1 Introduction

1.1 Motivation

While working at IBM in the early 1970, Edgar F. Codd invented the concept of the Relational Algebra used to model data stored in relational databases. In the 1971 paper "A Data Base Sublanguage Founded on the Relational Calculus", he introduced Alpha as the first database language. Based on Codd's work, Donald D. Chamberlin and Raymond F. Boyce, also working at IBM, developed SQL, one of the first commercial languages to operate on the relational model. Originally conceived to retrieve and manipulate data on IBM's System R, SQL quickly became the most widely used relational data query language and a de facto standard.

Despite its popularity, SQL has a number of flaws, some of which caused by the fact, that SQL was designed to resemble natural language. This design philosophy lead to arguably absurd constructs such as the IS NOT DISTINCT FROM operator, and, while working well for simple queries, leads to increasingly complex constructs for non-trivial queries. Because of this increase in complexity, tools that partially or completely automate the generation of queries have been investigated by both researchers and commercial companies alike.

Due to the widespread use of SQL, most tools are focused on a specific field of application, therefore limiting the complexity, but also power of the tool. Moreover, the inconsistent implementation of the SQL standard by most major database vendors (the exception being PostgreSQL) often leads to tools that are tailored to work with a specific implementation only, producing non-standard SQL code that does not work on other implementations.

During my semester break, I worked with Orbis, a clinical information system produced and marketed by Agfa Healthcare. Orbis is built on top of a relational database backend provided by Oracle and includes a visual query builder used to design queries to be executed on this database. While it is a powerful tool, I was never quite satisfied with it, wondering if there was a more fitting approach.

At the same time, I was starting to get interested in Blender, an open-source 3D modelling software. One feature that particularly caught my attention, was Blender's apporach to certain relatively complicated tasks such as texturing and post-processing, using a node or flow-based programming interface. This choice of interface has proven to be vastly superior to the interfaces used in previous iterations and has therefore been applied to more and more parts of the software.
The finding, that the node-based programming paradigm can be successfully applied to complex tasks combined with the apparent need for a more capable query builder lead to the idea of a node-based visual query builder, which is the subject of this thesis.

1.2 Background

To investigate the idea of a flow-based visual query builder, we first need to establish some background information on the major concepts involved. This includes a definition and some historical background as well as a look at some work previously done in the field.

1.2.1 Visual Query Builders

1.2.1.1 Definition

Visual Query Builders are a type of visual programming language designed to simplify the construction of queries on a database. Jeffrey Nickerson defines a visual programming language as a language that uses diagrams in the process of programming, where a diagram is a figure drawn in such a manner that the geometrical or topological relations between the parts of the figure illustrate relations between other objects. There are many other defitions, but most of them are very general and could be applied to most query builders that have a graphical interface.

1.2.1.2 History

The flowchart was introduced by Frank Gilbreth in 1921 as a structured method to model document process flow. However, the use of diagrams to visualize relations between objects can be dated back thousands of years. When computers were introduced in the second half of the twentieth century, they lacked a graphical interface and were therefore unsuited to work with graphical environments such as diagrams. This changed with the introduction of the Xerox Alto developed at PARC which greatly influenced Apple Lisa, the first personal computer to offer a graphical user interface, and more prominently the Apple Macintosh in 1984. However, despite the advances in technology and much interest in the field, visual programming did not find widespread adoption. As a result, the vast majority of programs are still written in a traditional text-based programming language.

1.2.1.3 Examples

To get an idea of the current state of the art of visual query builders, we will take a look at a list of examples. These include commercial products as well as solutions presented in research papers. The examples presented in this section are meant as an overview rather than a thorough analysis of the field. However, some of the examples presented here will be used later on to derive a set of specifications that will define the outline of the tool presented in this thesis.

EasyQuery EasyQuery is a commercial product developed by Korzh.com, located in Kyiv. It is presented as an "ad-hoc visual query builder [that] allows [the user] to construct a query visually: simply by assembling a phrase in natural language."

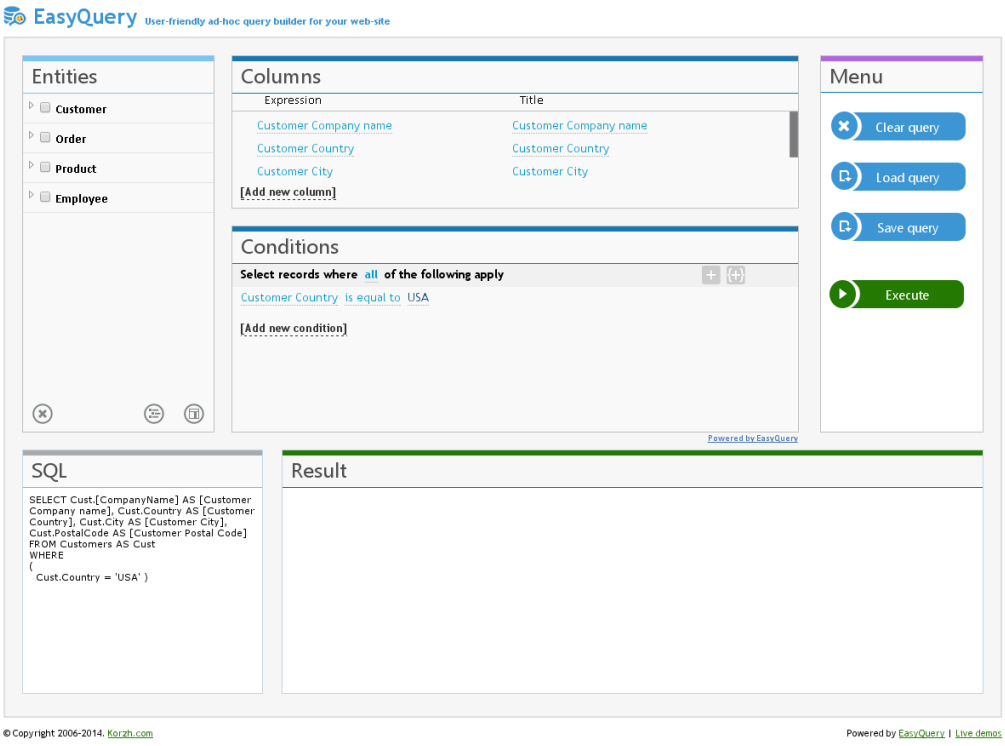

Figure 1.1: A screenshot of EasyQuery, showing its form-based interface

The user interface consists of lists for the entities, columns and conditions as well as two output text containers, one for the produced query and optionally the results from its execution. As such, by the definiton used earlier, EasyQuery cannot be considered a visual programming language, as it does not include diagrams or figures of any sorts. In spite of this, EasyQuery has a few unique features lacking from the other query builders examined, such as an implicit join notation, partial subquery support and a powerful, tool-assisted schema declaration.
The latter includes information not only about the tables but also their relations, therefore enabling the automatic joining of tables when necessary. For instance, if a user adds a condi-

tion comparing two columns originating from different tables, EasyQuery will automatically attempt to derive a join order, based on the foreign keys defined in the schema declaration. This feature is targeted at novice users unfamiliar with the concepts of relational databases. This comes at a cost however, greatly reducing the amount of control the user has over the structure of the query and limiting the number of queries that can be described. One such limitation is the inavailability of self-joins, as the query builder is unable to distinguish it from a simple comparison of two columns in the same table.

ExtJS Most commercial general-purpose query builders put the focus on joining the tables by means of visually connecting nodes. One examples is ExtJS, developed by cfsolutions.

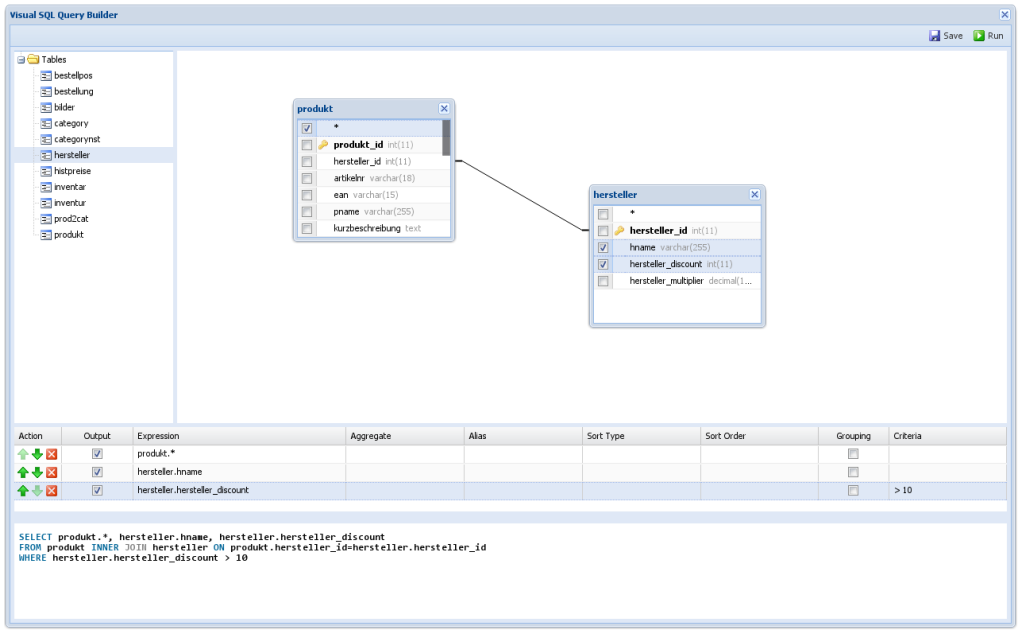

Figure 1.2: A screenshot of ExtJS, showing its node-based interface

Tables are represented as nodes, containing a list of the columns present in the table. A checkbox is used to designate whether a column is in the projection. Tables can be joined by dragging a column onto another one, resulting in a visual indicator, as can be seen in the screenshot.
Due to the use of nodes and connections between them, ExtJS and similar query builder satisfy our defintion of a visual programming language. However, projected columns are displayed in a list with the option to apply expressions, aggregate functions, aliasing, sorting and grouping as well as conditions, all using text fields. This non-visual approach is very powerful, allowing the user to design complex queries. However, it is very difficult to maintain type safety and therefore, it is possible to create syntactically incorrect queries in many different ways.

DFQL Dataflow Query Langugage is a visual query language proposed in a 1994 paper by Gard J. Clark and C. Thomas Wu. It is based on the concept of flow-based programming allowing incremental construction of queries.

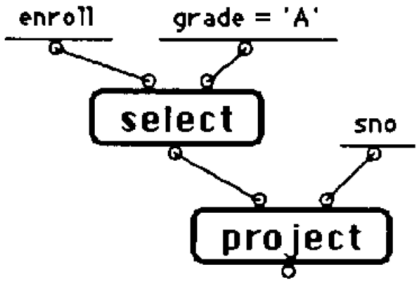

Figure 1.3: A simple query in DFQL

DFQL is very close to the idea resulting from the motivation, using flow-based progrmmaing to access data on relation databases. However, DFQL is not a query builder in the classic sense, as it is intended as a standalone query language. Unlike SQL, it is a strict implementation of the relational algebra and relationally complete. It maintains relational closure, meaning that the output of every node is again a relation, leading to a very intuitive way of constructing subqueries.
Inputs that are not relations, such as conditions and projections are entered in text form, represented as a straight line in the diagram. This approach is conceptually similar to conditions in ExtJS, albeit more tightly integrated into the interface. It shares the weaknesses and strengths of this approach of being very powerful while allowing the creation of syntactically incorrect queries.
DFQL makes strong use of the flow-based approach, allowing the grouping of multiple nodes into subnodes. This feature is also present in Blender and therefore part of the original idea for the tool discussed in the later sections.

1.2.1.4 Disscussion

Commercial and academic research has spawned a number of different visual query builder concepts, each with their own strengths and weaknesses. The flaws and inconsistencies of SQL make it very challenging to create a tool that is both powerful and intuitive, while ensuring correctness of the query.
EasyQuery shows almost perfect correctness at the cost of significantly reduced power and a less intuitive interface. ExtJS is a more visual as well as powerful approach, but does not feature a particularly intuitive interface either and generally not suited for novice users. DFQL on the other hand is vastly superior in both intuition and power, as it fully supports subqueries, but shares the correctnes weaknesses of ExtJS. Moreover, the fact that it does not produce SQL greatly reduces the practical use. From the analysis of DFQL it is clear though, that the flow-based approach offers some significant benefits.

1.2.2 Flow-based Programming

Definition

The term flow-based programming (FBP) has been used to describe a number of different concepts. Common characteristics include an architecture based on nodes and edges, which usually extends to the graphical user interface. Applications are defined as a network of black-boxed processes that exchange data across predefined connections. This network is usually defined in the form of a diagram, making FBP a visual programming language and essentially language-independent. FBP is closely related to Linda, a coordination and communication model developed by David Gelernter and Nicholas Carriero at Yale University, as both are ”coordination languages”.
Classical FBP, as defined by its inventor John Paul Morrison, has a focus on data processing and paralellism and therefore includes a number of features such as buffered connections, back pressure and packet lifetimes which are mostly missing from more general interpretations of the concept.
Since query builders are essentially compilers and therefore not concerned with data processing, the tool presented in this thesis is not a classical FBP application. However, with the recent rise in popularity, the term FBP has been somewhat generalized to include applications that are not based on stream data, but use flow-based networks to define the application structure.

History

Flow-based programming was invented at IBM in the early 1970s by John Paul Morrison. It was strongly influenced by simulation languages, GPSS in particular, which was also developed at IBM. Originally implemented under the name Advanced Modular Processing System (AMPS), FBP underwent a number of name changes until settling on its current designation with the release of the book *Flow-based Programming: A New Approach to Application Programming* by J. Paul Morrison in 1994.
In its first 40 years of existence, FBP has attracted little attention and therefore hasn't seen a lot of use. Recently however, interest in the concept has been growing strongly, mostly due to NoFlo, an FBP-based project started by Henri Bergius in 2013. In light of these developments, a number of major software companies such as IBM, Facebook and Microsoft, have announced new products based on the concept.

Examples

BankOfMontreal Mech The first flow-based application was a banking system designed to support the on-line banking operations of the Bank Of Montreal. Implemented in the mid-'70s, it provided real-time transaction processing, a feature that was not supported by the then dominant IBM COLT system used by most of the other major banks. It was mostly home-grown, with the support of an IBM team, including John Paul Morrison. A large part of the application was built using FBP, which turned out to be extremely powerful and resilient. In fact, parts of the software are still actively used as of 2014, over 40 years after their implementation.

NoFlo NoFlo is an open-source project lead by Henri Bergius aiming at creating a JavaScript programming environment based on the concepts of FBP.

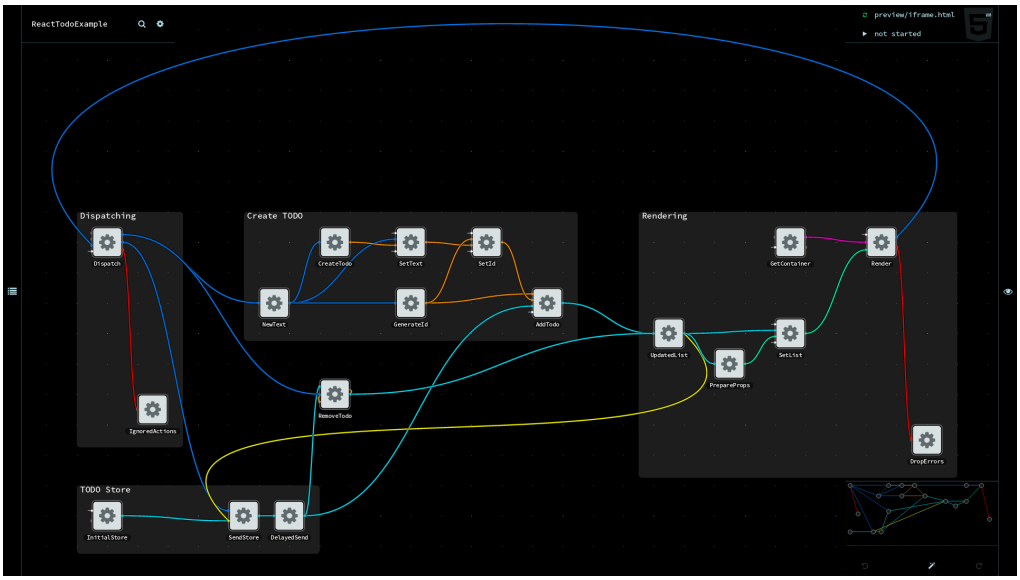

Figure 1.4: A NoFlo application

The first commit was pushed in June 2011 and in 2013, a Kickstarter campaign was started to secure funding for the development of a graphical interface to NoFlo, titled NoFlo-UI. The campaign was successful not only in securing the funds but spreading the word about the project and FBP in general.
NoFlo differs from classical FBP in a number of aspects as described in an article on John Paul Morrison's website. It suffices to say, that while classical NoFlo is targeted at data processing applications, NoFlo addresses a different problem space, focusing on application design instead. According to its website, NoFlo "refreshes the concepts [of FBP] and brings them to HTML5 and Node.js applications".

1.2.2.1 Disscussion

The strengths of flow-based programming lie in the concept of configurable modularity, allowing for the same black-box components to be used multiple times. Additionally, some implementations also allow grouping multiple nodes into subnodes, leading to a very high level of abstration. This also leads to great maintainabilty, an area often disregarded in literature, despite the fact that companies often spend 80-90% of their resources on maintaining old applications. Furthermore, applications designed with the concepts of FBP in mind display an impressive robustness while being surprisingly performant especially in a massively parallel setting.
FBP also exploits the developer's visual cognitive skills, and allows for incremental software development and maintenance. The focus on the data flow between components rather than the funcionality of the components themselves, introduces a natural notion of time as well as dependency, further contributing to the flexibility and expressiveness of the model.