on the fly.

Once the parameter is defined, it starts to be visible in the scope of query conditions and it is possible to build a query condition based on it using an *In operator* as shown in Figure 4.22.

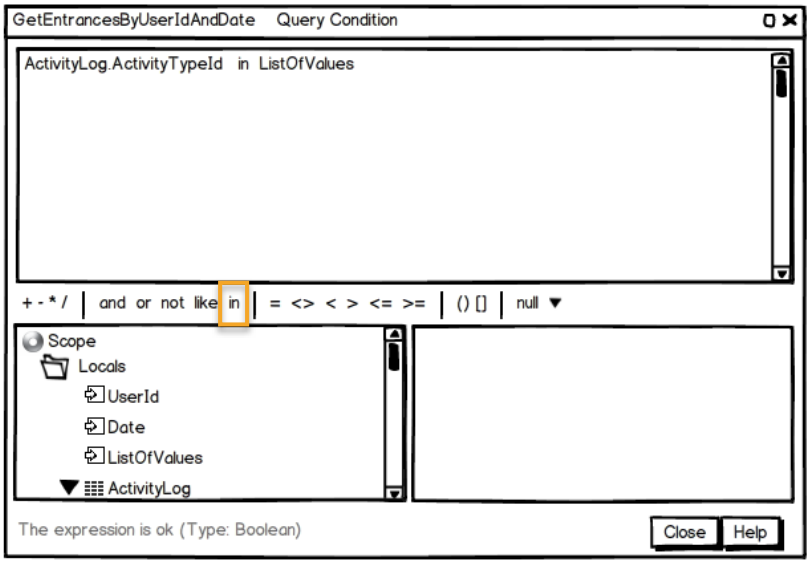


Figure 4.22: Mockup - extending *Simple Queries* with the fifth new feature (query condition editor)

Afterwards, at compilation time, the list of values needs to be converted in order to have the correct syntax to generate valid SQL code.

In summary, to extend the model with this fifth feature, we need to add the *In operator* inside the query condition editor, add *Record List* as a possible data type for query parameters, and last, but not least, restrict the *Record definition* of these parameters of *Record List* type, allowing only the *Structures* composed by one, and only one, attributes of a basic type as referred above.

At the moment, the intermediate model is composed by all the features supported by *Simple Queries*, as well as *Selection of columns*, *Distinct values*, *Aggregate functions*, *Complex joins* and *In operator*. There are only two features remaining to have the final model complete and these features are *Group By clause* and *Append Literals*.

### 4.3.6 *Group By* Clause

A plausible approach is adding a new folder called "Group By" to the current intermediate model obtaining a query tree as shown in Figure 4.23. The query tree is now composed by folders "Parameters", "Entities / Structures", "Conditions", "Group By" and "Order By".

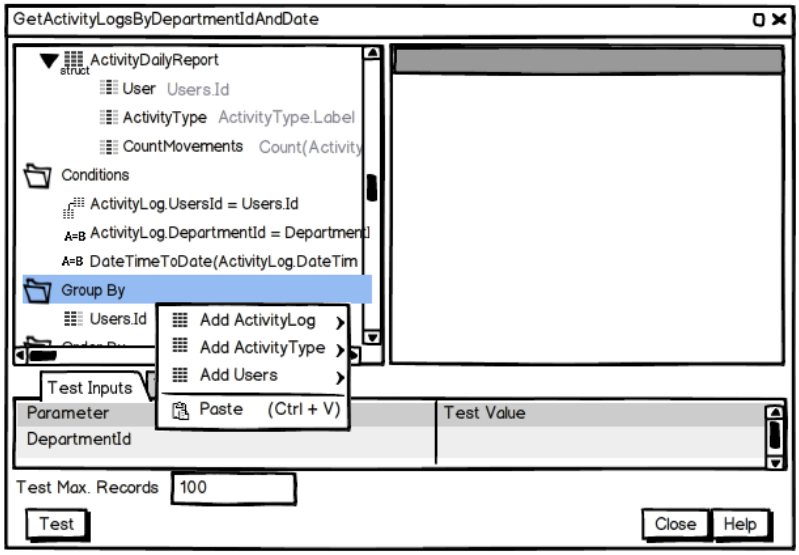


Figure 4.23: Mockup - extending *Simple Queries* with the sixth new feature

Figure 4.23 depicts the action of adding a new element to the *Group By*. To do that the developer just needs to do a right-click over *Group By* folder, after that choose one *Entity* between the entities available and finally choose the respective column that defines the *Group By* (Figure 4.24).

The behaviour of this feature is similar to *Order By*, since it is possible to choose any column from the entities involved in the query. Depending on the columns that are in the *Group By* folder, the property of an *Entity* that specifies if it belongs to the Output can change from *Yes* to *No* and vice-versa. If there is any *Group By* column defined, in order to have an *Entity* belonging to the Output, all the columns of that *Entity* should be added in the *Group By* folder.

Another characteristic of this feature is related with the columns that are used in the *Output Attributes*. All the columns that are not used with aggregate functions need to be on the *Group By* folder. Furthermore, all the *Output Attributes* that are using aggregate functions must be disabled on *Group By*, otherwise the aggregate function does not do any effect (will be always applied to one row).

The scenario presented in Figure 4.24 shows that *ActivityLog*, *ActivityType*, and *Users* entities are involved in the query, which is the reason why they are available to *Group By*. The goal of the query is to retrieve the number of movements grouped by Users.

Notice that we also give a suggestion to the developer about what column he should add next, putting the name of the column as bold text.

## 4.4 Suggestion Mechanism

In *Service Studio*, the validation engine TrueChange™ can be a great help afterwards in terms of validation of queries and suggestions, since it allows us to aware developers

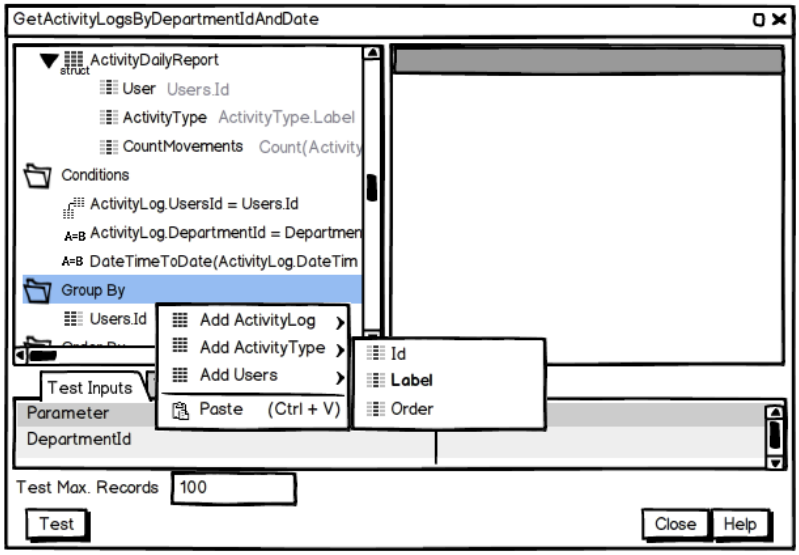


Figure 4.24: Mockup - extending *Simple Queries* with the sixth new feature (*Group By* column)

about some possible mistakes, or even aware them when an *Advanced Query* can be done as *Simple Query*.

The last point mentioned is relevant in order to aware existing users from *Agile Platform*™. Would be also interesting to provide a new command on Right-click over the query to automatically convert an *Advanced Query* into a *Simple Query*. Furthermore, it would be as well interesting to do the reverse, convert a *Simple Query* into an *Advanced Query* since it allows the developer to still writing the complex query that usually is based on a *Simple Query* that evolves over time.

## 4.5 Discussion

In this Chapter we presented the list of most frequent identified patterns that guided our model proposal along with the answers to some questions such as *"What characterizes each feature?"*, *"When does it occur?"* and *"Why should the model be extended with it?"*.

Afterwards, since the model proposal followed an incremental process, we presented an heuristic that allowed us to define a plausible order of implementation for the features, bearing in mind all the dependencies identified between them.

Furthermore, we defined our proposal to extend the model with the identified features, trying to validate all the decisions adopted with the R & D team. In the end, we also fulfilled the model proposal with suggestions taking advantage of the benefits offered by *Agile Platform*™ validation engine, TrueChange™.

The next Chapter presents the prototype implemented and it is split in two parts. The first focuses on what is extended in the prototype regarding the current *Simple Query* model, and once we need to submit the prototype to usability tests in order to validate

our proposal, the second part presents some important components of usability tests such as the scenario, the script, and the feedback from the testers.

# 5

# Prototype

After identifying the key features of the new model, we decide to implement a high-fidelity prototype. But, since we have some time constraints, we start to implement the first feature proposed, feature that brings more gains in terms of coverage of *Advanced Queries* that can start to be built as *Simple Queries*.

The implemented feature is the *Selection of columns*, however during its implementation we discover that it is easier to define the *Output Attribute* syntax as shown in Figure 5.1, opposing the initial proposal since in the prototype the domain of *Output Attributes* is now much more comprehensive. The Built-in Function on the syntax refers to functions available on *Service Studio*, that allow the developer to manage Date, Time, Numbers, Text, among others. Regarding the new domain of *Output Attributes* it is possible to say that the coverage of *Advanced Queries* that now can be done as *Simple Queries* will be greater than 17.8%, the percentage pointed to the feature of *Selection of columns* since the prototype implicitly and simultaneously implements other features such as *Append literals*, use of built-in functions on *Output Attributes*, and others.

| | | | |
|---|---|---|---|
| $Output\ Attribute$ | ::= | **null** | |
| | \| | **int** | (Integer literal) |
| | \| | **real** | (Real literal) |
| | \| | **string** | (String literal) |
| | \| | $Table.Column$ | |
| | \| | $BuiltInFunction(Output\ Attribute)$ | |
| | \| | $Table.Column\ +\ Output\ Attribute$ | |

Figure 5.1: *Output Attribute* Syntax in Prototype

In Figure 5.2 is shown a screenshot of the prototype where is possible to see a structure expanded as well as the *Output Attributes* from that structure, already with values assigned as may be seen after each *Output Attribute* name, surrounded by parenthesis.

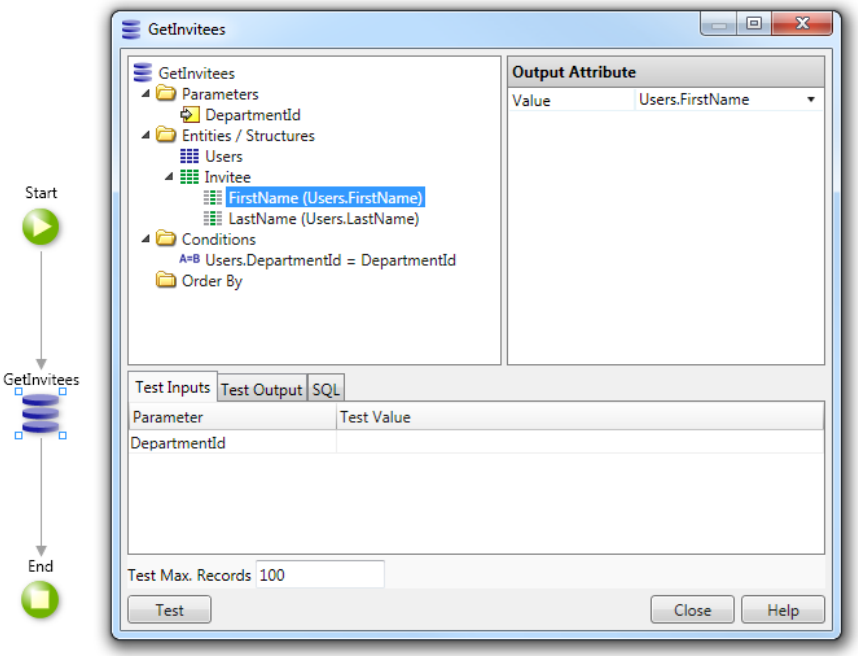


Figure 5.2: Prototype - *Output Attributes* from *Simple Query*

The prototype done will be submitted to usability tests that are the topic of the next section, such tests will allow us to validate the model.

## 5.1 Usability tests

> "*Why didn't we do this sooner?* — what everyone says at some point during the first usability test of their web site." [Kru06]

Although Krug mentions a web site, this situation can occur in other contexts, namely during the first usability test of a software.

To submit our prototype to usability tests we need a scenario, a script and tasks [Ros12, Fal12, oHS06]. In this particular case it is a single task since we are testing a unique feature.

### 5.1.1 Scenario

We are representing the campus IT team, and we know that will occur an event in a specified department in a few weeks.

A company of events management was engaged to set up everything:

- Space decoration,

- Snacks & drinks,
- Entrances,
- and so on...

Now, in order to manage the entrances, they are asking us the list of names of people that will be allowed on the event. Basically, we are talking about users belonging to that specific Department.

To this end, they provided us a *Web Reference* with an action *SendInviteesList* accepting a list of names of people.

### 5.1.2 Script

**If the tester uses *Advanced query*** Let him finish the action. After that, ask him to try again but, this time, using *Simple Query*.

**If the tester uses *Simple Query*** Let him finish the action, and

**If the tester uses a foreach** to iterate the list of users from the query and create a new list with the loop, then let him finish the action. After that, ask him why he did not use a structure. Then, ask him to try it using the structure.

**Just on the limit, advise him** to check "Add Another Entity / Structure..." with right-click. Maybe he will see the *Web Service* on the *Structures* folder from the *Picker*.

**In the end of the test** ask for suggestion of improvements that can allow to make the feature more usable and discoverable.

### 5.1.3 Feedback

In this section we present the feedback collected during the usability tests from the several iterations that we have done.

#### 5.1.3.1 First iteration

In this iteration we intend to collect feedback about several aspects such as scenario, proposed interface, new incoming ideas, among others. After that, we analyse all the received feedback and then, if necessary, we apply some changes according it.

**Tester 1**

- He followed the *Simple Query* path *a priori* (maybe due to having some knowledge about the project of add new functionalities on *Simple Queries*)

- He did not understand what was a list of details referred on the scenario (list to be consumed by the *Web Service*) -> Solution is to change some sentences on the scenario, and now we have "... an action *SendInviteesList* accepting a list of names of people" instead of "... an action *SendInviteesList* accepting a list of details".

- He did not find the output -> Possible solution is to change the Label "Entity / Structure" on the query tree or even change the label "(Add Another Entity / Structure...)" from the right-click command to add new entities on the query to "(Add Another Entity / Output Structure...)"

#### 5.1.3.2 Second iteration

Basically, in the previous iteration we needed to adjust the scenario description making it more understandable. Then, we are able to start a new iteration with new testers.

**Tester 2**

- He followed the *Simple Query* path

- Meantime, when he realized that Invitee was a structure, he deleted the *Simple Query* and created an *Advanced Query*

- After concluding the *Advanced Query* and finish the exercise, we asked him if there was another way to solve the scenario presented using a *Simple Query* and we encourage him to try

- Then, he created a query selecting the Users, however after that he was looking for a way to add a structure to the query

- After that he added the structure Invitee to the query, he expanded it and assigned the values for each one of the *Output Attributes*

- In the end, he exposed an idea, if a structure contains attributes with the same name and type of an attribute from any involved entity on the query, then such entity attributes should be suggested when the developer decide to click on the dropdown from the *Output Attribute* value expression.

**Tester 3**

- He followed the *Simple Query* path obtaining a list of Users

- He created a local variable that was a *Record List* of Invitees

- He created another local variable, this time was from the type Invitee, to assign the values from the Users list on each iteration

- Then, this last local variable Invitee was appended to the *Record List* of Invitees, and when all the Users list was covered he sent it to be consumed by the *Web Service*

- In the end, we asked him to try it without iterate the Users list, peeking the right-click option "(Add Another Entity / Structure...)"

- When he finished the action, he said that was expecting something closer to *Advanced Queries* that have a dedicated folder to Output (or the User Action that have also the concept of Output [parameter])

- Furthermore, when he looks for a *Simple Query* tree he links the "Entities / Structures" folder to the From clause of a query, the "Conditions" folder to the Where clause, and so on.

**Tester 4**

- He performed the scenario with a *Simple Query* getting a list of Users (maybe due to the fact that he has some previous knowledge about some new functionalities on *Simple Query*)

- However, he did not figure out how to manipulate the structure on the *Simple Query*, and decided to done it using an *Advanced Query* giving up on the *Simple Query*

- When he successfully finished the scenario using the *Advanced Query*, we asked him to try again guiding him to use *Simple Query* instead.

- He block and we said to take a look to "(Add Another Entity / Structure...)". He added the structure and finished the scenario without problems.

#### 5.1.3.3 Third iteration

We concluded with the previous iteration that all the testers were performing the scenario similarly and they never found the new feature which lead us to a question "*How should we aware developers about this feature?*".

Hereupon, we decide to change few things in the prototype, and if an *Advanced Query* is parseable as a *Simple Query* (new *Simple Query* model with support to *Selection of columns*) the *Platform* launches an *Info* message on TrueChange™ and aware the developer with the message "*Do you know that [queryName] could be built as a Simple Query?*". With this information we are avoiding some interaction with the testers in the end of their execution of the scenario presented. Figure 5.3 depicts the message that appears to the developer using TrueChange™ when a query is done using *Advanced Query* however is simple enough to do as *Simple Query*.

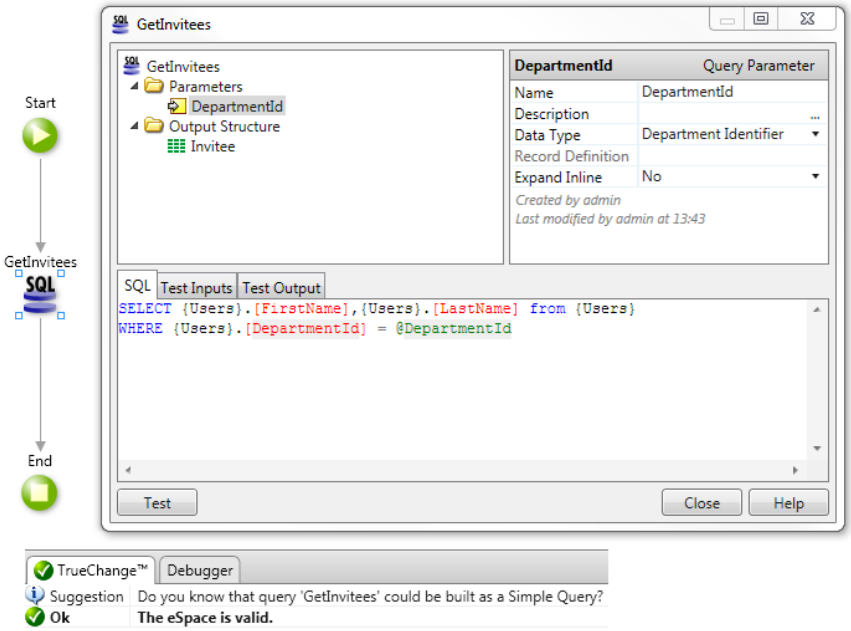


Figure 5.3: Prototype - message to aware developers about new functionalities of *Simple Query*

**Tester 5**

- He executed the scenario using *Simple Query*, then he obtains a list of Users
- After building the query, he checked what will be consumed by the *Web Service* and he discovered that will be a structure
- Created a new local variable that is a List of Records and another local variable that is a Record (more specifically an Invitee Structure)
- He iterates all the list retrieved from the list and on each iteration assign the values that he wants to the local variable Invitee and after that, append it to the local variable *Record List*
- After finishing the scenario following the previous steps, we asked him to try without iterate the list. Then, he added a structure and saw that it was expandable, however he almost ignore that fact but *"Why?"* -> The mandatory property of an output attribute should be confirmed, and if an attribute is mandatory but the value is not filled (even without the default value type) an error should appear on the TrueChange™ and the attribute should be distinguished with a red color maybe. This action would help to identify that there is something new in the *Simple Query* tree.
- Another problem that he detected was related with the fact that the query returns (A,B,C) but the *Web Service* just needs (B) and until now such thing was not possible,

*"Why is it happening now?"* Such thing can be accepted by developers if it is included in the new learning process.

### 5.1.4 Top Issues

Taking into account all the feedback collected during the previous iterations, there are some top issues detected that we list bellow:

- Understandability of the scenario (**SOLVED**)
- Discoverability of the feature (see *Discussion*, Section 5.2)
- Possibility to pass list of records of type (A,B,C) and consume just one of the types, e.g. a list of records of the type (B), is unknown (related with the previous one, see also *Discussion*, Section 5.2)

## 5.2 Discussion

In this Chapter we first referred to the prototype developed with some features according the model proposed in Chapter 4. After the implementation, the prototype was submitted to usability tests in order to validate the model, and we then presented all the feedback collected during the several iterations from these tests as well as some top issues detected.

One of the most important top issues was related with the discoverability of the feature, since its a new thing it is need to define methods to aware developers about it. Although it is not part of the scope of the problem that we are trying to solve, we present below some ideas to tackle it.

**How to aware new developers about this feature** Of course the first thing to do would be update the existing tutorials and videos from online training in order to explain how powerful are *Simple Queries* right now. Basically, we are talking about performing some changes on Academy Courses from *OutSystems Academy* [Aca13]. Moreover, it could also be integrated on the learning process from bootcamps, etc...

**How to aware existing developers about this feature** Maybe with workshops, training, tutorials, or even generating a suggestion through TrueChange™ that would allow to recognize when an *Advanced Query* could be specified as *Simple Query* (already implemented on the prototype). *How could that be done?* Through the grammars that we have built, we can now parse a query and check if it is syntactically correct as *Simple Query* or not. (Going deeper, it would be nice if it was possible to auto-convert with right-click).

Since all the top issues are solved or we have presented a solution for them, we can conclude that after fix some certain aspects on the prototype it can be integrated with the

product in a short-term. The referred aspects are related with the fact that mandatory *Output Attributes* should be filled otherwise the developer should get an error, the type of an *Output Attribute* should be also visible as a read-only property from it, and some additional visual awareness should be provided on the arrow to expand a structure on the query tree.

Hereupon, in the next Chapter we present the final remarks from this Master Thesis.

# 6

# Final Remarks

This thesis is integrated in the Research and Development (R & D) team of the *OutSystems* company and had two different parts.

During the first part of this dissertation we described a preliminary study about different subjects such as the use of SQL on industrial environments, query languages, clustering algorithms, searching for patterns on structured data, and visualization of data. Furthermore, we built our dataset of queries after extract these queries from thousands of applications from *OutSystems* clients and we also started an analysis over the dataset.

This analysis was composed by several steps from the histogram of terms to the searching tool implemented that supported searching for patterns using XPath expressions. Then, we proposed a possible solution for our tool that would allow us to improve its capabilities to discover patterns and to visualize and understand the dataset of queries. Some algorithms and techniques were studied as a basis for the decisions that we later made along the development phase of the improvements on the tool.

The first phase occupied 50% of the time available and was already made in collaboration with the company and with all the support from both sides, University and Company.

In a second part, the development of our tool, the implementation of the clustering algorithm, the new query model proposal and the prototype implementation were addressed on a full time basis at the company, and it was integrated in a team with other interns, each one responsible for a particular project regarding his Master Thesis. In this phase, it followed the *OutSystems Agile Methodology*, a methodology based on SCRUM Agile Methodologies [All01, Met01, SB01], for control and organization of projects.

The development of improvements in our tool and the implementation of the clustering algorithm to group queries according their structure were done in iterative process

as well as the proposal of the new *Simple Query* model, that suffered some changes along the time. For example, in some stages of the model proposal we were dealing with two or more different ways to implement a specific feature. With the arise of new challenges, we were forced to reinforce the proposal and rethink some decisions, reasons and arguments along time. Since we were integrated in a team of specialists, they provided us with valuable comments and interesting discussions. In the end, we were able to decide for the best proposal according the feedback obtained from these comments and discussions.

It was necessary to understand the functionalities and purposes of the main component of the *Agile Platform*$^{TM}$, *Service Studio*. It was also important to (partially) understand the DSL compiler and the *OutSystems* language. The next step was to start implementing the prototype, following the new model proposal.

The prototype implemented with this thesis need a few more days of work in order to be fully functional, nevertheless it can integrate the development branch of *Agile Platform*$^{TM}$ in a short-term. The usability tests done are a great help to the near future since they provided important information to the R & D team.

## 6.1 Conclusions

In the end, we can point out some existing problems that should be addressed in the future. The clustering algorithm could be more automatic explaining what are the patterns contained in each cluster, instead that we need to manually perform a visual analysis and some important facts can be missed due to possible human errors. There are some other limitations in what concerns the pattern search feature that we integrate with the Tool since it is using XPath, which means that it needs previous knowledge from the user.

We referred the issues of our approach. Nevertheless, now we present its benefits. Primarily we can mention the tool built for use with *OutSystems Agile Platform*$^{TM}$ with an extension where should be attached all the grammars that are intended to be used on the parse of a set of queries along with the clustering algorithm. This algorithm was implemented as a C# program and it can easily be adapted to a new set of queries, generating four different elements for analysis *per* cluster. These elements for analysis comprise a set of queries that belongs to a particular cluster, a complete and a detailed colored graphs, and an histogram of terms. Joining all these four elements, it is possible to overcome the limitations mentioned above regarding the possible human errors from the analysis of the clustering algorithm results.

Furthermore, since this tool has attached a search feature using a XPath processor (good approach for users with XPath knowledge), it is also possible to take a look at the parsed dataset and search for specific patterns from the business that would be interesting to understand if they are or are not relevant in a the specific context of that dataset.

The key features that should extend the current *Simple Query* model were identified, which opens a new door and helps the R & D team during the next decisions on their product development path.

Additionally, a prototype was implemented and submitted to usability tests that can be considered by the R & D team during the integration of some key features proposed with the new model.

## 6.2 Future Work

One interesting topic of future work can be to understand deeply why were the queries done as *Advanced Queries* if they could be done as *Simple Query*. This can lead the researchers to a new level of challenges, possible interface problems in the current *Simple Query* model or even the need to update some *OutSystems Academy* courses in order to direct users attention to *Simple Query* as well as making them more attractive.

Furthermore, improving the usability of the tool created can be another topic of study, discovering an easier way for looking for patterns in a big dataset of queries instead of using the XPath. Moreover, adjust the clustering algorithm or even apply a new kind of algorithm or technique that allows to discover patterns in a more automatic way, trying to avoid the visual analysis that can be a source of errors possibly due to human mistakes. If the visual analysis continue on the process of analysis, there are another point that could be reviewed, and it is related with the colored graphs used on this analysis since only the nodes are colored which means that a possible improvement could be done on its edges, coloring them as well according their occurrence frequency.

Regarding the heuristic defined to order the implementation of features on the new model, could be also interesting to have a tree as the one presented with the final path of the implementation, however with all the possible combinations of features. With that, we could understand better what would be the immediate gains obtained by implementing two features at a time (as unexpectedly happened on the prototype), or if there is a need to change the current path of implementation, the next step could be properly decided and justified with the obtained gains in terms of coverage.

## 6.3 Discussion

In this Chapter we have provided a sum up of the work experience that we live, followed by the work that was developed in this thesis, the main contributions and key aspects that could be improved in future work. These improvements include some more technical details of the tool implementation to improve its usability, improve the detection of patterns trying to make it more an automatic process, as well as understand the reason why were some of the queries specified as *Advanced Query* instead of using *Simple Query*.

# References


[Aca13] Academy. Outsystems. https://www.outsystems.com/Academy/, 2013. [Online; accessed 05-March-2013].

[All01] Agile Alliance. The agile manifesto. http://www.agilealliance.org/the-alliance/the-agile-manifesto/, 2001. [Online; accessed 15-Frebruary-2013].

[Bos12] Michael Bostock. D3.js - data driven documents. http://d3js.org/, 2012. [Online; accessed 12-March-2013].

[Cal09] Calitha. Calitha C# Gold Parser Engine. http://www.calitha.com/goldparser.html, 2009. [Online; accessed 14-June-2012].

[CG04] Luca Cardelli and Giorgio Ghelli. TQL: a query language for semistructured data based on the ambient logic. *Mathematical. Structures in Comp. Sci.*, 14(3):285–327, June 2004.

[CGA+02] Giovanni Conforti, Giorgio Ghelli, Antonio Albano, Dario Colazzo, Paolo Manghi, and Carlo Sartiani. The Query Language TQL, April 2002.

[Coo12] Devin Cook. GOLD Parsing System. http://goldparser.org/, 2012. [Online; accessed 14-June-2012].

[Dav12] Jason Davies. Parallel sets. http://www.jasondavies.com/parallel-sets/, 2012. [Online; accessed 12-March-2013].

[Eag12a] Eagereyes. Parallel sets. http://eagereyes.org/parallel-sets, 2012. [Online; accessed 12-March-2013].

[Eag12b] Eagereyes. Parallel sets. https://code.google.com/p/parsets/source/checkout, 2012. [Online; accessed 12-March-2013].

[Fal12] Trine Falbe. An epiphany: Usability testing on high fidelity prototypes. http://www.trinefalbe.com/?p=311, 2012. [Online; accessed 12-March-2013].

[GMT06] Calin Garboni, Florent Masseglia, and Brigitte Trousse. Sequential pattern mining for structure-based xml document classification. In *Proceedings of the 4th international conference on Initiative for the Evaluation of XML Retrieval*, INEX'05, pages 458–468, Berlin, Heidelberg, 2006. Springer-Verlag.

[Goo11] Google. High order bit. http://www.web2summit.com/web2011/public/schedule/detail/20861, October 2011. [Online; accessed 25-June-2012].

[Hel12] Service Studio Help. Outsystems agile platform 7.0. http://www.outsystems.com/help/servicestudio/7.0, 2012. [Online; accessed 02-August-2012].

[HM04] Elliotte Rusty Harold and W. Scott Means. *Xml in a nutshell, 3rd edition*. O'Reilly Media, Inc., 3 edition, 2004. Chapters 1,2,9.

[Kru06] S. Krug. *Don't make me think!: a common sense approach to Web usability*. Voices That Matter Series. New Riders, 2006. Chapter 9.

[LCMY04] Wang Lian, David Wai-lok Cheung, Nikos Mamoulis, and Siu-Ming Yiu. An efficient and scalable algorithm for clustering xml documents by structure. *IEEE Trans. on Knowl. and Data Eng.*, 16(1):82–96, January 2004.

[LCW93] Hongjun Lu, Hock Chuan Chan, and Kwok Kee Wei. A survey on usage of sql. *SIGMOD Rec.*, 22(4):60–65, December 1993.

[Met01] Scrum Methodology. Scrum methodology agile scrum methodologies. http://scrummethodology.com, 2001. [Online; accessed 15-Frebruary-2013].

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

[oHS06] U.S. Dept. of Health and Human Services. *Research-Based Web Design & Usability Guidelines*. U.S. Government Printing Office, 2006.

[Out12] OutSystems. Agile platform™. http://www.outsystems.com/agile-platform/, 2012. [Online; accessed 08-July-2012].

[Phi07] Phineas. Blog about Sankey Diagrams. http://www.sankey-diagrams.com/, 2007. [Online; accessed 25-June-2012].

[Pö95] Richard Pönighaus. 'Favourite' SQL-Statements — An Empirical Analysis of SQL-usage in Commercial Applications. In Subhash Bhalla, editor, *Information Systems and Data Management*, volume 1006 of *Lecture Notes in Computer Science*, pages 75–91. Springer Berlin / Heidelberg, 1995. 10.1007/3-540-60584-3_25.

[Ros12] Jim Ross. Ux matters - tips on prototyping for usability testing. http://www.uxmatters.com/mt/archives/2012/10/tips-on-prototyping-for-usability-testing.php, 2012. [Online; accessed 12-March-2013].

[SB01] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[SCDT00] Jaideep Srivastava, Robert Cooley, Mukund Deshpande, and Pang-Ning Tan. Web usage mining: discovery and applications of usage patterns from web data. *SIGKDD Explor. Newsl.*, 1(2):12–23, January 2000.

[Sim02] John E. Simpson. *XPath and XPointer: Locating Content in XML Documents*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002. Chapters 1-5.

[SKS10] Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., New York, NY, USA, sixth edition, 2010. Chapter 3.

[Tuf01] Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, Conn., 2nd ed. edition, 2001. Pages 40-41, 51.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

[Wal07] Priscilla Walmsley. *XQuery*. O'Reilly Media, Inc., 2007.

[Wik12] Wikipedia. Gold (parser) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/GOLD_(parser), 2012. [Online; accessed 14-June-2012].

# Appendix

## A.1 Glossary

- Domain Specific Language (DSL) - a Domain Specific Language is a programming language or executable specification language designed to express solutions to problems from a particular domain.
- Service Studio - a visual Integration Development Environment (IDE) that allows to edit an *eSpace* as well as publish it to a development environment, to be tested and analysed, or publish it to a production environment.
- eSpace - a web application project of *Service Studio*. It contains all definitions needed for developing and managing web application, from the logic layer to data layer elements.
- OML - stands for *OutSystems Markup Language* and is the format by which the *eSpaces* are saved to file. It is also the extension (.oml) for the *eSpaces*.
- Simple Query - visual query builder available on *OutSystems Agile Platform*™ that allows the developers to query the entities from a specific *eSpace* (recall Chapter 2).
- Advanced Query - element from *Agile Platform*™ that allows the developers to execute more complex query statements or any other SQL statements. This element can be used to manage your entities or to execute any other statement in the database.
- Web Service - it is a software function provided at a network address over the web or the cloud, it is a service that is "always on". In the context of *Service Studio*, it

is possible to add a Web Reference that, as a Web Service, provide the developers with several functions that can be used to consume or retrieve data.

## A.2 Patterns Detected on Clusters Visual Analysis

| Cluster | *Select clause* with | | *From clause* with | | *Where clause* with | | *Order By* | |
|---|---|---|---|---|---|---|---|---|
| | Columns | Aggr. function | Single entity | Inner Join | One condition | Several cond. | Used | Not Used |
| **C1** | X | | | | | X | X | |
| **C2** | X | | | X | X | | | X |
| **C3** | | X | | X | | X | | X |
| **C4** | | X | X | | X | | | X |
| **C5** | | X | X | | | X | | X |
| **C6** | X | | X | | X | | | X |
| **C7** | X | | | X | | X | X | |
| ***C8** | X | | X | | | X | X | |
| **C9** | X | | | | X | | | X |
| **C10** | | X | | X | | X | | X |

Table A.1: Summary table containing the patterns from clustering results