

Markku Rontu

Visual queries for a student information system

**Master's Thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science in Technology**

Espoo, 7th April 2004

**Teknillinen korkeakoulu Helsinki University of Technology
Tietotekniikan osasto Department of Computer Science and Engineering**

Helsinki University of Technology

Abstract of Master's Thesis

Author:	Markku Rontu
Title of thesis:	Visual queries for a student information system
Date:	7th April 2004
Number of pages:	64
Department:	Department of Computer Science and Engineering
Professorship:	T-106 (Software Technology)
Field of study:	Software Systems
Supervisor:	Lauri Malmi, Prof.
Instructor:	Ari Korhonen, Dr.Sc. (Tech.)
<p>Databases are everywhere. A great number of special purpose applications have been built to query information gathered into databases. There is still need for generic query systems that allow the making of arbitrary queries from any database. In particular, a visual query system enables the visual display of information and is in many cases easy to use.</p> <p>This work examines the visual query system approach in querying student information from the OREK database at Helsinki University of Technology. Main problems identified from the existing use are the complex database schema and the long time it takes to formulate queries. An easy user interface would enable more users to study the information than is possible with the current approach. Previous work is surveyed to find existing systems that could be used.</p> <p>A new diagrammatic visual query system named SEEQ (System for Enhanced Exploration and Querying) is presented as a tool for querying databases in general and student information systems in particular. The user interface, most important features and implementation are described. SEEQ is compared to existing systems and is found to have many useful features in common with them. Its utility in querying student information is evaluated by a set of example queries that are based on existing queries and research needs. SEEQ is found to be strongest in visualisation because, unlike many other systems, it is capable of result visualisations in addition to schema and query visualisations. Finally, improvements and future work are discussed.</p>	
<p>Keywords: visual query system, database querying, data visualisation, student information system, SEEQ, zoomable user interface</p>	

Teknillinen korkeakoulu Diplomityön tiivistelmä

Tekijä:	Markku Rontu
Työn nimi:	Visual queries for a student information system
Työn nimi suomeksi:	Visuaalisia kyselyjä opiskelijarekisteriin
Päivämäärä:	7. huhtikuuta 2004
Sivuja:	64
Osasto:	Tietotekniikan osasto
Professori:	T-106 (Ohjelmistotekniikka)
Pääaine:	Ohjelmistojärjestelmät
Valvoja:	Prof. Lauri Malmi
Ohjaaja:	TkT Ari Korhonen
<p>Tietokannat ovat tietotekniikan arkipäivää. Vuosien saatossa on rakennettu suuri joukko ohjelmia käsittelemään tietokantoihin tallennettua tietoa. Tyypillinen ohjelma rakennetaan tiettyä kyselyjoukkoa ja tietokantaa varten, mutta yleisempi kyselyjärjestelmä sallii kyselyjen tekemisen mihin tahansa tietokantaan. Erityisesti visuaalinen kyselyjärjestelmä mahdollistaa tietokantamallin, kyselyjen ja tulosten visuaalisen esittämisen. Visuaalinen kyselyjärjestelmä on myös usein helppokäyttöinen.</p> <p>Tässä työssä tarkastellaan kyselyjä Teknillisen korkeakoulun opiskelijarekisteriin (OREK) visuaalisen kyselyjärjestelmän kannalta. Merkittävimmät ongelmat nykyisessä käytössä ovat tietokannan mallin monimutkaisuus ja kyselyjen muodostamiseen kuluva aika. Helposti käytettävä kyselyjärjestelmä mahdollistaisi sen, että useat uudet käyttäjät voisivat tehdä kyselyjä. Ratkaisua etsitään katsauksella aiemmin toteutettuihin järjestelmiin.</p> <p>Ratkaisuna esitetään uusi diagrammipohjainen visuaalinen kyselyjärjestelmä SEEQ (System for Enhanced Exploration and Querying) sekä yleisesti tietokantojen että erityisesti opiskelijarekisterien käsittelyyn. Sen tärkeimmät ominaisuudet, käyttöliittymä ja implementaatio kuvataan. SEEQiä verrataan olemassa oleviin kyselyjärjestelmiin ja sillä on monia yhteisiä, hyödyllisiä ominaisuuksia. Sen käyttökelpoisuutta arvioidaan esimerkkikyselyjen avulla, jotka on muodostettu nykyisten järjestelmien käytön ja arvelujen tutkimustarpeiden perusteella. SEEQ havaitaan erityisen hyväksi visuaalisuudessa sillä, toisin kuin monet muut järjestelmät, se sisältää myös tulosten visualisoinnin tietokantamallin ja kyselyn visualisoinnin lisäksi. Lopuksi eritellään parannusehdotuksia ja jatkotutkimusmahdollisuuksia.</p>	
<p>Avainsanat: visuaalinen kyselyjärjestelmä, tietokantakysely, tiedon visualisointi, opiskelijarekisteri, SEEQ</p>	

Acknowledgements

It is done. I am satisfied with the results, although, it would have been nice to include usability testing and a few more iterations of development. There is not time for everything. The system is now going to my personal use and it will be improved further.

This project began roughly 18 months ago as I realised I would not have the time to do the countless user interface views required in a space strategy game. After contemplating code generation, I stumbled upon the idea of giving the user the power to make database queries using a visual interface. As the spring came, I discovered a seminar course for which visual query systems were a good topic. This was my first introduction to previous work. I was interested in implementing a visual query system and needed a thesis subject. Fortunately, there was interest in using one to query student information. To make a full circle, as this project now comes to an end, I will be going back to applying the visual query paradigm to other applications and databases.

I would like to thank Jan Lönnberg and Jukka Villstedt of many fruitful discussions and Kaj Björklund of implementation advice. I am also indebted to Lauri Malmi and Ari Korhonen for this great opportunity of working on this topic, for making my text much more scientific and for patiently commenting my partially completed versions. Thank you also to all my friends from school, work and beyond that have made the last five years the best years of my life.

Finally, I would like to thank my parents. Without their support I would not have been able to concentrate on the work and graduate so swiftly.

Otaniemi, 7th April 2004,

Markku Rontu

“SEQ and ye shall find.”

Contents

1	Introduction	1
2	Problem case	3
2.1	Student information system	3
2.1.1	Current use	3
2.1.2	Improvements	4
2.2	Example queries	5
2.2.1	Data model	5
2.2.2	Queries	6
2.3	Requirements	8
3	Related work	12
3.1	History and overview of visual query systems	12
3.2	Languages and environments	15
3.2.1	Draw an Object-Oriented Database Language (DOODLE)	15
3.2.2	Gql	17
3.2.3	Graphical User Interface for Database Exploration (GUIDE)	19
3.2.4	Query-By-Diagram* (QBD*)	20
3.2.5	Others	21
3.3	Evaluation criteria	22
3.3.1	Classifications	22
3.3.2	Formal differences	28
3.3.3	Visual differences	29
3.3.4	Miscellaneous differences	30
3.4	Comparison	31
3.5	Conclusion	32
4	Prototype	34
4.1	Example case	34
4.2	A more complex query	39
4.3	Discussion	40
4.4	Summary	42
5	Implementation	43
5.1	Query translation	44
5.2	Lessons learned	46
6	Evaluation	48
6.1	Initial goals	48
6.2	Requirements analysis	48
6.3	Comparison to other systems	50

CONTENTS

v

7 Conclusion	52
7.1 Improvement ideas	52
7.2 Future work	53
A Requirements results	54

Chapter 1

Introduction

Databases are prevalent. They are used, for instance, to store government records, to maintain bank and credit accounts, to keep medical information, to catalogue books in libraries and in the back-end of a wide variety of enterprise applications. In this work, databases store student information.

Besides governments, institutions and businesses, ordinary people can make use of databases in their daily activities, if the systems can be made simple to use. Databases can be used to store, for example, recipe, record, video and address information and to search for the right recipe, song, movie or address. Appointments or a calendar as well as a task list can be kept in a database and several kinds of useful views can be made to show the information grouped and sorted according to priority, date or completion status. Files on a computer can be stored in a database and easily searched. For example, photos can be explored with thumbnails and notes of the occasion. Even source code can be put into a database and efficiently processed by automated tools. Of course, all of these activities are possible without using databases and visual query systems, then the users have a set of regular applications (*e.g.*, calendar, address book, image viewer) they must use. The applications work in different ways, they may not interoperate well and certainly the applications cannot usually be customised very much by the user.

A significant problem of many existing query systems is that they require the user to know SQL (Structured Query Language) or a similar database query language. A query system with a complex textual query language is not a good tool for many users. A visual query system is an easier tool than a textual query system for novice users, and a visual query system can still retain the flexibility and power to satisfy expert users. Therefore, visual query systems are studied in this work to find a system that can be used to query student information. There are many existing systems that are surveyed such as DOODLE, Gql, GUIDE and QBD*. The systems are looked at from different perspectives to find the most suitable systems.

Regular applications are engineered for a single purpose. This is good because, with the help of usability experts doing usability tests, the application can be made to be optimal for the users and their particular use. The implementation is also more straightforward because it does not need to work in the general case. But designing an application for a single purpose is bad because the application cannot easily adapt to a change in the requirements. Sometimes it is impossible to please all users and uses of the application at the same time and sometimes a requirement is to be flexible and support different uses. Usually, if the user needs to have a new field displayed in a view, he must request it from the developer of the software, which is often expensive or impossible. There is a need for generic systems that are flexible. A generic system and in this case a generic query system has the advantage of being highly customisable by the user. However, a query system requires the user to know a query language and moves some of the burden of the implementation to the end user (*i.e.*, the making of the query and result view). The degree of genericity is a difficult trade-off.

Most applications can benefit from having the database separated from the application code. Two- (client-server) and three-tiered (database, business logic, user interface) architectures with database at the bottom are examples of how that can be accomplished. The database abstracts the storage in a generic way and usually offers additional features such as atomicity, consistency, isolation and durability. Indeed, if many applications are in theory just special purpose database front ends, it is good to analyse why they do not use full-fledged databases and why they are not accessible through a single generic front end. There are many reasons for this. For instance, real-time systems or performance hungry applications may have too high performance requirements. The existing generic query systems have problems like bad usability in various tasks. Good generic systems are difficult to make.

The term visual query system means in this work a database query system that displays the schema, the query or the results visually. The user interface leverages direct manipulation techniques such as dragging with the mouse. Regular non-visual query systems sometimes offer simple graphical representations of the schema. They usually require the user to enter SQL queries by hand and display the results as tables of data. A table is good for some information but less than optimal in some cases. For more visual needs, there are separate tools to construct database schemas visually, semivisual programs to send queries to databases and programs to visualise data. In this work, a system is presented that can do all of these tasks and offers a visual environment in each. There are existing visual notations like UML (Unified Modeling Language) for modelling programs and ER (Entity Relationship) diagrams for databases, and some applications do take advantage of them. Modern integrated development environments offer code generators based on UML and also reverse-engineer implementation diagrams from source code. There is no reason not to leverage the same advantages of visuality and integrate visual aspects into database query systems.

The goal of this work is to find effective visual query systems for querying student information. The characteristics of student information systems are studied. The student information does not have any special needs, when compared to databases of bank accounts, catalogues or music records. Therefore, the resulting query system SEEQ is not limited to databases of student information.

Chapter 2 begins with a description of the student information systems of Helsinki University of Technology and what problems there are in their use. A single system (OREK) is the most important database and its characteristics are discussed in detail. A set of queries is formulated to show examples of its use and evaluate visual query systems. In Chapter 3 the field of visual query systems is discussed and related work is examined. A few important systems are discussed in detail. Finally, a comparison of visual query systems is presented. In Chapter 4 a prototype visual query system named SEEQ is presented as the solution to the problems. Its features are discussed with an example case as well as comparison to related systems. The architecture of SEEQ and experiences from its implementation are outlined in Chapter 5. In Chapter 6 SEEQ is evaluated based on the goals and requirements from Chapter 2 and in comparison to the systems presented in Chapter 3. This work is concluded in Chapter 7 with improvement ideas and future work.

Chapter 2

Problem case

In this chapter, the characteristics of the application domain, namely student information systems are discussed. Also, the current situation regarding the use of student registers at HUT (Helsinki University of Technology) is presented. Some requirements are formalised for this work borrowing ideas from requirements specifications in software engineering. The motivation is to make the results of the work and the advantages and disadvantages of visual query systems more evident and more easily analysable.

2.1 Student information system

2.1.1 Current use

A student information system or student register, in general, is a collection of study related information relevant to a school. In this case, at HUT, the student register (OREK replaced by Oodi) consists of personal records such as the name, address and student number as well as course records and student grades, altogether hundreds of tables and hundreds of thousands of records. At HUT there are also other separate databases such as the OSR (Osasuoritusrekisteri) of the Computer Science department, which holds data mainly consisting of partial completions of courses by students. Legacy databases still contain information to be moved into Oodi at a later date. There are dedicated users managing and accessing the information. For this work, I interviewed the users of the OREK and Oodi student registers (K. Olamo and P. Solin, personal communication, October 6, 2003).

Common uses for the stored information are, for example:

- querying how many students are present or absent,
- examining which courses a particular student has completed and
- listing the degrees granted.

Typical queries can involve joins of five to seven tables. If the output must have textual explanations of enumeration type fields, an additional join of the explanation table to the result is required for each type. Many queries are needed periodically, for example, asking who the new students are each year. It is useful to store these recurring queries and modify them to suit each new situation instead of writing them from scratch all the time. Some queries, like checking for errors in recently entered data, may occur only once.

Developing new queries takes a significant amount of time. From several hours to days depending on the user's experience and how well he remembers the technical details of the concepts in question. Even for a frequent user, who remembers the details, the significant factor is the time that it takes to formulate the query using the concepts of the database model and not the actual entry. It does not help that there are numerous technical details

and many intricate relationships that need to be considered to get valid results out of the query. Technical details like expiration dates, filtering out graduated students and validating information are such that they must always be considered regardless of the query system and are such that they can be more easily manageable with a better system.

Information is also processed and given to third parties to use. The output can be simple formatted text or spreadsheets in this case. It should be noted that privacy is a very important concern and any tool to be used to access the student register must be designed with that in mind.

Currently, RapidSQL [Rap] is used to make SQL queries of the data. It is a traditional tool that offers the possibility of viewing and searching the database model details easier than browsing papers with the model details. It also does syntax checking for SQL queries. Query results can be exported to formats other programs can use and the queries themselves can be saved for later use.

Having now described the current system and its environment, it is easy to see there is room for improvement. Next, there are some improvement ideas and features for a hypothetical new visual query system.

2.1.2 Improvements

There are several potential types of users and uses for the new system. First, student coordinators, course organisers and other staff sometimes want to ask queries that are not supported by the current systems (*e.g.*, hard coded queries in OSR). A new flexible query system would not have limitations in the kind of queries that can be made. It is almost impossible to compete in the ease of making a particular supported query with custom made tools, such as the queries in OSR now. That is because those queries can be executed by pressing a single button or at most entering a handful of values. However, it is easy to improve a static set of entry fields and enable almost limitless potential for query refinement. But giving the users a free SQL entry field as the interface is not a wise choice. It is not sensible to expect that every potential database user knows SQL nor is it sensible to teach it to all of them. And even users who do know SQL can use something more efficient. It is laborious to produce meaningful queries with SQL, so tools that ease the query formulation process would be practical.

Also, a new visual query system would be a platform that enables better use and research of the data available in the student register. An effective query system is related to data mining. It enables the discovery of new meaningful information. It is important for the school to find and fix bottlenecks and problems, help students in their studies and ensure smooth operation (and ultimately graduation). Therefore, providing researchers, course planners and staff access to the information, and in this case the possibility to experiment easily with queries, would support their efforts.

Considering more practical work and the current use of the student register, it should be possible to save a lot of time by improving how queries are made. Making simple joins easier to accomplish is straightforward in a visual query system and can save time. A more time-saving improvement would be to ease the designing of the query, particularly browsing of the model in search of the right concepts. And for any serious use, it would be convenient, if the validation of the query and its results would be easy. It is important to know that the query and the results are in fact the right query and the right answer to the original question.

It should be possible to have a concept search feature, list of concepts and a graphical view of the whole data model. For the user to be able to effectively grasp the few hundred or so database tables, the system must present the model as a simplified graphical overview and enable some kind of hierarchical zooming into the details when necessary. There are so many concepts that it is impractical to display the entire model on paper. However, an interactive presentation has many advantages and would be very helpful in many cases. There are some solutions to the problem of a large model in previous work and those are

discussed in the next chapter. Lots of intricate details, which are unavoidable because they are part of the problem domain, will still remain to complicate querying.

It would be advantageous to make a new system, which would represent a common interface to all the different heterogeneous sources and possibly access them all at the same time. Monolithic large databases have a habit of becoming, at least perceivably, too large and unwieldy, which often becomes a desire for the various parties involved (*e.g.*, departments) to move into using separate independent databases. If such a new system would aggregate, on demand, the information from the different sources, there would be no need for a single large database, because the user's view to the data is uniform. The largest problem in this is that accessing multiple databases, especially when some resemblance of efficiency is required, is harder than accessing a single large database and downright impossible in some cases. Also, keeping data available and handling concurrency properly makes any query system a very complex program.

Besides being a solution to the aforementioned practical needs, a new visual query system also serves several other research goals unrelated to the student register. For quite some time, I have been disappointed by the modern user interface libraries and the way we construct programs that manipulate databases. A generic visual query system is a potential solution to many of the problems because a user interface is not constructed at a low level but queries are formulated instead. There are also many applications that use databases and the users of those applications can benefit from a truly easy and practical database query system. Examples are customer relationship management, financial data analysis, medical databases, mind maps and similar. Even recipe, record and movie databases would benefit from a unified powerful interface. These other needs manifest themselves in this work as additional requirements to the implementation. It should be generic enough to be easily applicable also to other types of databases and to other problem areas.

Last but not least is the central idea of having a visual system instead of a traditional one. We humans can take advantage of the visual display of information and it also offers advantages in usability for many of the potential users as well as a more satisfying experience (see Subsection 3.3.4). Despite that the research of visual query systems has a long history, as will be explained in the next chapter, it is far from over. A new visual query system can help push the research forward.

2.2 Example queries

It is quite hard to pin down all useful query types. As a matter of fact, that is a good indication that there is need for a real query system with the necessary generality and adaptability. As described in the first chapter, the common approach to accessing databases is to fix the set of queries at design stage and then implement them and only them. To facilitate research, the new system must be able to adapt to the requirements of the researchers, whatever they may be (as long as they are sensible). However, there are already some interesting queries that must be possible to do in a new system as well as existing queries made by the users of current systems.

Data modification and entry is an important part of many databases. Therefore it should be present in a generic visual query system. However, it is a large topic where visibility may be of less use. Certainly, it is out of the scope of this work because of the excessive validation needs present in the student register. It has also been left out of many of the existing visual query systems.

2.2.1 Data model

The example queries all relate to a hypothetical student register since the actual database schema of OREK has many more concepts than is necessary. The database model is shown in Figure 2.1. Roughly speaking, there are students who participate on courses and other

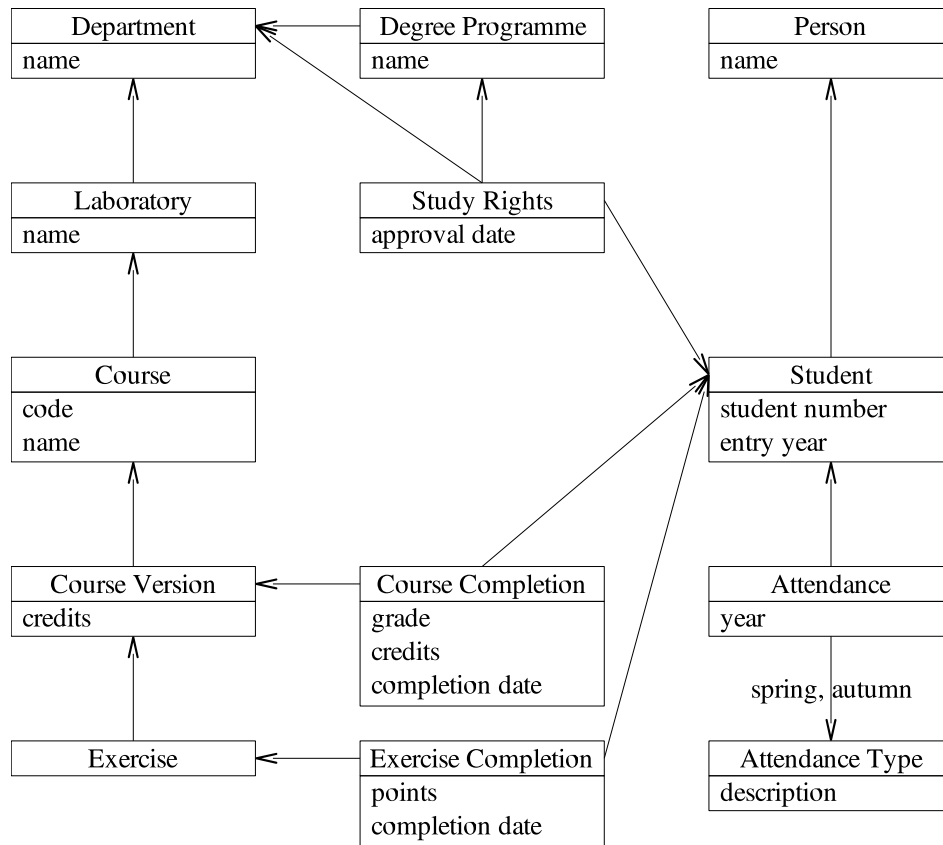


Figure 2.1: Example data model of a student information system

Table 2.1: Explanations of the used prefixes

Prefix	Source
Oodi	Motivated by Oodi
OSR	Queries in OSR
Pi	Pure invention
EQ	Example query (useful queries selected as examples)

people like the users of the register. All persons have names but students also have a student number. Each course belongs to a laboratory in a department. Each course has a unique code and a name. Each course version has a certain number of credits, a lecturer and a beginning and an ending date. Each course version has a set of exercises and completing them will give the user points. Students will receive a grade on completing a course.

The term course version is used to refer to the particular instance of a course organised during a particular semester. For example, a course on data structures and algorithms is organised yearly and may be organised by different people each year. The instances have the same name and course code but some attributes may change like the lecturer or the number of credits and certainly the beginning and ending dates.

2.2.2 Queries

Next, there are the actual queries. Each query is identified using a prefix and an ascending serial number. The prefixes that depend on the class of the query are identified in Table 2.1.

Queries Oodi1-3 in Table 2.2 were motivated by examples of real queries made in Oodi.

Table 2.2: Queries from Oodi

ID	Query
Oodi1	Which students participated in course C arranged at time T?
Oodi2	Which students are new in year Y and are not absent the whole year, or were new in year Y - 1 but were absent the whole year?
Oodi3	Which students have changed degree programme to programme P in department D in year Y or changed department from department D?

Table 2.3: Queries from OSR

ID	Query
OSR1	Who got the best grades from every exercise during the course version V?
OSR2	Who have not passed any exercises of course version V?
OSR3	Who have not yet completely passed course version V?
OSR4	Who have multiple uncompleted versions of course C?
OSR5	Who failed course version V?
OSR6	Who failed exercise E three times or more?
OSR7	How many students passed course C (or course version V or exercise E) during time frame T1-T2 (or before date D)?

If the new system is to replace the systems currently in use, it needs to be able to duplicate their functionality.

The purpose of queries OSR1-6 in Table 2.3 is to find students that did very well or students that might have had problems with their studies and show where the problems have been. It is useful to follow the number of students passing courses and their exercises over the years. OSR7 shows the number of students. Furthermore, it is useful to show the differences between the course versions that are arranged. The new system should also be able to plot these as various graphs.

Pi-queries in Table 2.4 are ideas of interesting queries that might be useful. Actual use of a visual query system will most likely add a lot more to the list and necessitate the refinement of these queries.

Some additional filters and weights that are applicable for several of the queries are: gender, age, graduation, success in basic mathematics and physics courses, number of points in entrance examinations, department, degree programme and group by starting year.

Next, there are a few representatives of the presented queries to be used later for testing of the visual query system. They also provide the reader an overview to some aspects of the problem domain. They form a test for the system and its potential. Because the test data comes from the previous student register OREK, there are some limitations to the queries that can be chosen. For example, there is no data available of partial completions, exercises or course versions as there is in OSR.

The example queries are in Table 2.5. The first query EQ1 is representative of the quer-

Table 2.4: Queries that were thought interesting

ID	Query
Pi1.1	What courses have students, who began their studies in year Y, passed?
Pi1.2	How many students have not passed any courses or only a few?
Pi2.1	What are the dependencies of courses visualised as a graph?
Pi2.2	Add to edge weight, if a student has passed a dependency of a course. Visualise edge weights.
Pi2.3	Which courses have had students participate many times?

Table 2.5: Queries selected as examples

ID	Query
EQ1	Which students are new in year 2000 and are not absent the whole year, or were new in year 1999 but were absent the whole year?
EQ2	Which students that began their studies in year 1998 have passed less than five courses?
EQ3	Show a graph of courses. If a student has passed a course C1 before course C2, make a precedence edge between the courses or add 1 to an existing precedence edge weight. Show the weights on the edges.

ies done using the current system. It is the modified Oodi2 question but now with concrete years. The second EQ2 is an example of the research use. Partial completions are not available in the OREK database but it is possible to query course completions. The query that stresses the visual output capabilities most appears last. EQ3 visualises the order in which students have actually passed courses. It would be useful to compare the actual order of a group of students into the recommended order and, for instance, try to see if students with troubles in their studies have had a different order.

2.3 Requirements

Here are important questions that I try to answer in this thesis. They form a set of requirements for the system implemented. They are in a semiformal notation like requirement specifications in software engineering. First, there is always a concrete question the system tries to answer. Second, the explanation and details. Finally, a validation criteria to see if the system does satisfactorily answer the question. The alternatives are emphasised. Some questions are open-ended and there is no list of alternatives. Some questions have more precise supplementary questions that will together answer the main question.

ID	Description
Q1	Can a visual query system be used to gather information from a (student register) database?
	If the system is not able to show the information recorded in the database(s), it is worthless. The system must be able to show all recorded information. Most research has focused on simple alphanumeric data and it is common data in databases as well as in the student register. Further types need to be supported for other kinds of databases but are not the scope of this work.
A1	The system can display none / some / all of the information.

Q2	Can the system answer the example queries?
	The system should be able to answer all the example queries as long as there is appropriate data in the database.
A2	The system can answer none / 25% / 50% / 75% / all of the queries.

Q3	Can a reasonable implementation be made for a visual query system?
	It is important to know the problems behind a generic visual query system and particularly its implementation. If there are no significant problems, it will be a possible choice also for commercial products.
A3	The required implementation effort is insignificant / significant / too much .
Q3.1	Are there any particular problems with visual query system implementation?
	Since the system in question is visual in nature, it will require more processing power from the client side of the query process than a regular text-based query system. It is likely that this is a hurdle for visual query systems even though modern personal computers are very well capable of displaying complex graphics.
Q3.2	What technical difficulties are there or would be?
	Technical problems should be possible to overcome. At least it would benefit future implementations if the technical problems are known when initially designing a system. What hurdles still remain?
Q3.3	Does it take too much time to develop?
	It may be possible that a good visual query system is impossible to develop well with the modern software engineering practices and programming tools. It would be useful to assess the difficulty of the implementation task.
Q3.4	Is the implementation effort comparable to conventional systems?
	A conventional program is such that there is a view for each distinct query with entry fields for each parameter. A conventional query system is such that there is a text-based query language and the query is written using the language with the help of the user interface. The former requires the implementation of many views while the latter requires skill in the query language.

Q4	How usable is the visual query system?
	The classical question, yet impossible to answer as it is. There are countless aspects in usability and this thesis is not about performing a usability evaluation, but about implementing a visual query system. Common sense and user feedback will be used to evaluate the usability. The general idea is to build a system that is more effective to the frequent users than the existing systems and to provide a tool that enables the other users to access the data as well. To fully support the latter goal, the new system should be easy to learn. However, it is certain that a powerful generic system is more complex to handle than a simplistic system intended for novice users and limited use. Furthermore, a commercial quality polished product is out of the scope of this thesis as well, so some requirements in the following questions are meant to be taken as general guidelines for potential further development and research in the field of visual query systems.
Q4.1	Can a user do a complete simple query in "a few seconds" and with "a few actions"?
	The query "What courses has a student completed?" is a query that is repeated often, by the students themselves, too. This is an example of a simple query and simple queries should be simple and quick to do even by a novice user. A <i>novice user</i> is a user that has no or very little experience with the system. An <i>expert user</i> on the other hand is someone who has used the system lately and for more than a hundred queries.
A4.1	A novice user can make the query in less than ten seconds / a minute / ten minutes . An expert user can make the query in less than ten seconds / a minute / ten minutes .

Q4.2	Can a user repeat an old query in "a few seconds" and with "a few actions"?
	It is likely that some queries need to be repeated often, so there should be some easy way of storing, retrieving, and executing queries.
A4.2	The user can / cannot record queries for later use. A novice user can repeat the query in less than ten seconds / a minute / ten minutes . An expert user can repeat the query in less than ten seconds / a minute / ten minutes .
Q4.3	Once a user has an idea of a question, how long does it take to formulate a query?
	The objective of the user is to see the results, not manipulate the query, <i>per se</i> . Additionally, the visual query language, just like regular programming languages, should provide constructs with which to think about the problem at hand. When there is a query that needs to be done, the mental model of the query should be close to the actual representation in the visual query system to be easily formulated. As mentioned previously, the time it takes to formulate the query is a decisive factor. The query for this requirement is EQ1.
A4.3	The query takes seconds / minutes / hours / days to formulate.
Q4.4	How easy it is to modify a query?
	Modifying a part of the query should not require changes to unrelated parts let alone reconstructing the entire query again. An advantage of a good direct manipulation system is that the user can just start the query process and manipulate the query on the fly based on the results instead of patiently trying to formulate a syntactically and semantically correct formal statement in the first place. Therefore, users will likely want to refine queries as they go based on the results of the intermediate stages. For example, if they see at an intermediate stage that there are too many or too few results and need to change some filters. The system must be able to support this. When reusing stored queries, they might need refinement like changing a date or filtering based on some new criteria. Sometimes, especially for exploratory or <i>ad hoc</i> queries, the user might not know exactly what the resulting query will be when he starts formulating it. Possibility of easy modification supports this kind of a strategy that will also be useful when learning the database model.
A4.4	Modifying a query requires the user to change just the necessary details / also change unrelated parts / rewrite the whole query . The user can / cannot see intermediate results while formulating a query.
Q4.5	How easy it is to find and fix errors in a query?
	Related to the previous question, people also make mistakes. It is an undeniable fact of life so the visual query system should enable the user to fix errors easily. An effective (programming) environment constantly provides useful feedback to its user. For example, if there are syntax errors, they should be indicated as soon as they are detected.
A4.5	To fix an error the user can change only the erroneous part / must rewrite the query . The system provides / does not provide feedback on errors. Feedback is given after the query is done / while formulating the query .
Q4.6	How responsive is the system? Is it interactive?

	<p>Making queries should be fast. Usability can suffer if the feel of the user interface degrades due to slow response times. In a direct manipulation system like many visual query systems the user interface itself has some requirements for efficiency. Actual query processing time cannot realistically be better than in an SQL-based system. There are well optimised production quality DMBSs (DataBase Management System). If the visual query system internally uses a regular DBMS and its query processing, then the visual query system will be almost as fast. For other kinds of implementations a useful comparison is to some existing available DMBS. The difference between visual query parsing and textual query parsing should be an insignificant difference. The visual query system should be faster to parse as the representation is already a syntax tree in the memory instead of an array of characters. However, if the visual query is transformed into a textual query that is passed on to a regular DBMS, the visual query system adds more overhead. This is meaningless compared to the time the DBMS uses for computing the results and what the visualisation may use.</p> <p>A visual query system with output visualisation can also spend significant amounts of time laying out graphs. This should be considered in the user interface. Existing graph layout systems can be examined for comparison. The more generic the layout system is, the slower it will be, simply because various commonly used simplifications and optimisations are practically impossible to use in it.</p> <p>Suitable systems for comparison include PostgreSQL for querying and Graphviz for layouts.</p>
A4.6	<p>The system and query formulation behaves interactively with short response times / pauses for seconds regularly.</p> <p>The visual query answers queries EQ1-2 slower than / in comparative time to / faster than a regular DBMS.</p> <p>Output visualisations are too slow for practical use / fast enough to be usable.</p> <p>The output layout performs slower than / comparatively to / faster than a regular layout system.</p>

Q5	Is the visual query system better than a textual query system in this case?
	There should be advantages of visuality in the visual query system. However, what is the general opinion of users? Can a visual query system replace a textual query systems in the general case? And specifically, is the visual query system better than the current situation?
A5	No / some / most / all users like the new system more than a regular query system.

Q6	Is there potential for further experimentation?
	Last but not least is the future. Does it make sense to develop systems based on the visual query paradigm?

There are many problems in using databases. For the student register at HUT, the most significant problems are formulating queries in the complex database schema and providing an effective tool for research. For databases in general, there is the problem of providing access to the stored information that coincides with the research tool needs. From these needs, I have elaborated a set of example queries and a set of requirements. Next, I will present previous work and existing solutions from the field of visual query systems.

Chapter 3

Related work

In this chapter, I discuss the history of visual query systems, highlight some interesting visual query systems in more detail and finally evaluate them with the problem case in mind.

3.1 History and overview of visual query systems

The history of visual query systems is relatively long. The history itself has not been an active research topic and no paper describing the developments through the years has come to my attention. There are two fairly recent surveys: in [Chan1997] visual query languages are surveyed for ER databases and in [Catarci1997] for traditional databases with alphanumeric data. This section is mostly based on these surveys and some article reference analysis.

In [Catarci1997] visual query systems are defined as database query systems that use visual representations to depict the domain of interest and express related requests. They are divided into four classes based on what the database and queries look like: **form-based**, resembling paper forms, **iconic**, using symbolic icons in their representation, **diagrammatic**, using various forms of graphs and diagrams, and **hybrid**, combinations of the other three (see Subsection 3.3.1 for more details).

In [Chan1997] the survey is proposed as a checklist of features for future articles on visual query languages. There are tables showing which concepts from the ER model a system supports (*e.g.*, complex attributes and inheritance), what kind of conditionals are possible (*e.g.*, logical nesting and set operations) and what kind of interaction techniques are supported (*e.g.*, subqueries and undo).

Depending on the definition, the first visual query systems emerged in the 1970s. QBE (Query-By-Example), presented by Zloof around 1975, is one of the first, if not indeed the first visual query system¹. It is a form-based system, where the user fills templates of tables with example values. Another often quoted early system is the query language CUPID by McDonald also around 1975², which allows the specification of constraints between tables as lines between their representations on screen, labelled with the operator.

Since the 1970s, a number of different approaches have been developed. Many authors adopted the Entity-Relationship notation as the basis of their systems (*e.g.*, [Wong1982, Angelaccio1990] and all papers from [Chan1997]). Others based on classic forms or their own idea of graphs, using other formalisms (*e.g.*, [Goldman1985, Kuntz1989, Cruz1992] and [Benzi1999, Lee2001, Balkir2002]). The strategies used for interacting with the system, calculating the results of the queries of the users and the countless other details in

¹Implied in [Vadaparty1993, Benzi1999] as the first and no other suitable systems before 1975 are referenced in papers (except CUPID).

²Many papers have references to the 1975 version, in [Wong1982] to a paper from 1974.

the systems have varied greatly. There are some ideas that have been experimented with more often than others (*e.g.*, how to effectively represent conditionals [Goldman1985, Kuntz1989, Murray2000]) and there still remain many problems to study (see the end of this section).

There are many different querying strategies with various advantages and limitations. The queries can be based, for example, on:

1. **A set of examples and possibly counter-examples.** The user gives the system some values that he would like to see in the result and the system generalises the question. [Kuntz1989, Orman1996]
2. **Patterns to be matched against.** The user specifies what kinds of structures are searched, like all students whose last name begins with R and who have taken the course Algorithms and data structures. The system searches the data structure consisting of the data in the database trying to look for matches. [Cruz1987, Consens1990]
3. **Tables, relations or classes and constraints for their columns or attributes.** The user is shown the database model and he can select (*e.g.*, by mouse clicking or other graphical manipulation) the part of the schema he wishes to see the data of. Further constraints can be added to the columns. [Wong1982, Goldman1985, Angelaccio1990, Dennebouy1995]
4. **Data-flow diagrams.** The user creates a data-flow diagram that specifies how the system computes the query. The data "flows" out of the tables through some filters and other operations that combine, affect and select the data and is finally shown on screen. [Murray2000]
5. **Graph rewriting algorithms.** The database is represented by a (large) graph. The system takes a set of rewriting instructions from the user and rewrites the graph into another (result) graph that is then shown to the user. The instructions can add, remove or replace some parts of the graph. [Gyssens1990, Rodgers1997]
6. **Visual programming.** The user defines a program that uses the database as input and computes the desired output. The program is specified using a visual programming language that is like a regular textual programming language, but with visual operations.

The querying strategy greatly affects the outlook and the query formulation details of the language. For instance, can the user express his interest declaratively (1, 2, 3) or does he have to specify details of how the computation is to be done (4, 5, 6)?

Besides the earliest systems already mentioned, often cited systems are GUIDE, ISIS, QBD*, G and GraphLog. GUIDE [Wong1982] is a pioneer from the early days compared to many others but nonetheless an interesting visual query system that uses ER diagrams with (3) as the approach. ISIS [Goldman1985] is a graphical database tool that enables the user to query as well as create and update the database. It uses the Semantic Data Model with (3). QBD* [Angelaccio1990] is a visual query system that uses ER diagrams with (3). G and GraphLog [Cruz1987, Consens1990] both try to balance between computational completeness and efficiency. They use logic programming like pattern matching (2). More details of these systems can be found in the next section.

The most common goal for the researchers seems to be to build systems that are more usable than traditional query systems. There are several systems that should be usable by novices [Wong1982, Goldman1985, Angelaccio1990, Benzi1999]. A different kind of goal is to build systems that are easier for recursive queries than traditional query languages such as [Cruz1987] or for querying a particular kind of databases like scientific databases [Balkir2002], geographic information systems and spatial databases [Sebillo2000,

Aufaure2001], temporal databases [Silva1999] or multimedia databases [Cruz1997b]. Also, supporting a particular kind of more expressive data model such as ER [Wong1982] or object [Murray2000, Lee2001] has been a motivation.

In [Papantonakis1994] three main origins for visual query languages are identified: those arising from ease-of-use or human factors studies, those that visualise some mathematical model or concept and those that are based on the visualisation of a textual language.

Many authors present in their papers a new kind of a visual appearance and then proceed to describe the semantics of the new visual language using their preferred formalism. A common choice is to map the visual constructs into, for example, SQL (Structured Query Language), OQL (Object Query Language) or some form of logic programming (such as [Angelaccio1990, Cruz1992, Lee2001, Balkir2002]). In other words, the semantics are defined operationally using the chosen lower level language. Mapping into existing query languages is useful, for example, because the work can then be concentrated on the graphical details and new features of the query language instead of implementing a whole query system from scratch. That is, if it is possible to map the semantics of the new visual query language to the available system in the first place, which is true for most current systems.

Over the years, the query languages have followed the trends in user interface development. In the early years the computer hardware itself limited the visions. There were few computers, the computers and their operating systems were not built for interactive use and the input and output capabilities were limited. That has not prevented good work (*e.g.*, [Wong1982]).

Since the invention of the now ubiquitous WIMP (Windows, Icons, Menus, Pointers) metaphor and the development of windowing toolkits, it has become significantly easier to develop programs with graphics such as visual query systems, although, it seems it is still difficult. It seems difficult to finish a usable system, let alone break the mould and do something ground-breaking while at it. The objectives of the various research projects have not been tools for real use either. The difficulty related to toolkits is that the task of specifying necessary user interface detail is too great. It takes too much work to implement all the desirable features. Some advances must be made in the toolkits and perhaps there is need to rethink the way we construct user interfaces.

There exist some common trends in the latest papers:

- **Experimentation with the third dimension.** It is an area with potential. The capabilities of personal computers have grown significantly due to powerful modern graphics accelerators. To be able to interactively display thousands or even millions of objects is going to help in visualising large databases. Also, the third spatial dimension enables the packing of this information on the screen, while still remaining uncluttered and understandable. Virtual reality applications are also possible. Visualisation of the query results in three dimensions will be useful for some kinds of data. [Rapley1994, Boyle1996, Murray2000]
- **Usability studies.** The heightened awareness, in general, of the importance of usability has brought usability evaluation and testing into the world of visual query system research. [Catarci1995b, Catarci1995a, Badre1996, Chan1997, Catarci2000, Murray2000]
- **XML and the Internet.** The popularisation of the Internet in the late 1990s has brought a computer to many homes and offices. Now every user has a wealth of information at his fingertips but it is hard to search it effectively. Is keyword-based text search the best we can do or can we start adding and using meta-data? Also, XML (eXtensible Markup Language) has gained in popularity and some of the recent query systems are related to querying it. [Erwig2003, Oliboni2002, XML]

In [Chan1997] it is reported that the trend of proposing new visual query languages will likely continue, as there is no generally accepted language, and suggests to concentrate on

two areas: supporting more advanced modelling concepts and innovative ways to specify complex conditions. There is also urgent need for validating all kinds of results empirically.

In [Catarci1997] a detailed list of open issues in visual query systems is presented:

1. Further improvement of the available visual representations by using the third dimension, animation and virtual reality features.
2. Extension of available data types, like natural language texts, geographic maps, images and sounds.
3. Generation of a user model that formally describes the user's interests and skills and that should be dynamically maintained by the system according to the analysis of the story or interactions.
4. Definition of appropriate metrics and methodologies to evaluate the system usability.
5. More experiments of the system, both prototypes and final version, with real users validating it and evaluating its usability.
6. Provision of formal semantics.
7. Enrichment of the expressive power towards more powerful classes in Chandra's hierarchy (see Subsection 3.3.2).
8. Definition of a different query hierarchy based on the concept of the "usefulness" of the various queries for users; it could then be compared against Chandra's hierarchy to find the most significant intersections.
9. Use of mechanisms for inferring implicit knowledge from stored facts.
10. Handling of indefinite knowledge, metaknowledge, multiple agents, approximate questioning (where queries are allowed that do not specify exactly the desired result), *etc.*

This work represents the experimentation of the utility of a visual query system in a real problem case, although a thorough usability evaluation is out of scope for now.

3.2 Languages and environments

There are many visual query systems and many of their differences are superficial, like what symbol is used where, however, there are usually some new ideas or variations in each system. Next, there are detailed descriptions of a few interesting query systems and brief descriptions of several others. The systems that are described more thoroughly have influenced this work most.

3.2.1 Draw an Object-Oriented Database Language (DOODLE)

DOODLE [Cruz1992, Cruz1997a, Cruz2001] is a visual query language used in the Delaunay Visualization System and it is interesting for many reasons. Unlike other systems, the user can effectively build his own visualisations for the database and is not limited to a set of predefined visualisations. Another distinctive feature is the use of a constraint solver to resolve the layout.

The roots of the system lie in F-logic, which is capable of encoding the most common object-oriented features. A query is a mapping from, for example, the objects of the database to graphical objects (*i.e.*, a visualisation), objects of the database to other data objects (*i.e.*, a classic query), from graphical objects to graphical objects (*i.e.*, visual transformation), or from graphical objects to data objects (*i.e.*, storing the visual information).

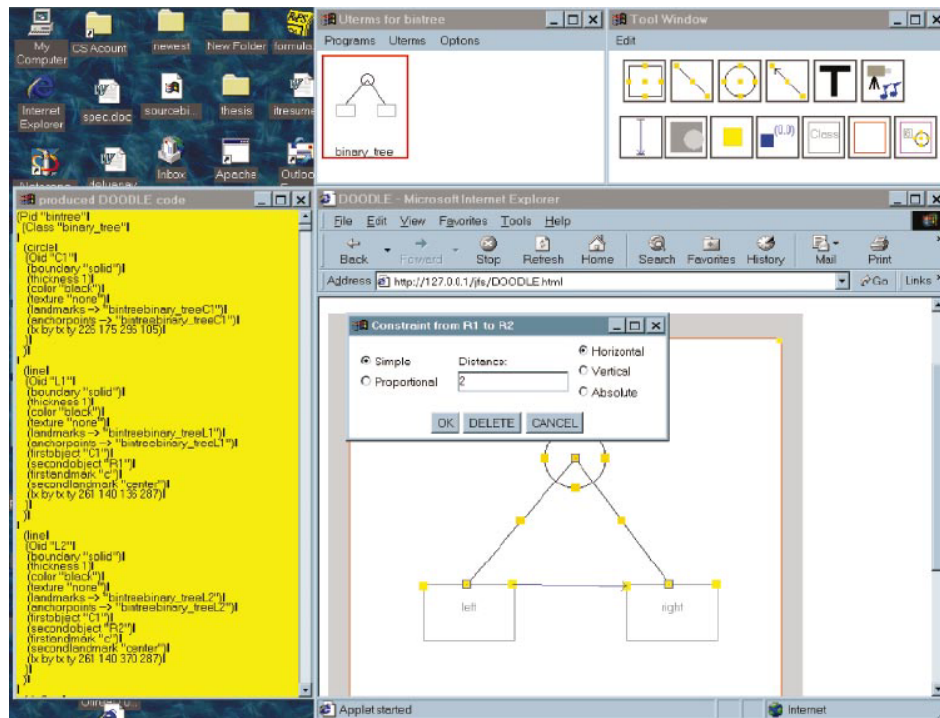


Figure 3.1: A user defining a visualisation for binary trees in DOODLE. Using the tools from top right, the user defines a set of terms. The user is defining a new term in the sketch window and adding a constraint that the distance between the left and right subtrees is two units. On the left, there is the DOODLE code for this visualisation.

To define a visualisation, the user combines three kinds of symbols: prototypical symbols, constraints and key symbols. The querying is essentially defining a visualisation for the data using the symbols. The tool window with the symbols and the sketch window, where new terms are defined, are depicted in Figure 3.1.

Prototypical symbols are box, text, circle, straight line, arrow and double arrow. Each has predefined landmarks that can be used as reference points for constraints such as the center (all objects), midnorth, mideast, midsouth and midwest (polygons and circles), the head and tail (lines and arrows) and arrowhead and arrowtail (arrows). It is also possible to add anchorpoints anywhere on the border of a graphical symbol.

Constraint symbols specify spatial relationships between the graphical symbols. There are two types of constraints: length and overlap. Length is used to specify that the two landmarks or anchorpoints should be a set distance apart. It can be the horizontal, vertical or absolute distance. The overlap constraint specifies that one symbol should be shown on top of another.

Key symbols are the defbox, refbox and origin symbol³. Refboxes are used to reference the visual representation of some object that matches in a rule that is being defined. Defboxes are used to define which part of the object refboxes will refer to. The refbox will be replaced by the actual representation of the object, the part inside a defbox, when the rule is used by the system. There is also the origin symbol which shows where the coordinate origin should be.

In Figure 3.1 the user has used the circle to denote a binary tree node and lines to denote links to subtrees. The user is adding a constraint to set the distance between the two subtrees. There are two refboxes that refer to the subtrees of the node.

³There is some variation of features in the papers. [Cruz1992, Cruz1997a] also describe the grouping box.

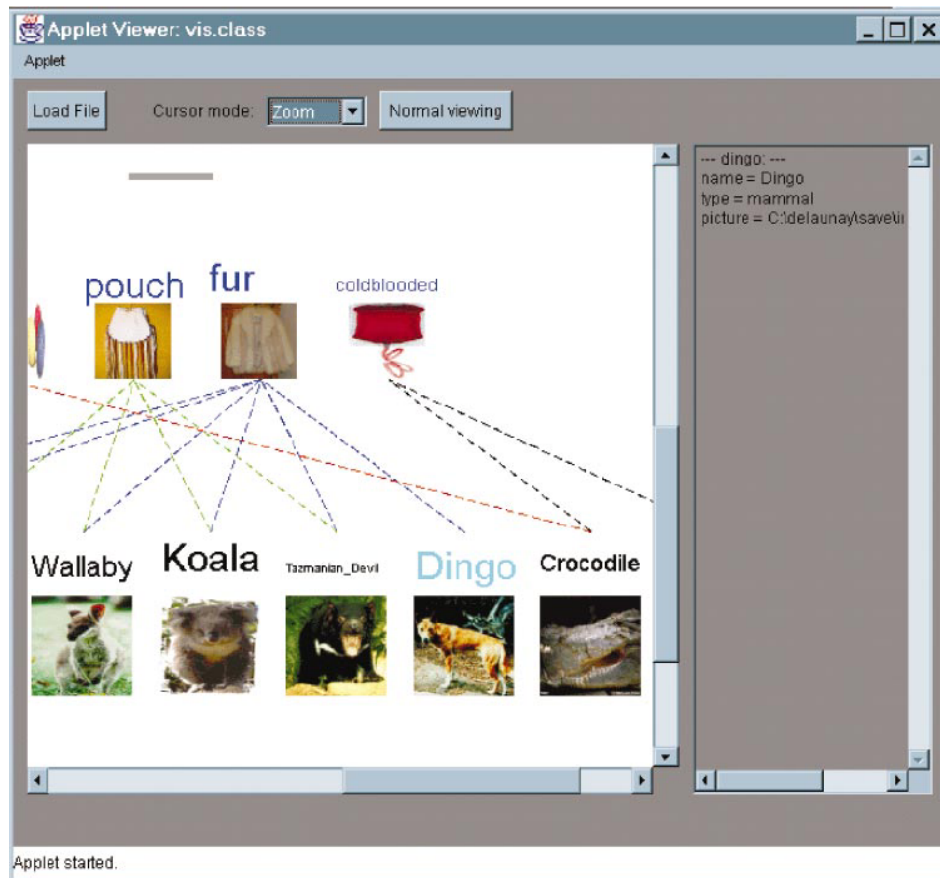


Figure 3.2: The visualisation of a classification of Australian animals in DOODLE. On the left, there is the output pad. The properties of the selected object are displayed on the right.

Once the visualisation has been defined, a constraint solver will be used to solve the layout (*i.e.*, the system of symbols and constraints). The system instantiates enough objects from the database and layouts them, according to the results of the solver. Constraint solving is not trivial, and to overcome problems with an existing solver, a new efficient solver was made. There are still problems with the solver, if the system is over- or under-constrained. The results are put on an interactive output pad which can be panned and zoomed, fish-eye views can be applied and details of the objects can be inspected. The result visualisation is static and cannot be modified. Figure 3.2 shows a result visualisation.

Specifying all the details of the layout at the level of elementary constraints can be a burden to the user. Future work will include usability improvements like providing a library of basic charts and making the visualisations reusable.

3.2.2 Gql

Gql [Papantonakis1994] is a graphical declarative query language based on the functional data model. Its aim is to enable users, with no programming experience, to make non-trivial *ad hoc* queries. The philosophy behind it is summarised into five points:

1. Design with end-users in mind. The language should be easy to learn, use, understand and remember, at least for relatively simple use.
2. Simple queries should be expressed simply. It must also be possible to express complex queries and expressivity as well as ease-of-use should scale up naturally.

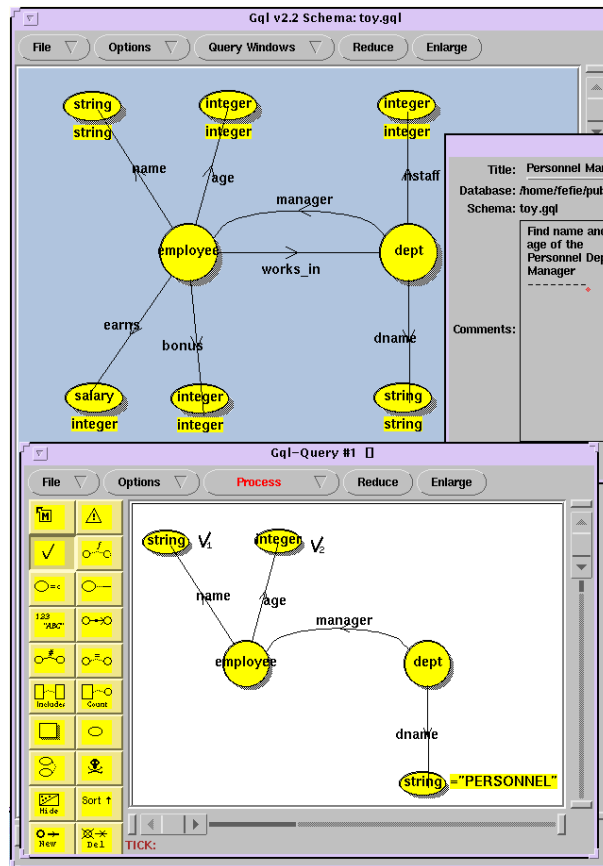


Figure 3.3: The user is defining a query in Gql. The schema is shown in the window above and the query is below.

3. Every query should be representable fully and unambiguously by a single diagram.
4. The definition of the language should be fully separated from any user interaction procedures.
5. The syntax and semantics of the language should be defined formally.

Gql uses the binary relational model which consists of entities and binary relations between them. There are two kinds of entities: abstract or non-lexical entities which correspond to real world concepts (*e.g.*, student, department, course) and lexical entities which have a definite representation and stand for themselves (*e.g.*, number 5, string "Data structures and algorithms"). The relations define two functions that map from one end to the other. The user interface is based on a diagrammatic representation of the entities as nodes and the functions as labelled directed edges. Conditions can be defined to restrict lexical entities. Other constructs are constants, arithmetic operations, boxes, aggregate functions, negation, union, disjunction and universal quantification. Boxes are a way to visualise a collection of entities.

The implementation of Gql (`gql_int`) is a direct manipulation graphical user interface which consists of three types of windows: the schema window, query windows and the output window. The user can pick entities and functions from the schema window and copy them to a query window. Tools are used to elaborate the query by, for instance, adding conditions, specifying entities for output and deleting entities. The diagrams can be translated

and zoomed. The user can specify queries in any order. The query is processed after pressing a button and checked for errors. Some errors are also detected during the formulation of the query. The system can show the generated FDL language code to the user. Figure 3.3 shows the schema and query windows.

3.2.3 Graphical User Interface for Database Exploration (GUIDE)

GUIDE [Wong1982] is a visual query system based on the Entity-Relationship model. As GUIDE represents research done over two decades ago, it is somewhat disheartening to discover that many of the issues discussed by Wong and Kuo are still prevalent. The main issues identified in the paper are:

1. **The user has to remember too many things.** GUIDE provides menus, examples, illustrations and help messages for the user. For example, when entering constraints like having a certain type of student denoted by an integer number, the system can show examples of such constraints, all the possible (integer) type values from the database and a natural language explanation text for each integer.
2. **The data models are semantically poor.** The Entity-Relationship model is used to represent the relationships between entities explicitly, whereas relationships do not exist in relational databases. Extensions, like generalisation relationships, aggregate functions, cyclic queries and dynamic relationship creation (for arbitrary joins), are left for further research.
3. **The user gets no feedback during the query process.** In GUIDE the query is built in piecemeal fashion. The query is formulated by traversing a path in the network made up of the entities. Constraints on the values of attributes in entities or relationships can be added. Preliminary results can be shown to the user while he is building the query so that, for example, he can see the constraints were correct. Colours are used to differentiate separate local (sub-)queries. The textual query so far, in the query language STRAND, is displayed at the bottom.
4. **The schemas lack levels of detail.** The entities and relationships have a rank describing their relevance to a kind of user, who can select which ranks should be displayed, thus removing unnecessary details. Having chosen a central entity to focus on, the user can also limit the radius at which other entities are shown. Figure 3.4 shows the use of the level of detail tools in GUIDE.
5. **There is no meta-data browsing facility.** GUIDE has two kinds of hierarchical directories the user can browse: a subject directory to display and enable the search of all concepts in the database, grouped logically by their subject, and an attribute directory for each entity or relationship with attributes, to show their composition.

Almost all of the ideas from GUIDE are still relevant for any query system. Computers and their hardware have advanced most since the writing of the paper, which allows for more pleasant interaction between the user and the computer. However, the principles described are sound. Also, many more complex data models have since been introduced, but relational databases are the most common.

The formulation of the query consists of four stages: schema definition, schema exploration, query expression and output display. First the database schema is defined and loaded into the system. Hierarchical directories may be built and entities and relationships can be ranked according to their importance. At the schema exploration stage the user can use the directories as well as explore the graphical representation. The representation can be edited to remove irrelevant objects and the importance ranks can be used to filter out uninteresting objects. At the query expression stage the user builds the query by defining a path and adding conditions on the objects. The parts of the path define local queries that together

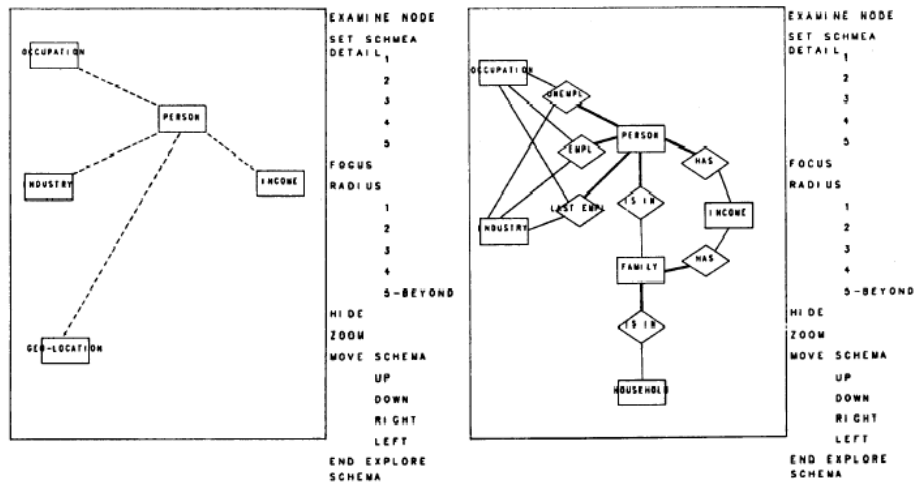


Figure 3.4: GUIDE before and after using the level of detail tools. On the left, there is a picture of the GUIDE interface, where only the highest ranked schema elements are displayed. On the right, the schema detail level has been increased and previously hidden elements are displayed. It is also possible to select an element to focus on and display only the other elements that are near it. Furthermore, single nodes and links can be hidden and the view zoomed and translated.

make up the whole query. It is possible to display partial results, a textual description of the query and examples and explanations of objects on screen.

3.2.4 Query-By-Diagram* (QBD*)

QBD* [Angelaccio1990] is a diagrammatic visual query language for the ER model. The motivation comes from the experiences of usability difficulties. The traditional query languages are not usable by the casual user. Novice users can formulate the queries graphically, simply by navigating in the schema and expert users have many options for complex conditions. In Figure 3.5 is an example of a user navigating through a schema.

The user can access four libraries: **schema library** that contains schemas of different applications, **schema user library** that consists of modified schemas that have been tailored to be useful for certain use, **top-down schema library** representing hierarchical aspects of the schema from which the user can select the appropriate level of detail, and **user query library** that stores queries for reuse. The user can build a temporary view starting with a schema from one of the libraries. By manipulating the schema to bring it closer to the query, the query becomes easier to express.

Navigational operations are used to define a path in the schema. The path contains the elements that the query consists of. It begins with a main concept and can be formed by following existing relationships between connected entities or defining new relationships between previously unconnected entities. For complex queries, several paths can be specified. Conditions can be placed on attributes.

The user manipulates an ER schema using graphical operations. A Translator translates non-recursive queries into relational algebra expressions and recursive queries into programs in a host language. A DBMS Interface translates algebraic expressions into terms of a target DBMS.

QBD* is capable of recursive queries and is relationally complete. Generalisation is included in the model. It has been further extended into QBD** for more expressive power.

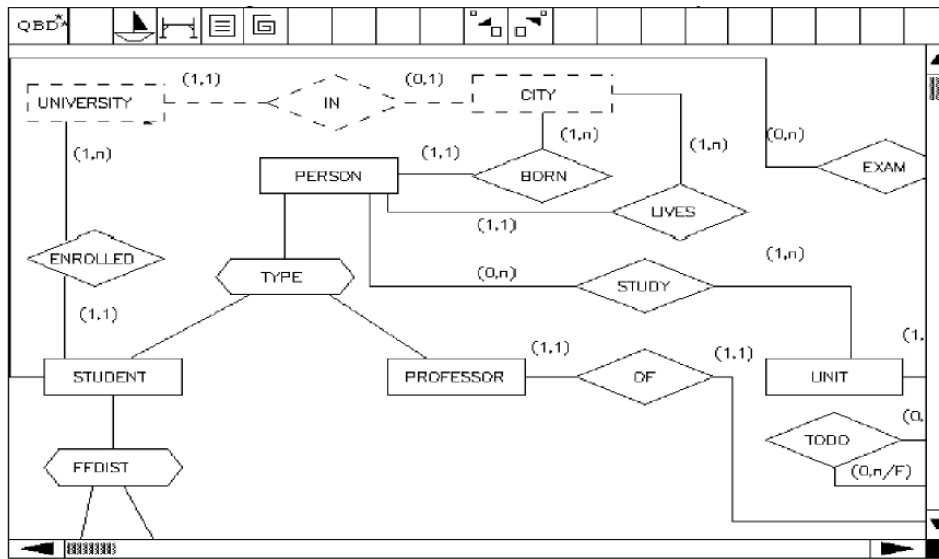


Figure 3.5: A user querying in QBD*. The user has selected a path in the schema containing the entities University and City and the relationship In. The path can also include new relationships built during the path selection.

3.2.5 Others

Kaleidoquery [Murray2000] is a visual query language that uses a filter flow style, conceptually the whole mass of information in the database is filtered to obtain the desired result. The problems identified in existing (non-visual) query languages that are attempted to solve are: the complexity of parentheses in large queries, the conflict of the meaning of logical AND and OR with natural language meanings, having two distinct ways to write queries over relationships depending on whether the relationships are single- or collection-valued and structuring of results not always being separated from the rest of the query. Using a suitable graphical filter flow style conveniently solves these problems. Concepts in the language can include icons in their representation in addition to the textual name of the concept. Kaleidoquery offers most of the capabilities of OQL but with considerations to obtain better usability (for example, said ordering of results).

Spider [Rodgers1997] is a visual language based on a graph rewriting paradigm. It is based on a few simple ideas that make it computationally complete. Complex queries written in Spider may become rather involved and they are possibly not intuitive for people used to regular (procedural) programming languages. Everything is represented graphically but the diagrammatic notation does not allow for the user to define his own appearances. It is possible to view the schema graphically as well as the instances of the database, except that the latter view would often be too cluttered for real databases containing thousands or more instances.

ISIS [Goldman1985] allows users to construct, maintain and query a database using a graphical interface. It uses the Semantic Data Model with inheritance and grouping capabilities. The user can operate on the schema level or at the data level. The schema level has three views: First, the inheritance forest for showing the classes of the system and their attributes, super- and subclass relationships and groupings. Second, the semantic network, showing a class with its attributes and relations to other classes, useful for navigation. Third, the predicate worksheet, for constructing predicates that define groupings and subclasses. The data level shows the instances of a class, makes it possible to navigate to related concepts (*i.e.*, to the class of an attribute) and modify data. Each class is denoted in the user interface by a system provided unique fill pattern that eases recognition.

GraphLog [Consens1990] uses a pattern matching approach to match queries into data, both of which are expressed as graphs. The queries consist of multiple query graphs (similar to subqueries) which are graph patterns. The patterns are matched against nodes and edges in the database. Edges, defined by regular expressions, can match either a single edge or long multiple edge paths between nodes. By restricting the expressive power suitably, efficient evaluation of the result can be guaranteed, but a useful class of queries can still be provided. The prototype implemented can also show the results as a graph that can be manipulated (*e.g.*, filtered).

G [Cruz1987] is a language proposed not to replace general purpose query languages but to complement in the formulation of recursive queries through a graphical user interface. The data is thought to form a graph, which is not transformed into, for instance, SQL for querying, but graph algorithms are used to compute the result from the query. Regular expressions can be used to match graph edges. Again, like GraphLog, the expressive power is limited to guarantee the efficient evaluation of recursive queries.

Pasta-3 [Kuntz1989] is a visual query system where the approach has been to use the graphical direct manipulation techniques and experiences from actual use to concentrate on previously neglected aspects, instead of going overboard with formal semantics and design. The contributions are: the use of direct manipulation techniques (query editing through clicking and dragging, *etc.*), the cooperative environment and expressive power. With the cooperative environment, three things are meant: Handy values, having a quick to use menu of values if there are less than 15 values to choose from (arbitrary choice, fits the screen), values can be copied around and finally entered with the keyboard. Automatic path completion, the system will try to connect entities with relationships, where there are unconnected parts, and it will ask for the right choice, if there are many possible paths. Edit-and-Reevaluate loop, as soon as a query has been built and evaluated, results are displayed so the user can see if the query was correct or not. The query can be easily modified and re-evaluated. Complex boolean expressions are hierarchically visualised. Subqueries can be built and Prolog code can be mixed in for very complex queries.

VQL [Vadaparty1993] is based on a unifying vision of visual query systems. It introduces a declarative visual language that can handle relational, nested and object-oriented models with using a provably minimal set of constructs including the restricted universal quantification. Many constructs of other languages where their nature and implications are unclear, can be simulated with the quantification such as negation, grouping, minimum and maximum. It has formal semantics. The resulting language is semigraphical and the implementation, which is said to be underway, is not described.

3.3 Evaluation criteria

Next, I present ways to evaluate visual query systems and languages. In [Vadaparty1993] it is noted that there are over fifty visual query systems. I begin with existing classifications to provide criteria, which can be used while looking for a system that might fill the requirements. Then I also look at the existing systems from formal, visual and many miscellaneous perspectives to find some differences between the existing systems that would tell me which of them are more suitable than others.

3.3.1 Classifications

Classification by database and query representation [Catarci1997]

In [Catarci1997] visual query systems are surveyed from the point of view of the user. They divide existing visual query systems into four groups based on the database and query representation: form-based, iconic, diagrammatic and hybrid.

Course	Person	Laboratory
courseID	personID = P	laboratoryID
code	name = ?	name
name		
Department	Student	Study Rights
departmentID = D	personID = P	departmentID = D
name = ?	studentID = S	studentID = S
	student number = ?	approval date
	entry year	
Degree Programme		
degreeProgrammeID		
name		

Figure 3.6: Example of what a form-based system can look like. Find the names, student numbers and departments of students.

Form-based denotes systems that consist mainly of text. They are a version of paper forms, so to say, where blanks are filled with desired values to formulate a query. Another possibility is to have table like and spreadsheet like structures and, generally, to layout the text to present visual clues. This level can be described as semivisual.

An example of a form-based system is in Figure 3.6. The user fills in values for some of the table attributes. For the attributes he is interested in he enters a question mark. For attributes where there must be a certain value there is a variable or a definite value. Joins are made by ensuring two attributes have the same variable as value. Forms are present in systems like Pasta-3 and VQL.

An **iconic** visual query system uses icons to display concepts. A query is formulated by manipulating the icons. An icon (*e.g.*, a currency symbol, a cross or an image of a printer) may very well have some natural intrinsic meaning (*e.g.*, money, religion, printing) and it is exploited by these kind of iconic systems. Another possibility for an icon is that it may be a picture of the concept represented (*e.g.*, picture of a specific device or a person). Icons are not without their problems. It is difficult to come up with effective and natural icons for new concepts, icons may have multiple conflicting interpretations in some contexts and they are only useful with particular kinds of concepts.

The Figure 3.7 presents an example of an iconic query. The user selects a part of the schema on the left to the query. Joins are made by making the icons overlap. This is the same query as with the form-based system but no single attributes are considered. QBI [Massari1995] is an iconic query system and Kaleidoquery uses icons in its representations, although not in similar sense.

A **diagrammatic** or diagram-like visual query system uses notations based on graphics that use the position, size, colour and shape to display the properties of objects and, usually, lines or arrows between the objects to denote relationships.

The Figure 3.8 has an example of a diagrammatic query. The database schema is displayed as a diagram. The query is done by selecting the part of the schema that is in the query (shown with dashed lines). Some of the attributes are selected as the output (bold face). The joins can be done by including predefined relationships between the tables. Example systems are the GUIDE, QBD* and Gql.

The fourth and final system, a **hybrid** system uses a combination of features from two or three of the representation types. Kaleidoquery can be described as a hybrid system for combining icons with data-flow diagrams. VISUAL [Balkir2002] uses a combination of

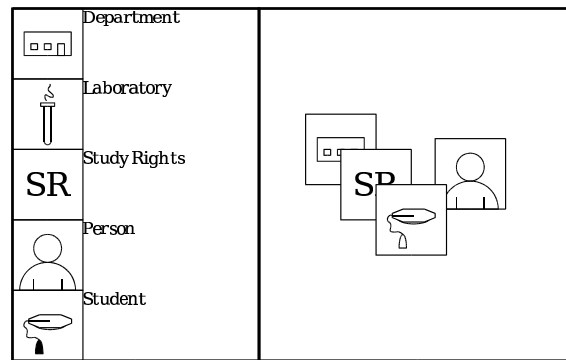


Figure 3.7: Example of what an iconic system can look like. Find the students and the departments.

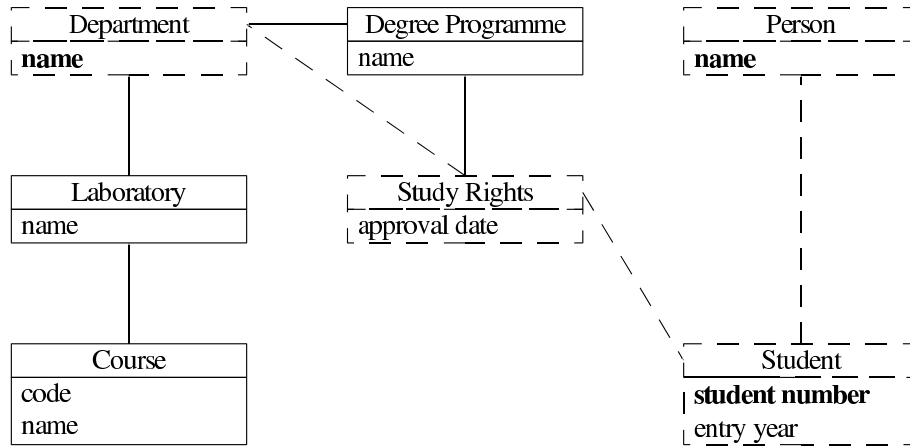


Figure 3.8: Example of what a diagrammatic system can look like. Find the names, student numbers and departments of students.

forms and icons and many other systems incorporate forms with diagrams in some manner.

This classification is used both at the level of database schemas and at the level of the query. Either or both may use these constructs. Furthermore, the same terminology can be applied to the visualisation of the query results. Very few query systems yet provide complex output visualisation (however, DOODLE does).

Classification by understanding strategy [Catarci1997]

Queries are also characterised in [Catarci1997] based on how the user interacts with the system. The different understanding strategies, ways in which the database schema can be understood by the user and ways in which the relevant parts of the query can be found are: top-down, browsing and schema simplification. Examples of the strategies can be found in Figure 3.9.

The **top-down** approach to understand a database means that the user is first presented high level general aspects of the database and then he can interactively open or view more detailed levels of the hierarchy or zoom in to see the details. GUIDE and QBD* have top-down interaction.

Browsing means that the user begins with one interesting concept - acquired by some

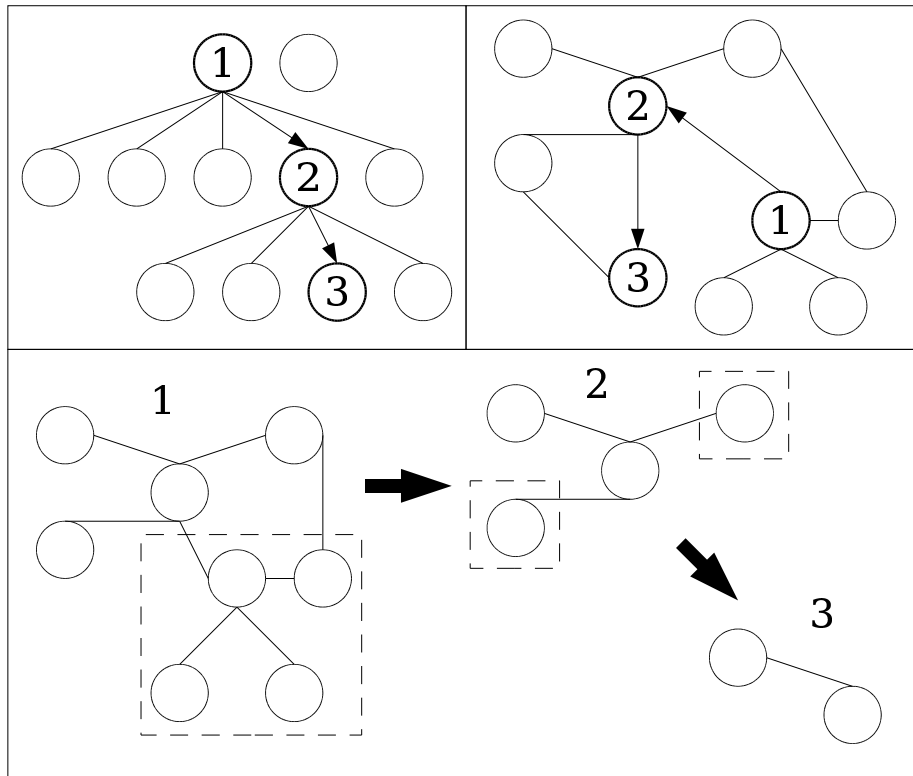


Figure 3.9: Understanding strategies for visual query systems. Top left, there is the top-down strategy. The user is first presented with only a few high level concepts. Then, he can choose to open more and more levels of details. Top right, there is the browsing strategy. The user begins with one concept and moves to an interesting neighbouring concept. The third strategy at the bottom is schema simplification. The user is first presented with the whole schema and he hides the irrelevant parts (dashed line). Finally, only the important parts of the schema remain.

means - and then proceeds to concepts that are connected to or just near of the concept currently focused. The focus of the user therefore moves based on local decisions made by the user. Some balance must be made between displaying only immediate concepts and displaying all concepts. If the horizon is too close, the result is pretty much analogous to trying to find your way while looking at your feet only. If too many concepts are displayed at the same time, it becomes difficult for the user to find the right concept. The amount of information might also confuse the user. Some systems allow browsing in the schema (*i.e.*, intension), some in the data (*i.e.*, extension) and some in both of them possibly mixing the modes. Examples of the use of browsing are QBD* and SUPER [Dennebouy1995].

In **schema simplification** the idea is to manipulate an initial schema by combining and transforming it so that it is a better fit for the intended query. Unnecessary concepts can be removed, concepts can be combined and new relationships can be formed. The result is a simplified schema that makes it easier to make the query itself. Example systems are yet again QBD* and SUPER as well as VISIONARY [Benzi1999].

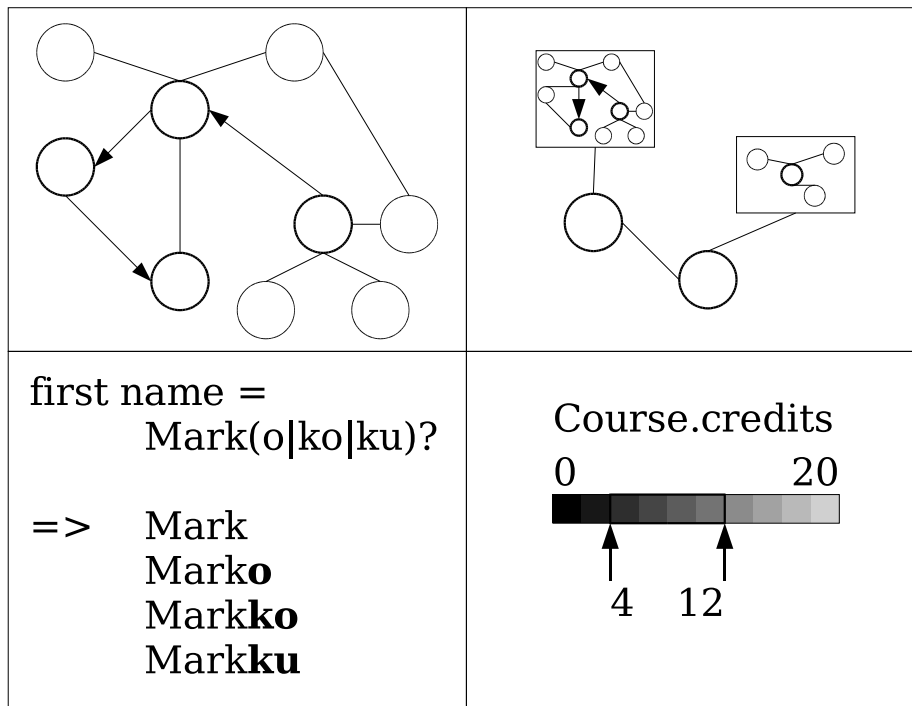


Figure 3.10: Query formulation strategies for visual query systems. Top left, there is the schema navigation strategy. The user begins with one concept and proceeds to neighbouring concepts defining a path that is the query. Top right, there is the subqueries strategy. The user combines two smaller subqueries with some new concepts to form his query. The third strategy at bottom left is matching. The user has entered a pattern to specify that he wants people with a particular first name. The patterns can also be graph-patterns matched against the graph of all the database instances. The fourth strategy at bottom right is range selection. The user specifies with the two sliders that the courses he is interested in give 4-12 credits on completion.

Classification by query formulation [Catarci1997]

Query formulation also varies between systems. The different strategies are: schema navigation, subqueries, matching and range selection. Figure 3.10 explains the strategies visually.

Schema navigation means specifying a path in the schema that indicates the desired concepts and joins between them. Additional constraints may be put along the path. The path can be an arbitrary connected path, a connected hierarchical path or an unconnected path (*i.e.*, create new paths between unconnected concepts by specifying the join conditions). This is done in, for example, QBD*, SUPER, Gql, Pasta-3 and ISIS.

In the case of **subqueries**, the query is formulated by using smaller queries. Composition of concepts means that the output concepts are produced by composing input concepts (*e.g.*, overlapping two icons to join them). Using stored queries means using previously stored queries or queries provided by the system in a library to form new queries. Subqueries can be used, for instance, in Pasta-3 for composition and in GUIDE for storing queries.

Matching can be done using examples and the system will generalise the query from them or patterns can be specified to match against the data. Matching is used in, for example, GraphLog and DOODLE.

Finally, **range selection** is choosing the range of values and the system will return all

the concepts that have values in the range. The specification of the ranges can be done using, for instance, dials, sliders or lists. In DEVise the user can select the range by dragging with the mouse on the visualisation.

Classification of Papantonakis and King [Papantonakis1994]

Visual query systems are classified in [Papantonakis1994] by six criteria:

1. **Data model.** Popular data models for visual query systems are the relational, Entity-Relationship, semantic, *etc.*, and various extensions.
2. **Targeted users.** The users may be novices, experts in computer science, experts in query languages, domain experts and this affects the design of the query system.
3. **Visual programming or visual interaction.** The system can be meant for general visual programming or it can use a visual language in a particular form of visual interaction.
4. **The degree of visual expressivity.** Everything does not have to be a visual element.
5. **Expressive power.** Some systems allow more complex queries than others.
6. **Design origin.** The system can be designed for ease-of-use, to visualise a mathematical model or concept or visualise a specific textual language.

Classification of Dennebouy *et al.* [Dennebouy1995]

In [Dennebouy1995] it is asserted that visual query systems are of three types: browsers, assertional query editors and procedural query editors.

Browsers are for finding and displaying the contents of the database using simple interaction mechanisms. For example, using a schema diagram acquiring information about the network of classes and then, through clicking one, getting information about the actual objects stored in a table. Pasta-3 is described as a browser.

Assertional query editors are such that statements (*i.e.*, assertions) are made of the desired result and the system formulates the query based on them. These editors can use syntactic editing, which is using visual aids for forming textual queries. Likewise, they can use formulation by example, which is deducing the required query from a set of examples. Assertional query editors can also be graphical editors, which present the database schema as a diagram and use menus, point-and-click or other kinds of interaction techniques for the formulation. For example, QBD* and Gql are members of this class.

Procedural query editors are such that queries are, for example, expressed by designating elements in a diagram and choosing operators from a palette or a menu. It is effectively about forming a procedural program using the editor that satisfies the query needs of the user. None of the systems in this work are procedural query editors.

Discussion

In general, it would be useful to have even more specific classifications, for selecting good visual query systems for a particular use. For instance, saying that a system is diagrammatic, does not say very much of its utility or quality. The classification in [Chan1997] does go into specifics. However, another problem is that many features are such that all commercial quality systems ought to have them - but there are no good systems available.

The systems that seem most suitable for the student register and that are most interesting, are of the hybrid type. It seems best to use simple lists and tables where they are appropriate, especially in the output, but support a diagrammatic representation of at least the data model and query, to take advantage of visualisations. Diagrams are also needed in

the output to effectively display course dependencies and similar graphs. Icons can be used to provide more visual clues to the user. However, it seems hard to come up with a usable set of icons for all the relevant concepts of the student register.

The advantage of assertional editors compared to procedural editors is that the algorithm for obtaining the result need not be specified. Browsers are generally for simpler needs. In the student register, the diagrammatic representation of the schema should satisfy the browsing needs of the users. It is not yet clear, if the declarative nature is an advantage to the users of the student register, but it is a good start.

3.3.2 Formal differences

For the second approach of narrowing down and selecting a suitable query system, I have chosen to examine formal differences in the systems. Like the well known Chomsky language hierarchy, in [Chandra1988] a hierarchy for meaningful query classes is proposed, which is expanded in [Catarci1997]. However, comparing the expressive power of query languages in the formal sense is not very relevant for this work. The most important aspects of the expressive power, in this case, are actually the implications that the expressive power has to the overall system usability and that it matches the power of SQL, as used in the student register.

One consideration that has been a research topic in many papers is how can the query result be computed efficiently. Limiting the expressive power allows more efficient algorithms. This topic has been handled in [Cruz1987] and [Consens1990] in the context of recursive queries, in [Cruz2001] as a constraint satisfaction problem, and in [Vadaparty1993] regarding the semantic and implementation advantages of the strict universal quantification.

The chosen data model also affects the query system. The more expressive the model, the more naturally the interesting application domain concepts can be represented. For example, if the system is capable of inheritance, generalisation type relationships become most natural to represent in the system. Without being available, the modelling of the domain must be done using less suitable constructs, where important issues may be hidden since they are not explicitly mentioned. This is noted in [Wong1982]. On the other hand, having few powerful constructs that expert-users can use to define the more specific constructs for other users, can also be an advantage. This is discussed in [Vadaparty1993].

Discussion

For novice users, it is obviously easier to grasp a simple system with few limited constructs than a system with many possibilities and a wide range of constructs. On the other hand, frequent users will require a different kind of usability, where convenience and capabilities matter most. Finding a good compromise that caters to the various kinds of users is important.

Specifying the language formally is seldom useful to the users. Only system implementers or computer science experts can hope to understand the specifications and only implementers are required to do so. But hopefully the implementers use a clean underlying formalism and implement a logically working, usable system.

Most interesting, both sides, those favouring practicality and those who favour a formal approach, claim their point of view has been neglected (*e.g.*, [Kuntz1989, Vadaparty1993]). While both approaches are obviously needed, it is my opinion that more interest could be raised by a more practical approach. Some complete query systems need to be put on-the-shelf, so to speak, to generate interest and new perspectives.

To be usable in a wide variety of situations and for as many query types as possible, the system should offer some level of Turing-completeness to the user. Then the user can adapt the system to all kinds of queries, different needs in visualisation and different forms of data access. However, it is debatable whether this flexibility should be in the visual programming capabilities of the system or in the system architecture in the form of well-defined extension

Table 3.1: Visual elements of query languages

forms	particularly laid-out, structured text [Wong1982, Kuntz1989] lists, one-dimensional tables [Goldman1985, Kuntz1989] two-dimensional tables, spreadsheets, hierarchical boxes [Vadaparty1993, Balkir2002]
diagrams	simple graphical elements such as boxes, circles and polygons [Wong1982, Goldman1985, Angelaccio1990] lines or arrows connecting elements [Wong1982, Goldman1985, Angelaccio1990]
icons	images, photographs [Cruz1997b, Cruz2001] non-photorealistic symbolic pictures with interpreted meaning [Benzi1999, Murray2000]

points. There are many proficient programmers and well-known techniques for producing programs with regular textual programming languages so using one for extending a visual system has advantages before there is a stable visual programming language and experience in its use. The intended usage, as described in Chapter 2, suggests that it is unnecessary to have a Turing-complete visual language right from the start, so this work concentrates on different issues.

As noted in Chapter 2, it is hard to say what the exact requirements for exploratory queries are and there are multiple sources of data so there is need for the generality in the form of expressive languages and generic systems. The example queries from Section 2.2.2 and Table 2.5 form the basis of the measurement of sufficient expressive power.

Most languages discussed in the literature are expressive enough for simple use, it is good that it does not matter which one is chosen, but the creation of new visualisations is limited to only a few systems, for example, DOODLE [Cruz2001]. Other systems may have pluggable modules such as in DEVise [Livny1997]. Plugin-type extensions are usually more of an implementation issue than a theoretical impossibility.

3.3.3 Visual differences

Third way of trying to find suitable query systems is to compare them visually. Not only does the word visual (and graphical) mean different things to different people, but there are several different kinds of visuality as well. In [Catarci1997] there are four basic classes of visual queries: forms, diagrams, icons and hybrids, which use multiple approaches. Some of the common visual elements that are present in the visual languages are outlined in the Table 3.1 along with a few example systems.

There are many differences in visual query languages in what kind of an element is used in a particular context. These are mostly superficial differences where the choice of a particular notation seems to be a question of aesthetics, personal preference or ease of use. Research into usability and human perception is needed to provide solid answers here.

In many cases, the desirable appearance depends on the particular use case. For example, personnel records may contain an image and a name of each person. Sometimes it is useful to query a list of just the names of the persons but sometimes it might be useful to show the images along with the name. There is not just one right representation in the general case so the visual query system must provide means to select the right appearance.

Some systems have the capability of visualising the results of a query (*e.g.*, DOODLE and DEVise). The elements are generally the same as those found in the Table 3.1. Depending on the data there might be various useful types of visualisations like plotting (positioning the objects geometrically, if they contain coordinates), drawing charts, trees or graphs. Many systems settle for simple lists and data exporting capabilities. Then any program that can read the format can be used for processing the data further.

Discussion

In some visual languages the appearance is tightly connected to the underlying formalism (e.g., ER diagrams). In many cases it is impossible for the user to customise the appearance, only the implementer of the visual query system can change it. Customisability is useful to let the users of a visual query system make the user interface reflect their notations when they are working on a particular database. The terms and formalisms used by the originator of the visual language are irrelevant to the users in an ideal case. Thus, having customisations closes the gap between a generic visual query system and a regular application, because the user can modify the appearance and extend the system to fit his notation and does not have to settle for the lowest common denominator. The implementers of regular applications observe the users to find the best outlook and interaction model for best usability for the particular use case they are implementing but a generic query system does not have this advantage. It can be adapted only after it has been taken into use.

Adaptation work can be done, if abstractions can be built on top of the basic data model, hiding implementation details and such. Not all users need to be able to do the adaptation but a domain expert or an expert user should be. Going too far with the adaptation brings about the downsides of a regular application since there is need for an application programmer with programming skill to do the customisation. The difference is what language or tool is used to make the customisation, how much of the original generality is maintained, how powerful and numerous the available tools are and who does the work. There is no single perfect programming language. Most of the advantages should be attainable by merely building a system that is open for extensions that does not use esoteric technology or highly proprietary libraries. The user should be able to build extension modules separate from the system developer and should not need the source code for the system to do that.

I think that application programmers will not program more features than absolutely required so there is no possibility of having as generic and powerful tools with the regular approach to building applications. It will only be possible with a visual query system because somebody else did the hard work, so to say, by implementing the powerful generic tools - once and for all. The adaptation to the problem is then more or less routine work whether it needs programming skill or not.

The ideal system for the student register enables the building of some abstractions for casual users and "conveniences" for the power users. Not as much on the level of how concepts look but on the level of how they are used effectively. Most of the time the users have the ability of using other programs for complex visualisations, so it is not strictly necessary to have output visualisation. The openness of the implementation to extensions is attained by using common technologies.

3.3.4 Miscellaneous differences

There are many kinds of small differences between the existing visual query systems. We have looked at the visual query systems through the classification, formal and visual perspectives. In this section, I will present some miscellaneous differences.

It is obvious that most research systems were never intended for commercial use. The required implementation effort is simply too much for a research project and it is besides the point of the research work. The different levels of implementation seem to be: prototype, test use, real use and commercial use.

First there are prototypes. Their purpose is to quickly test ideas, perhaps only on paper, and they are by far the most common type. If further research is made, or the system is made to solve a particular problem, then the next step is test use. The system is tested in a real situation, or in a mock-up that is close to the real situation, with possibly real users. The results of the use, however, are not used, but the experiences and observations of utility. The third use is real use in which the results from the use of the system become important. The last level of implementation is commercial use. The system is polished, marketed and

sold. I have found mentions of systems that would be possible to buy off-the-shelf but they are only semivisual systems at best. There is a comparison in [Aslandogan1995].

In [Catarci1995b] QBD* and SQL are compared and visual query systems are not seen to be markedly better for expert users, however, they provide a more satisfying experience. They are easier to understand and help to avoid many usual (syntactic) errors. In [Badre1996] QBD* and QBI are compared and the comparison shows that experts perform better using QBD*, while non-experts are slightly better using QBI. However, the query type makes a larger difference than the language choice and some queries suite one language markedly better than the other. Queries containing paths were performed better using QBD* and queries containing cycles using QBI. There are further results from usability testing in [Catarci1997, Catarci2000].

Discussion

Real scenarios often have interesting practical considerations like a huge number of concepts compared to a test database. Although some solutions have been found, they are not in use in every system. Also systems with lots of temporal data (*e.g.*, birthdays, timestamps) or spatial data (*e.g.*, locations of mobile phones) are in some situations different from alphanumeric query systems. If these features and different types of data are important, more effort should be spent on examining these kinds of systems. As noted in Chapter 2, support for exploration is important. I have evaluated the elements of the support for exploration of various visual query systems and included this aspect into the Comparison.

3.4 Comparison

In this section, promising candidates of existing visual query systems are compared while trying to find systems to suit the requirements from Chapter 2. The selected systems have mostly appeared in the previous sections of this chapter. Additional systems were added for a more thorough comparison. The emphasis is on diagrammatic and hybrid solutions as they were seen most useful based on the classifications. As for the other requirements, formal differences were less important for the needs of the student register but do limit the simplest systems out. More diagrammatic and otherwise graphical systems have been included than iconic or form-based to concentrate on more visual systems. Finally, some systems have been included for comparison purposes (*e.g.*, PSQL⁴).

The criteria used in this work to compare the visual query systems are shown in Table 3.2. Each criterion has three levels denoted by stars. It is also possible that a system has no stars, if it does not support something or the support is unknown. This is denoted by a dash (-).

As we can see in the Table 3.3, the form-based systems as well as most systems in general do not have output visualisation and for this work a more visual system was desired. QBI as the sole icon-based system does not fulfil the requirements of the student register as it is not as suitable for complex queries as its competitors. There are no suitable icons for the concepts of the student register either. One good feature that would be useful for any system is that the queries (and conditions) have a generated natural language description. Although, in the case of a complex query, the generation of a meaningful, complete and minimal description is hard. Many of the diagrammatic systems would be usable in some form for this work, DOODLE and Spider are the most promising from the visualisation point of view. None of the systems offer user definable components (*i.e.*, extending the user interface) so no system is perfect from the implementation generality aspect. Also, none of the systems have powerful abstraction mechanisms for making very complex queries. QBD* has a lot of schema modification features that offer abstraction but it is missing the possibility of a user adding new operations. No system gets three stars for explora-

⁴PSQL is a command-line SQL query system in the PostgreSQL system.

Table 3.2: The grading used in the comparison

Criterion	*	**	***
Query visualisation	textual constructs	structured text and graphical schema	visual queries with visual operators
Output visualisation	lists, tables	simple charts, plots	graphs
Level of use	prototyping	test use	real use
Implementation generality	hard-coded schema	applicable to any schema and database	user-defined components
Customisability	hard-coded queries	custom queries	custom visualisations
Expressiveness	only limited constructs available	powerful constructs	high level constructs, good abstraction facilities
Support for exploration	a list of concepts to choose from	graphical display of concepts, modification of query	fully interactive output visualisation

tion support because the output visualisations are not completely interactive like the query visualisation often is.

It would be useful to try to compare languages and implementations separately because some languages undoubtedly suffer from lack of attention to implementation even though the language itself is free of problems. On the other hand, when comparing to other systems, a particularly well implemented system with attention paid to usability may misrepresent some qualities of the language by downplaying obvious deficiencies by having many usability aids. In [Murray2000] this problem is noted as a defence for paper and pencil tests. For practical use the implementation matters. Nevertheless, since none of the compared query systems are commercial products, they all have some deficiencies and missing features that would be fixed, if they were completely finished.

3.5 Conclusion

Visual query systems have a long history starting from the mid-70s. Their development has not reached the end yet. There are many existing systems but they are mainly of prototype level of implementation. Therefore, they are missing certain desirable features. It is fairly straightforward to implement a visual query system to the prototype level, as is the case with most systems. However, adding the missing desirable features and polishing to release quality demands much greater effort and is usually impossible for pure research projects. Most of the existing system have only been tested with small example data models instead of real production quality and complexity models. However, by using ideas from several systems it should be possible to effectively handle large models.

As a summary, the existing systems do not satisfy all of the requirements mainly because of implementation issues and lack of testing with real-sized scenarios. The developed prototype that is described in the following chapters fulfils two main goals. It is a generic enough implementation meant for public use. There are very few visual query systems publicly available even for testing let alone for use. The second goal is to test new ideas in the field of visual query systems. Thus, the prototype is a platform for research and development but also a tool for real use.

Table 3.3: The compared systems

System	Language	Year	Query visualisation	Output visualisation	Level of use	Implementation generality	Customisability	Expressiveness	Support for exploration
Form									
Pasta-3		1989	**	*	**	**	**	** ^a	**
QBE		1975	*	*	**	**	**	*	*
Icon									
[Massari1995]	QBI	1995	**	^b	*	**	**	*	**
Diagram									
Delaunay	DOODLE	2001	** ^c	***	**	**	***	**	**
gql_int	Gql	1994	**	*	** ^d	**	**	**	**
[Consens1990]	GraphLog	1990	**	**	*	**	**	**	**
[Angelaccio1990]	QBD*	1990	**	*	**	**	**	**	**
[Cruz1987]	G	1987	**	*	*	**	**	**	-
ISIS		1985	**	*	*	**	**	**	**
GUIDE		1982	**	*	**	**	**	**	**
Hybrid/Other									
PSQL	SQL	2003	-	*	*	**	**	**	-
[Balkir2002]	VISUAL	2002	*	*	-	**	**	**	*
Kaleidoscape	Kaleidoquery	2000	***	-	*	**	**	**	**
DEVise		1997	** ^e	***	***	**	**	*	**
[Rodgers1997]	Spider	1997	***	***	*	**	**	**	**

^aThere is a possibility to mix in Prolog code to the queries. This was not yet implemented in the system.

^bThe implementation is not discussed in detail in the paper.

^cMaking a query is making a visualisation for it. Afterwards, more details can be acquired by clicking on the concepts, which produces an inspector like view.

^dUse is experimental but the system is robust enough for real use.

^eQuerying is done by clicking on interesting concepts and selecting ranges of concepts in the visualisation .

It is out of the scope of this work to implement all useful features and I will concentrate on those that enable research of the student register. The use of existing systems would enable some research of the applicability of visual query systems to the problem case but it is in my interests to also produce a new implementation that can be used to try ideas not yet covered by existing implementations and previous work.

Chapter 4

Prototype

A prototype visual query system named SEEQ (System for Enhanced Exploration and Querying) was implemented to examine new ideas as well as to try a visual query system firsthand. Another program was implemented to insert a data dump of the old student information system into a database and prepare a database schema, in the form of a data source *specification file*, for the prototype to load. In the next section, there is an example interaction session with SEEQ.

4.1 Example case

This section contains an example use case. The user is trying to figure out what courses a student has taken. First, he must understand the content of the database by exploring the data model. After deciding the relevant part of the model, he formulates the query. When the query is done, he explores the result.

Figure 4.1 shows the prototype after loading the student register specification from file along with some explanations. On the left there is a *canvas* where the database schema or *model* is displayed and on the right we have a currently empty canvas that is used during querying. The canvases function as infinite workspaces the user can use to manipulate graphical objects. The objects displayed are also called *representations*, since they represent the underlying model, query and result objects graphically in the user interface. A particular feature that is shown in Figure 4.1 is the *silhouette* drawing. When working with a practically infinite canvas, objects may be outside of view, when the view is transformed (*i.e.*, moved or zoomed). The silhouettes of such objects are drawn on the edges of the visible area to remind the user of their existence.

The model consists of *classes*, *attributes* and *associations*. Classes and attributes are mapped directly to table and column names in the physical database. However, they may be given aliases for display. In Figure 4.1 we can see that all the internal Finnish names of the database have been replaced by their English translations. Associations can be introduced to specify relations between table attributes, even though they do not actually exist in the physical model. The associations are shown as arrows with a label. The benefit of associations is to clearly show the user where attributes refer to other tables and, therefore, table joins are natural. Arbitrary joins can still be done, though not as conveniently. Another type of an approach is to use a more powerful data model as discussed in [Wong1982], but the available student information data is made for a relational database, and thus relational model is the most straightforward choice for a data model.

An unavoidable consequence of trying to solve problems in a real database is that the model canvas gets rather cluttered as is seen in Figure 4.1. Thus, in Figure 4.2, the user has used the *filter buttons* above the model canvas to filter out certain parts of the model, that is, those parts that are not used in his current use case. The domain and model specific

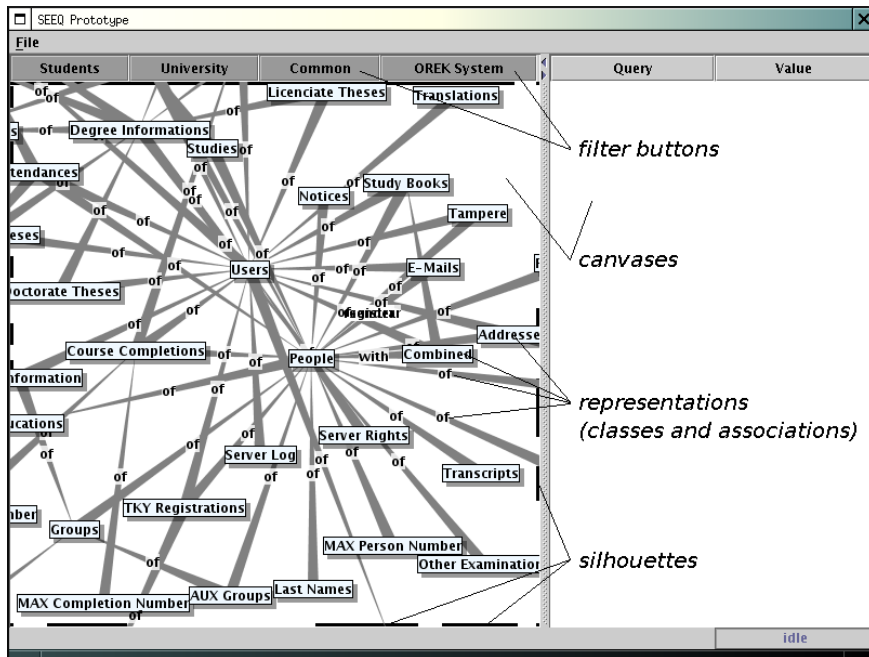


Figure 4.1: Situation after loading the student register specification. On the left, there is a canvas where the database schema is displayed. On the right, there is a currently empty canvas that can be used during querying. The user can translate and zoom the canvases. When representations go out of view their silhouette is displayed in the border. The filters can be used to hide uninteresting representations.

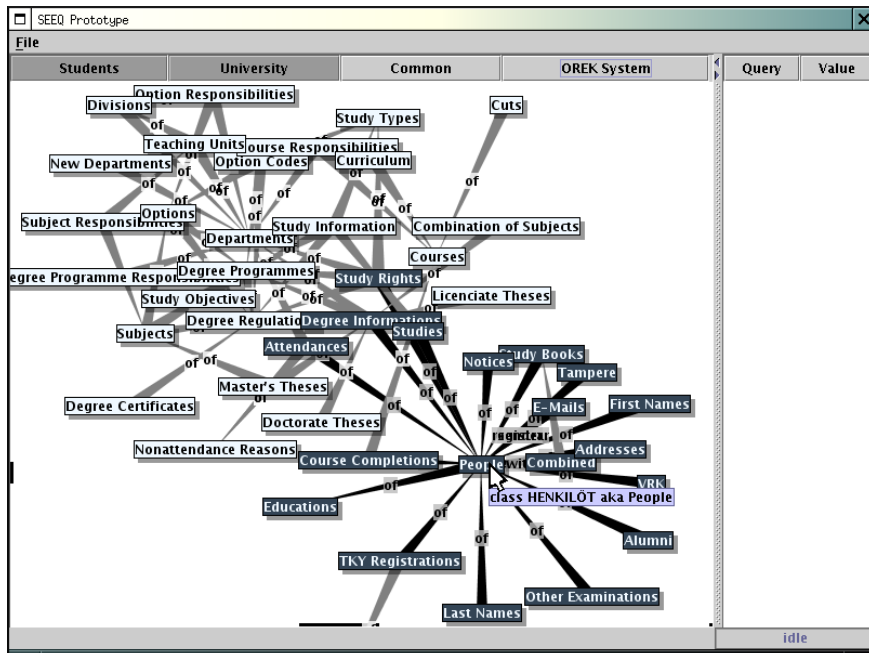


Figure 4.2: The user has applied two filters and is examining concepts related to People. The neighbourhood of the class People is shown highlighted.

filters are specified along with the model in the specification file. Different kinds of users can have different kinds of filters and some filters of the same user can show the model in different levels of detail. Any user can add his own filters although it should often be enough if an expert user defines one common set for everybody. Tables that are more or less implementation details can be gathered into one filter, such as the OREK System filter in Figure 4.2, or can be discarded when specifying the model in the first place.

In Figure 4.2 the user is concentrating on student and university information. He has moved the mouse cursor on the People class. The system highlights all directly related classes and their connecting associations, as well as shows a tooltip with information of the People class. The user has also zoomed out and resized the canvases to show most of the objects in view. The prototype implements the basic functionalities of ZUIs (Zoomable User Interface). Text size is not changed for now, only the positions are scaled. The zooming can easily be enhanced to *semantic zooming*, where faraway objects are shown in smaller font and closeup the insides of objects (*e.g.*, attribute list of a class) can be shown automatically.

Figure 4.3 shows how the user, now having decided upon the interesting part, has begun formulating the query. He has created a new *query statement* in the working canvas and dragged the classes that interest him inside it. In this example, the user is trying to show the courses of a student so he has dragged the Courses and People classes to the query. The basic interaction technique used is dragging, which is used for copying and moving objects around the canvases and also connecting objects like operator parameters and their values. The query statement contains two subcanvases of which the left one is for the query and the right one is for the result. The user has also inspected the two classes in the model canvas (by clicking) and their composition in the form of an attribute list is shown. Hiding the attribute lists and similar details inside the objects by default helps to keep the view largely uncluttered. Next the user has selected the classes (shown by the thick rounded blue border) in the query in preparation of connecting them automatically. The user has clicked on the canvas to focus on it (shown by the light blue tint) to send the connection command to that canvas. The automatic connection finds the shortest path between the two selected representations in the model and copies the connecting representations to the canvas the connection operation was performed in. The mouse cursor is hovering over the canvas so the canvas is also highlighted (shown by the blue instead of gray border). Finally, the user has viewed the *values menu* of the gender attribute (of class People) that are found in the database and they are now shown in a separate window. The values menu enables quick access to common significant values without first doing a separate query to review them. Although, if complex conditions are required, a separate query is still necessary.

In Figure 4.4, we can see that the user has minimized the model canvas and filled the freed up space by expanding the query. The automatic connection has introduced the Course Completion class and the associations between it and the Course and People classes. The query has been further refined with attributes name, first names and last name and a condition on the first names attribute. The user has begun entering a further constraint on the last name attribute (by double-clicking it). For swift and effortless entry the user can enter new values and constraints with the keyboard. Double-clicking on the canvas creates new values or operator statements and similarly double-clicking on an attribute creates an operator but the left hand side is already connected to the attribute. The semantic of double-clicking is therefore "enter and connect here".

The name (of class Course) and last names (of class People) attributes are shown differently (with a thin rectangular blue border) to show that they are the attributes the user is interested in and are included in the query results. The attribute first names is present in the query as it is part of a constraint but the user is not interested in its value. By default, all attributes are included in the results but they can be removed by clicking that attribute.

The user has formulated the desired query of the courses of the particular student and now he can ask for the result. In Figure 4.5 we can see the results and that the user has organised the query objects as well as the results using *layouts*. Besides moving objects by

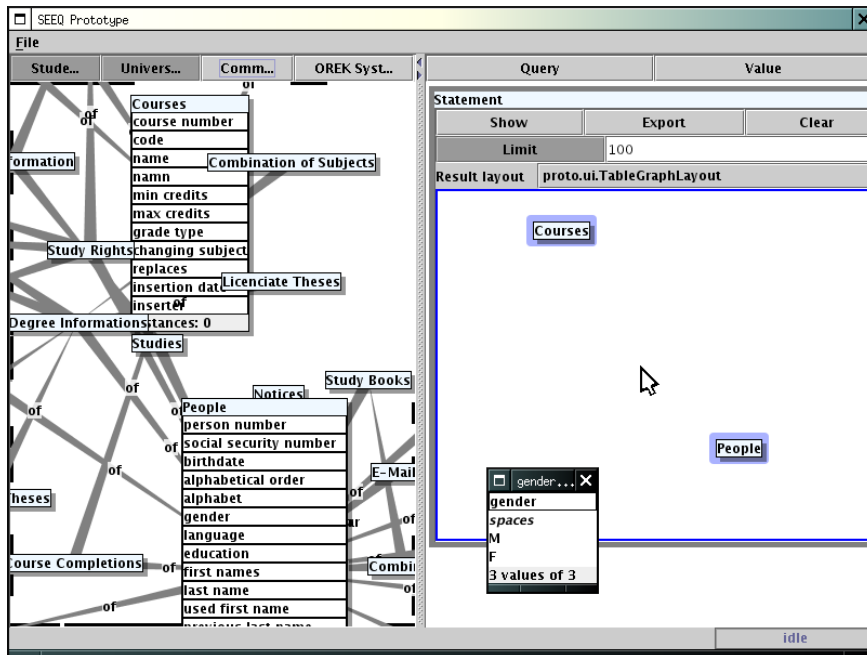


Figure 4.3: The user has begun formulating a query. The model canvas on the left has been zoomed to contain the desired representations Courses and People. The two representations have also been opened to show their attribute lists. On the right is a new query object where copies of the classes have been dragged. Below is a window that contains the gender attribute and the values it has in the database.

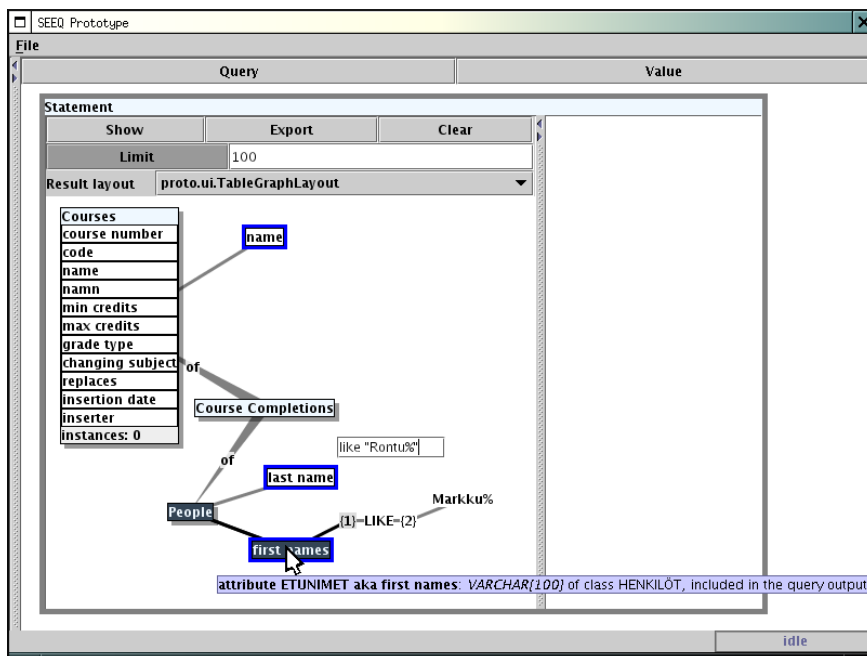


Figure 4.4: The user is finishing the formulation of a query by adding constraints. The name attribute of the Courses class has been dragged into the query to include it in the result. The user has double-clicked on the last name attribute to enter a new constraint. A previous constraint on the first names attribute is already in view.

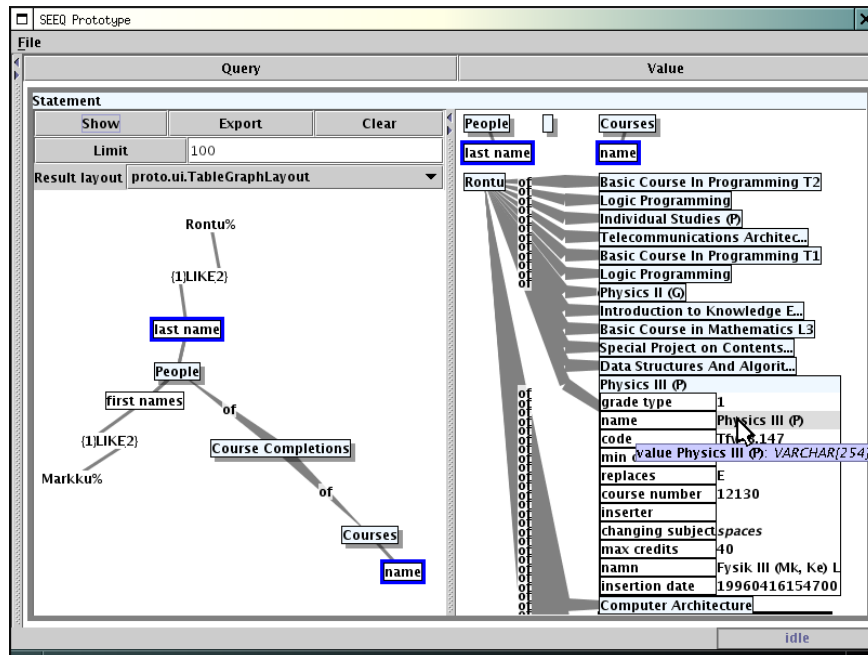


Figure 4.5: The query has been evaluated and results are displayed. The completed query is on the left and the result is on the right. The `TableGraphLayout` has been applied to the results. The output visualisation is fully interactive and the user has opened a `Courses` instance to display its details.

mouse, the user can use a spring-mass simulation to layout the contents of any canvas. Other layouts are possible such as aligning objects vertically in alphabetical order. A separate concept involving layouts is the *query result layout*. The user can select a layout that is applied to the result of the query after its results have been computed. In an ideal situation there would be a single layout system.

In the case of Figure 4.5, the `TableGraphLayout` is selected and has also been applied to the result. `GridLayout` using the spring-mass simulation is also possible. Furthermore, a user can write his own layouts and more generically his own visualisations. Many previous visual query systems do not have custom layouts or data visualisation aspects at all. In SEEQ, the user can build his own layouts by implementing the `LayoutManager` interface or more finely controlled visualisations by implementing the `CanvasManager` interface. The user interface shows all visualisations specified in a configuration file as options for the user and dynamically loads the extension as required. SEEQ does not require the user to modify the program but he must be able to implement visualisations using some programming language capable of producing compiled Java bytecode.

Once the query is complete, in addition to specifying the layout, the user can limit the number of resulting rows to a certain amount, which is useful if there is a lot of data and the system would not handle the amount interactively. Also limiting the amount can be used to speed up initial exploratory queries and the limit can be removed for the final query. The `Show` button sends the query to the underlying database and then displays the results in the result canvas. `Export` button can be used to export the results to a simple character separated text file, straight without going through the result canvas. The text data can easily be imported into spreadsheets and other programs for further processing, visualisation or be given to a third party as is often the case with the student register.

The `TableGraphLayout` is much like a simple table format used in many query systems for output. The main difference is that in all query results shown in the system (*i.e.*, also

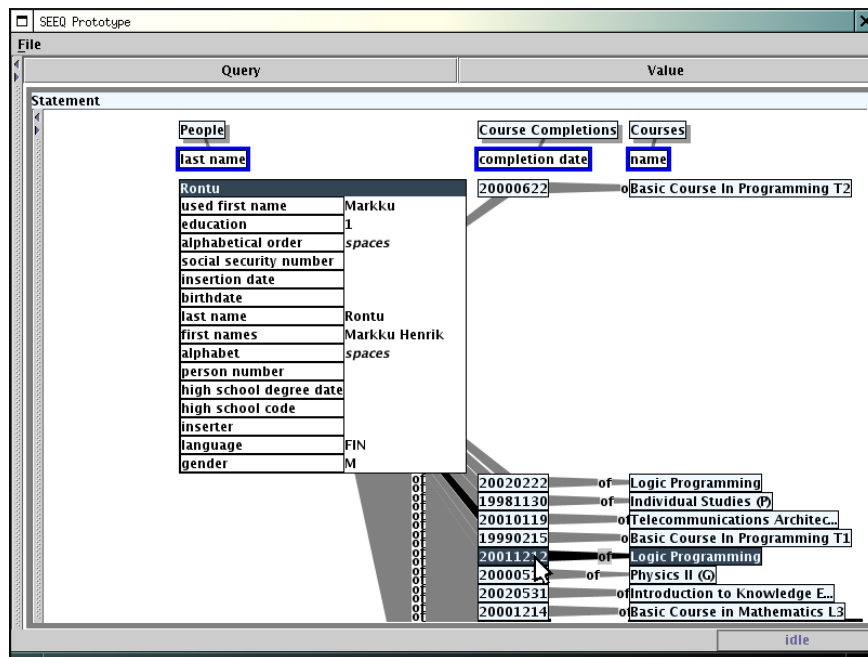


Figure 4.6: The user has modified the output with an attribute and is inspecting data. The completion date attribute has been dragged into the output visualisation to change the visualisation of Course Completions objects to display the completion date. Previously the Course Completions instances were hidden. The user has also inspected the People instance.

including other layouts), instead of repeating results like is usual, only one graph object is shown (*i.e.*, in the join table of Figure 4.5 the Last Name is repeated for each row in the database output but is unified when displayed in the user interface). Also, association instances are drawn between objects, which is natural considering that rows do not have the same meaning as in a regular result table. All the objects of the result class are full objects in the sense that they can be inspected and all their attribute-value pairs can be shown. The result canvas is no special case as the user can drag some of the result classes or some particular attribute value to other canvases to assist in formulation of subsequent queries.

Finally, in Figure 4.6 the user has refined the results. He has maximised the result canvas, added the completion date to the Course Completions and is exploring the data. Attributes can be added to the result which affects the information the result objects display but, in any case, all the information is available inside the objects (accessible by clicking on them).

4.2 A more complex query

Figure 4.7 shows the query EQ3 from Section 2.2. The query on the left does not contain any hints of its visualisation except that it uses the `CompletionOrderCanvasManager` result layout. This layout uses the `People`, `Course` and `Course Completion` instances to make its visualisation where only the `Course` instances are visibly included. There is clearly need to make the parameter mapping adaptable by the user in the interface and not hard-code it into the layout code.

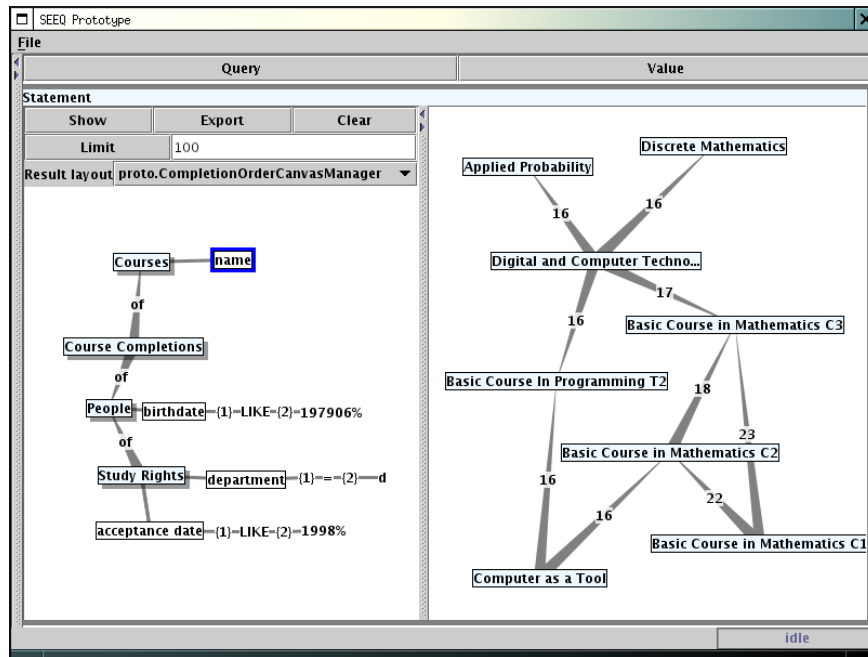


Figure 4.7: The query EQ3: "Show a graph of courses. If a student has passed a course C1 before course C2, make a precedence edge between the courses or add 1 to an existing precedence edge weight. Show the weights on the edges." The students in this query are those who began their studies in 1998, who study computer science and who were born in June 1979 (a random pick). There are altogether 20 students and precedencies that had at least 15 students are shown. Some students had taken a single course multiple times as is shown by the weight of 23.

4.3 Discussion

A fundamental idea in the user interface is the presentation of the model, query and results using the same graphical approach. The user can often do most parts of the query by clicking and dragging with a mouse. The entry of some values can be done using the keyboard, which is many times more convenient. Some values can be acquired from the database using other means such as dragging from a result subcanvas or the previously discussed attribute values menu. The implications of this idea are twofold. On one hand, the user interface is nicely uniform as everything is a manipulatable graphical object. On the other hand, some operations are awkward such as manipulating operator statements graphically, where a simple editable text field would suffice. The experiences from using the prototype suggest that not everything has to be graphical in an effective system, although rich graphical presentations have definite advantages. For example, it is debatable whether it is useful to build complex visualisations using a visual language instead of a regular programming language. The advanced users who need to create their own visualisations can be familiar with regular (*i.e.*, textual) programming languages.

In Subsection 3.2.2 there are five philosophical points of Gql of which some apply to SEEQ. SEEQ was designed with users in mind, simple queries are simply expressed and the language is separated from the user interface operations. When complex visualisations are required, the whole query is not expressible in the diagram but textual code must be written. The query language and its translation has not been specified formally although it would have advantages.

Of the issues identified in [Wong1982] and discussed in Subsection 3.2.3, SEEQ an-

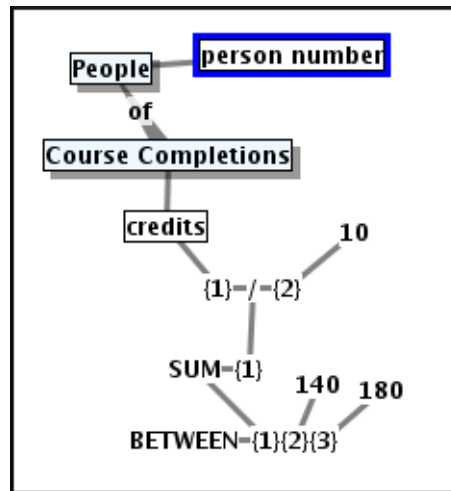


Figure 4.8: A complex operator statement. Find People who have Course Completions totalling between 140 and 180 credits. Credits are stored as integers multiplied by 10.

swers all in some manner. The issue of the user having to remember too much detail is solved by having manipulatable canvases with at least the data model in view all the time. Operators can be brought in from menus and example values of an attribute can be shown easily. The poor data model is offset by introducing associations to reduce the need to specify joins explicitly. The user can get feedback during querying by showing the results so he can verify the correctness of the query in small steps. The lack of control of the level of detail of the schema is solved by introducing filters that may be made to be hierarchical or to match certain common use cases. Finally, the meta-data browsing ability is handled by making the model canvas, or in fact any canvas, searchable and grouping the attributes inside the classes. Nevertheless, some improvements can be made for each of these, especially in showing of syntactically incorrect forms and in handling of errors.

The schema libraries of QBD* described in Subsection 3.2.4 would be useful in SEEQ. Schema library would be useful to store workspaces for different kinds of users, although, this could be done by having a database specific web page to download the schemas from. Each user can store his own modified workspaces on his computer but a centralised schema user library would be a way to distribute improved schemas easily to other users. Top-down schema libraries are possible when different levels of detail of the schema are stored in the filters. Finally, a user query library storing individual queries is possible on the user's own computer but a distributed searchable central location would be better.

In Kaleidoquery, one motivation for using a graphical language was to make complex parenthesised statements more readable. SEEQ accomplishes this because the statement is shown using graphical connections and there is no need for parentheses. Pasta-3 used a hierarchical visualisation for boolean statements. An example of an operator statement in SEEQ is in Figure 4.8. Readability might be improved in SEEQ by structuring the statements like parse trees. Now the user can (and has to) layout the statements as he likes.

Pasta-3 has a cooperative environment (see Subsection 3.2.5 for a description). It can be shown that SEEQ also has the same capabilities. Copying by dragging, entry with a keyboard and the attribute values menu like in Figure 4.3 are the "Handy values" of Pasta-3. Automatic path completion is possible, although, instead of asking the user for a choice in the case of multiple possible connections, the user is given the shortest route with the option of specifying intermediate nodes, if a specific path matters. Finally, the "Edit-and-Reevaluate loop" is made possible by enabling the user to evaluate the query while formu-

lating the query, at any time, pressing the Show button¹, and by making the query easily modifiable.

4.4 Summary

SEEQ is a visual query system that has a number of useful features that are summarised here. The user interface consists of *canvases* containing directly manipulatable graphical *representations* of the database model. The user can translate and zoom the canvases as well as search for any concept by name. *Silhouettes* of representations outside the view are shown in the borders. *Filters* can be used to hide uninteresting parts of the model and *aliases* to adapt or translate the names in the model. Queries are formulated by dragging representations of classes, associations and other objects inside a query object. Values used in the query can be entered with the keyboard, copied from other queries or from a *values menu*. A *layout* algorithm can be applied to any canvas and particularly to the result layout. Users can write their own *visualisations* and include the bytecode for the system to use. The results can also be exported to a file.

¹Results are not automatically updated as the query is modified for three reasons. First, there is the performance trouble that repeated complex queries would bring. Then, there is the fact that the query is often in a syntactically and semantically incorrect state, so the result is not useful. The user knows this best. Third and least is that sometimes the user may want to copy some parts of the results into the query to refine it and changing the result would prevent this.

Chapter 5

Implementation

In this chapter, I describe the implementation of the SEEQ prototype system. An overview of the system architecture is depicted in Figure 5.1. The user manipulates graphical representations of query objects through the user interface. Initially the query objects consist of objects loaded from a data source specification file, that is, the database schema. A *data source* is, for example, an address and a username the system uses to access a database. Through the user interface, the user can add query statements, operators and values or modify them and the schema. When the representations have been manipulated to form the desired query, the actual query objects within the representations are translated into a form the data source understands. The only data source type currently implemented is database with JDBC (Java DataBase Connectivity) drivers, so the statement is translated into SQL and given to the DBMS (DataBase Management System) for processing with JDBC. The results are parsed back to query objects that are then visualised to the user. The state of the system and the possibly modified schema can be saved back to a file. Results can also be exported into a file which bypasses the parsing and visualisation and writes the results obtained from the JDBC driver directly to disk.

The SEEQ prototype is implemented in the Java programming language. Java was chosen to ensure sufficient portability to the mixed environments used. There was also sufficient prior knowledge of its use in database connectivity (JDBC) and user interface implementation (Swing). There has been no evidence of any particular needs for a programming language used in implementing a visual query system. The availability of suitable libraries and developer experience and preference in the used tools are decisive.

The user interface was built using Swing from Java standard libraries. This proved to be a mixed blessing. On one hand, it was easy to start working on the user interface with ready-made components, but on the other hand the peculiarities of a visual query language caused a lot of extra code, because a lot of custom components were also needed. For example, Swing offers no support for drawing connections between arbitrary components, no zooming and no automatic drawing of silhouettes, magic lenses or non-linear projections. However, Swing does offer traditional user interface components such as labels, text fields, buttons, menus and tooltips. There are many comparable choices for other programming languages.

To be able to concentrate mostly on the visual aspects of SEEQ, general implementation work and considering the large amount of data in the student register (just over 800 MB), it was decided that the system should leverage existing relational databases and SQL queries and work on top of them. It allows the use of the visual query system in many existing relational databases and, if developed further, alongside with the current tools in use for the student register. Most of the architecture does support other kinds of data sources.

No locking, transactions or live data streams are supported, so the data displayed in the user interface may not represent the current state of the data source. This is not a problem because only `SELECT` queries are supported and the prototype will not be used to manipu-

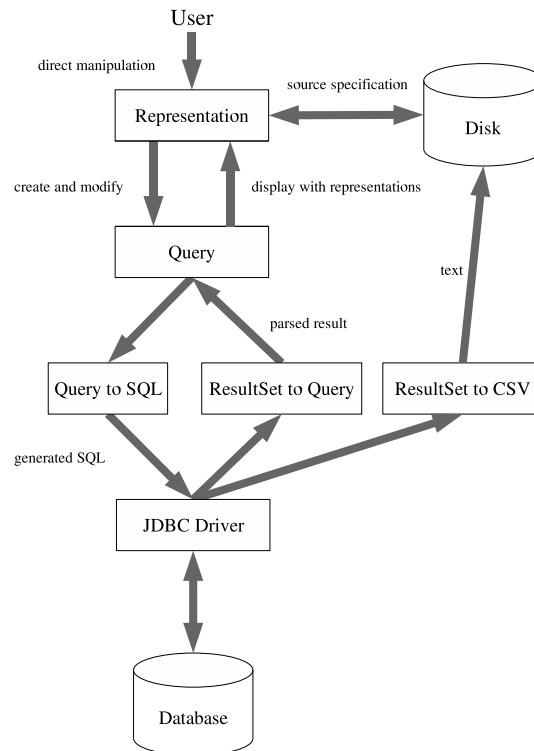


Figure 5.1: Prototype system architecture

late the real student information database. Multiple data sources can be queried at the same time, as all representations track the source they came from, but representations from multiple sources cannot be members of the same query, because there is no query processing code in the prototype. It is obvious that one database cannot join tables with another database without additional processing.

The database schema is presented to the system in the form of an XML file. Writing or generating this file is the only work needed before SEEQ can be used to query a new database. For the purposes of the student register, a program was made to translate the SQL CREATE statements of the data dump into the specification for the prototype. To facilitate the quick introduction of new data sources some further tools would be useful. The CREATE statements could be parsed more comprehensively, it would be useful to have controls in the user interface for the introduction of new classes, attributes and associations. It would also be useful to have reflection capabilities, if the data source allows the programmatic inspection of the data model. An example specification is in Figure 5.2.

The databases used were PostgreSQL¹ for the actual student information and also Axion² and Mckoi³ for testing. Of these, PostgreSQL offers best support and is a real DBMS but the others are small Java databases that can be distributed with the query system, so the user does not have to install a DBMS himself to use the prototype.

5.1 Query translation

The translation of the visual query into SQL works in the following manner:

¹<http://www.postgresql.org/>

²<http://axion.tigris.org/>

³<http://www.mckoi.com/database/>

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<sources>
  <database name="OREK"
    type="sql"
    address="jdbc:postgresql:orek"
    user="mrontu"
    password=""
    driverClass="org.postgresql.Driver">
    <model>
      <class name="People">
        <attribute name="person number" type="INTEGER" />
        <attribute name="gender" type="CHAR\cite{1}" />
        <attribute name="language" type="CHAR\cite{4}" />
      </class>
      <class name="Course_Completion">
        <attribute name="Completion number" type="INTEGER" />
        <attribute name="person number" type="INTEGER" />
        <attribute name="course number" type="INTEGER" />
        <attribute name="grade" type="INTEGER" />
      </class>
      <association type="unidirectional"
        name="Course Completion - People"
        table1="Course Completion"
        table2="People"
        id1="person number"
        id2="person number"
        multiplicity1="one"
        multiplicity2="many" />
    </model>
    <view name="Students">
      <class name="People" alias="" />
      <class name="Course_Completion" alias="Course Completion" />
      <association name="Course Completion - People" alias="" />
    </view>
  </database>
</sources>

```

Figure 5.2: An XML specification of data sources for the SEEQ prototype. Data source details (*i.e.*, type, address and user account) and contents (*i.e.*, classes, attributes and associations) are specified in a specification file that can be imported into the system. The filters are specified in the `view` definitions.

- All classes are directly mapped into database tables and they form the `FROM` part of the SQL query
- Associations store the tables and attributes they connect and are included in the `WHERE` part of the SQL query
- To facilitate the construction of complete objects from the results back in the user interface, all attributes of each class are always included in the `SELECT` part with the exception of queries containing code in the `HAVING` part of the SQL query necessitating the use of `GROUP BY`
- All operators included in the query are, depending on their exact type, translated into additional clauses:
 - Boolean operators such as `<`, `<=`, `!=` and `BETWEEN` with nested arithmetic operators such as `+`, `-`, `*` and `/` are translated into `WHERE` clauses

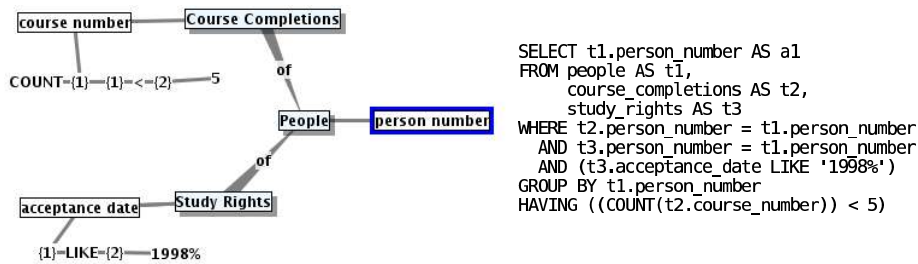


Figure 5.3: The query EQ2 in SEEQ and its translation in SQL. Show the students who began their studies in year 1998 and who have less than five Course Completions.

- Aggregate operators such as `COUNT`, `SUM` and `AVERAGE` appearing as the root operator and concerning attributes are translated into statements in the `SELECT` part where the attribute would normally appear
- Aggregate operators that appear inside boolean operator statements are translated into `HAVING` clauses
- The `GROUP` operators are translated into `GROUP BY` clauses
- Operators expecting subqueries such as `ANY`, `ALL`, `EXCEPT` and `EXISTS` are translated into the respective operators and the subqueries are translated recursively.

An example of a query and its translation can be seen in Figure 5.3. The left side is the query EQ2 from Section 2.2 and the right side is the corresponding SQL statement SEEQ sends to the database. The query contains three classes so the `FROM` clause contains three table names. The constraint for the course number attribute contains the `COUNT` aggregate so it is translated into the `HAVING` clause. This also causes the output attribute `person number` to be grouped. The acceptance date attribute is compared to the year and is translated into the `WHERE` clause. The associations are translated as the two remaining `WHERE` clauses. Note that all attribute and table names are uniquely renamed and the database names are different from the aliases displayed in the query representation.

5.2 Lessons learned

In the beginning of the implementation, it seemed sufficient to have connections that are between representations inside the same canvas. Later the inclusion of subqueries showed that they must be able to refer into the enclosing query as well, which is a parent canvas of the subquery. This was not a problem for the query functionality that is isolated from the representations and canvases, but it was a problem for the visual representation. The drawing of connections was handled in the background of the Swing component that was used to implement the canvas. This meant the connection would be painted over by either the subquery or the enclosing query depending on which component held the actual connection representation. Connections need to span component boundaries so painting becomes non-trivial using Swing.

Another restricting technological choice was the use of the native drag and drop of Java. While this is necessary at some point to enable the use of the system clipboard and to transfer information between the prototype and other applications, the native drag and drop is not good enough in the long term. It does not support the drawing of the dragged object in the new position while dragging but instead uses a generic dragging icon. Updating connections to point to the dragged object is also impossible while the dragging is still in effect.

A complicated implementation issue was that of properly isolating the querying code from the user interface details, three layers of abstraction were needed: data model, query and representation. The data model contains the absolute basics of the enhanced data model: classes, attributes, associations, objects and association instances. On top of this is the query level with query classes, query attributes, *etc.*, and additionally operators, statements and values. Finally, there is the representation level where each of the query objects has a corresponding representation: class representation, association representation, *etc.* This means a lot of code must deal with three interconnected class hierarchies. The part of the program that suffers most from this complexity is the serialisation code that is responsible for saving and loading the workspace and must deal with all the hierarchies. Using a suitable persistence framework would help to reduce the amount of code and would make it easier to implement undo.

From the state of the visual query systems examined in this work and from the experiences in implementing SEEQ, it is rather obvious that further research would be much easier, if there were suitable graphical user interface libraries available. Many similar graphical primitives and operations are used in the visual query systems and many systems can benefit from a generic implementation. By using common libraries, it can be guaranteed that all systems have the useful features. An example of a step towards this direction is the Graphical Editing Framework or GEF [Gra]. GEF is an attempt to build a framework that enables the easy creation of graphical editors, which have a lot in common with visual query systems. Both need the visualisation and graphical manipulation capabilities. Reusable libraries are, of course, not only applicable to visual query systems in general or to user interface libraries in particular. Some other good candidates for libraries and reusable subsystems for use in visual query systems are graph layout algorithms and visualisation tools, visual language tools (*i.e.*, compilers, interpreters and compiler-compilers), constraint programming and global optimisation frameworks and graph algorithms. For example, SEEQ could have used existing good layout algorithms, a visual language compiler-compiler to specify the visual language formally and define its translation into SQL, constraint programming for either the user interface layout (as in DOODLE) or query computation scheme, global optimisation frameworks for layout (as in [Coleman1996]) and graph algorithms for the automatic connection in queries. Some libraries already exist, in free and commercial form, so if the intention is to build a system for real use their inclusion should be investigated seriously. Research systems can also make use of them by allowing the implementers to focus on the new features and possible usability test results would be more in line with what is expected from a real system. For example, undo support, tab-completion and meta-level search are mainly uninteresting implementation details but their lack may affect usability results in practice.

Chapter 6

Evaluation

In this chapter the SEEQ prototype is evaluated against the requirements in Chapter 2 and systems in Chapter 3. The evaluation begins with ideas described in Section 2.1.2 and then proceeds to analysing the requirements from Section 2.3. Finally, SEEQ is compared to the other systems listed in Section 3.4.

6.1 Initial goals

As described in Section 2.1.1, the development of new queries in current systems takes a lot of time. The database model consists of a large number of tables and attributes. There are a lot of joins in the queries and they span many tables. The queries are often stored for later use when they may be modified to suit new requirements. Results are often exported for other parties to use.

SEEQ helps the user's work by offering a graphical view of the database schema with tools such as filters and search to examine it. There is no need to remember all the details because the schema can be explored. Furthermore, it is easy to experiment with different versions of queries, store queries for later use, modify queries as needed and show results while still developing the query. The explorative nature of SEEQ helps the discovery of information. Minor improvements are an easier way of making joins, having joining opportunities to be explicit and a concept search in every canvas. Moreover, results can be exported.

Compared to RapidSQL, which is the commercial product used currently, problems are the lack of context sensitive help, error detection and data modification queries as well as the incomplete keyboard operation and the immaturity of SEEQ.

6.2 Requirements analysis

This section analyses SEEQ in light of the requirements in Section 2.3. Appendix A contains the exact results for each evaluated requirement.

Q1 Can a visual query system be used to gather information from a (student register) database?

The implemented prototype is obviously not a complete product. For example, it does not come with a comprehensive online documentation. However, it is capable of displaying all of the information of the student register. It can also be extended to support natural representations of others kinds of data (*e.g.*, images).

Q2 Can the system answer the example queries?

Yes, the system can answer all the selected example queries (EQ1-3). EQ1 is a straightforward join and has not been included in this work but EQ2 is shown in Figure 5.3 and EQ3 in Figure 4.7.

Q3 Can a reasonable implementation be made for a visual query system?

The SEEQ prototype is around 15 KLOC¹ and is the work of a single programmer. Since it was successfully implemented and the amount of code is small, it is safe to assume visual query systems are a significant amount of work but not too much. Commercial quality polish would cause a lot more work but then there would typically be more implementers. The availability of suitable libraries would further reduce the work. For example, the user interface library of the prototype is approximately 7.2 KLOC, layouts 0.7 KLOC and query translation 1.7 KLOC. Most of the user interface library features would be covered by a generic graphical editing framework. The few layouts implemented and many more would be available in a graph layout library. Finally, the query translation process would be specified more clearly and with less lines of code, if a visual language compiler-compiler were to be used.

Some particular problems with the implementation were described in Chapter 5 but the largest difficulty is the complexity of the whole system. A visual query system consists of multiple complex subcomponents that must be integrated. Databases, query processing and concurrent use are combined with visual languages, graphical user interfaces and data visualisation. All the fields have been studied extensively but there is a need to combine them into a single program. The best combination allows the features to work together but they are orthogonal in implementation, so their continuing separate development is possible. When new developments in a single field are sufficiently interesting they can be reintroduced into the visual query system. This is done by a modular structure with clearly defined interfaces. More research on this area would be useful as would the actual implementation of the libraries and subcomponents.

The development time is hard to assess accurately. The whole project lasted ten months but a significant amount of time was spent converting data, examining related work and writing this thesis. Many parts of the prototype were refactored aggressively.

If a regular application with the example queries would be developed, it would be less work because a comprehensive graphical editor and a visual language translator would be unnecessary. Most work would go into implementing the example query EQ3 that requires the building of a suitable visualiser, just as it requires in SEEQ. However, the advantages of the visual query system are that it works with any database. No additional work, except implementing new visualisations, would be required to support all of the queries in Section 2.2. In a regular application, a programmer would implement new views, typically one for each query, and this implementation work consists of more than just the specification of the query (*i.e.*, what input and output components the view contains and where they are). On the other hand, the main advantage compared to a conventional textual query system is the interactive model canvas with filters, silhouettes and more as well as an interactive result canvas with layouts. Textual query systems do not usually have any of these or only a primitive graphical schema.

Q4 How usable is the visual query system?

Thorough usability testing was left out right in the beginning as it would have been too much work and out of scope for this project. Even the planned small amount of tests was in the end deemed to be best left for later work. Some initial impressions are available that

¹LOC, Lines Of Code, no comments or blanks included

have been gained by using the system for the example queries and *ad hoc* queries during demonstrations.

- The visual query system is useful for understanding the database contents (*i.e.*, the exploratory aspect of the problem) but the query formulation is slightly unintuitive for complex queries. Complex queries can be formulated quickly and with few actions, if the user understands the translation process well. It would be interesting to know how long it takes before a new user acquires a sufficiently accurate mental model of the process through experimentation and what kind of improvements can be made with the use of interactive tutorials, automatic natural language descriptions of queries, good feedback on errors, *etc.*
- Graphically manipulating objects works but the entry of conditions could be better. Simple conditions might well be made using an editable text field and they could be visualised as formatted text more clearly than now as graph elements. Although, as noted in Section 4.3, complex operator expressions may be better visualised in other ways. Ideally, any piece of code could be written as text and included as a condition.

Q5 Is the visual query system better than a textual query system in this case?

Yes, the database model is much easier to understand through the graphical representation. The graphical query formulation is better in some cases but the lack of comprehensive keyboard support means querying is more cumbersome in others. Integrated result visualisations cannot be a part of a textual system so they are a definite advantage, although, not all users of the student register need them.

Q6 Is there potential for further experimentation?

Yes, some aspects of the new system are promising and a lot can be done to improve the system. Improvement ideas and future work are outlined in Chapter 7.

6.3 Comparison to other systems

In Section 4.3 the features of SEEQ are discussed in relation to other systems with similar features. There are many small features in common, such as the easy copying of values, visualisation of operators and automatic path completion. This section considers SEEQ from another perspective. The Section 3.3 describes a number of differences in existing visual query systems that were used to focus on suitable systems. Using the classifications from [Catarci1997] SEEQ is a diagrammatic visual query system, although, lists are used, for example, to display class attributes instead of having them in the already densely populated model graph. The diagrams are used in schema, query and result visualisation. It supports all the different understanding strategies (*i.e.*, top-down, browsing and schema simplification). The query formulation uses a combination of schema navigation and sub-queries. In the classification of [Dennebouy1995], SEEQ belongs into the assertional query editor class.

Considering the formal aspects of SEEQ, its query language is mapped into SQL but it offers an extensible architecture for visualisations that may also be used for arbitrary computations on the result. A better defined translation would be good for formal analysis. It is not equivalent to SQL because it does not support every language feature.

The visual elements used in SEEQ are boxes, text, lines and arrows. With the implementation in Java Swing, it would be straightforward to include other shapes, icons and pictures where deemed useful. However, in the student register, they are not very useful. The appearance of the objects has been fixed by the system. The only adaptation that can be done is choosing descriptive aliases to implementation specific terms and being able

Table 6.1: SEEQ using the grading in Section 3.4

System	Language	Year	Query visualisation	Output visualisation	Level of use	Implementation generality	Customisability	Expressiveness	Support for exploration
SEEQ		2004	***	***	**	**	***	**	***

to make domain specific visualisations. A significant improvement would be to make the implementation more easily extensible so that domain specific adaptations would be easier to make. Ideally there is no need to recompile the system when doing adaptation but in that case a separate scripting system or an implementation with a more dynamic language is required.

There has been no real use of SEEQ yet but it has been used with a real database and queries. The next step is to put it into real use and start improving it based on feedback. Table 6.1 contains grades SEEQ received using the grading explained in Section 3.4. Its strengths are clearly in visualisation and exploration where it scores full three stars. Visualisation is not handled by most existing systems except for the schema. The high grade for exploration comes from the fully interactive output visualisation.

The strongest competitors from Section 3.4 in visualisation are DOODLE and Spider. The output visualisations of SEEQ are very much like those of Spider. Indeed, they were inspired by Spider, although, the paradigms of the two systems are otherwise completely different. QBD* also has an output visualisation that avoids the duplication of values. DOODLE is unsurpassed in its visual definition of layouts. The constraint programming approach to defining user interfaces and layouts is a very exciting direction for future research. The output pad of DOODLE described in Subsection 3.2.1 is very similar to what SEEQ result canvases can do. As an advantage, the result canvas in SEEQ is just another element in the user interface with the same interaction methods. Also, the fish-eye view of DOODLE addresses the same concerns as the silhouettes of SEEQ.

SEEQ has less than perfect scores in level of use, implementation generality and expressiveness. DEVise as well as many traditional query systems have been used a lot more than SEEQ at this point. For three stars, implementation generality would require better defined interfaces and user definable components. The representations offered by SEEQ are not customisable enough. Improving expressiveness can be done at two levels. It is possible to support all SQL constructs and make sure that all SQL queries map to some SEEQ queries. Another alternative is to stop using SQL and use something more expressive, maybe even a completely new visual language. A different form of query processing is needed to support queries that query multiple databases at the same time.

Chapter 7

Conclusion

In this work, I have described how a visual query system can be used to solve problems in a student information system. The main problem in the OREK student register, as well as many other databases, is that there is a huge number of concepts in the database schema and the user has to remember too many things. Another problem is that the people who need to access the data often do not know a textual query language and those who do know would like an efficient and user friendly system. The solution offered is a visual query system named SEEQ.

SEEQ solves the problem of complex database schemas by having an interactive manipulatable graphical schema. Concepts can be hidden by using filters and the view can be zoomed. The diagrammatic or graph-like appearance is also used in the query and result visualisations. Another particular strong point is the support for exploration of the data. The user can easily experiment with queries because modifying them is easy. Objects in the output visualisation can be inspected for details. SEEQ does not only limit itself to student registers but any database may be queried. In all fields compared in this work, it is as least as good as the other systems. Many existing systems do not have result visualisation at all but SEEQ offers user definable and fully interactive visualisations.

7.1 Improvement ideas

- Parametrisable queries and user defined operators are needed to enable the implementation of domain specific higher level abstractions. These can be used as building blocks in queries instead of always needing to consider implementation details of the database. For example, when querying students it is often useful to exclude students who have graduated. This must be done by adding the right constraints or copying them from existing queries. However, in an ideal situation these constraints might be implemented as a domain specific operator that is included into the query as needed.
- A more complex and comprehensive keyboard support in addition to an easily manipulatable graphical user interface is needed for efficient use.
- The reusability of layouts should be investigated. The mapping of query results to the input parameters of the result layout algorithm should be easily definable by the user so that the same layout algorithm can be used for many queries with similar visualisation requirements. For example, a data-flow diagram interface could be intuitive. Another feature to improve reuse would be to make a parametrisable query type. Parametrised queries could be reused by binding values to their parameters and the user would not need to view the internal implementation details.
- One improvement would be to be able to systematically manage previous queries in

libraries so that they would be easier to find and use. Stored queries might be named, described, classified into categories and also indexed by them.

In general, there is need for good libraries and reusable subsystems to reduce the implementation effort. They include a graphical user interface library with features like zoomable user interface and magic lenses or silhouettes, a graph layout library with a variety of customisable layouts, constraint satisfaction and global optimisation frameworks, visual language tools and graph algorithms.

7.2 Future work

The next phase is to start using the prototype and develop it further based on feedback. It should be used for many kinds of databases, not just for the student register. Once the initial usability concerns have been solved and missing important features have been implemented, the usability should be thoroughly and systematically tested. Further development should be based on the results of the testing. The SEEQ prototype will be packaged and made available for public use.

Appendix A

Requirements results

Here are the analysis results for the requirements. Some of the questions could not be answered in this work and are denoted by a dash.

ID	Description
Q1	Can a visual query system be used to gather information from a (student register) database?
A1	The system can display all of the information.

Q2	Can the system answer the example queries?
A2	The system can answer all of the queries.

Q3	Can a reasonable implementation be made for a visual query system?
A3	The required implementation effort is significant.
Q3.1	Are there any particular problems with visual query system implementation?
A3.1	See Section 6.2.
Q3.2	What technical difficulties are there or would be?
A3.2	See Section 6.2.
Q3.3	Does it take too much time to develop?
A3.3	See Section 6.2.
Q3.4	Is the implementation effort comparable to conventional systems?
A3.4	See Section 6.2.

Q4	How usable is the visual query system?
Q4.1	Can a user do a complete simple query in "a few seconds" and with "a few actions"?
A4.1	-
Q4.2	Can a user repeat an old query in "a few seconds" and with "a few actions"?
A4.2	-
Q4.3	Once a user has an idea of a question, how long does it take to formulate a query?
A4.3	-
Q4.4	How easy it is to modify a query?
A4.4	-
Q4.5	How easy it is to find and fix errors in a query?
A4.5	-
Q4.6	How responsive is the system? Is it interactive?

A4.6	The system and query formulation behaves interactively with short response times. The visual query answers queries EQ1-2 in comparative time to a regular DBMS. Output visualisations are fast enough to be usable. The output layout performs slower than a regular layout system. The prototype is interactive and responsive in most situations. When calculating query results the system freezes. On the other hand, when the spring-mass simulation is used the system does not freeze but displays a progress bar.
------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Q5	Is the visual query system better than a textual query system in this case?
A5	-

Q6	Is there potential for further experimentation?
A6	yes

Bibliography

- [Angelaccio1990] Angelaccio, M., Catarci, T. and Santucci, G. QBD*: A fully visual query system. *Journal of Visual Languages and Computing*, 1(2):255–273, 1990.
- [Aslandogan1995] Aslandogan, Y. A. *VQL: A Visual Query Language for Uniform Database Access*. Master’s thesis, Case Western Reserve University, 1995.
- [Aufaure2001] Aufaure, M.-A. and Trépied, C. What approach for searching spatial information? *Journal of Visual Languages and Computing*, 12(4):351–373, 2001.
- [Badre1996] Badre, A. N., Catarci, T., Massari, A. and Santucci, G. Comparative ease of use of a diagrammatic vs. an iconic query language. In *Proceedings of the 3rd International Workshop on Interfaces to Databases*, pages 1–14. 1996.
- [Balkir2002] Balkir, N. H., Özsoyoglu, G. and Özsoyoglu, Z. M. A graphical query language: VISUAL and its query processing. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):955–978, 2002.
- [Benzi1999] Benzi, F., Maio, D. and Rizzi, S. VISIONARY: a viewpoint-based visual language for querying relational databases. *Journal of Visual Languages and Computing*, 10(2):117–145, 1999.
- [Boyle1996] Boyle, J., Leishman, S. and Gray, P. M. D. From WIMPS to 3D: The development of AMAZE. *Journal of Visual Languages and Computing*, 7(3):291–319, 1996.
- [Catarci1995a] Catarci, T. Diagrammatic vs. textual query languages: A comparative experiment. In *Proceedings of the Third IFIP 2.6 Working Conference on Visual Database Systems*, pages 69–83. 1995.
- [Catarci1995b] Catarci, T. and Santucci, G. Are visual query languages easier to use than traditional ones? an experimental proof. In *Proceedings of HCI ’95*, pages 323–338. 1995.
- [Catarci1997] Catarci, T., Constabile, M. F., Levialdi, S. and Batini, C. Visual query systems for databases: A survey. *Journal of Visual Languages and Computing*, 8:215–260, 1997.
- [Catarci2000] Catarci, T. What happened when database researchers met usability. *Information Systems*, 25(3):177–212, 2000.

- [Chan1997] Chan, H. C. Visual query languages for entity relationship model databases. *Journal of Network and Computer Applications*, 20:203–221, 1997.
- [Chandra1988] Chandra, A. K. Theory of database queries. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–9. 1988.
- [Coleman1996] Coleman, M. K. and Parker, D. S. Aesthetics-based graph layout for human consumption. *Software—Practice And Experience*, 26(1):1–25, 1996.
- [Consens1990] Consens, M. P. and Mendelzon, A. O. GraphLog: a visual formalism for real life recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 404–416. 1990.
- [Cruz1987] Cruz, I. F., Mendelzon, A. O. and Wood, P. T. A graphical query language supporting recursion. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 323–330. 1987.
- [Cruz1992] Cruz, I. F. DOODLE: a visual language for object-oriented databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 71–80. 1992.
- [Cruz1997a] Cruz, I. F., Averbuch, M., Lucas, W. T., Radzysinski, M. and Zhang, K. Delaunay: a database visualization system. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 510–513. 1997.
- [Cruz1997b] Cruz, I. F. and Lucas, W. T. A visual approach to multimedia querying and presentation. In *Proceedings of the Fifth ACM International Conference on Multimedia*, pages 109–120. 1997.
- [Cruz2001] Cruz, I. F. and Leveille, P. S. As you like it: Personalized database visualization using a visual language. *Journal of Visual Languages and Computing*, 12(5):525–549, 2001.
- [Dennebouy1995] Dennebouy, Y., Andersson, M., Auddino, A., Dupont, Y., Fontana, E., Gentile, M. and Spaccapietra, S. SUPER: Visual interfaces for object+relationship data models. *Journal of Visual Languages and Computing*, 6(1):73–99, 1995.
- [Erwig2003] Erwig, M. Xing: a visual XML query language. *Journal of Visual Languages and Computing*, 14(1):5–45, 2003.
- [Goldman1985] Goldman, K. J., Goldman, S. A., Kanellakis, P. C. and Zdonik, S. B. ISIS: Interface for a semantic information system. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, pages 328–342. 1985.
- [Gra] Graphical Editing Framework. Last checked: 1st March 2004.
URL <http://www.eclipse.org/gef/>
- [Gyssens1990] Gyssens, M., Paredaens, J. and Gucht, D. V. A graph-oriented object model for database end-user interfaces. pages 24–33. 1990.

- [Kuntz1989] Kuntz, M. and Melchert, R. Pasta-3's graphical query language: direct manipulation cooperative queries, full expressive power. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 97–105. 1989.
- [Lee2001] Lee, S. K. and Whang, K.-Y. VOQL*: A visual object query language with inductively defined formal semantics. *Journal of Visual Languages and Computing*, 12(4):413–433, 2001.
- [Livny1997] Livny, M., Ramakrishnan, R., Beyer, K., Chen, G., Donjerkovic, D., Lawand, S., Myllymäki, J., and Wenger, K. DEVise: integrated querying and visual exploration of large datasets. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 301–312. 1997.
- [Massari1995] Massari, A. and Crysanthos, P. K. Visual query of completely encapsulated objects. In *Proceedings of the Fifth International Workshop on Research Issues in Data Engineering*, pages 18–25. 1995.
- [Murray2000] Murray, N. S., Paton, N. W., Goble, C. A. and Bryce, J. Kaleidoquery—a flow-based visual language and its evaluation. *Journal of Visual Languages and Computing*, 11(2):151–189, 2000.
- [Oliboni2002] Oliboni, B. and Tanca, L. A visual language should be easy to use: a step forward for XML-GL. *Information Systems*, 27(7):459–486, 2002.
- [Orman1996] Orman, L. V. Queries = examples + counterexamples. *Information Systems*, 21(8):615–635, 1996.
- [Papantonakis1994] Papantonakis, A. and King, P. J. H. Gql, a declarative graphical query language based on the functional data model. In *Proceedings of the Workshop on Advanced Visual Interfaces*, pages 113–122. 1994.
- [Rap] Rapid SQL. Last checked: 26th January 2004.
URL <http://www.embarcadero.com/products/rapidsql/>
- [Rapley1994] Rapley, M. H. and Kennedy, J. B. Three dimensional interface for an object oriented database. In *Proceedings of the 2nd International Workshop on Interfaces to Databases*, pages 133–158. 1994.
- [Rodgers1997] Rodgers, P. J. and King, P. J. H. A graph-rewriting visual language for database programming. *Journal of Visual Languages and Computing*, 8(5):641–674, 1997.
- [Sebillo2000] Sebillo, M., Tortora, G. and Vitiello, G. The Metaphor GIS Query Language. *Journal of Visual Languages and Computing*, 11(4):439–454, 2000.
- [Silva1999] Silva, S. F., Catarci, T. and Schiel, U. A graphical notebook as interaction metaphor for querying databases. In *Proceedings of the XIV Simpósio Brasileiro de Banco de Dados*, pages 171–185. 1999.
- [Vadaparty1993] Vadaparty, K. V., Aslandogan, Y. A. and Özsoyoglu, G. Towards a unified visual database access. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 357–366. 1993.

[Wong1982] Wong, H. K. T. and Kuo, I. GUIDE: Graphical user interface for database exploration. In *Proceedings of the Eighth International Conference on Very Large Data Bases*, pages 22–32. 1982.

[XML] XML Query. Last checked: 28th January 2004.
URL <http://www.w3.org/XML/Query>