

Università degli Studi di Modena e Reggio Emilia

Dipartimento di Ingegneria "Enzo Ferrari" di Modena

Corso di Laurea Magistrale in Ingegneria Informatica

INTERROGAZIONE E RAPPRESENTAZIONE VISUALE
DI GRANDI MOLI DI DATI DI STREAM

VISUAL QUERYING AND RESULTS VISUALIZATION
OVER VERY LARGE DATA STREAMS

Relatore:

Prof. Sonia Bergamaschi

Candidato:

Alberto Malagoli

Correlatore:

Dott. Ing. Massimo Mecella

Anno Accademico 2011/2012

Contents

I	Summary in Italian	7
II	Introduction	13
III	Smart Vortex	19
1	Project Overview	21
2	Main concepts	25
2.1	Meta-data	25
2.2	Database Management System (DBMS)	25
2.3	Query	26
2.4	Query Language	26
2.5	Data Stream	26
2.6	Data Stream Management System (DSMS)	26
2.7	Continuous Query (CQ)	27
2.8	Continuous Query Language (CQL)	28
3	Data Streams	31
3.1	SCSQ	32
3.2	AmosQL	34
3.3	SCSQL	34
4	Requirements	37
4.1	User interface requirements	37
4.2	Use cases	41

4.2.1	ISP-1: Data streaming computation from stored data	42
4.2.2	ISP-2: Computation of real-time data streams	43
4.2.3	ISP-3: Data streams within collaborative environments	44
4.2.4	Meta Use Case	44
IV	State of the Art	49
5	Visual Query Systems	51
5.1	Query By Diagram	53
5.1.1	QBD*	53
5.1.2	MURAL	53
5.2	Query By Icon	54
5.2.1	QBB	54
5.2.2	QBI	55
5.2.3	The Flow Metaphor	56
5.2.4	Kaleidoquery	58
5.2.5	Visual	59
5.3	Visual Query Comparison	60
6	Visual Analysis	63
6.1	Data and View Specification	64
6.1.1	Visualize	64
6.1.2	Filter	64
6.1.3	Sort	65
6.1.4	Derive	65
6.2	View Manipulation	65
6.2.1	Select	65
6.2.2	Navigate	66
6.2.3	Coordinate	67
6.2.4	Organize	68
6.3	Process and Provenance	69

<i>CONTENTS</i>	5
V Results	71
7 Visual Query Language (VQL)	73
7.1 VQL elements	74
7.1.1 Object	74
7.1.2 Function	74
7.1.3 Select	75
7.1.4 Conditions	75
7.1.5 Parameters	76
7.1.6 Connection	77
7.1.7 Handle	78
7.2 Visual Query example	79
8 Query translation	81
8.1 Definitions	81
8.1.1 Boolean operators	82
8.1.2 Objects names	83
8.1.3 Generic graph	83
8.1.4 Functions used in the algorithms	84
8.2 Conditions of translatability	84
8.2.1 Hypothesis	85
8.2.2 Correctness tests	85
8.3 Algorithm	91
8.4 Examples	99
8.4.1 Simple example	99
8.4.2 Complex example	102
9 Prototype	111
9.1 Technologies	111
9.2 Architecture	112
9.2.1 Web application	112
9.2.2 FDSMS bridge	114
9.2.3 Architecture overview	114
9.3 Web application	115
9.3.1 Visual Query Editor	115

9.3.2	Data visualization	128
9.3.3	Query execution	130
9.3.4	Dashboard	133
9.3.5	Saved queries	134
9.3.6	Templates	135
9.3.7	Comments	135
9.3.8	Overall view	136
9.4	FDSMS Bridge	137
9.4.1	Query execution	138
9.4.2	Types and functions retrieving	142
9.5	User DataBase	143
VI	Conclusions and Future Works	153
10	Implemented requirements	155
11	Considerations and limitations	159
11.1	Visual Query Language	159
11.2	Query translation algorithm	160
11.3	Visual Query Editor	160
12	Future Works	163
VII	References	167

Part I

Summary in Italian

La progettazione e la produzione di prodotti innovativi e altamente specializzati comporta un ciclo di vita complesso, a partire da un'idea che passa attraverso lo sviluppo, la fabbricazione, il funzionamento, la manutenzione e lo smaltimento.

Durante l'intero ciclo di vita del prodotto, vengono generati molti flussi di dati di diverso tipo, tra cui:

- dati provenienti da sensori e apparecchiature di analisi;
- flussi di dati di simulazione e di collaudo;
- flussi di dati di progettazione;
- flussi di dati multimediali di collaborazione;
- flussi ricavati da eventi di alto livello.

Questi flussi di dati vengono creati ed utilizzati in ogni fase del ciclo di vita del prodotto.

Il progetto di ricerca europeo Smart Vortex si colloca in questo contesto. Nell'industria, la quantità di dati prodotti e raccolti dai flussi di produzione precedentemente citati è in continuo aumento. Questo porta a problemi e difficoltà di analisi degli stessi, da parte degli operatori umani; a volte, dati che sarebbero importanti non vengono nemmeno presi in considerazione. Per migliorare le prestazioni nei processi decisionali, nella progettazione ingegneristica e nella decisione collaborativa, il progetto Smart Vortex si propone di studiare un modello che catturi le informazioni pertinenti e le renda disponibili nel posto giusto al momento giusto.

Il progetto affronta il problema della estrazione di informazioni utili da una grande mole di dati; inoltre, un obiettivo principale è quello di migliorare il lavoro collaborativo ed il processo decisionale, condividendo queste informazioni tra le persone. Il lavoro di collaborazione in sé genera flussi di dati che devono essere raccolti ed utilizzati. Più in generale, lo scopo del progetto Smart Vortex "... è quello di fornire una infrastruttura tecnologica costituita da una suite completa di strumenti interoperanti, servizi e metodi per la gestione intelligente e l'analisi di flussi di dati molto grandi, con lo scopo di raggiungere una migliore collaborazione e una migliore presa di decisioni in progetti di collaborazione su larga scala, riguardanti progettazioni industriali innovative".

Smart Vortex produrrà diversi componenti, per implementare le funzionalità necessarie. Un Data Stream Management System (DSMS) raccoglierà i flussi di dati, eseguirà manipolazioni su di essi e offrirà funzionalità per

l'interrogazione e il recupero dei dati. Verrà creato inoltre un insieme di regole ed un piano di accesso ai dati, per permettere il lavoro collaborativo tra individui, ed anche tra industrie in competizione tra loro: tale modello permetterà l'accesso a dati condivisi, pur garantendo la preservazione delle proprietà intellettuali.

Inoltre, un componente chiave del progetto sarà una interfaccia visuale per l'interrogazione dei flussi di dati, che ne permetterà il recupero e l'analisi attraverso un Visual Query Language (linguaggio visuale di interrogazione), semplificando tale operazione e permettendo la costruzione di interrogazioni anche complesse ad utenti non esperti. L'interfaccia grafica mostrerà poi i risultati delle interrogazioni attraverso opportune visualizzazioni, portando in risalto ed estraendo informazioni utili da tali flussi di dati.

Il lavoro presentato in questo documento di tesi si focalizza sulla realizzazione di tale componente.

Questo lavoro è stato svolto durante il secondo anno, su quattro, del progetto Smart Vortex, ed è stato indirizzato alla realizzazione degli obiettivi preposti, ovvero la progettazione di un Visual Query Language per la composizione semplificata di query, e la visualizzazione dei risultati dell'interrogazione sul DSMS adottato, seguendo i principi definiti dal progetto. Una interfaccia utente completa delle necessarie funzionalità è quindi stata progettata, comprendente strumenti per la composizione di query visuali, per la definizione di visualizzazioni appropriate dei risultati, e per la organizzazione di tali visualizzazioni in cosiddette "dashboard". Infine, è stato realizzato un prototipo funzionante dell'interfaccia utente, implementato come applicazione web, ed un server TCP per la comunicazione con il sottostante DSMS.

Il Visual Query Language che è stato progettato, è pensato per la rappresentazione in forma visuale dei costrutti definiti da SCSQL, il linguaggio per la formulazione di query, anche su stream, adottato nel progetto Smart Vortex. Query scritte in SCSQL vengono poi eseguite su SCSQ, il DSMS su cui si appoggia il progetto; esso utilizza oggetti e tipi per la rappresentazione dei dati, e fa un forte uso di funzioni (o metodi). Il Visual Query Language presentato segue quindi questi stessi principi, facendo proprio il concetto di composizione di funzioni ed estendendolo anche ad altri costrutti del linguaggio. Nella sua definizione si sono poi adottati concetti generali della rappresentazione dell'informazione, quali ad esempio l'uso di forme geometriche e colori distinti per una migliore categorizzazione e riconoscimento dei vari elementi visivi. Non tutti i costrutti introdotti da SCSQL sono però stati tradotti in una loro controparte visuale, ma solo quelli necessari alla composizione di query del tipo SELECT-FROM-WHERE (dall'SQL). Questo non rappresenta però una vera limitazione, in quanto tale linguaggio visuale di

query è pensato proprio per questo tipo di utilizzo, ed altre operazioni, quali la manipolazione e la definizione dei dati sottostanti, esulano dai suoi scopi.

È stato quindi proposto un algoritmo per la traduzione di una query visuale in una testuale, scritta nel linguaggio SCSQL. L'algoritmo fa uso di una rappresentazione a grafo della query visuale, e tramite una navigazione dello stesso viene composta la sua controparte testuale. Prima della traduzione vera e propria, l'algoritmo esegue alcuni controlli di correttezza della query visuale, che non può essere tradotta se non rispetta questi vincoli.

Il prototipo sviluppato, come già accennato, si compone di due parti: una applicazione web, che implementa l'interfaccia utente, ed un server TCP, chiamato FDSMS Bridge, per la comunicazione con il DSMS, SCSQ.

L'applicazione web, scritta utilizzando la tecnologia JSP su web server Tomcat, si preoccupa, come detto, di creare l'interfaccia utente, e di gestire le comunicazioni tra il client web ed il DSMS, per l'esecuzione delle query. All'utente viene messa a disposizione una interfaccia per la composizione delle query visuali, chiamata Visual Query Editor, completa di funzionalità che lo aiutano e guidano in questo compito; tali funzionalità evitano, almeno in parte, la costruzione di query formalmente non corrette. Un'altra interfaccia è poi adibita alla definizione delle necessarie visualizzazioni per la rappresentazione dei dati ritornati dalle interrogazioni. Una serie di grafici (a linee, a barre, a punti, ...) vengono messi a disposizione dell'utente, il quale può decidere in che modo comporre tali visualizzazioni in modo da estrarre dai dati le informazioni più interessanti o più consone ai suoi obiettivi. Infine, le visualizzazioni dei risultati di query possono essere composte ed organizzate su una "dashboard", così da poter analizzare o monitorare dati provenienti da diverse sorgenti, comparandoli tra loro o localizzando eventuali malfunzionamenti. Il web server fa uso di un DataBase MySQL per la memorizzazione delle query visuali e di altri dati di configurazione dell'interfaccia utente.

FDSMS Bridge è, come accennato, un server che permette la comunicazione col sottostante DSMS attraverso messaggi testuali scambiati col protocollo TCP. Scritto in Java, fa uso di una apposita libreria, scritta anch'essa in Java, per lo scambio di dati con SCSQ. Il suo compito è quello di ricevere richieste di esecuzione di query da parte del client, passarle ad SCSQ per l'esecuzione, e ritornare al client i risultati. Essendo tali query eseguite su stream di dati, il flusso di risultati ottenuto è possibilmente infinito; le connessioni tra client e FDSMS Bridge possono quindi essere persistenti, ed eseguite su thread paralleli. Il client ha poi il compito di recuperare i nuovi risultati, man mano che questi vengono messi a disposizione da SCSQ. Tutte le comunicazioni tra i diversi componenti sono asincrone, e fanno uso di messaggi testuali in formato JSON.

Il prototipo è stato progettato pensando ad un suo utilizzo da parte di un utente singolo, senza funzionalità per la collaborazione con altri utenti, come da specifiche per questo anno di sviluppo del progetto. Per l'anno successivo, tuttavia, è pianificata l'estensione dell'interfaccia per l'utilizzo in un ambiente di lavoro collaborativo, in cui è possibile scambiare e accedere a risorse comuni, tra cui query visuali e dashboard. È inoltre previsto lo sviluppo di un prototipo su altre piattaforme operative, quali dispositivi mobili e tablet. Un ulteriore obiettivo sarà quello di eseguire dei test di usabilità sulle interfacce progettate, così da evidenziarne i punti deboli e apportare dovute modifiche.

Part II

Introduction

The design and the production of innovative and highly specialized products imply a complex lifecycle, starting from an idea which goes through development, manufacture, operation, maintenance and disposal [1].

During the entire lifecycle, many kind of data streams are generated. Among the others, these data streams are:

- streams from sensors and analysis equipment;
- streams of simulation and testing data;
- streams of design data;
- streams of multimedia collaboration data;
- streams of higher level inferred events.

These data streams are created and even consumed in every phase of the product lifecycle.

The European research project Smart Vortex is placed in this context. In the industry, the amount of collected data in each phase of a product lifecycle is rapidly increasing. This leads to issues and difficulties for human operators on analyzing them; sometimes important data are even not processed at all.

To improve performances in the decision making process, the engineering design and the collaborative decision, the Smart Vortex project aims for studying a framework that captures the relevant information and delivers it to the right place at the right time.

The project addresses the problem of the extraction of hidden but useful information from a great amount of data; moreover, a main vision is to improve the collaborative work and the decision making process by sharing this information between people. The collaborative work generates itself data streams that need to be collected and consumed. More generally, the purpose of the Smart Vortex project “...is to provide a technological infrastructure consisting of a comprehensive suite of interoperable tools, services, and methods for intelligent management and analysis of massive data streams to achieve better collaboration and decision making in large-scale collaborative projects concerning industrial innovation engineering” [1].

The Smart Vortex project will produce several components, to implement the needed functionalities. A Data Stream Management System (DSMS) will collect data streams, will perform manipulations on them and will offer functionalities to query them. A rules and policy framework will also be created. Collaborative work is a key point of the project, and sometimes

even several competing industry can collaborate. While working together, the framework will allow control over cross-organizational IPR¹ management. Furthermore, a key component of the project will be a Visual Query Interface to perform querying operations over data streams through a Visual Query Language, easing the task and giving the opportunity to compose complex queries even to non-expert users. The graphical interface will also display the composed queries results with appropriate visualizations, exploiting useful information extraction from data streams.

The work presented in this document is part of this component, that is the Visual Query Interface.

Since this thesis work has been done during year 2 of the entire project cycle, lasting four years, it has addressed the objective of designing a Visual Query Language to easily compose queries over the adopted DSMS, and to visualize their results in a proper way, following project's principles and requirements. A comprehensive user interface has been designed, with tools to compose Visual Queries, define visualizations to display the queries results, and organize these visualizations on a so called "dashboard". A working prototype of this user interface has been then realized, implemented as a web application; the web application communicates with another ad-hoc TCP server, developed in parallel during the same thesis work, which sends queries to execute to the DSMS, and gets results back.

The second part of this document presents a more detailed description of the Smart Vortex project. Data streams are contextualized, and the adopted FDSMS, SCSQ, is introduced, along with its own query language, SCSQL. Project requirements are discussed, with usage scenarios describing a plausible interaction between the system and people in a real work example, as suggested from the industrial partners.

In the third part, an overview of the Visual Query Systems and Visual Analysis tools are presented. A Visual Query System (VQS) can be defined as a system that uses a visual representation for both the domain of interest and the related requests on it. Visual Query Systems are studied since the early 1990s. In the last years there has been an increasing interest, as shown by the amount of published research papers. A theoretical background is provided and some relevant example are presented. Visual Analysis is a field derived from Information Visualization. While Information Visualization is focused on a static presentation of a great amount of data, Visual Analysis focuses on analytical reasoning through visual interfaces. The main concepts are presented and some valuable example are shown.

¹IPR, Intellectual Property Rights.

The fourth part is dedicated to the description of the accomplished results. The designed Visual Query Language is presented, along with its basic constructs and the explanation of design choices. The translation algorithm, from a Visual Query to a textual query written in the SCSQL language, is described, giving a mathematical explanation of it. After that, the developed prototype is presented. A first overview of the adopted technologies and the overall architecture is given, highlighting how the different components interact and communicate. A single section is then dedicated to each component: the Web Application, the FDSMS Bridge and the User DataBase. The Web Application is the real core of the system, which handles the communications between the Web client and the FDSMS through the web server. It also generates the user interface, where a user can compose Visual Queries by the Visual Query Editor, choose how to visualize queries results, and combine different visualizations into a “dashboard”, for monitoring or analysis purpose. The FDSMS Bridge is a TCP server used to execute SCSQL queries on SCSQ (the FDSMS), and gets results back. These queries are the ones composed by the users through the Visual Query Editor, translated into textual queries and then executed; when results are returned from SCSQ, they are displayed on the chosen visualizations. The User DataBase is used to store composed Visual Queries and user-related configurations.

The fifth part summarizes the achieved results and the implemented project’s requirements, with a discussion about design choices and their limitations, addressing possible improvements and solutions to known problems. It ends with a description of the work planned for the next year of the project.

This work is therefore totally related with the Smart Vortex project. Part of the information presented in this document contributed to the composition of the project deliverable’s documentation. Furthermore, the realized prototype has been presented as a work demonstration during the Smart Vortex second year review meeting.

Part III

Smart Vortex

Chapter 1

Project Overview

In the last years there a trend in the industry emerged which fosters the offer of intangible assets within vendor-customer relation even if tangible assets represent a high volume and a very precious part of the delivery. These underlying business models are called “Service Oriented Models” where the product offer is no longer the equipment itself, but the contractual agreement to specific performances of intangible features like “transportation volume, reliability, physical parameters like torque, power, cutting speed or other measurable parameters” which are taken as contractual basis for between vendor and customer. While leasing and rental models still calculate the financing of the used equipment based on price list items which are seen as goods service oriented models define the goods only as means to an end to reach a specific business goal.

From the end 90s first business models based on novel licensing schemas like “Pay per Use”, “Term Deals with License Remix”, “All you can Eat” and other service based licensing models changed the mindset how to make business started in the software oriented IT and CAD/CAM¹ industry and has now found strong interest from the Manufacturing Industry. While software tools with their licenses had the flavor of easy to reproduce assets, that easily can be applied to similar business models with tangible assets like machinery, cars, electronic or other capital intensive goods that have been seen as critical and difficult to handle in service oriented business models. To enter in similar business models in the manufacturing industries requires exact knowledge about the behavior of the goods, equipment or the systems, their usage conditions, early detection of upcoming failure modes or maintenance requirements and dependencies with the delivered systems. Additionally it

¹CAD/CAM are computer aided technologies, specifically Computer Aided Design and Computer Aided Manufacturing.

requires a new way of engineering and design which is called “Functional Engineering” and Design requires much higher efforts in simulation and validation of the goods prior to production to reduce the risk of failing once such business models get applied. One very significant change in the design methodology established functional engineering due to the split of design tasks and input parameters from each other used for the design and construction activities. The use and applicability of the parameters that will be taken for the design requires better knowledge about risk factors, criticalities, boundary conditions and system behavior.

This deeper understanding of functional dependencies is combined with measurement of actual data derived from test systems or out of predicted simulation values which are taken to validate the design and operation behavior of the systems and to optimize the used simulation and design models applying the achieved results. To get representative and exact results either high volumes of actual sensor measurement data or data from a high volume of similar systems need to be reviewed and analyzed. One method which reduces time and efforts is based in a technology where data streams produced either by life operating sensors and or derived from computation activities which have produced extremely large data files are validated during the streaming mode to recognize immediately threshold violations, risk situations, uncontrolled usage patterns which require immediate action and care and uncritical values that can be excluded from further investigations.

A core objective of the Smart Vortex is to effectively use this data for supporting collaboration and to improve the product lifecycle. Sustainable collaboration is a critical skill and competence in organizations. As systems and products become more complex, experts and engineers need to collaborate to manage integration and communication of the subsystems they design and develop. Parallel to this, users, clients and stakeholders are becoming increasingly often included in various phases of a product lifecycle, including design, user feedback, innovation and improvement cycles. Collaboration in the product’s lifecycle can be supported by technology; e.g. information systems for planning and controlling the production process as well as for archiving, administrating and providing product related data. In complex design and engineering phases, different tools and techniques are useful at different points in the design. However, groups of engineers often do not have the knowledge to choose or select effective tools and techniques for specific situations. Furthermore, groups might not have the skills or the knowledge to effectively use them in the appropriate way, in order to get relevant information out of the data streams. Smart Vortex addresses those existing gaps in the collaboration work and will provide intelligent support for the selection

and appropriation of collaboration support tools based on a group's current interaction; it will also enhance each process phase and the availability and accessibility of information.

In the domain of Smart Vortex, organizations are not isolated, but are members of federations of partners in which data is shared and tasks are worked into collaboratively across organizational borders. Nowadays, data exchange between organizations is quite commonplace and explicit calls are used to place orders, to inquire about business deals or to send other data which may be important in a B2B² scenario. However, on the technical side, these calls are often isolated in nature and there is the lack of support for a full-blown middleware solution. Smart Vortex has the vision of a cross-organizational middleware, meaning, a framework which can automatically connect organizations seamlessly, while it can still protect the intellectual property of the partners and mind the different company policies. Instead of proposing a monoculture community, where every partner has to adopt central concepts, every organization is free to use its own terms, definitions and structures which will automatically be translated or mapped for the other partners, or hidden if other members of the federation are not allowed to see the data. Using this mechanism, Smart Vortex allows for a more fluid and agile user experience in collaboration sessions, while it is still able to guarantee the highest standards regarding data protection and privacy.

To improve the acceptance of application engineers that are not programming specialists, easy access to the data which concern the design engineers is required and need to be addressed to speed up the design and validation process. Support given by graphical access methods can solve these deficits and allow the engineers to concentrate on their mainstream activities of design and validation. Smart Vortex addresses those existing gaps in the current design methodology and will open the door to novel service oriented business models reducing risk for the industries and minimizing the resource consumption to make the European industries more competitive.

²Business-to-Business, commerce transactions between businesses.

Chapter 2

Main concepts

2.1 Meta-data

Meta-data, often referred to as “data about data” can be defined as “...structured information that describes, explains, locates, or otherwise makes it easier to retrieve, use, or manage an information resource.” This can for instance be information about the origin of the data (the data source), the purpose of the data, the time and date of creation, etc.

2.2 Database Management System (DBMS)

A Database Management System (DBMS) is a system software for scalable management of large data volumes usually stored on disk.

DBMSs can be categorized according to the DataBase model, which is a type of data model that determines the logical structure of a DataBase, and fundamentally determines in which manner data can be stored, organized, and manipulated. The most popular example of a DataBase model is the relational model. DBMSs also provide methods to maintain the integrity of stored data, running security and users access, and recovering information if the system fails.

DBMSs provide for languages and components to search and update DataBases, as well as for storing meta-data called the DataBase schema about the stored data.

2.3 Query

A query is a request for the retrieval of some data from a DataBase. A query is executed immediately to retrieve the desired information, for example: “How many machines of type X located Y have installed sensors of type Z?” This is called a passive query since the system processing the query passively waits for users to send the queries for execution.

The execution result of a query through a DBMS returns a table of tuples reflecting current state of the DataBase tables.

2.4 Query Language

Queries are expressed in terms of some query language. For example, queries to relational DataBases are usually expressed in the query language SQL, while queries to RDF based information models are usually expressed in SPARQL. Queries provide for users and programmers a very general way to specify data selections (filters), combinations (joins), and computations over data stored in DataBases.

2.5 Data Stream

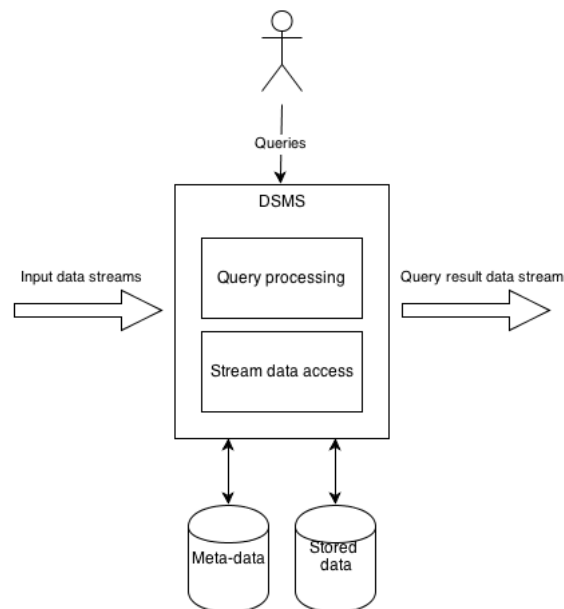
A data stream is a continuous flow of tuples (events) of measurements from some artefact. A data stream can be seen as an ever growing sequence of tuples. A data stream can be live in case it is communicated directly from its source without being stored on some media. At a given point in time a user can save a snapshot of the current state of a stream on disk or some other media.

2.6 Data Stream Management System (DSMS)

A Data Stream Management System (DSMS) is similar to a DBMS with the difference that while a DBMS allows searching only stored data, a DSMS in addition provides query facilities to search directly in data streaming from some source(s). therefore, the result of a DSMS query can be not only a set of tuples as in SQL, but also a potentially infinite stream of tuples.

The argument about using a dedicated DSMS, and not a traditional DBMS, is that DBMSs are not able to efficiently, or at all, handle large amounts of streaming data and can be greatly outperformed by a DSMS [2].

Following figure shows the main building blocks of a DSMS:



2.7 Continuous Query (CQ)

Queries that involve streams are called Continuous Queries (CQs).

DSMSs could execute a continuous query that is not only performed once, but is permanently installed. therefore, the query run indefinitely, or until they are terminated, while conventional queries are executed on demand and run until all requested data is delivered. The result of a CQ is a stream itself, which is continuously updated as new data appears in the queried stream(s).

A conventional DataBase query executes once and returns a table of tuples. Each row in a DataBase table is called a tuple. Analogously, an item in a data stream is also called a tuple, but unlike a conventional DataBase query, the result of a continuous query is a stream.

An example of CQ where some machines continuously deliver streams of temperature readings may be: “Continuously show me the temperature readings for sensor X on equipment of model Y operating at 20% higher temperature than what is recommended.”

Data streams are often of such a high rate that saving them to disk is not desirable or feasible. Furthermore, results of CQs have to be delivered as soon as possible, putting requirements on the response time. In many cases, the

applications require non-trivial analysis, leading to CQs involving expensive processing. To provide scalability of such expensive CQs over high-volume streams, the execution of the CQs must be parallelized.

When optimizing one-time queries, the query optimizer may use meta-data and statistics on the tables. In the same fashion, a CQ optimizer may use meta-data and statistics on the data streams. An executing CQ plan continuously reads input data streams and may access stored data (see figure above).

2.8 Continuous Query Language (CQL)

A Continuous Query Language (CQL) is a query language with specific operators for Data Streams.

Different CQLs have been proposed, like CQL from Stanford [3] or Stream-SQL [4] to name a few; however, they all share the same operators. A common approach taken by several DSMS systems, in processing data streams, is to form sliding windows upon the data stream [3].

The sliding window can be thought of as view upon a stream that, at any point in time, reflects the viewed part of the stream as a finite relation. As time flows and new tuples arrive, the window moves over the stream and the content of the relation is changed to reflect the current view. This continuously changing view is called a continuous relation, and by being finite it can be processed by relational operators. Continuous relations are created by windowing operators; the most relevant ones are time, counting, and partitioned windowing operators.

All windowing operators are common in that they specify a window size and a window slide. For a time window size and slide are specified in terms of time units. The sliding window may contain any number of stream tuples as long as the tuples' timestamps (all tuples have an associated timestamp) are larger than the window starting timestamp, and smaller than or equal to its ending timestamp ($\text{startingtimestamp} + \text{size}$). When the window is full, i.e., a new stream tuple has a timestamp larger than the window ending timestamp, the window increments its starting timestamp by slide. For a counting window size and slide are specified as numbers of tuples. That is, the maximum number of tuples that a counting window may contain is size. Once full, the window drops slide of its oldest tuples. For a partitioned window size and slide are also specified as numbers of tuples, as in the counting window. Unlike counting window, a partition window will split the stream into sub-streams such that each is uniquely identified by one or several stream

tuple attributes - much like the GROUP BY operator found in the relational algebra. Each sub-stream will then be processed by a counting window using the given size and slide. Finally, the resulting continuous relation is formed by taking the union of all counting windows. The partitioned window operator is often used to construct continuous relations on which grouping and aggregation can later be meaningfully applied.

Chapter 3

Data Streams

Along a McKinsey report on “Big Data”, analyzing large data sets will become a “key basis for competition, underpinning new waves of productivity growth, innovation and surplus as long as the right policies and enablers are in place”. McKinsey sees the power of “innovating new business models, products and services” and “supporting of human decision making” in the computation of “Big Data” [41].

A similar development can be seen regarding data streams. In industrial applications high volume data streams are generated by simulations, sensors or actuators etc. in for instance machining operations or collaboration processes, which can interact with the previously mentioned data streams.

The goal of Smart Vortex is to provide a technological infrastructure consisting of a comprehensive suite of interoperable tools, services and methods for the intelligent management and the analysis of massive data streams produced in all phases of the product and design life cycle, to achieve better collaboration and decision making in large-scale collaborative projects concerning industrial innovation engineering. Data streams will be used for validation, behavior prediction, simulation and simulation model improvement. This will be done during all the phases of the product life cycle, and especially for the steady improvement of the offered services, combined with product enhancements. These improvements are used to perform those service offers and collaboration data to make design decisions traceable, and also to improve the communication between the stakeholders.

Both data stored in repositories and produced in real-time as data streams need to be processed. A particular kind of repository data is meta-data describing properties about artifacts (machinery, collaborations, simulations, etc.). Operational data from regular enterprise databases are used by Smart Vortex for example for accessing customer, pricing, materials, or marketing

data. If desired and possible, streams can be saved in a repository as stored streams by a program (e.g. a simulator) and replayed later.

Data streams can either be raw data streams produced by some artifact or human, or derived data streams whose data is continuously computed based on other data streams in combination with product meta-data and data stored in repositories. Derived streams can contain both continuous and discrete values. An example of a continuous value is the power consumption of some equipment in use estimated by a mathematical model over other data streams and product meta-data. A discrete value can be, e.g., an action event generated when non-expected behavior of equipment is inferred by a validator.

As said, a central technology in the Smart Vortex project is the ability to search and analyze high volume data streams in a distributed environment by means of a DSMS. The searches and analyzes are specified using CQs expressed in a query language. Other aims of CQs are, for instance, monitoring of critical parameters of in-use products, monitoring how operators use the equipment, and supporting collaborative interactions.

CQs over DSMS are different from conventional DataBase queries written in, for example, SQL, where a query requests data from tables stored in the DataBase. The result of a DSMS query can be not only a set of tuples as in SQL, but also a potentially infinite stream of tuples (often called events). Furthermore, stream queries are CQs in that they run indefinitely, or until they are terminated, while conventional queries are executed on demand and run until all requested data is delivered. In the DSMS, values are computed by procedures based on stored rule sets and stored metadata.

In Smart Vortex, the DSMS technology is built in as a federated system which gives higher flexibility, scalability and enables parallel computing, while providing integration with systems that rely on DBMSs like PLM, ERP or other relational oriented systems. Furthermore, DSMS functions can be installed in a distributed environment.

The chosen federated DSMS for the Smart Vortex project is SCSQ, which has the ability to perform expensive computations for decision support in real-time over streams, and adopts SCSQL as CQ language.

3.1 SCSQ

Super Computer Stream Query processor (SCSQ) [5] [6] is a DSMS prototype developed at Uppsala University where the CQs are specified in a query language called SCSQL (presented later) that includes types and operators

for sets, streams, and vectors. Vector processing operators enable queries to contain numerical computations over the input data streams. Composite types are allowed, which enables useful constructs such as vector of stream. The system is extensible so that programmer can define customized data structures and query operators. Furthermore, the query language is extended with Stream Processes (SPs) and parallelization functions, which allow the user to specify customized parallelization and distribution of queries. Parallel computations were defined as sets of parallel sub-queries, where each sub-query executed on one SP.

SCSQ is implemented over Amos II (Active Mediator Object System) [7], which is an extensible mediator DataBase system allowing different kinds of distributed data sources to be queried. This system is centered around an object-relational and functional query language, AmosQL (presented later). Amos II can store data in its main-memory object store. Furthermore, wrappers can be defined for different kinds of data sources and external storage managers accessed to make them queryable. Several distributed Amos II peers can collaborate in a federation.

SCSQ extends Amos II in the following ways [8]:

- Stream query coordinators start parallel processes dynamically.
- SPs provide mechanisms for iteration over streams in a distributed environment.
- Primitives for network stream connections provide an infrastructure for communicating SPs.
- Numerical vectors represented in binary form, and functions operating over these vectors, provide efficient processing of stream tuples.
- Postfilters extend stream processes by reducing and transforming their output streams.
- Query language parallelization functions provide declarative parallelization of CQs.
- Physical windowing functions provide network bound data stream rates between stream processes.
- Performance tools allow profiling of parallelized query execution.

The superior continuous query processing rate of SCSQ is enabled by the combination of two key technologies: parallelization of stream splitting in

conjunction with the use of physical windows. Parallelizing stream splitting speeds up the distribution of data in a federated DSMS, whereas the use of physical windows saves communication cost.

3.2 AmosQL

AmosQL [9] is an object-relational and functional query language.

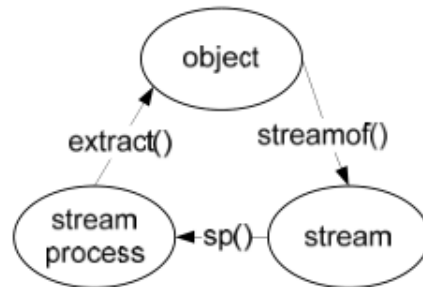
Data, and information in general, is represented in AmosQL through types. As in object-oriented programming, a type has attributes that describe the type itself, and associated procedures known as methods. types can have an inheritance relationship with other types; having defined a type, other types can inherit attributes, methods and behavior from this type, called base type, supertype, or parent type. The resulting types are known as derived types, subtypes, or child types. The relationships of types through inheritance gives rise to a hierarchy. Given a certain data representation through types, instances of these types are called objects.

AmosQL is based on the function composition notion, and even types attributes and methods are represented as functions themselves. AmosQL has a set of predefined functions, but new ones can be defined through proper statements. These functions can be then composed in order to fulfill user needs.

3.3 SCSQL

SCSQL extends AmosQL with stream and parallelization primitives. In summary, SCSQL is an extensible query language for both stored and streamed data. SCSQL allows continuous and ad hoc queries over these data sources to produce derived streams. The foreign function interface of SCSQ allows any external data and processing to be plugged into SCSQ. Finally, the parallelization functions of SCSQL allow stream processing to be massively parallelized.

All data in SCSQ is represented by objects in SCSQL. The relation between Streams, Stream Processes and Objects in SCSQL is illustrated in the following figure:



A stream is an object that represents (possibly unbounded) sequences of any kind of objects, a bag represents relations, and a vector represents bounded sequences of objects (for example, vectors are used to represent stream windows, and vectors of streams are used to represent ordered collections of streams). The result of a continuous subquery is a stream. Continuous subqueries are assigned to SPs. Users of SCSQL define parallel and distributed stream computations by assigning continuous subqueries to SPs.

An important property of SCSQL is that it is extensible so that programmers easily can define own foreign functions in some conventional programming language such as C or Java. For example, new kinds of stream event formats produced by particular kinds of communication protocols can be accessed through foreign functions and used in continuous queries. Data filtering or mining algorithms can be defined as foreign functions and used in queries.

Another central feature of SCSQL is the facilities to define distributed and massively parallel queries. The query language includes stream processes (SPs) and parallelization functions, which allow the user to specify customized parallelization and distribution of queries. This is enabled by high-level primitives for scalable stream splitting and for specifying distributed and parallel computations [5] [10].

Chapter 4

Requirements

One of the goals of this framework is to facilitate the use of data to be processed by an analyst to ease the decision-making process. As the collaboration data streams contain high amounts of information (coming from the raw data) it would be very complex for a human analyst to process all of the raw data without any filtering mechanism that can transform the high amounts of data into processed information, reducing its volume. The transformation and aggregation of raw data into more useful information is automatically performed, as explained, by the DSMS, applying proper functions and filters to reduce its volume. For example, the system is able to calculate from the data streams the fuel's lifecycle usage of an excavator not just its statistics for hours or days.

At an higher abstraction level, however, users have to take decisions and perform tasks based on analyzed data. This can be achieved through the support of a graphical interface, and appropriate tools, which aid the user to reach her goal.

Industrial partners have provided, during a first analysis phase, a description of their requirements, taken as a starting point for the creation of different use cases. These use cases have been taken into consideration during the user interface requirements gathering phase.

4.1 User interface requirements

The user interface of the system should provide the end-users the resources to collaborate on, consume and reason about the raw and derived data streams as well as to manipulate and create new data stream computation instances. Using graphical and multi-modal query languages, the complexities of the

underlying, textual query language are hidden from the end-users for a more intuitive access to the information contained within the data streams (figure 4.1).

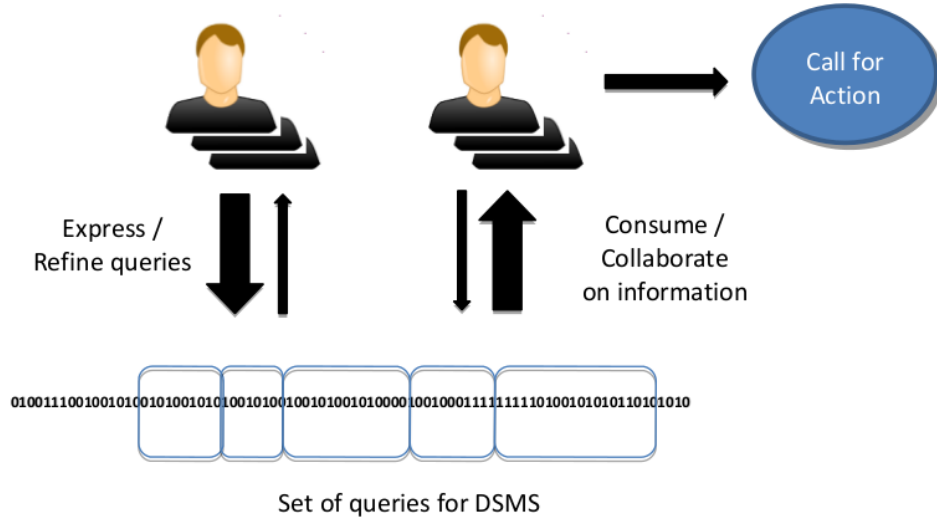


Figure 4.1: Multimodal Interaction with Data Streams.

To achieve these goals, several requirements are identified and categorized according to different parts of the Smart Vortex project. Looking at the Smart Vortex Framework architecture (figure 4.2), the subproject of interest is the one related to the graphical user interface, for the query writing process and the subsequent query results consumption: the Visual and Multimodal Query and Data Presentation macro-component.

The interaction conceptualizations are to be integrated into a collaboration tool, for multiple end-users with their roles, to reason about the information and coordinate and agree upon and take actions. Furthermore, part of the user interface will use appropriate modalities to proactively notify end-users about noteworthy occurrences regarding the system's state with respect to the users' observable activities. The collaborative visual environment should enable two different types of interactions: provide support to synchronous collaboration between co-located users and asynchronous distributed collaboration between remote users. To support these tasks and enable collaboration among different workers, the user interface has to identify and provide accessible, flexible and editable artifacts for both individual-based and group-based exploration of data streams. This requires the definition of a group interaction space for collaborative data analysis, where users can interactively access data streams, as well as create, layout, arrange and share

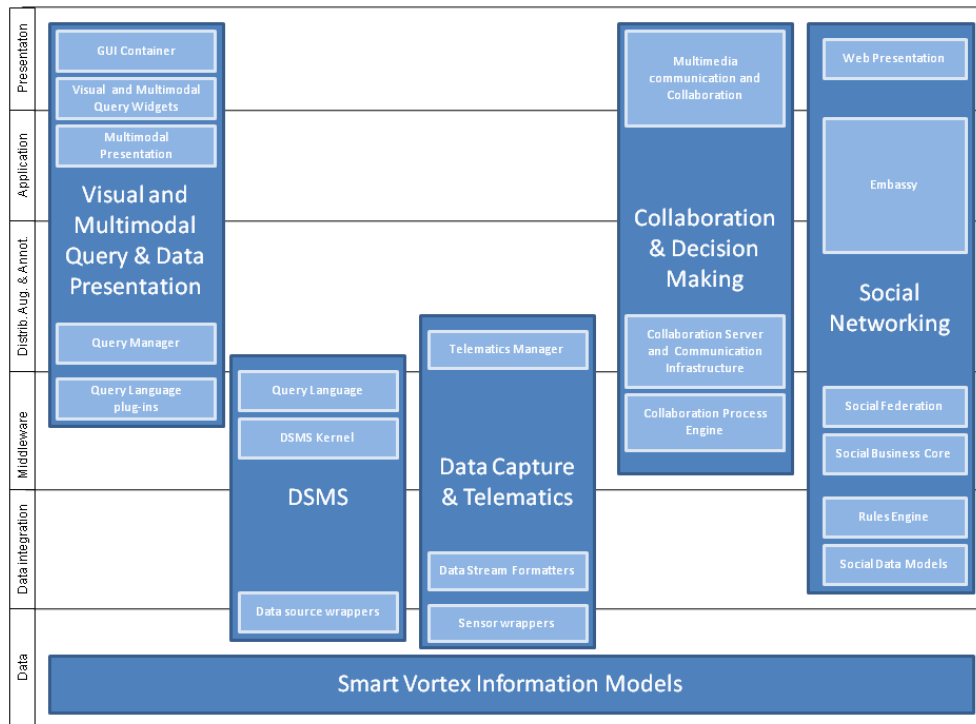


Figure 4.2: Smart Vortex Framework architecture.

artifacts connected with data streams. Each artifact should be enriched with contextual information (such as labels, scales, legends, etc.) that allows users to correctly interpret the information being displayed. Users collaborating in a shared group interaction space should be able to jointly interact with a single shared view of data streams, or to create different views of the same data streams in a personalized workspace that can optionally be shared with other participants and mapped to, or merged with, other views. By enabling the sharing and coordination of multiple views of data streams in a common workspace, the visual environment allows users with different backgrounds and experience to reason on complementary data stream representations.

Additional requirements, particularly targeted to asynchronous distributed collaboration, include the ability to:

- allow users to trigger conversations and remote social interactions via the provided collaboration tools;
- use notifications of actions performed by other users (edited views, updates in artifacts, requests or comments issued by other users, etc.);

- use metaphors to represent the timing of performed actions and to access and display the actions' history for the artifacts;
- provide representations of user profiles, including their roles, backgrounds and skills (as provided by social networks).

Moreover, the interface should provide users having different skills, roles and capabilities with the possibility to explore and analyze complex data streams with little or no assistance from technical practitioners. To effectively support inexperienced or non-technical users, display and interaction techniques should partially or totally automate the selection of metaphors and interaction modalities, in order to constrain the configuration parameters that users have to define. On the other side, experienced users and technical people should be able to fine tune the appearance and behavior of their displays via the user interface.

As collaboration practices often include mobility and remote interaction, the user interface layer should also provide representations and interaction models for pervasive computing scenarios, where multiple devices with different capabilities (e.g., in terms display size, screen resolution, available devices and computational resources) and demanding little or no technical knowledge (e.g., PDAs and tablet computers).

Apart from all technical features mentioned above, the user interface should support also the main marketing related user criteria such as:

- easy to learn with as much as possible intuitive handling elements;
- fast reaction capabilities to risk situations;
- easy adaptability if extensions are required.

More specifically, the graphical user interface offers to the end user two main features: the creation of data processing instances as queries over the DSMS, and the consumption of the information delivered by these queries.

Hiding the complexity of the textual query language potentially limits the expressiveness for queries to be expressed. The available semantic subset of the textual query language is a function of the employed modality. To select meaningful subsets requires the identification of suitable trade-offs between expressiveness and ease of use for the different modalities and user's expertise. Similar considerations apply to the consumption of the information from the expressed queries. The different modalities and employed interaction techniques determine the dimensionality and frequency of information from the

DSMS to be perceived and processed by the user with acceptable cognitive load.

The visual and multi-modal query language should support the access to data streams (and related metadata) for both local and remote users in various formats. It should allow users to define queries over data streams, parametrize them and create and position resulting views in the personal or shared workspace. Query building and configuration should be based on conceptualizations with a formal syntax and semantics to be matched with the syntax and semantics of the underlying textual query language. The query building environment has to provide visual and multi-modal tools that allow unskilled end-users to perform data stream queries, navigate the results and retrieve other hidden relevant information. Graphical and multi-modal primitives, as well as basic visual and multi-modal queries, should be made available via a user query library extendable by the user by adding and storing composed queries. The environment should enable multiple combinations and compositions of conceptualizations to reflect advanced query constructs and allow users to aggregate/disaggregate, filter and browse data in order to focus on relevant information. Depending on the textual query language capabilities, visual and multi-modal query formulation tools should allow both intensional and extensional queries operating on the data stream model or their representation.

Despite the requirements for the user interface refer to a collaborative environment, the objectives for year 2 of the project (the one this work refers to) are for software components available in “single-use mode”, i.e., with no collaboration/sharing features. For this reason, all references to the collaborative environment have not been taken into consideration, except for those ones presented in the meta use case (presented in the next section).

Furthermore, only a web-based environments is targeted, and a web application is to be designed.

4.2 Use cases

The Smart Vortex project handles some aspects of data stream management in industrial environments within three integrated sub-projects (ISP-1/2/3), based on the industrial partners' visions of the innovative Functional Product business model.

For each sub-project, some general use cases, envisioning possible working scenarios, have been proposed. The selected use-cases provide a high inno-

vation potential in the areas of the following main topics:

- Processing of (i.e. search in, analysis of , and computation on):
 - Large data streams generated by sensors in equipment, simulations, collaboration activities (cross organizational, cross team, and cross domain) etc.
 - Inferred data streams, i.e. resulting from processing other data streams.
 - Non-streaming data sets generated in product development or product life-cycle management.
- Visualization of the computational results like trend, risk and threshold validation in a new graphic interface also supporting querying data streams.

In order to identify the users' needs and the related functionalities, a meta use case upon the existing ISPs is created. This fictive use case describes a hypothetical work session; it also helps to figure out how users interact with the system and in which role. There can be different kinds of users and they may need different functionalities. The system analysis is then applied to the three ISPs and their respective use cases.

Only a brief description of the use cases is given. A deep insight in the use cases can be found in [42].

4.2.1 ISP-1: Data streaming computation from stored data

Use-case 1: Data streams derived from complex simulations (Use case provided by Fe-Design GmbH)

In this use case, frequency oriented trend calculation are done about complex stress, vibration or sound simulations. Possible examples of actions possible through the visualization functionalities: prediction calculation out of trend analysis on relation to expected overshoot of given thresholds in stress, sound or vibration, stop of current simulation run; adaptive visualization of prediction and restart of the simulation with new parameters if overshoot is predicted, etc.

Use-case 2: Data stream computation of data streams that have previously been stored in file sets (Use-case provided by Hägglunds Drives and Volvo CE)

Due to missing communication infrastructure or expensive connections, it can happen that both Hägglunds Drives and Volvo CE store the data in either large file repositories or dump them from random access memories [42][43].

In this scenario the main difference with the other ones is that the data are not transferred in real time, although the functionalities needed for the data analysis have no differences with the use cases presented in ISP-2.

4.2.2 ISP-2: Computation of real-time data streams**Use-case 1: Real-time data streaming and computation from actual sensor data in hydraulic systems (Use case provided by Hägglunds Drives)**

Hägglunds needs to monitor their systems to learn about system performance and avoid unplanned stops. Data streams from sensors have to be analyzed in combination with derived data. Data streams will be initially stored at a service provider, and then the data will be delivered to the data stream management system. The result of the analysis process is a cockpit chart which allows for an overview of many machines' behavior in a limited space. A list of functionalities and rationale is provided. Especially about the visualization features and the visual query editor, some functionalities will be added or modified when the selected query language and a set of query is available.

Use-case 2: Real-time data streaming and computation from actual sensor at milling tools (Use case provided by Sandvik Coromant)

At this stage we refer to the functionalities needed in the use case 1. The kind of data is the same and a list of query is not yet available; in this use-case we assume to implement all the functionalities identified in the use case 1.

Use-case 3: Real-time data streaming and computation from actual sensor in construction equipment (Use case provided by Volvo Construction Equipment)

At this stage we refer to the functionalities needed in the use case 1. The kind of data is the same and a list of query is not yet available; in this use-case we assume to implement all the functionalities identified in the use case 1.

4.2.3 ISP-3: Data streams within collaborative environments

In this fictive use case, Sandvik, Hägglunds and Volvo CE would work in a collaborative environment to design and develop functional products. Following, a list of capabilities that will be implemented to allow users to cooperate in a collaborative session, while the visualization and query tools functionalities are the same as in the other ISP.

4.2.4 Meta Use Case

An “administrator” needs to set up the system, before engineers can start to work. The administrator can manage both users and analysis tools and creates users’ rights and permissions to access different parts of the system as well as data sources. The administrator also manages the collaborative environment and the related issues, e.g., who can create new collaborative sessions, who can participate, who is able to share or edit documents, etc. The administrator can also create an initial set of queries (or query templates) through a visual query editor and store them in a query repository.

The visual query editor provides many graphical icons representing different kinds of operators, divided in palettes according to their operator category such as spatial, logical, mathematical or comparison. Once the system is configured, the engineers can access it using their user credentials. The end users (referred to as engineers in the following, but they could as well be consultants, etc.) can start their work session, load a query stored in the query repository, modify it, or write a new query through the visual query editor. Engineers have the possibility to visualize the results of queries in several ways since several visualization modalities are provided, e.g., canonical charts (bar chart, histogram, line chart) or combinations of those (cockpit chart), and advanced features like zoom to a specific data source. This

process can be repeated to further investigation, modifying for example the numeric values of the query.

Engineers can also start or join a collaborative work session, depending on their permission. In a collaboration environment it is possible to cooperate in several ways, such as sharing the desktop, sharing or editing a document, creating or joining a conference call, and all the other available modalities of interaction. The improvements made in a collaborative session can be stored for analysis of further individual contribution and work progress.

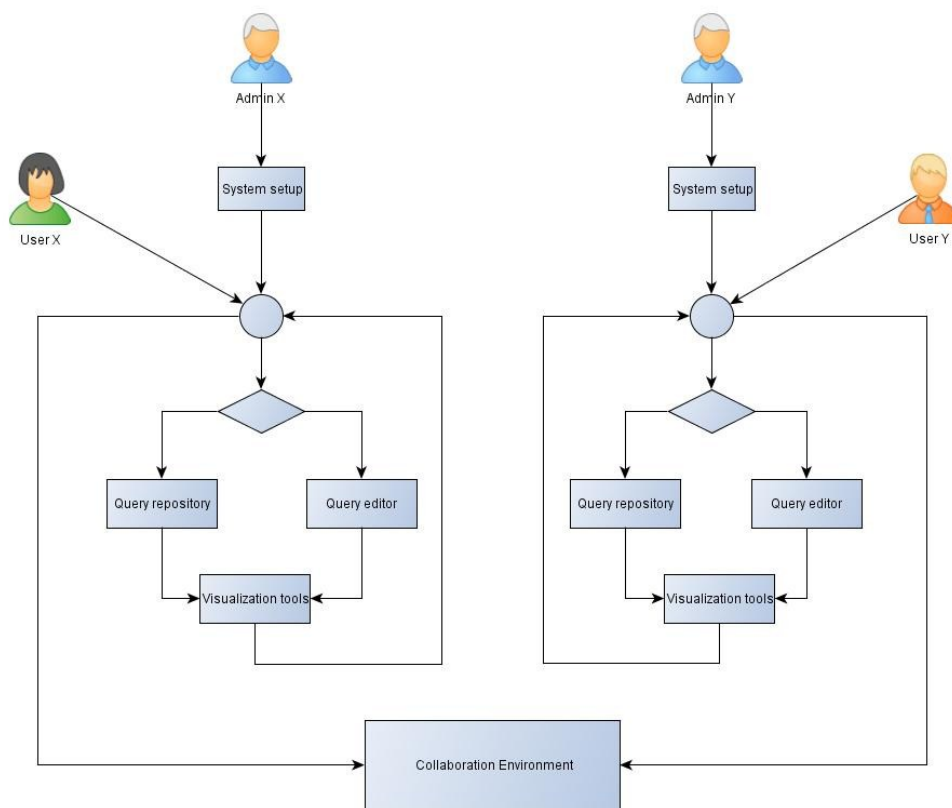


Figure 4.3: Meta use case and roles in the Smart Vortex system.

Starting from this meta use case, the User Layer functionalities can be analyzed and identified. These functionalities, presented below, are taken as a design model for the prototype developed during year 2 of the project. Notice, like already mentioned, that collaboration functionalities, even if described in this meta use case, are not taken into consideration during this research and design phase.

PCA-3 Visualization Functionalities (PCA-3_VF)

PCA-3_VF01	Line Chart Visualization
Rationale	The interface shall provide a line chart visualization to show, for example, the system behavior over the time.
PCA-3_VF02	Histogram Visualization
Rationale	The interface shall provide a histogram visualization to show the distribution of continuous data.
PCA-3_VF03	Bar Chart Visualization
Rationale	The interface shall provide a histogram visualization to show the distribution of discrete data.
PCA-3_VF04	Graphical Data Source Representation
Rationale	The interface shall provide an intuitive graphical way to represent the data sources in the query formation process. E.g., a graphical representation of the sentence “all the sensor X on the machine Y in the country Z”
PCA-3_VF05	Zoom Visualization
Rationale	Users shall be able to zoom in on a desired part of the visualized data (e.g., once showed the n machines as result of a query, it shall be possible to zoom and visualize the overall status of that machine).
PCA-3_VF06	Cockpit Chart
Rationale	The interface shall provide a cockpit chart as result of the data stream analysis process. Through the cockpit chart it is possible to have an overview of the system, monitor many machine/sensors and many historical trends at the same time combining many visualization modalities.

PCA-3 Query Functionalities (PCA-3_QF)

PCA-3_QF01	SPARQL Visual Query Language
Rationale	The interface shall provide a SPARQL Visual Query Language to query the RDF model.
PCA-3_QF02	SCSQL Visual Query Language
Rationale	The interface shall provide a SCSQL Visual Query Language to pose queries to the DSMS.

PCA-3_QF03	Query Repository
Rationale	The interface shall allow users to easily load queries saved in the repository, providing also an intuitive visualization to identify the desired query.
PCA-3_QF04	Visual Query Editor
Rationale	The interface shall provide a Visual Query Editor where users can write new visual queries and also modify a query stored in the repository.
PCA-3_QF05	Visual Spatial Query Operators
Rationale	The interface shall provide Visual Spatial Operators to express spatial constraints on the data sources.
PCA-3_QF06	Visual Time Query Operators
Rationale	The interface shall provide Visual Time Operators to express time constraints on the data sources.
PCA-3_QF07	Visual Logical Query Operators
Rationale	The interface shall provide Visual Logical Operators to express logical constraints on the data sources and spatial and temporal query operators.
PCA-3_QF08	Visual Arithmetic Query Operators
Rationale	The interface shall provide Visual Arithmetic Operators to express arithmetic and statistical constraints on the data sources.
PCA-3_QF09	Visual Comparison Query Operators
Rationale	The interface shall provide Visual Comparison Operators to express comparison constraints on the data sources.

PCA-3 Administrator Functionalities (PCA-3_AF)

PCA-3_AF01	Create user
Rationale	The administrator shall be able to create new users.
PCA-3_AF02	Manage user
Rationale	The administrator shall be able to modify rights and permission of each user.
PCA-3_AF03	Write new visual query
Rationale	The administrator shall be able to write a new visual query and store it in the query repository.

Part IV

State of the Art

Chapter 5

Visual Query Systems

A Visual Query System (VQS) can be defined as a system that uses a visual representation for both the domain of interest and the related requests on it. The first graphical query language was introduced in the mid-1970s with the name of Query By Example (QBE). A wide range of implementations were built using the QBE concepts and there are several tools using this paradigm today. Two forms of query creation (SQL and QBE) were compared through an experiment [11].

The authors found that the time requested for query formulation applying QBE-based approach was shorter than the time requested using a SQL approach; the participants were also more comfortable during the creation process while adopting the QBE paradigm instead of the traditional SQL language. Interestingly, there were no remarkable differences regarding the accurateness of the queries between the two approaches. Even considering some limitation of the experiment (e.g., the choice of the participants can be questionable, QBE does not cover all the possible visual approaches and visual query languages properties), it is quite clear there are some advantages using a visual approach while writing queries.

A general overview and classification of VQSs can be found in Catarci et al. [12].

According to the visual representation adopted for the DataBase and the queries, VQSs are categorized into Form-based, Diagram-based, Icon-based or a combination of them. A Form is a generalization of a table, and it is possible to represent relationships among cells, subset or the overall set, allowing a three level answer. There are VQSs in which it is possible to manipulate both the intensional and extensional part of the DataBase, focusing on different parts of the DataBase. Diagrams are frequently adopted in VQSs, and they generally use some visual components (shapes, colours,

arrows) that are univocally mapped into a concept. In an Icon-based system, there is a mapping between a real concept or analogy and an icon that hides the schema of the data. It is possible to query the DataBase combining icons according to spatial concepts. The main issue while designing an iconic system is to define a non-ambiguous mapping; different attempts are made to find a common mapping, but there still are not any universal standards. Another possible categorization is made by considering the strategies to understand the reality of interest. The filtering of the information of interest can be accomplished using a top-down strategy. The implementation can be made in several ways: iterative refinement, selective or hierarchical zoom, or user-system dialogue. A different approach is browsing, which enables getting more knowledge by exploring the neighbourhood concepts. Browsing can be specialized in extensional, intensional or mixed browsing. An alternative approach is schema simplification, which "brings the schema close to the query". This can be done by transformations of concepts of the original schema in a user view, which can not be extracted by the original schema. Transformations are made to produce a better query representation. The Visual Query Languages are also classified according to the query formulation strategy. In a schema navigation strategy the user starts from a concept and can reach the other concepts of interest. There can be different paths to navigate the schema. The first possibility is to use an arbitrary path to explore the schema, reach the concepts of interest and apply condition on them. It is also possible to select one concept from the DataBase and then navigate the schema by a hierarchical view build using the chosen concept as a root. Moreover, users can choose the start concept and then build their own relationships. A second possible strategy in the query formulation process is by using sub-queries. This can be done following two approaches: by composition of concepts, usually in iconic based languages in which several icon are combined to write the final query, or using stored queries previously created or stored in a system library. Another strategy for query formulation is by matching, which can be done by example or by pattern. In a matching by example strategy, users can provide an example of a query and the system generalizes the example and builds the query. Using a pattern matching strategy, the system searches in the DataBase for a pattern specified by the user. The last strategy for query formulation is by range selection, where it is possible to specify a range on different data-set through graphical widgets.

5.1 Query By Diagram

5.1.1 QBD*

An example of diagram-based Visual Query Language is QBD*[13].

This system is an Entity-Relationship (E-R) Oriented Data Model, which provides a relationally complete query language. The graphical interface, which is the same for both schema specification and query formulation, relies on a language that allows also recursive queries. The main architecture is composed by three main modules: the Graphic Interface, the Translator and the DBMS Interface. Users can interact with the graphical interface in four different ways. In the “E-R schema library” there are the schematas of the applications, and in the “E-R schema user library” there are the user views of the schemata stored in the schema library. For each schema in the schema library, there is also a set of schematas at a higher abstraction level stored in the “E-R top-down schema library”. Users can store graphical queries in the “user query library” and reuse them when needed. In the second main module, there is a translation from graphical queries into relational algebra, or into suitable programs if the original query is a recursive query. Following, the DBMS Interface translates the relational algebra into a query in the underlying DataBase language. In QBD*, the query formulation process has different interchangeable steps. Firstly, a user can explore the conceptual schema by a top-down browsing mechanism. Secondly, using graphical primitives called location primitives it is possible to focus on the subschema of interest. This can be done by direct extraction, expressing query on the schema or using the schemata stored in the library. Further, it is possible to manipulate the schema by graphically replacing primitives bringing it “close to the query”. After this transformation, the schema can result in a different schema, which is non-isomorphic to the original schema. Subsequently, the query is completed by graphical operations such as navigation or selection upon the DataBase schema. In QBD* it is only possible to define queries on the DataBase schema, that means defining on the intensional level.

5.1.2 MURAL

A similar approach to QBE can be found in the Visual Query Language MURAL[14],

which is intended for multiple data sources integrating different types of data. The main objects are entities and relationships. The entities are all different data sources, and the relationships represent ways of correlating data sources.

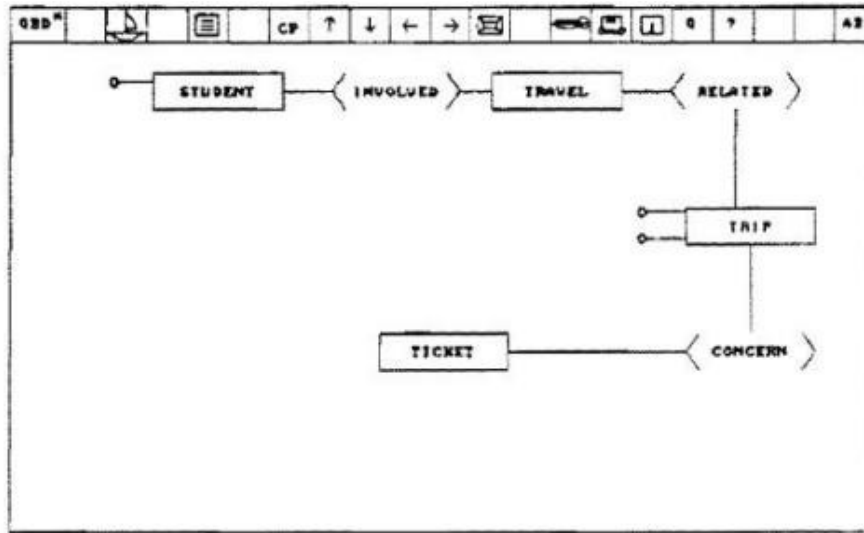


Figure 5.1: QBD* schema representation.

An entity can represent a tuple from a relational DataBase, an object from an object DataBase or a C++ or a Java object. Each entity has several fields defined over a set of domains like primitive type of data (strings, integers) or references to other entities. A relationship can be a simple way to relate entities in one set with those in one other or a more complex relationship. MURAL introduces several concepts in order to facilitate the creation of complex queries: the notion of combination of both entities and relationships to express: “AND” “OR” conditions, restrictions and fields. It is also possible to save a query for a sub-model, so that users can instantiate such sub-models as needed for building new complex queries. The expressive power of MURAL is at least the same of the relational algebra or calculus.

5.2 Query By Icon

5.2.1 QBB

A framework developed following the paradigm Query By Browsing (QBB), which allows both intensional and extensional queries, is described in the work by Polyviou et al. [15].

This framework adopts the same metaphor as most current operating systems, i.e. the desktop paradigm and the related concepts like folders, documents and applications. These objects are displayed in a tree mode, starting

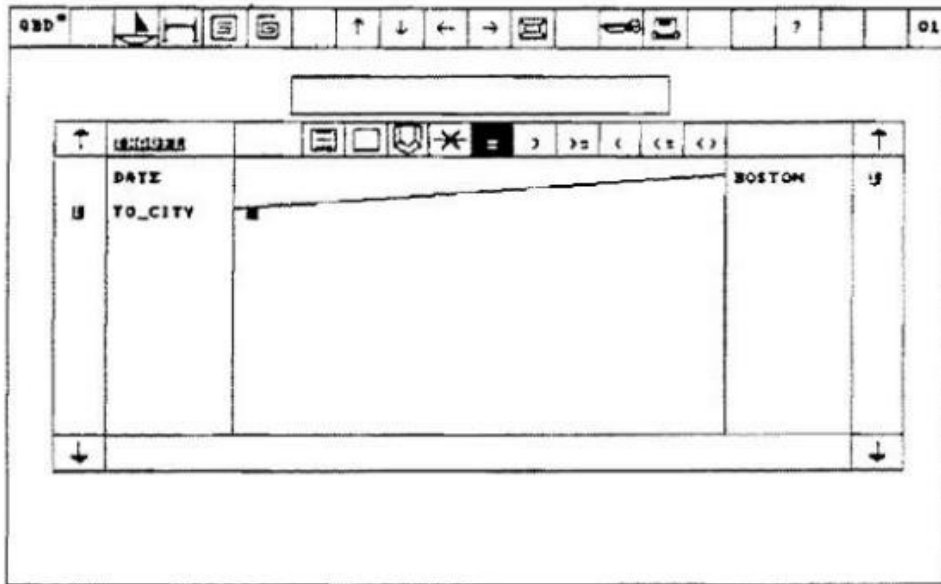


Figure 5.2: QBD* interaction.

from a root folder with subfolders. In QBB, both the schema and query are represented by a folder hierarchy. The concept of a folder is highly related to the table of a DataBase, and the subfolders are all the folders related to the parent folder. Documents and applications are views on the data. Documents are used to display the data whereas the manipulation of the data, such as insert or delete, is performed by the applications. By filtering, a special kind of application, it is possible to restrict the record in the parent folder. There are filter templates which are highly related to the SQL predicates, but it is also possible to build custom filter for other data types. In QBB the distinction between the DataBase navigation and the query formulation is represented by the activation of a folder; by an implicit or explicit activation it must be clear whether the folder is involved in a query or if it is only being browsed.

5.2.2 QBI

QBI [16] is a pure iconic Visual Query Language, which provides tools for an intensional browsing of DataBases. A user can formulate queries without knowing the underlying structure of the DataBase and the path specification, plus a single icon hides the path expression that is automatically generated by QBI. The external view is made up with only two concepts, that is to say a class of objects and attributes of a class. The entire DataBase is

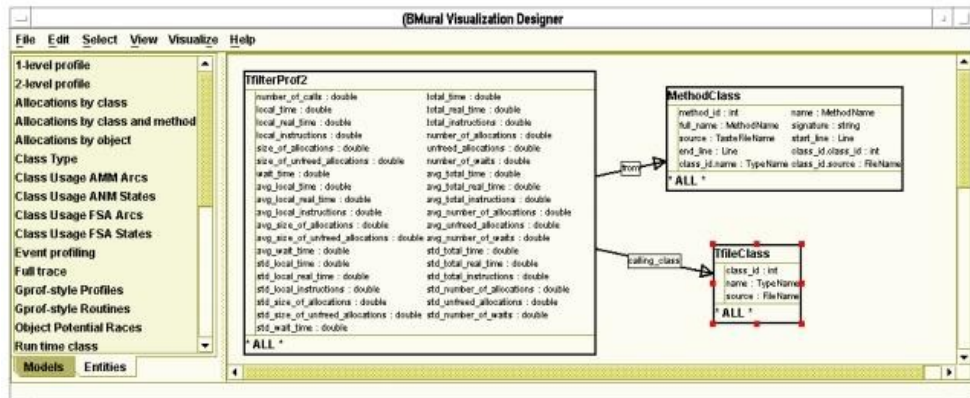


Figure 5.3: MURAL overview.

expressed by a set of classes with several properties called Generalised Attributes (GA). GA represents a generic property of one or more classes and can encapsulate both implicit and explicit relationships. Both classes and GAs are represented by icons. In order to avoid disambiguation, a natural language descriptions generated by the system is added to the icon visualization. The schema of the DataBase is made according to a semantic data model called Graph Model. The schema consists of a labelled graph that captures both structural information, such as classes, relationships, and consistency constraints. The classes of objects, which are nodes in the semantic model, are connected through paths. This concept is related to the GAs but not all the paths are equally meaningful; therefore the QBI system defines a semantic distance function to estimate the meaningfulness of a path. This is done only in order to present to the user a restricted number of useful GAs, that otherwise would be shown in an infinite number. The query process formation follows the select-project paradigm. The user first defines the conditions that determine a subset of class and then specify the GAs that will be part of the output result.

5.2.3 The Flow Metaphor

Morris et al. [17] designed a Visual Query Language for spatial DataBases. Such DataBases focus on representing and formulating queries about data related to objects located in the space. The proposal is valid for both spatial and non-spatial DataBases, and all the operations are expressed consistently. Some insights can be found in this implementation, where the most important is the metaphor used to define the query. Queries are visualized by a flow of information from the data source to the result. In between, there is a

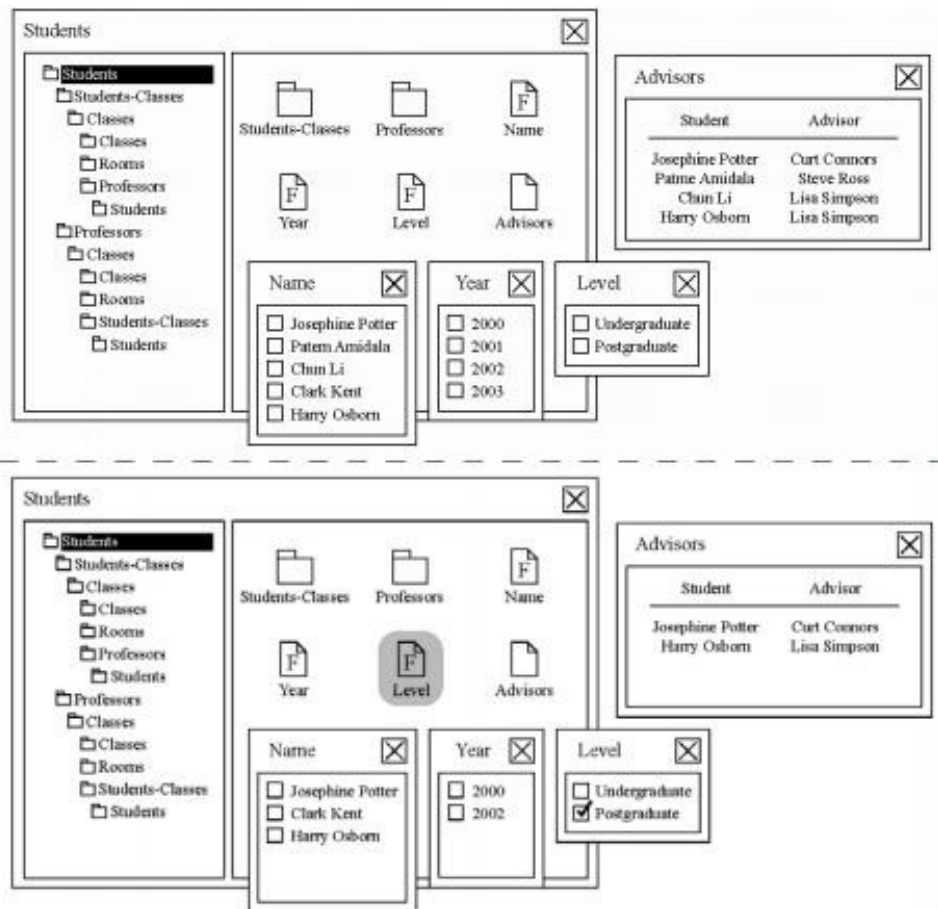


Figure 5.4: QBB query example.

filter process where constraints can be applied. The flow starts with an icon representing an object. A simple filter expressing constraints can be applied and the flow will pass through the filters only as long as the constraints set are satisfied. Boolean conditions can be created combining filters. The AND condition is represented by two filters in a series, whereas two filters in parallel represent the OR condition. If a filter has a double border it means a join condition, and these kinds of icons are associated with more than one object type. Combining these basic constructs makes it possible to build complex queries.

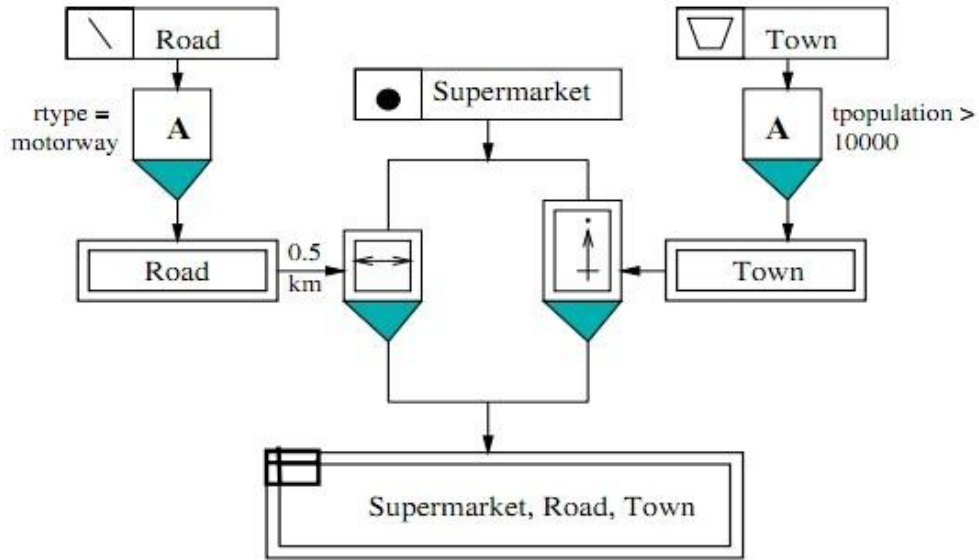


Figure 5.5: A visual query in the flow metaphor.

5.2.4 Kaleidoquery

The flow metaphor is also the basic idea for the Kaleidoquery [18], which is a visual query language for object DataBases. As Morris states in his work [19], class instances and their extents enter the query; in the flowing process there are one or more filtering steps in which some constraints are applied on the attributes of the classes. The results of the query can be visualized or further used as inputs to start a new querying flow. Classes and extents are represented by a combination of icons and text descriptions to give a better understanding than a pure iconic or pure textual visualization [20].

As a user becomes more familiar with the system, the user will rapidly associate the icon with the text without fully reading it in order to understand the meaning of the query. The extents visualization consists of the extent name surrounded by an oval box. Kaleidoquery provides different icons to describe boolean operators, which can be easily combined using parallel or serial connections as seen in Morris' work. It is possible to express basic constraints, such as: "equal", "greater than", and "less than" operators, adding them in the flow and restrict the result. In figure 5.6, the iconic visualization of the operators can be easily identified. Two extents involved in a join are identified with an equal condition applied on the join's attributes; aggregation operators, such as sum, maximum, average, are visualized by an oval

surrounding the extents; the membership test and the universal and existential quantification are displayed with an oval arrow surrounding a textual description. Kaleidoquery relies on OQL, an object query language used to work with query DataBases. One limitation of this language is that while writing the query, the user must concentrate also on the desired structure of the result. To facilitate the structuring process, the system allows the user to apply all the visualization conditions directly on the query results. This means that, starting from the final extent of the flow, it is possible to apply grouping, order by, or other conditions before visualizing the desired output.

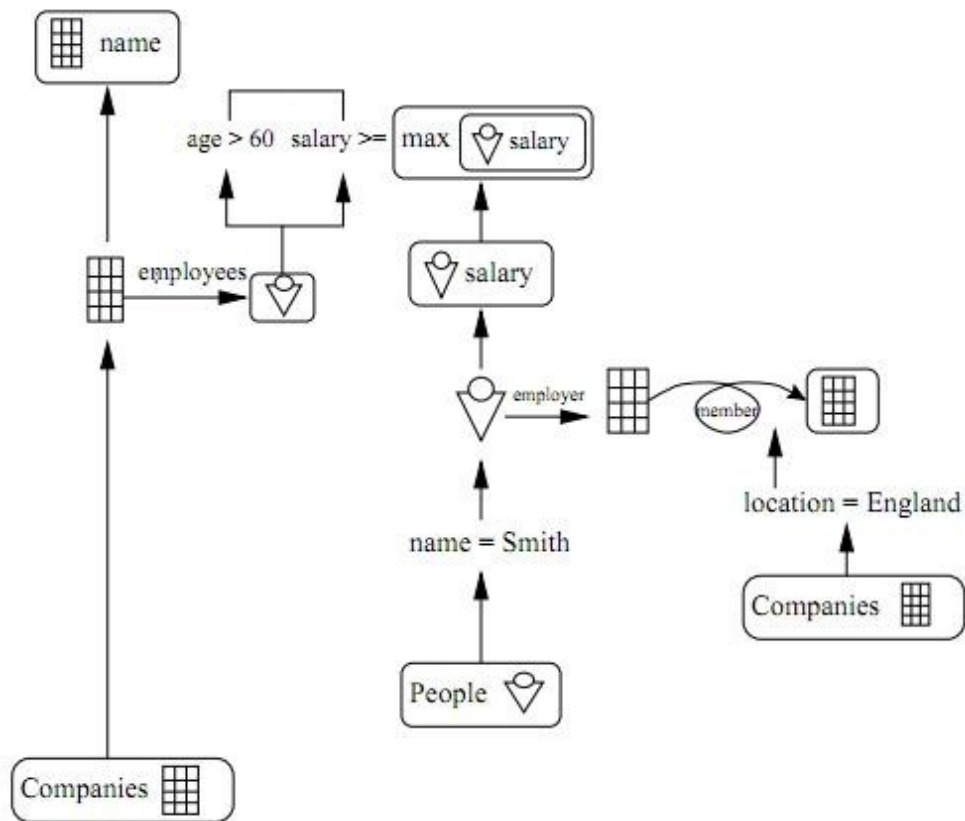


Figure 5.6: A query example in Kaleidoquery.

5.2.5 Visual

Another example of an icon-based object-oriented query language is VISUAL [21], which is a system addressing scientific DataBases. The system design is aimed for large volumes of data with real time constraints and spatial prop-

erties. The query part is implemented as an object and, while processing the results can be communicated in between different query objects. Security, synchronization and time-constraints issues are better managed in this object-oriented approach. In VISUAL, there is a client-server approach for the query object model, where a query object acts as a client when requesting services from another query object, which becomes a server. Every query object is described using interpretation semantics, while there can be different execution semantics: in this way objects can communicate through the interpretation semantics and can be executed in different frameworks. The query is represented by a window, divided in query head and query body. In the head there is a name of the query, input and output parameters, and an output specification. VISUAL is strongly typed, and each output parameter must be specified as a single object or a collection of objects. The body contains several iconized objects, condition boxes and links to other queries. Every iconized object has some properties, such as color or shape, that clearly identify it. There can be four classes of iconized objects: domain objects, method objects, range objects and spatial enforcement region objects. VISUAL focuses on spatial and hierarchical concepts, in which objects can intersect each other or can be contained in to another object specifying the relationships among them. The object oriented architecture allows an easy change of the domain of interest: the domain is the lowest layer of the system architecture. Thus, it is possible to build a new application writing only the lowest layer of the architecture.

5.3 Visual Query Comparison

A Visual Query Language (VQL) should provide different kinds of interaction because there is not a unique paradigm that leads to the best results. An empirical experiment [22] about the ease of use of two different query languages shows that there can be some advantages as well as disadvantages, both in iconic and diagram-based approaches. In the experiment comparing QBI and QBD systems, different strategies are used for the query formulation (navigation vs. composition) as well as a different visual formalisms (diagrams vs. icons), which are basic aspects of VQL. Thus, the results can be extended to larger classes of the VQL system. The experiment focuses on discovering which relation occurs between the query language type and both the query class and the experience of the user. In particular, the queries are classified according to the semantic distance of the path involved in the query and the overall number of the cycles in the query. The notion of path derives from the Graph Model described in QBI. The main result is that both accuracy

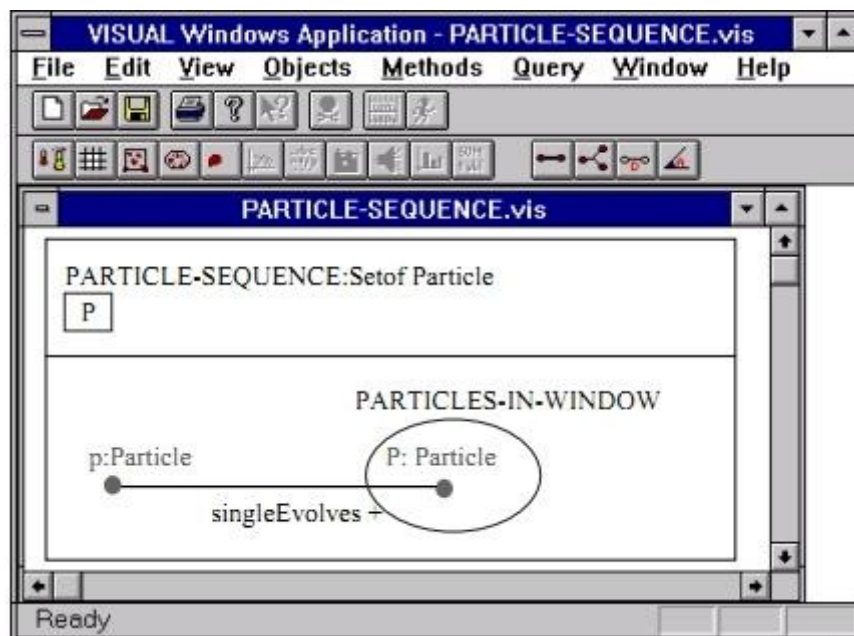


Figure 5.7: A visual query in VISUAL.

and response time seem to be highly sensitive to the semantic distance of the query path, while QBD shows independence for both criteria. In addition, QBD is less accurate and requires more time when there are cycles in the query. Furthermore, QBI seems not to be affected by the presence of cycles.

Chapter 6

Visual Analysis

The increasing scale and availability of digital data provides an extraordinary resource for scientific discovery, business strategy, and even our personal lives. To get the most out of such data, however, users must be able to make sense of it, to answer questions, uncover patterns of interest, and identify (and potentially correct) errors. Beside data-management systems and statistical algorithms, analysis requires contextualized human judgments regarding the domain-specific significance of the clusters, trends, and deviations discovered in data.

Visualization provides a powerful means of making sense of data. By mapping data attributes to visual properties such as position, size, shape, and color, visualization designers boost perceptual skills to help users discern and interpret patterns within data [23]. A single image, however, typically provides answers to, at best, a handful of questions. Instead, visual analysis typically progresses in an iterative process of view creation, exploration, and refinement. Meaningful analysis consists of repeated explorations as users develop insights about significant relationships, domain-specific contextual influences, and causal patterns.

Visual analysis can be divided into task types, grouped into three categories:

- data and view specification (visualize, filter, sort, and derive),
- view manipulation (select, navigate, coordinate, and organize),
- analysis process and provenance (record, annotate, share, and guide).

6.1 Data and View Specification

To enable users to explore large data sets involving different data types (e.g., multivariate, geospatial, textual, temporal, networked), flexible visual analysis tools must provide appropriate controls for specifying the data and views of interest. These controls enable users to selectively visualize the data, to filter out unrelated information to focus on relevant items, and to sort information to expose patterns. Users also need to derive new data from the input data, such as normalized values, statistical summaries, and aggregates.

6.1.1 Visualize

Perhaps the most fundamental operation in visual analysis is to specify a visualization of data: users must indicate which data is to be shown and how it should be depicted. A common data visualization method is represented by the chart, where users can select the type (bar charts, scatter plots, map views, etc.), assign data variables to X/Y/Z axes and choose size or color of visualized marks.

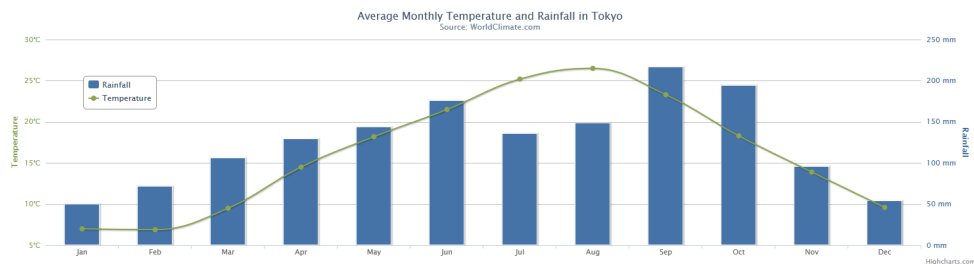


Figure 6.1: Chart example, with X/Y axis and bars and line representations overlapped.

6.1.2 Filter

Filtering of data values is intrinsic to the visualization process, as users rarely visualize the entirety of a data set at once. Instead, they construct a variety of visualizations for selected data dimensions. Given an overview of selected dimensions, users then often want to shift their focus among different data subsets, for example to examine different time slices or isolate specific categories of values.

6.1.3 Sort

Ordering (or sorting) is another fundamental operation within a visualization. A proper ordering can effectively surface trends and clusters of values [24] or organize the data according to a familiar unit of analysis (days of the week, financial quarters, etc.). The most common method of ordering is to sort records according to the value of one or more variables. Sometimes specialized sort orders (such as weekday or month names) are necessary to reveal important patterns.

6.1.4 Derive

As an analysis proceeds in iterative cycles, users may find that the input data is insufficient: variables may need to be transformed or new attributes derived from existing values. Common cases include normalization or log transforms to enable more effective value comparisons. Derived measures are often used to summarize the input data, ranging from descriptive statistics (mean, median, variance) to model fitting (regression curves) and data transformation (group-by aggregation such as counts or summations).

6.2 View Manipulation

Once users have created a visualization through data and view specification actions, they should be able to manipulate the view to highlight patterns, investigate hypotheses, and drill down for more details. Users must be able to select items or data regions to highlight, filter, or operate on them. Large information spaces may require users to scroll, pan, zoom, and otherwise navigate the view to examine both high-level patterns and fine-grained details. Multiple, linked visualizations often provide clearer insights into multidimensional data than do isolated views. Analysis tools must be able to coordinate multiple views so that selection and filtering operations apply to all displays at once and organize the resulting dashboards and work spaces.

6.2.1 Select

In visual analysis, reference (or selection) is realized through a limited set of actions, such as clicking or lassoing items of interest. Common forms of selection within visualizations include mouse hover, mouse click, region selections (e.g., rectangular and elliptical regions, or free-form “lassos”), and

area cursors (e.g., “brushes” [25] or dynamic selectors such as the bubble cursor [26], which selects the item currently closest to the mouse pointer). These selections often determine a set of objects to be manipulated, enabling highlighting, annotation, filtering, or details-on-demand.

6.2.2 Navigate

How users navigate a visualization is in part determined by where they start. One common pattern of navigation adheres to the widely cited visual information-seeking mantra: “Overview first, zoom and filter, then details-on-demand.” [27].

Users may begin by taking a broad view of the data, including assessment of prominent clusters, outliers, and potential data-quality issues. These orienting actions can then be followed by more specific, detailed investigations of data subsets. A common example is geographic maps: an overview might show an overall territory, followed by zooming into regions of interest.

Of course, starting with an extended overview is not always advisable. Another navigation approach can be summarized as “Search, show context, expand on demand.” [28].

In either case, visualizations often function as viewports onto an information space. Users need to manipulate these viewports to navigate the space. Common examples include scrolling or panning a display via scrollbars or mouse drag, and zooming among different levels using a zoom slider or scroll wheel (figure 6.2). Zooming does not have to follow a strict geometric metaphor: semantic zooming [29] methods can modify both the amount of information shown and how it is displayed as users move among levels of detail. In the calendar in figure 6.3, the display magnifies selected regions as users navigate from months to days to hours. Semantic zooming reveals more details within focal regions. Additionally, dynamic query widgets, such as range sliders for the X and Y axes of a scatter plot, can filter the visible data range and thus provide a form of zooming within a chart.

Visualizations can provide cues to assist users’ decisions of where and how to navigate. The controls for view manipulation have often been invisible, such as zooming/panning by mouse movement. Improved strategies facilitate discovery by users and provide visible indication of settings in legends or other ways, such as scrollbar positions, that provide informative feedback. An important challenge is to show selected items, even when they are not in view. For example, the results of a text search that are not currently in view might be shown by markers in the scrollbar [30] or the periphery of the display.

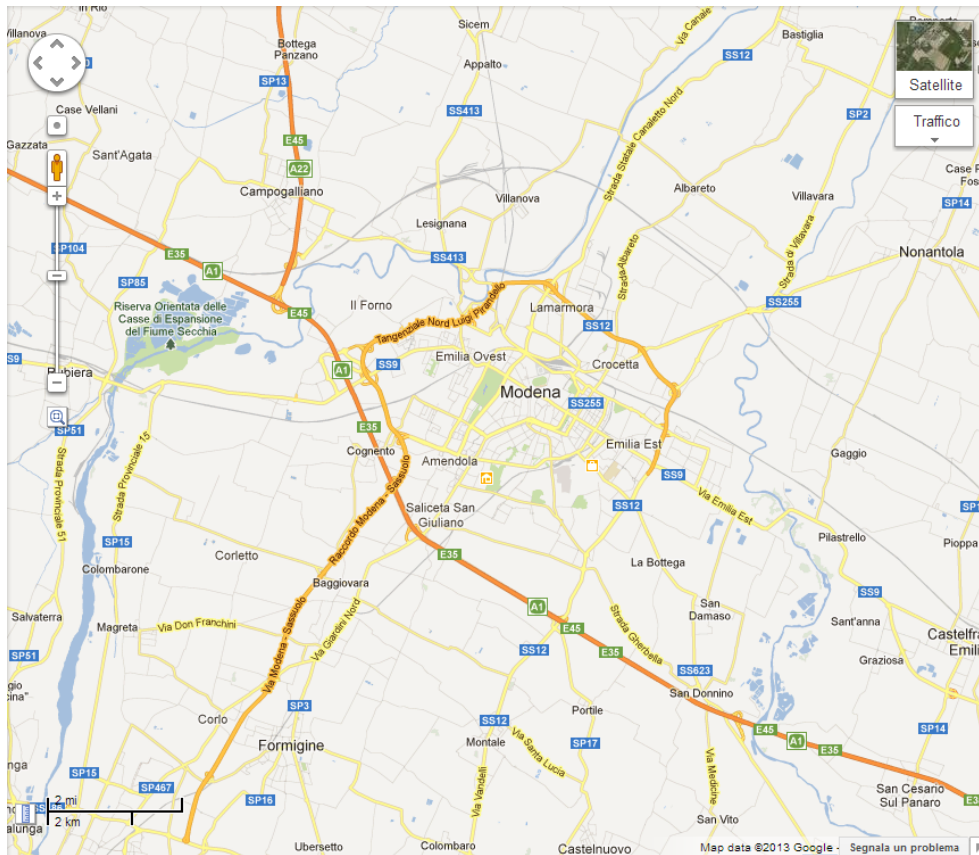


Figure 6.2: Map navigation.

6.2.3 Coordinate

Many analysis problems require multiple views, that enable users to see their data from different perspectives, and can facilitate comparison. For example, Edward Tufte [31] advocates the use of small multiples: a collection of visualizations placed in spatial proximity and typically using the same measures and scales. As in figure 6.4, these small multiples enable rapid comparison of different data dimensions or time slices. The repetition of the chart form supports comparison among sectors. Plotting all the data in one chart would otherwise clutter and obscure individual trends.

Alternatively, multiple view displays can use a variety of visualization types, such as histograms, scatter plots, maps, or network diagrams, to show different projections of a multidimensional data set. Automatically generated legends and axes are important for providing accurate annotations for users and meaningful explanations when visualizations are shared. Legends and

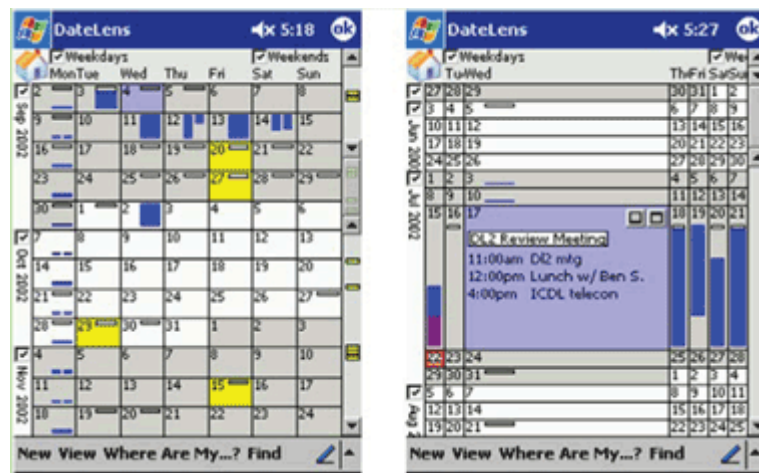


Figure 6.3: Fisheye view.

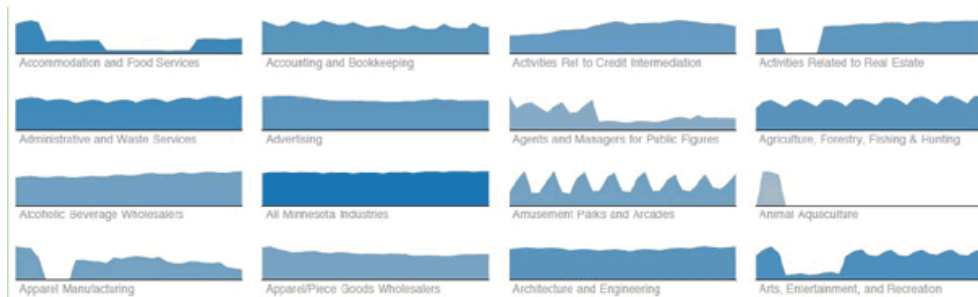


Figure 6.4: Multiple views.

axes can also become control panels for changing color palettes, marker attributes, variable ranges, or provenance information [32].

6.2.4 Organize

When users make use of multiple views, they face the corresponding challenge of managing a collection of visualizations, laying them out in the most useful way, given more space to important data, and placing views into comfortable locations on screen.

Typical systems allow users to add, remove and reorganize views on the workspace. As larger and multiple displays become more common, layout organization tools will become decisive factors in creating effective user experiences. Similarly, the demand for tablet and smartphone visualizations will promote innovation in layout organizations that are compact and reconfigurable by simple gestures.

6.3 Process and Provenance

Visual analytics is not limited to the generation and manipulation of visualizations, it involves a process of iterative data exploration and interpretation. Visual analytics tools should preserve analytic provenance by keeping a record of user actions and insights so that the history of work can be reviewed and refined. Textual logs of activity have benefits, but visual overviews of activity can be more compact and comprehensible. If users can annotate patterns, outliers, and views of interest, they can document their observations, questions, and hypotheses. In a networked environment, users should be empowered to share results and discuss with colleagues, coordinate the work of multiple groups, or support processes that may take weeks and months. Moreover, analysis tools can explicitly guide novices through common analysis tasks, provide progress indicators for experts, or lead viewers through an analysis story.

Part V

Results

Chapter 7

Visual Query Language (VQL)

In this section, the definition of the Visual Query Language (VQL) is provided, showing, for each visual element, the graphic symbol, and providing its semantics expressed in natural language. This VQL is intended to map the SCSQL query language (section §3.3). Its translation produces SCSQL queries, and its main constructs (Functions and Objects) are mapped one to one on the same SCSQL constructs. In addition to these constructs, which have a direct mapping on SCSQL, the VQL consists of other graphic symbols, needed during the query construction phase (e.g., Connections and Handle), but which have no mapping with SCSQL. The translation is one-way, i.e., the system can translate a Visual Query into SCSQL, but the inverse passage (from textual SCSQL query to Visual Query) is not foreseen.

Since SCSQL has been chosen as the Smart Vortex CQL reference, and since SCSQL is based on AmosQL (section §3.3), which is based on the function composition notion and the use of types and objects, the same approach has been followed on designing the VQL. Because of these principles, this Visual Query Language can be categorized as part of the already known Diagram based language class (section §5.1). Diagrams frequently adopted in VQSs use, as basic (visual) components, either points or simple geometrical figures. Generally, a diagram adopts visual components that have a direct correspondence with specific concept types. Lines denote logical relationship types among elements. Sometimes labels are also included in diagrams for denotational purposes, while inclusion of geometrical figures are used to represent structural relationships. Diagrams emerge as the most popular visual formalism used in existing VQSs [12].

Following the same approach on designing the presented VQL, simple geometrical figures (e.g. rectangles, circles) have been employed for base elements, while lines denote logical relationship among SCSQL elements. As better

explained later, circles represent SCSQL types or objects, and rectagles have been chosen as a representation for SCSQL functions or function-related elements. In order to ease elements recognition and categorization, colors have been adopted; for example, blue is associated with Types and Objects elements. When the user composes a query, she is aided by these facilitations on recognising different elements, accelerating the composition process.

7.1 VQL elements

An overview of all graphical elements composing the Visual Query Language is presented. Object, Function, Select and Conditions elements have a direct mapping with SCSQL statements. The Handle is used as one of the visual composition facilitations.

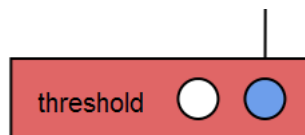
7.1.1 Object



An Object is an instance of a Type, and can have a Value that depends from its Type. It is directly mapped into corresponding AmosQL/SCSQL Object. It is used to characterize both primitive Types (Integer, Number, Boolean, Charstring, Date, Time, Timeval, Bag, Vector, Stream, etc...) and complex Types, such as an hypothetical “Sensor” Type.

The graphical representation is an elliptical shape with a blue background. In figure above, from left to right, an Integer Object, an Integer Object with a Value equals to three, and a Bag of Boolean Object (unordered set of Boolean Objects with duplicates allowed) is shown.

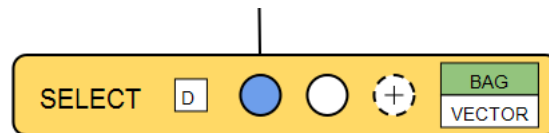
7.1.2 Function



Function elements (like their mathematical counterparts) have a set of fixed, ordered, Function dependent Parameters of a certain Type, and returns a Function dependent Object (the result of Function computation). It is directly mapped into corresponding AmosQL/SCSQL function.

The graphical representation is a rectangular shape with a red background. Inside the rectangle, a string represents the Function name, and a set of circles represent Function's Parameters.

7.1.3 Select



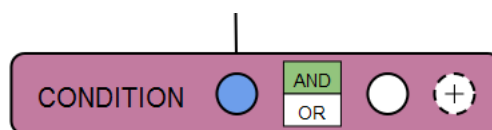
The Select construct is used to specify which elements will be mapped into the SELECT SCSQL statement. It acts like a Function, its Parameters accepts Object Types, and it can be composed with other Select constructs.

When creating a SELECT-FROM-WHERE statement, the DISTINCT clause can be specified, in order to remove duplicates from result.

A SELECT statement returns results represented as a Bag (unordered set of Objects) or as a Vector (ordered sequence of Objects).

The graphical representation is a rectangular shape with a yellow background. Inside the rectangle, the square with a “D” inside represents the DISTINCT clause; clicking on it activates (green) the clause, which will be added to the SELECT statement during translation. Circles represent Parameters to pass to the SELECT statement, followed by a specific construct to add/remove new/old Parameters (more on this later). The last two rectangles maps the Type representation of returned results. These are like toggle buttons, therefore only one Type a time can be activated (green).

7.1.4 Conditions



The Conditions construct is used to specify which elements will be mapped into the WHERE SCSQL statement. It acts like a Function, and it can be composed (connecting it) with other Conditions constructs, in order to create more complex Boolean expressions. Its Parameters accept Boolean Types, and returns a Boolean Object.

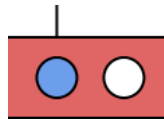
Conditions Parameters can be added or removed at need using the already mentioned construct. Each Parameter represents a WHERE condition, separated each other by an AND/OR Boolean operator.

The graphical representation is a rectangular shape with a purple background. Circles represent Parameters to pass to the WHERE statement, followed by a specific construct to add/remove new/old Parameters (more on this later). Parameters are separated by two rectangles representing Boolean operators AND and OR; these rectangles are like toggle buttons, therefore only one Boolean operator a time can be activated (green) by clicking on it.

7.1.5 Parameters

As explained, Functions, Selects and Conditions elements have associated a set of Parameters, which represent arguments to pass to these elements. Each Parameter has associated a Type; a Parameter will accept only associated Types, or its Subtypes (so, for example, if Integer is a Subtype of Float, a SUM(FLOAT, FLOAT) Function will accept Floats and Integers as Parameters, but not Charstrings). The Type accepted by the Parameter is not recognizable from the base construct, but useful functionalities are developed to assist the user during the query creation process (see 9.3.1, where an explanation about how to know which Types are accepted by a Parameter is provided).

The graphical representation is a white (when unconnected, see section 7.1.6) or blue (when connected) circle:



As already mentioned, two special Parameters are used to add and remove Parameters from Select and Conditions elements: Add Parameter and Remove Parameter respectively. Select and Conditions elements are represented as special kinds of functions, and they both accept other elements as

arguments, but unlike Function elements, where Parameters are Function-dependent, in these elements Parameters can be added or removed at will; the only constraint is that they cannot have less than one Parameter.

The graphical representation for the Add Parameter is a dashed circle with a plus inside, and for the Remove Parameter is a dashed circle with a minus inside. When clicking on them, a new Parameter is added, or an old one is removed.



7.1.6 Connection

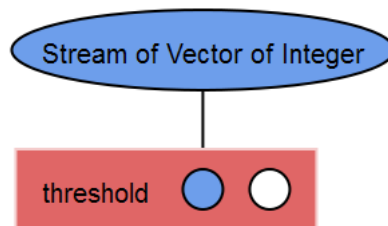


Figure 7.1: Connection between an Object and a Function Parameter.

Connections are used to specify a logical association between elements. In particular, Connections specify that a certain Object, or a result of a Function computation, is passed as Parameter to other elements.

The graphical representation is a line between connected elements.

Not all elements can be connected each other. Connections can be created only between an element and a Parameter, but with some restrictions; in particular, they can be created between:

- Object and Function/Select/Conditions Parameters;
- Function output and Function/Select/Conditions Parameters;
- Select output and Function Parameters;
- Conditions output and Conditions Parameters.

Conversely, Connections cannot be created between:

- Select output and Conditions Parameters;
- Conditions output and Function/Select Parameters.

Assuming:

- O: Object,
- FP: Function Parameters,
- FR: Function result,
- SP: Select Parameters,
- SR: Select result,
- CP: Conditions Parameters,
- CR: Conditions result,

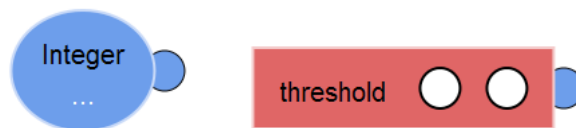
the following table resumes permitted Connections (vertical elements can or cannot be connected with horizontal elements):

	O	FP	FR	SP	SR	CP	CR
O		X		X		X	
FP							
FR		X		X		X	
SP							
SR		X					
CP							
CR						X	

7.1.7 Handle

An Handle represents the Object returned from a Function, Select or Conditions element, and the Object itself for an Object element. It is used to connect elements with Parameters.

The graphical representation is a blue circle, placed on the right side of the corresponding element, and is shown only when the element is selected (9.3.1).



7.2 Visual Query example

Following, a Visual Query example shows how all the presented Visual Query elements are put together. For this purpose, this SCSQL query is proposed:

```
select first_n(heartbeat(1), 10);
```

which takes the sub-stream ending at position 10 of the stream *heartbeat(1)*, that emits a stream of numbers, starting from zero, at a frequency of one second.

To build the query, following elements are needed:

- the “heartbeat” Function, with a Number Object as Parameter, instantiated to one;
- the “first_n” Function, where the first Parameter is the stream of Numbers returned by the “heartbeat” Function, and the second Parameter is a Number Object instantiated to ten;
- the Select construct with the “first_n” Function as Parameter, and with Bag as returned Type.

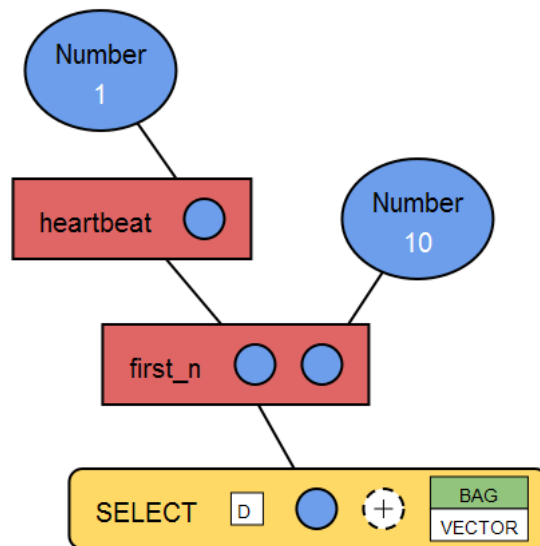


figure above shows how the Visual Query looks like once it has been created.

Chapter 8

Query translation

Visual Queries have to be translated into its textual representations, in order to be executed on the FDSMS. This chapter explains how the Visual Queries can be represented, from a mathematical point of view, with oriented graphs, and how this representation leads to a textual form.

A mathematical definition of each visual element is given, with properties connected to them, and with a generic representation through oriented graphs.

The graph model must be formally correct, otherwise the translation algorithm cannot be applied. Therefore, some conditions of translatability must be verified before proceeding with the translation.

The description of the translation algorithm then follows, with an explanation of its implementation in code. Two examples of Visual Query translation are given, to better describe how it operate.

8.1 Definitions

$$G = (V, A) : V = S \cup C \cup F \cup O \cup P, A = (v, w) : v, w \in V$$

G is an oriented, connected graph, with no cycles.

V is a set of all vertices, composed by:

- S , a set of Select vertices,
- C , a set of Conditions vertices,
- F , a set of Function vertices,
- P , a set of Parameter vertices,

- O , a set of Object vertices.

These arrays are used in code to represent previous sets:

```
var selects = [];
var conditions = [];
var functions = [];
var parameters = [];
var objects = [];
```

A is a set of ordered pairs of vertices, from v to w , called arcs.

In code, these arcs are partially represented by this array:

```
var connections = [];
```

Arcs between Functions, Selects and Conditions are not stored here, because they are directly associated with the elements.

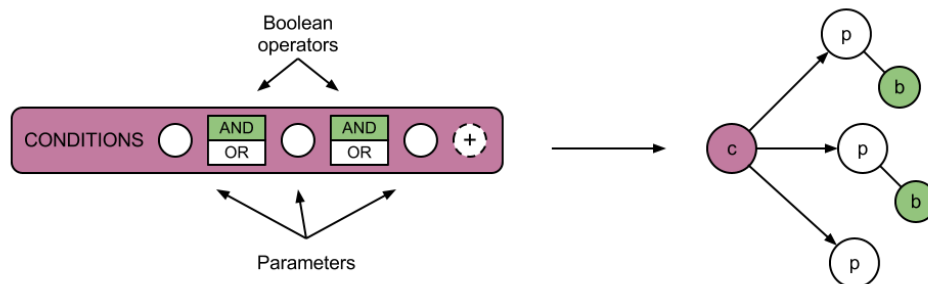
For a vertex, the number of inward arcs is called the *indegree* of the vertex, and the number of outward arcs is its *outdegree*. The indegree is denoted $deg^-(v)$ and the outdegree as $deg^+(v)$. Notice that the *output* of a Function (or a Select/Conditions) is represented by an inward arc, and parameters *input* by an outward arc, despite its names.

table 8.1 summarizes, for each element, its visual representation and the corresponding graph.

8.1.1 Boolean operators

$B = b : b$ is a Boolean operator

$Bp = (p, b) : p \in P, b \in B$: association between a parameter and a Boolean operator.



In code, each Conditions element has an array associated, which stores the list of Boolean operators:

```
conditionsElement.booleanOperators = [];
```

To retrieve the i -th Boolean operator:

```
conditionsElement.booleanOperators[i].value
```

8.1.2 Objects names

In code, an array is used to store the Objects, and each Object has some properties, like Type and Value. When the query is translated, each Object without a Value is translated into a variable with a unique name, in order to generate the FROM clause. The variable name is composed of a string “v”, followed by the index of the Object in the array, plus one; so, for example,

```
objects[3]
```

will have “v4” as associated variable.

This is performed by the following function:

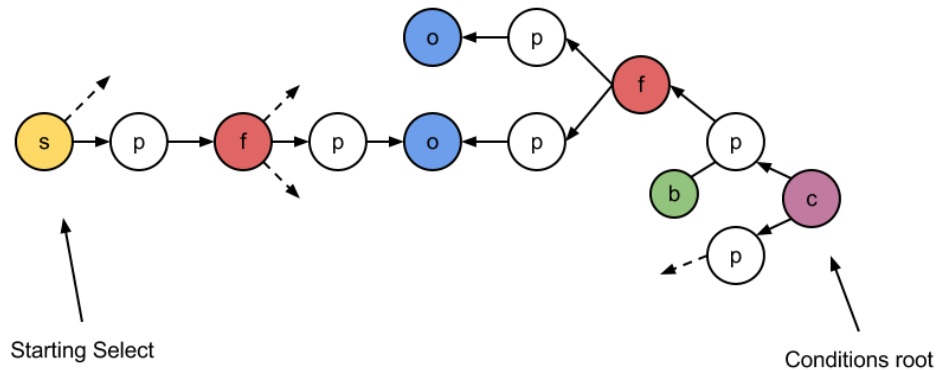
```
function assignVariable(object) {
    var v = new Object();
    v.type = object.objectType;
    v.name = "v" + array.indexOf(objects, object) + 1;
    return v;
}
```

which, given an Object, returns a representing variable.

In this mathematical model is assumed that each Object already has an associated variable, which can be retrieved through the function $name(o)$ (explained later).

8.1.3 Generic graph

The following figure depict a generic Visual Query representation through its associated graph model. The starting Select and the Conditions root are also highlighted, and are deeply described later.



8.1.4 Functions used in the algorithms

$adj(v)$: returns the set of vertices adjacent to v , i.e. connected to v by an arc, following the arc direction ($adj(v) = w \in V : (v, w) \in A$).

$name(v)$: returns the name of the corresponding vertex, so for example:

- for a Function named “ abs ”: $name(f) = "abs"$,
- for an Object: $name(o_1) = "v1"$.

$type(o)$: returns the Type of given Object.

$value(o)$: returns the Value of given Object, or NIL if it has no associated value.

$distinct(s)$: returns *true* if the “*distinct*” clause is selected on given Select element, *false* otherwise.

$boolop(p)$: given a (Conditions) parameter, returns the associated boolean operator ($boolop(p) = b \in B : (p, b) \in B_p$), or NIL otherwise.

8.2 Conditions of translatability

Following conditions must be true, in order to have a translatable query:

- G is connected,
- G doesn't contain closed loops composed of F and S vertices,
- $\forall s \in S, deg^-(s) \in \mathbb{N}_0, deg^+(s) \in \mathbb{N}$,

- only one $s \in S$ has $deg^-(s) = 0$, every other one has $deg^-(s) > 0$ (formally: $\exists!s \in S : deg^-(s) = 0, \forall s' \in S - s, deg^-(s) > 0$),
- $\forall c \in C, deg^-(c) \in \mathbb{N}_0, deg^+(c) = n, n \in \mathbb{N}, n \geq 2$,
- $\forall f \in F, deg^-(f) \in \mathbb{N}, deg^+(f) \in \mathbb{N}$,
- $\forall p \in P, deg^-(p) = 1, deg^+(p) = 1$,
- $\forall o \in O, deg^-(o) \in \mathbb{N}, deg^+(o) = 0$.

Some of these conditions are assured by the hypothesis, some others are verified by the correctness tests.

8.2.1 Hypothesis

These conditions are constraints given by the VQE, therefore they are always true:

- $deg^-(s) \in \mathbb{N}_0$: Select elements can have from 0 to n output connections;
- $deg^+(s) \in \mathbb{N}$: Select elements can have from 1 to n parameters (By default, they have 1 parameter. Parameters cannot be less than 1, and can be added without limits);
- $deg^-(c) \in \mathbb{N}_0$: Conditions elements can have from 0 to n output connections;
- $deg^+(c) = n, n \in \mathbb{N}, n \geq 2$: Conditions elements can have from 2 to n parameters (By default, they have 2 parameter. Parameters cannot be less than 2, and can be added without limits);
- $deg^-(p) = 1$: The arc between a Function (or Select or Conditions) and a Parameter is directly implied by the Function creation;
- $deg^+(o) = 0$: Objects cannot have outward arcs.

8.2.2 Correctness tests

Following tests are used to check these three conditions:

- G is connected,
- G doesn't contain closed loops composed of F and S vertices,

- only one $s \in S$ has $deg^-(s) = 0$, every other one has $deg^-(s) > 0$.

If these conditions are verified, the query can be translated.

Elements connections

G is connected if:

- all parameters are connected: $deg^+(p) = 1, \forall p \in P$ or $\forall p \in P, \exists a \in A : a = (p, v), v \in V - P$;
- all functions have at least one output connection: $deg^-(f) > 0, \forall f \in F$ or $\forall f \in F, \exists a \in A : a = (p, f), p \in P$;
- all objects have at least one connection: $deg^-(o) > 0, \forall o \in O$ or $\forall o \in O, \exists a \in A : a = (p, o), p \in P$.

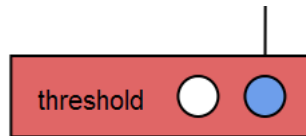
To check if all parameters are connected, the following algorithm is used:

```

var allParametersConnected = true;
for ( var i = 0; i < parameters.length; i++) {
    var parameter = parameters[i];
    if (!parameter.connected) {
        allParametersConnected = false;
        break;
    }
}

```

This is an example of a Function where not all parameters are connected:



To check if all functions have at least one output connection, the following algorithm is used:

```

var allFunctionsHaveConnections = true;
for ( var i = 0; i < functions.length; i++) {
    var f = functions[i];
    if (f.connections.length == 0) {
        allFunctionsHaveConnections = false;
        break;
    }
}

```

The previous example is also an example of Function without output connections.

To check if all objects have at least one connection, the following algorithm is used:

```

var allObjectsHaveConnections = true;
for ( var i = 0; i < objects.length; i++) {
  var object = objects[i];
  if (object.connections.length == 0) {
    allObjectsHaveConnections = false;
    break;
  }
}

```

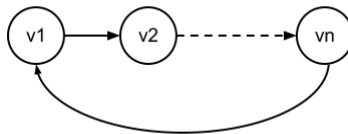
This is an example of Object without connections:



Closed loops

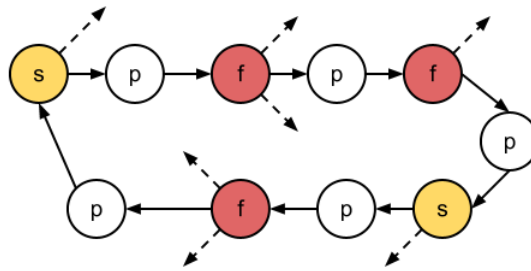
Closed loops are meant as Function and Select elements connected each other. This is not permitted, otherwise the translation algorithm falls into an infinite loop and the textual query cannot be created.

Definition of cyclic directed graph: $C_{n \in \mathbb{N}} = (V, A), V = v_1, v_2, \dots, v_n, A = (v_i, v_{i+1} : i \in \mathbb{N}_{n-1})(v_n, v_1)$.



In this context, a closed loop is a subgraph of graph G , isomorphic to a cyclic directed graph C_n , composed only of S and F vertices. Formally: $L = (V', A') \text{ closed loop} : V' \subseteq S \cup F \cup P, \#V' = n, n \in \mathbb{N}, n \geq 4, A' \subseteq \binom{V'}{2} \cap A, L \text{ isomorphic to } C_n$.

An example of closed loop:



In order to have a translatable query, G cannot contain closed loops: $L : L \text{ closed loop} = \emptyset$

Following, the algorithm used to check the existence of closed loops. When a closed loop is found, the algorithm stops, and the query cannot be translated.

FindClosedLoops(r, c) :

1. *if* $r = c$ *then*
2. $\langle \text{closed loop found} \rangle$
3. *end.*
4. *for each* $v \in \text{adj}(c)$
5. *if* $v \in S \cup F \cup P$ *then*
6. *FindClosedLoops*(r, v)
7. *end.*

This is a recursive function, which walks through the graph, starting from a Select or a Function vertex (the root r); it follows only the arcs connected with Select, Function or Parameter elements, and if the root is reached again, the graph G contains a closed loop. c is the currently evaluated vertex.

In code:

```
function closedLoopFound(rootElement, currentElement) {
    if (rootElement === currentElement) {
        return true;
    }
    if (currentElement.type === "object"
        || currentElement.type === "conditions"
        || (currentElement.type === "select"
            && currentElement.connections === [])) {
        return false;
    }
    for (var i = 0; i < currentElement.connections.length;
        i++) {
        var connectedElement = currentElement.connections[i]
            .destination.element;
```



```

        if (closedLoopFound(rootElement , connectedElement))
            {
                return true;
            }
    }
    return false;
}

```

1. *for each* $v \in S \cup F$
2. *for each* $w \in \text{adj}(v)$
3. *FindClosedLoops*(v, w)
4. *end.*

Previous function is applied on every Select and Function vertex.

In code:

```

for ( var i = 0; i < selects.length; i++) {
    var select = selects[i];
    for ( var i = 0; i < select.connections.length; i++) {
        var connectedElement = select.connections[i].
            destination.element;
        if (closedLoopFound(select , connectedElement)) {
            <error>
            return;
        }
    }
}
for ( var i = 0; i < functions.length; i++) {
    var f = functions[i];
    for ( var i = 0; i < f.connections.length; i++) {
        var connectedElement = f.connections[i].destination
            .element;
        if (closedLoopFound(f, connectedElement)) {
            <error>
            return;
        }
    }
}
}

```

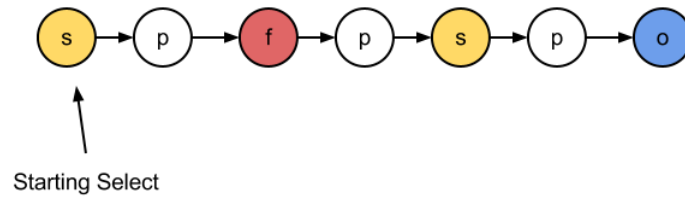
Find starting Select element

All Select elements must have at least one output connection, except from one, which must have no output connections. This latter one is the starting Select, that is the Select element from which the query translation will start.

Let's take for example a query like:

select count(select p from Person p);

where we want to count how many people there are on the DataBase. In this case there are two Select statements, and is it visually built using two Select elements. But only the first one, on the left, is the Select element which the query starts with.



If one, and only one, starting Select can be found, then the third condition is verified, and the the query translation can start.

Following, the algorithm used to verify the third condition:

1. *startingselect* \leftarrow *NIL*
2. *for each* $s \in S$
3. *if* $\text{deg}^-(s) = 0$ *then*
4. *if* *startingselect* = *NIL* *then*
5. *startingselect* \leftarrow s
6. *else*
7. \langle *more than one* s *with* $\text{deg}^-(s) = 0$ \rangle
8. *end.*
9. *if* *startingselect* = *NIL* *then*
10. \langle *no* s *with* $\text{deg}^-(s) = 0$ \rangle
11. *end.*

Every Select element is evaluated.

Line 3: check if current Select has no output connections.

Lines 4-5: current Select has no output connections, and if this is the first encountered one, set it as the starting Select.

Lines 6-7: current Select has no output connections, therefore it could be a valid candidate for the starting Select; but a starting Select has been already found, therefore there is more than one possible candidate, which means that the third condition is not verified.

Lines 9-10: if no starting Select has been found, the third condition is still not verified.

Lines 7 and 10 tell that the query cannot be translated.

In code:

```

var startingSelect = undefined;
for ( var i = 0; i < selects.length; i++) {
  var select = selects[i];
  if (select.connections.length == 0) {
    if (startingSelect === undefined) {
      startingSelect = select;
    }
    else {
      <error: more than one starting Select found>
      return;
    }
  }
}
if (startingSelect === undefined) {
  <error: no starting Select found>
  return;
}

```

8.3 Algorithm

Following, the algorithm used to translate the Visual Query into a Textual Query is presented.

GetConnectedVarsToConditions(v, connectedvars) :

1. *if* $v \in O$ *then*
2. *if* $v \notin \text{connectedvars}$ *then*
3. $\text{connectedvars} \leftarrow \text{connectedvars} \cup v$
4. *else*
5. *for each* $p \in \text{adj}(v)$
6. *GetConnectedVarsToConditions*($\text{adj}(p)$, connectedvars)
7. *end.*

This function walks through the graph, starting from a certain Conditions vertex, and collects all the reachable Objects into the *connectedvars* set.

Lines 1-3: if given vertex v is an Object, and if it doesn't belong to the *connectedvars* set, add it to the set.

Lines 4-6: otherwise (v is not an Object) recursively walk through the graph, following all the arcs.

In code:

```
function getConnectedVarsToConditions(element,
connectedVars) {
  if (element.type === "object") {
    var v = assignVariable(element);
    // if v not in connectedVars
    if (array.indexOf(connectedVars, v) === -1) {
      // add it to connectedVars
      connectedVars.push(v);
    }
  }
  else {
    // for each conditions parameter
    for ( var i = 0; i < element.parameters.length; i
      ++ ) {
      var parameter = element.parameters[i];
      getConnectedVarsToConditions(parameter.
        connection.source, connectedVars);
    }
  }
}
```

FindConditionsRoot(conditionsroot, fromvars) :

1. for each $c \in C$
2. if $\text{deg}^-(c) = 0$ then
3. $\text{connectedvars} \leftarrow \emptyset$
4. $\text{GetConnectedVarsToConditions}(c, \text{connectedvars})$
5. if $\text{fromvars} \cap \text{connectedvars} \neq \emptyset$ then
6. $\text{conditionsroot} \leftarrow c$
7. end.

Function used to find a Conditions root for current SELECT subquery. Conditions elements can be combined, in order to create more complex WHERE clauses, therefore the algorithm must find the most external one.

Lines 1-2: consider only Conditions without output connections.

Lines 3-4: get the Objects reachable from current Conditions.

Lines 5-6: if at least one of these Object is reachable also from currently considered Select element (so it's contained into the *fromvars* and the *connectedvars* sets), the Conditions root has been found.

In code:

```
function findConditionsRoot(fromVars) {
  for ( var i = 0; i < conditions.length; i++) {
    var condition = conditions[i];
    if (condition.connections.length == 0) {
      var connectedVars = [];
      getConnectedVarsToConditions(condition,
        connectedVars);
      for ( var connectedVar in connectedVars) {
        for ( var fromVar in fromVars) {
          if (connectedVar.name == fromVar.name)
            {
              return condition;
            }
          }
        }
      }
    }
  }
}
```

BuildQueryFromParameter(query, v, fromvars) :

1. *if* $v \in O$ *then*
2. *if* $value(v) \neq NIL$ *then*
3. $query \leftarrow query + value(v)$
4. *end.*
5. *else*
6. *if* $v \notin fromvars$ *then*
7. $fromvars \leftarrow fromvars \cup v$
8. $query \leftarrow query + name(v)$
9. *end.*
10. *if* $v \in F$ *then*
11. $query \leftarrow query + name(v) + "("$
12. *for each* $p \in adj(v)$
13. $BuildQueryFromParameter(query, adj(p), fromvars)$
14. *if* p *is not the last one then*
15. $query \leftarrow query + ", "$
16. *else*
17. $query \leftarrow query + ")"$

```

18.   end.
19.   if v ∈ S then
20.     BuildQueryFromSelect(query, v)
21.   if v ∈ C then
22.     query ← query + "("
23.     for each p ∈ adj(v)
24.       BuildQueryFromParameter(query, adj(p), fromvars)
25.       if p is not the last one then
26.         query ← query + "," + boolop(p) + " "
27.       else
28.         query ← query + ")"
29.     end.
30. end.

```

Builds the Textual Query, starting from a Parameter, and walking through the graph, following the arcs. This is a recursive function.

Lines 1-9: when an Object is reached, if it has a Value, add the Value to the Textual Query, otherwise add the variable name to the Textual Query, and add the Object to the *fromvars* set.

Lines 10-18: when an Function is reached, add its name to the Textual Query, and build the Textual Query recursively, for each adjacent Parameter. Separate Parameters with commas, and when the last Parameter (for this Function) is reached, add a closed parenthesis to the Textual Query.

Lines 19-20: when an Select is reached, start to build a new SELECT sub-query.

Lines 21-29: when an Conditions is reached, build the Textual Query recursively, for each adjacent Parameter. Each Parameter is separated by the corresponding Boolean operator.

In code:

```

function buildTextualQueryParameter(element , fromVars) {
  switch (element.type) {
  case "object":
    if (element.value !== undefined) {
      return element.value;
    }
  else {
    var v = assignVariable(element);
    // if v not in fromVars

```

```

    var vIsInFromVars = false;
    for ( var i = 0; i < fromVars.length; i++) {
        var fromVar = fromVars[i];
        if (v.name === fromVar.name) {
            vIsInFromVars = true;
        }
    }
    if (!vIsInFromVars) {
        // add it to fromVars
        fromVars.push(v);
    }
    return v.name;
}
break;
case "function":
    // textualQuery will be like foo(par1, par2, par3)
    var textualQuery = element.name + "(";
    for ( var i = 0; i < element.parameters.length; i
    ++ ) {
        var parameter = element.parameters[i];
        textualQuery += buildTextualQueryParameter(
            parameter.connection.source, fromVars);
        // if is not the last parameter
        if (i < element.parameters.length - 1) {
            textualQuery += ", ";
        }
        else {
            textualQuery += ")";
        }
    }
    return textualQuery;
break;
case "select":
    return buildTextualQuerySelect(element);
break;
case "conditions":
    // textualQuery will be like (par1 and|or par2 and|
    or par3)
    var textualQuery = "(";
    for ( var i = 0; i < element.parameters.length; i
    ++ ) {
        var parameter = element.parameters[i];
        textualQuery += buildTextualQueryParameter(

```

```

        parameter.connection.source, fromVars);
    // if is not the last parameter
    if (i < element.parameters.length - 1) {
        textualQuery += " " + element.
            booleanOperators[i].value + " ";
    }
    else {
        textualQuery += ") ";
    }
}
return textualQuery;
break;
}
}

```

BuildQueryFromSelect(query, s) :

1. $fromvars \leftarrow \emptyset$
2. $selectclause \leftarrow \text{"select"}$
3. *if* $distinct(s) = true$ *then*
4. $selectclause \leftarrow selectclause + \text{"distinct"}$
5. *for each* $p \in adj(s)$
6. *BuildQueryFromParameter*($selectclause, adj(p), fromvars$)
7. *if* p *is not the last one then*
8. $selectclause \leftarrow selectclause + \text{","}$
9. $conditionsroot \leftarrow NIL$
10. *FindConditionsRoot*($conditionsroot, fromvars$)
11. $whereclause \leftarrow NIL$
12. *if* $conditionsroot \neq NIL$ *then*
13. $whereclause \leftarrow \text{"where"}$
14. *for each* $p \in adj(conditionsroot)$
15. *BuildQueryFromParameter*($whereclause, adj(p), fromvars$)
16. *if* p *is not the last one then*
17. $whereclause \leftarrow whereclause + \text{" " + boolop}(p) + \text{" "}$
18. $fromclause \leftarrow NIL$
19. *if* $fromvars \neq \emptyset$ *then*
20. $fromclause \leftarrow \text{"from"}$
21. *for each* $fv \in fromvars$
22. $fromclause \leftarrow fromclause + type(fv) + \text{" " + name}(fv)$

23. *if f v is not the last one then*
24. *fromclause* ← *fromclause* + ", "
25. *query* ← *query* + *selectclause*
26. *if fromclause* ≠ *NIL* *then*
27. *query* ← *query* + *fromclause*
28. *if whereclause* ≠ *NIL* *then*
29. *query* ← *query* + *whereclause*
30. *end.*

Builds a SELECT subquery, starting from a given Select, and walking through the graph, following the arcs. This is a recursive function.

Lines 1-2: initialize variables.

Lines 3-6: build the SELECT subquery from each Parameter adjacent to current Select.

Lines 7-8: find the Conditions root.

Lines 9-15: if a Conditions root has been found, build the WHERE clause.

Lines 16-22: build the FROM clause, if fromvars set is not empty.

Lines 23-27: build the SELECT subquery.

In code:

```
function buildTextualQuerySelect(element) {
    var fromVars = [];
    // build SELECT clause
    var selectClause = "select ";
    if (element.distinctSelected) {
        selectClause += "distinct ";
    }
    for ( var i = 0; i < element.parameters.length; i++) {
        var parameter = element.parameters[i];
        selectClause += buildTextualQueryParameter(
            parameter.connection.source, fromVars);
        // if is not the last parameter
        if (i < element.parameters.length - 1) {
            selectClause += ", ";
        }
    }
    var conditionsRoot = findConditionsRoot(fromVars);
    if (conditionsRoot !== undefined) {
        // build WHERE clause
        var whereClause = " where ";
    }
}
```

```

    for ( var i = 0; i < conditionsRoot.parameters.
        length; i++) {
        var parameter = conditionsRoot.parameters[i];
        whereClause += buildTextualQueryParameter(
            parameter.connection.source, fromVars);
        if (i < conditionsRoot.parameters.length - 1) {
            // separate parameters with boolean
            operator
            whereClause += " " + conditionsRoot.
                booleanOperators[i].value + " ";
        }
    }
}
if (fromVars.length > 0) {
    // build FROM clause
    var fromClause = " from ";
    for ( var i = 0; i < fromVars.length; i++) {
        var v = fromVars[i];
        fromClause += v.type + " " + v.name;
        if (i < fromVars.length - 1) {
            fromClause += ", ";
        }
    }
}
var textualQuery = selectClause;
if (fromClause !== undefined) {
    textualQuery += fromClause;
}
if (whereClause !== undefined) {
    textualQuery += whereClause;
}
return textualQuery;
}

```

1. $query \leftarrow ""$
2. $BuildQueryFromSelect(query, startingselect)$
3. $query \leftarrow query + ";"$

Build the Textual Query from the starting Select. Add a semicolon to close the query.

In code:

```

var textualQuery = buildTextualQuerySelect(startingSelect);
textualQuery += ";";

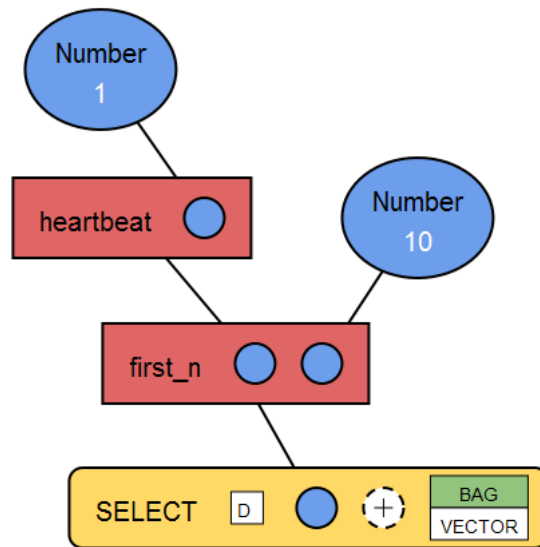
```

8.4 Examples

These examples describes how the translation algorithm is applied to a Visual Query, and which values each element assumes.

8.4.1 Simple example

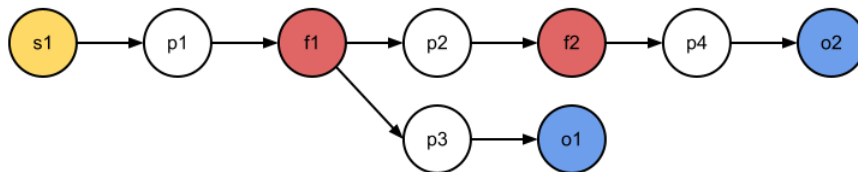
Let us take back the already presented Visual Query example (section §7.2), where the Visual Query is represented by the following figure:



and where the corresponding textual query is:

```
select first_n(heartbeat(1),10);
```

The corresponding graph is:



and its properties are:

$$S = s_1, C = \emptyset, F = f_1, f_2, P = p_1, p_2, p_3, p_4, O = o_1, o_2,$$

$$A = (s_1, p_1), (p_1, f_1), (f_1, p_2), (p_2, f_2), (f_2, p_4), (p_4, o_2), (f_1, p_3), (p_3, o_1),$$

$$value(o_1) = 10, type(o_1) = "Number",$$

$$value(o_2) = 1, type(o_2) = "Number",$$

$$name(f_1) = "firstn", name(f_2) = "heartbeat",$$

$$deg^-(s_1) = 0, deg^+(s_1) = 1,$$

$$deg^-(p_1) = 1, deg^+(p_1) = 1,$$

$$deg^-(f_1) = 1, deg^+(f_1) = 2,$$

...

$$adj(s_1) = p_1, adj(p_1) = f_1, adj(f_1) = p_2, p_3, \dots$$

Correctness tests

G is connected:

- $\forall p \in P, deg^+(p) = 1 \rightarrow \checkmark$,
- $\forall f \in F, deg^-(f) > 0 \rightarrow \checkmark$,
- $\forall o \in O, deg^-(o) > 0 \rightarrow \checkmark$.

G does not contains closed loops composed of F and S vertices.

Let's find the starting Select:

1. *startingselect* : *NIL*
2. s_1 :
3. $deg^-(s_1) = 0 \rightarrow \checkmark$
4. *startingselect* = *NIL* $\rightarrow \checkmark$
5. *startingselect* : s_1

All conditions are verified, therefore the query can be translated.

Query translation

query : ""

BuildQueryFromSelect(*query*, *startingselect*)

fromvars : \emptyset

selectclause : "select"

$distinct(startingselect) = true \rightarrow \times$

$p_1 :$

$BuildQueryFromParameter(selectclause, f_1, fromvars)$

$f_1 \in F \rightarrow \checkmark$

$selectclause : "select firstn("$

$p_2 :$

$BuildQueryFromParameter(selectclause, f_2, fromvars)$

$f_2 \in F \rightarrow \checkmark$

$selectclause : "select firstn(heartbeat("$

$p_4 :$

$BuildQueryFromParameter(selectclause, o_2, fromvars)$

$o_2 \in O \rightarrow \checkmark$

$value(o_2) \neq NIL \rightarrow \checkmark$

$selectclause : "select firstn(heartbeat(1"$

$p_4 \text{ is not the last one} \rightarrow \times$

$selectclause : "select firstn(heartbeat(1)"$

$p_2 \text{ is not the last one} \rightarrow \times$

$selectclause : "select firstn(heartbeat(1),"$

$p_3 :$

$BuildQueryFromParameter(selectclause, o_1, fromvars)$

$o_1 \in O \rightarrow \checkmark$

$value(o_1) \neq NIL \rightarrow \checkmark$

$selectclause : "select firstn(heartbeat(1), 10"$

$p_3 \text{ is not the last one} \rightarrow \times$

$selectclause : "select firstn(heartbeat(1), 10)"$

$conditionsroot : NIL$

$FindConditionsRoot(conditionsroot, fromvars)$

$C = \emptyset \rightarrow \checkmark$

$whereclause : NIL$

$conditionsroot \neq NIL \rightarrow \times$

$fromclause : NIL$

$fromvars \neq \emptyset \rightarrow \times$

$query : "select firstn(heartbeat(1), 10)"$

fromclause \neq *NIL* \rightarrow ✗

whereclause \neq *NIL* \rightarrow ✗

query : "select firstn(heartbeat(1), 10);"

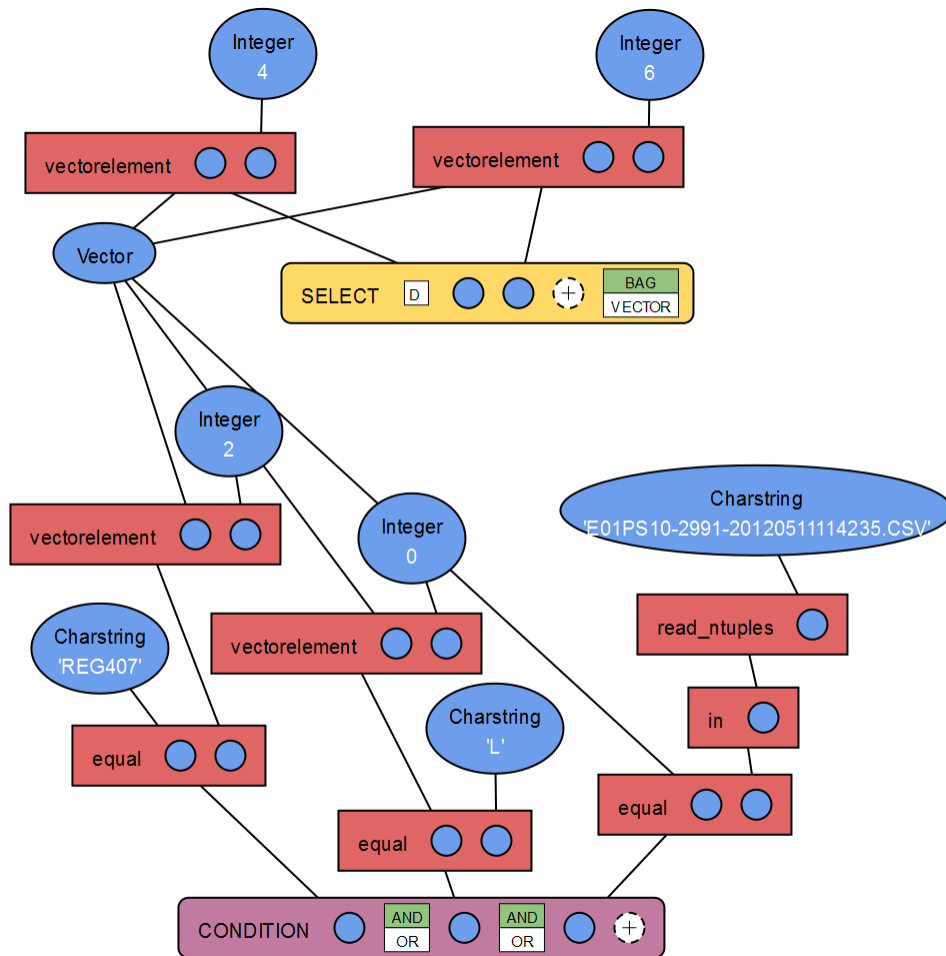
8.4.2 Complex example

Let us now see a more complex example. This example comes from a user scenario where data stored in log files, containing analog/digital measurements produced by a proper control system, is considered. This control system measures data from a wood waste shredder, that includes two driven machine shafts, three electric motors and four hydraulic flow pumps. Is now supposed that the user, an engineer, wants to retrieve the trend related to the pressure B-side pump1, and she wants to compare it with a certain threshold.

The log file is column based (\langle block \rangle, \langle tag \rangle, \langle value1 \rangle, \langle value2 \rangle):

- there are four possible blocks (header, alarms/warnings, settings and log data);
- tag depends on the type of a block and provides more details about it;
- value represents the values associated with the tag or, in case of log data, the timestamp and the measure.

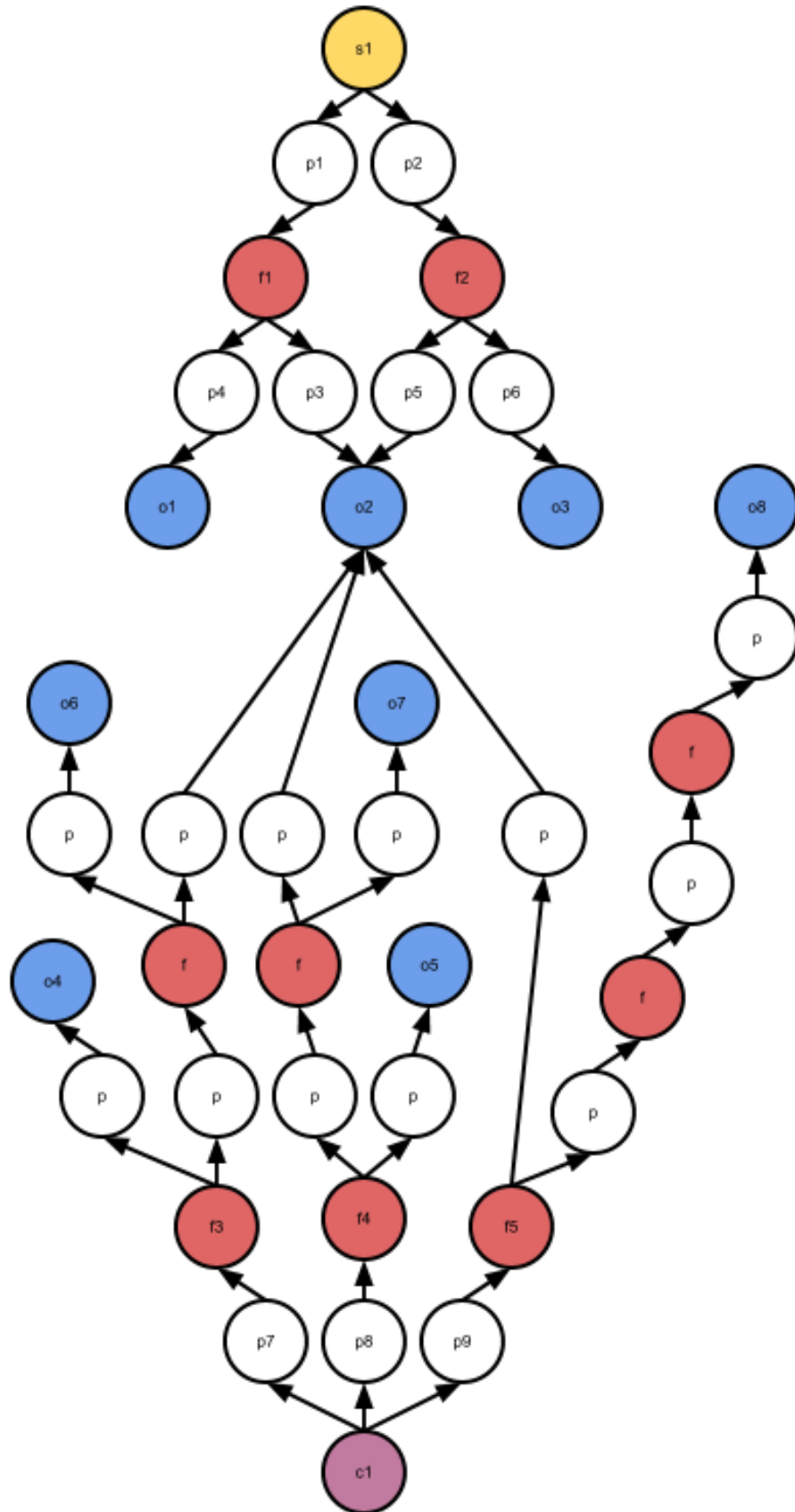
This is the Visual Query used to retrieved desired data:



The corresponding textual query is:

```
select vectorelement(v01, 4), vectorelement(v01, 6) from Vector
v01 where equal('REG407', vectorelement(v01, 2)) and
equal(vectorelement(v01, 0), 'L') and equal(v01,
in(read_ntuples('E01PS10-2991-20120511114235.CSV')));
```

and here's the corresponding graph:



$S = s_1, C = c_1, F = f_1, f_2, \dots, f_9, P = p_1, p_2, \dots, p_{21}, O = o_1, o_2, \dots, o_8,$
 $A = (s_1, p_1), (s_1, p_2), \dots,$
 $value(o_1) = 4, type(o_1) = "Integer",$
 $value(o_2) = NIL, type(o_2) = "Vector", name(o_2) = "v01" ,$
 $value(o_3) = 6, type(o_3) = "Integer",$
 $value(o_4) = "REG407", type(o_4) = "Charstring",$
 $value(o_5) = "L", type(o_5) = "Charstring",$
 $value(o_6) = 2, type(o_6) = "Integer",$
 $value(o_7) = 0, type(o_7) = "Integer",$
 $value(o_8) = "E01PS10-2991-20120511114235.CSV", type(o_8) = "Charstring",$
 $name(f_1) = "vectorelement", name(f_2) = "vectorelement", name(f_3) =$
 $"equal", ...$
 $boolop(p_7) = "and", boolop(p_8) = "and",$
 $deg^-(s_1) = 0, deg^+(s_1) = 1,$
 $deg^-(p_1) = 1, deg^+(p_1) = 1,$
 $deg^-(f_1) = 1, deg^+(f_1) = 2, ...$
 $adj(s_1) = p_1, adj(p_1) = f_1, adj(f_1) = p_3, p_4, ...$

Correctness tests

G is connected:

- $\forall p \in P, deg^+(p) = 1 \rightarrow \checkmark,$
- $\forall f \in F, deg^-(f) > 0 \rightarrow \checkmark,$
- $\forall o \in O, deg^-(o) > 0 \rightarrow \checkmark.$

G does not contains closed loops composed of F and S vertices.

Let's find the starting Select:

1. $startingselect : NIL$
2. $s_1 :$
3. $deg^-(s_1) = 0 \rightarrow \checkmark$
4. $startingselect = NIL \rightarrow \checkmark$
5. $startingselect : s_1$

All conditions are verified, therefore the query can be translated.

Query translation

query : ""

BuildQueryFromSelect(*query*, *startingselect*)

fromvars : \emptyset

selectclause : "select"

distinct(*startingselect*) = true $\rightarrow \times$

*p*₁ :

BuildQueryFromParameter(*selectclause*, *f*₁, *fromvars*)

*f*₁ $\in F \rightarrow \checkmark$

selectclause : "select vectorelement("

*p*₃ :

BuildQueryFromParameter(*selectclause*, *o*₂, *fromvars*)

*o*₂ $\in O \rightarrow \checkmark$

value(*o*₂) $\neq \text{NIL} \rightarrow \times$

fromvars : *o*₂

selectclause : "select vectorelement(*v*₀₁"

*p*₃ is not the last one $\rightarrow \times$

selectclause : "select vectorelement(*v*₀₁, "

*p*₄ :

BuildQueryFromParameter(*selectclause*, *o*₁, *fromvars*)

*o*₁ $\in O \rightarrow \checkmark$

value(*o*₁) $\neq \text{NIL} \rightarrow \checkmark$

selectclause : "select vectorelement(*v*₀₁, 4"

*p*₄ is not the last one $\rightarrow \times$

selectclause : "select vectorelement(*v*₀₁, 4)"

*p*₂ :

...

selectclause : "select vectorelement(*v*₀₁, 4), vectorelement(*v*₀₁, 6)"

conditionsroot : *NIL*

FindConditionsRoot(*conditionsroot*, *fromvars*)

*c*₁ :

deg⁻(*c*₁) = 0 $\rightarrow \checkmark$

connectedvars : \emptyset

$GetConnectedVarsToConditions(c_1, connectedvars)$
 $c_1 \in O \rightarrow \times$
 $p_7 :$
 $GetConnectedVarsToConditions(f_3, connectedvars)$
 \dots
 $GetConnectedVarsToConditions(o_2, connectedvars)$
 $o_2 \in O \rightarrow \checkmark$
 $o_2 \notin connectedvars \rightarrow \checkmark$
 $connectedvars : o_2$
 $p_8 :$
 $GetConnectedVarsToConditions(f_4, connectedvars)$
 $p_9 :$
 $GetConnectedVarsToConditions(f_5, connectedvars)$
 \dots
 $fromvars \cap connectedvars = o_2 \neq \emptyset \rightarrow \checkmark$
 $conditionsroot : c_1$
 $whereclause : NIL$
 $conditionsroot \neq NIL \rightarrow \checkmark$
 $whereclause : " where "$
 $p_7 :$
 $BuildQueryFromParameter(whereclause, f_3, fromvars)$
 $f_3 \in F \rightarrow \checkmark$
 $whereclause : " where equal("$
 $p :$
 $BuildQueryFromParameter(whereclause, o_4, fromvars)$
 \dots
 $p :$
 $BuildQueryFromParameter(whereclause, f, fromvars)$
 \dots
 p_7 is not the last one $\rightarrow \checkmark$
 $whereclause : " where equal('REG407', vectorelement(v01, 2)) and"$
 $p_8 :$
 $BuildQueryFromParameter(whereclause, f_4, fromvars)$
 \dots

p_8 is not the last one $\rightarrow \checkmark$
whereclause : " *where equal('REG407', vectorelement(v01, 2))*
and equal(vectorelement(v01, 0), 'L') and"
 p_9 :
BuildQueryFromParameter(whereclause, f₅, fromvars)
 ...
 p_9 is not the last one $\rightarrow \times$
fromclause : *NIL*
fromvars $\neq \emptyset \rightarrow \checkmark$
fromclause : " *from*"
 o_2 :
fromclause : " *from Vector v01*"
query : " *select vectorelement(v01, 4), vectorelement(v01, 6)*"
fromclause $\neq \text{NIL} \rightarrow \checkmark$
query : " *select vectorelement(v01, 4), vectorelement(v01, 6) from Vector v01*"
whereclause $\neq \text{NIL} \rightarrow \checkmark$
query : " *select vectorelement(v01, 4), vectorelement(v01, 6) from Vector v01*
where equal('REG407', vectorelement(v01, 2)) and
equal(vectorelement(v01, 0), 'L') and
equal(v01, in(readntuples('E01PS10-2991-20120511114235.CSV')))"
query : " *select vectorelement(v01, 4), vectorelement(v01, 6) from Vector v01*
where equal('REG407', vectorelement(v01, 2)) and
equal(vectorelement(v01, 0), 'L') and
equal(v01, in(readntuples('E01PS10-2991-20120511114235.CSV')));"

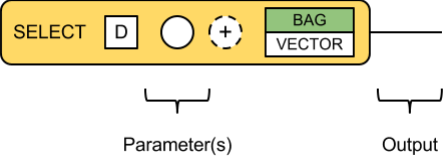
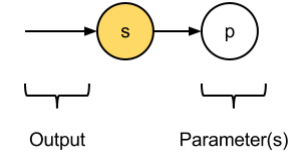
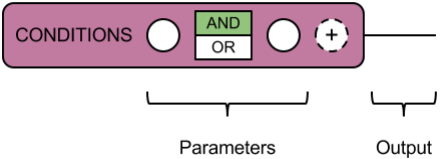
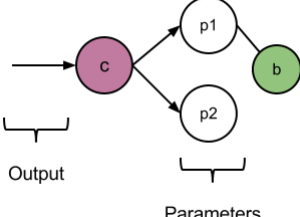
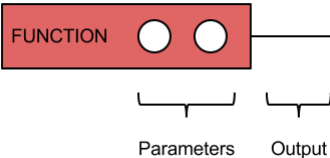
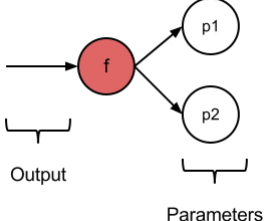
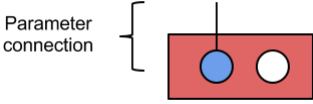
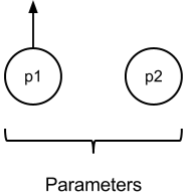
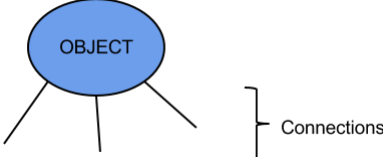
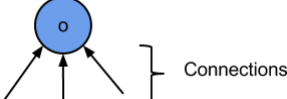
Element	Visual Element	Corresponding Graph
Select		
Conditions		
Function		
Parameters		
Object		

Table 8.1: Elements and corresponding graphs.

Chapter 9

Prototype

In this chapter, the prototype developed to implement the Smart Vortex project's requirements is presented. The prototype can be divided in two parts: the web application and the FDSMS bridge. The web application can be seen as the client side application, used by users to compose Visual Queries, choose how to visualize their execution results, and create groups of queries, called dashboards, with a monitoring or data analysis aim. The FDSMS bridge is used to execute queries on SCSQ and get results back.

An overview of adopted technologies is firstly given. The overall architecture of the prototype is then described, illustrating how the elements are related and how they communicate.

A comprehensive description of each component is then given, which elements compose them, how they are implemented in code.

9.1 Technologies

In the web application, web pages are rendered from HTML and CSS/LESS, using JavaScript to control the browser and communicate asynchronously with the web server. Following, a comprehensive list of the adopted technologies:

- LESS [33] is an extension of CSS, which add dynamic behaviour to it, and it must be compiled or translated into CSS in order to be read by web browsers.
- JavaScript is used to dynamically operate on HTML elements and to retrieve data from the web server, using an AJAX paradigm with JSON as data exchange format.

- Bootstrap from Twitter [34] is a LESS-based front-end framework, used to build the application user interface. Some plugins, like modal dialogs, alerts etc., have been employed too.
- Dojo [35]. A platform-independent JavaScript framework. It is used as the main foundation for DOM manipulation, events handling and web server asynchronous communications. It also has support for canvas drawing, therefore it has been used to create the Visual Query Editor.
- jQuery [36] is a JavaScript library. Bootstrap plugins and following libraries depends on it.
- jQuery UI [37] is a User Interface library built on top of jQuery. Some of its widgets have been integrated in the User Interface, like the Input spinner.
- Timepicker [38]. Date and time picker, used to set values for Date, Time and Timeval objects.
- Highcharts [39] is a charting library written in pure JavaScript. Used to render lines, bars and points charts.
- Canvas Gauge [40] draws gauge charts using pure JavaScript and HTML5 canvas.

Tomcat has been chosen as the web server to handle user requests. Java is the main programming language, and JSP is used to render web pages at server side. MySQL has been adopted to implement the User DataBase, and JDBC drivers are employed to access it through the web server.

FDSMS bridge is written in the Java language, and uses TCP as communication protocol and JSON as data exchange format.

9.2 Architecture

This section describes the general architecture adopted for the overall prototype, and how single elements are related each other. A deep explanation about how each component is organized in files is also given.

9.2.1 Web application

The web application is now presented. It can be seen as the client side application, and can be divided into three components: presentation layer and data exchange, web server and User DataBase.

The presentation layer is the real connection between the user and the whole application. Its purpose is to let users work on queries and visualize their results, through a web-based user interface. Being composed of web pages, it is accessed by a desktop web browser. These web pages are dynamically updated and modified using JavaScript instructions, which send asynchronous requests to the web server, retrieving data to show up on web pages.

The web server is used to handle users requests, replying with web pages and/or data retrieved from the User DataBase. It also handles the communications with the FDSMS. Since it has responsibility for communication between different application actors, it can be seen as the core of the prototype. It is composed of different resources and files, in order to handle these tasks:

- `Index.jsp`: web application starting point, it creates a new `DBmanager` instance and it stores it into the user session, which is then retrieved by other pages in order to access DataBase data and build HTML pages.
- `Home.jsp`: web page with dashboards and templates lists. Dashboards and templates can be created, duplicated and removed.
- `Dashboard.jsp`: displays a dashboard, which is a collection of queries and their visualizations. Queries can be created, duplicated or removed on the dashboard.
- `Query.jsp`: web page used to create a Visual Query and define a visualization for its results.
- `Template.jsp`: same as `query.jsp`, but for templates. Templates are not meant for execution, therefore they cannot have a results visualization.
- `Exec.jsp`: replies to asynchronous client requests. In fact, it serves as a “gate” between the presentation layer and the User DataBase or the FDSMS bridge.
- `DBmanager.java`: it handles User DataBase connection and communication, using convenient methods to retrieve and to store data.
- `Dashboard.java`, `Query.java`, `Template.java`, `Comment.java`: classes representing corresponding User DataBase entities.
- `FDSMSclient.java`: it implements methods to handle communications with the FDSMS bridge, to execute queries on it and to retrieve execution results.

- JavaScript files: Bootstrap plugins, JavaScript libraries and other scripts used by web pages. They contains dynamic code used to interact with the HTML pages, and to exchange asynchronous data with the web server.
- CSS/LESS files: web pages style-related files. Some CSS files are used by JavaScript libraries.
- Images: graphic resources incorporated in web pages.

The User DataBase stores user related data, such as created queries and templates, dashboards and comments. Data retrieved from the DataBase is used to build web pages.

figure 9.1 describes the web application architecture, and how the elements are related. Each color describes a different kind of interaction: red lines indicate a recall of other resources inside the web application; blue lines denotes a use of related Java classes to store and represent DataBase data; green lines represents communications with components external to the web application.

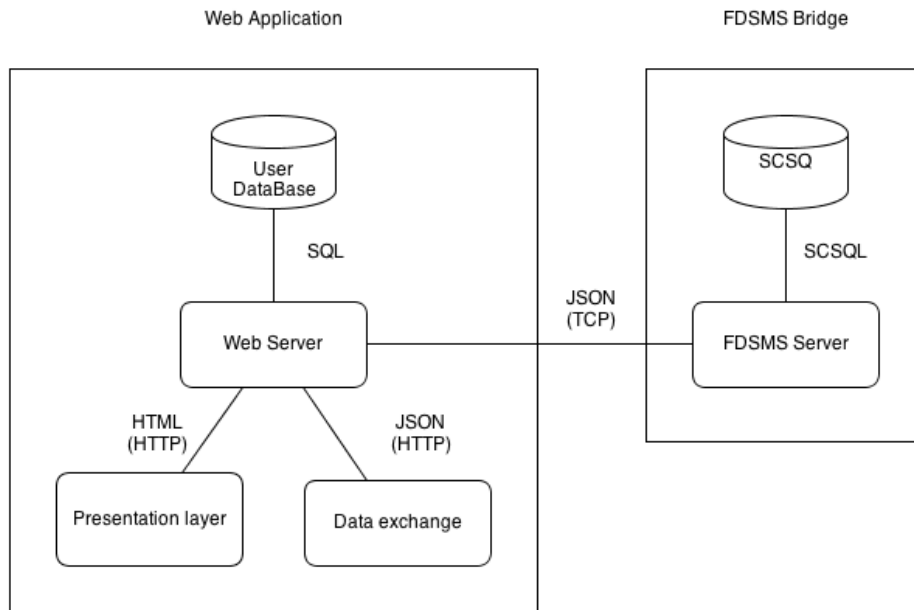
9.2.2 FDSMS bridge

The FDSMS bridge represents a connection layer with SCSQ, the FDSMS, and it executes queries over it, returning their results back. It is a multithread server that forwards requests to SCSQ and return results properly formatted in JSON. It also responsible for retrieving all the types and functions declared on SCSQ. The FDSMS bridge exposes a complete set of methods, supporting previously described operations.

SCSQ is the backend where queries written in SCSQL are executed.

9.2.3 Architecture overview

Figure below summarizes how the system is composed:



It can be seen as a two tiers system, represented by two rectangles. The left one represents the web application, containing the presentation layer and data exchange, the web server and the User DataBase. The right block represents instead the FDSMS bridge. Over each connecting line, the data exchange format and the communication protocol (inside parenthesis) are provided.

9.3 Web application

The web application, as already described, is composed of different elements. In this section, however, only the web server and the presentation layer are deeply analyzed and illustrated, postponing the description of the remaining elements in the following sections.

In particular, elements composing the user interface are presented, describing their functionalities, and how they are implemented in code.

9.3.1 Visual Query Editor

The Visual Query Editor (VQE) is presented, is the main interface for the Visual Queries creation. It collects and presents to the user all the elements

from the FDSMS, and it assists the Query composition with different facilitations.

VQE components

The VQE (figure 9.2) is composed of three main components: the Canvas, the Elements Toolbox and the Toolbar.

Canvas The Canvas is used to create the Visual Query. Here elements are added, can be removed, and can be connected, in order to compose the final query. Elements can also be freely arranged on the canvas.

Elements Toolbox The Elements Toolbox (ET) contains all the VQL elements which can be added to the Canvas, grouped by: Types, Functions, Data Sources and Instructions. Each ET element is a button, used to add the corresponding element to the Canvas. Types and Functions are retrieved from SCSQ, so they represent all possible types and functions declared in it. Functions are represented as tables (figure 9.3), where the header, in red, is the function name, white left-aligned rows represent parameters' types, and blue right-aligned rows represent result's type. Types and Functions groups have also a filter functionality, which is presented later. The Data Sources group gather Objects representing sources from where data is collected. In figure 9.4, for example, "Mill A", "Mill B" and so on, points to data streams originated from sensors installed on milling machines. Finally, the Instructions group contains the SCSQL statements, useful to compose the Visual Query. Despite in figure 9.5 are shown many different statements, as explained in chapter 7, only Select and Conditions are currently supported.

Toolbar The Toolbar is placed over the Canvas, and it contains two buttons: "Set object value" and "Remove/Clear all elements" (see figure 9.6). The "Set object value" button is used to set a Value over an Object (further details are provided below). The "Remove/Clear" button removes an element or clear all the elements from the Canvas.

VQE interactions

This section explains how to interact with the VQE, and how it gives assistance on composing a Visual Query.

Create Elements can be created using the ET. Each ET element is a button, and clicking on it creates a new corresponding VQL element on the Canvas.

Place Elements can be rearranged on the Canvas by drag and drop.

Change properties Element properties, like the DISTINCT clause for Select elements, can be changed by clicking on them (see figure below).



Select/Deselect Clicking on an element selects it. Selected elements are highlighted, with a lighter border, and remain highlighted even when mouse leaves the element; they can then be removed or connected with other elements through the Handle. As explained in details later, an element selection activates corresponding ET filters. Clicking again on the element deselects it.

Remove Only selected elements can be removed. After selecting an element, the Remove button on the Toolbar becomes enabled, and the selected element can be removed from the Canvas by clicking on it. To remove a connection, the connected parameter must be selected. The Canvas can also be cleared from all elements by clicking on the dropdown placed at the right side of Remove button, and selecting “Clear all elements” (figure 9.7).

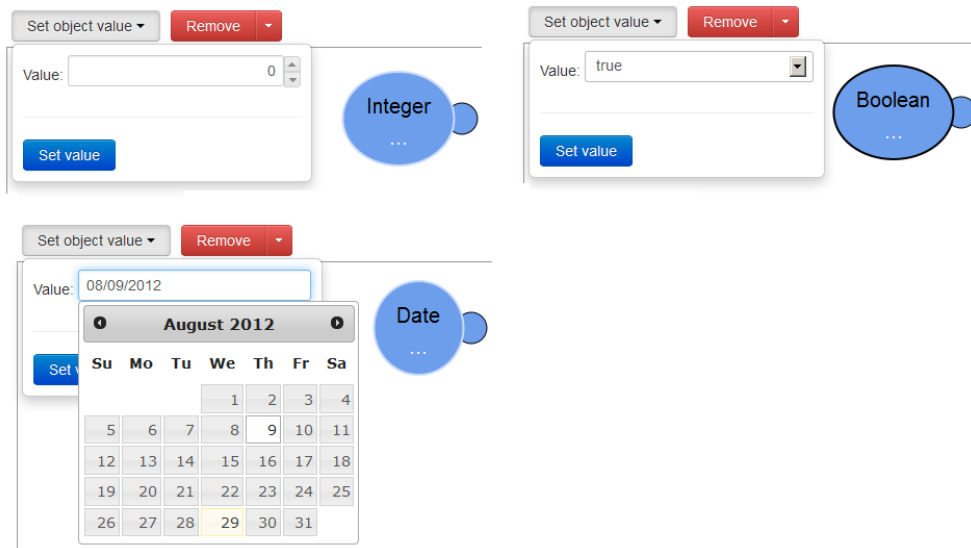
Connect In order to connect an element with another one, firstly it must be selected, so the Handle becomes visible. The Handle is used to connect the element by dragging it, and dropping it over the target (see figure 9.8). Not all of the elements can be connected each other; table 9.1 summarizes permitted connections. When the target is a Parameter, a previous Type check is performed: only if the Parameter accepts the Handle Type, a connection is created. The target can also be the Canvas; in this case, when the handle is dropped on it, a new Object of the same Type is created.

Select and Condition Parameters Select and Conditions Parameters can be added and removed at need, as explained in chapter 7.

	Connectable elements
Object	Parameters which accept the same Type
Function, Select, Conditions	Parameters which accept returned Type
Parameters	Objects of accepted Type, and Functions/Selects/Conditions which return accepted Type

Table 9.1: Permitted connections.

Set object value Certain Types can have a Value, e.g. Charstring, Integer, Number, Boolean, Time, Date, Timeval. Each of these Types have associated an ad-hoc panel, used to set its Value; by now, only some default Types are supported, but further panels can be added in future. To set a value, firstly the Object have to be selected. When an Object is selected, the “Set object value” button in Toolbar becomes enabled, and clicking on it, the associated panel is shown. After having specified the desired value, clicking on “Set value” sets it on the Object. Some examples of panels are shown in the following figure:



Composition facilities

During the query creation, the VQE supports the user with different facilities, in order to ease the composition.

As already mentioned, the Types and Functions groups in the ET present filters, which allow to easily find and use desired elements. These filters depends on the ET group.

Types have two different kinds of filters:

- **Name contains:** a string can be specified, and only Types matching that string are displayed.
- **Accepted by [function or parameter]:** selecting an element (Function/Select/Conditions), only Types accepted by selected element's Parameters will be shown.

Functions have four different kinds of filters:

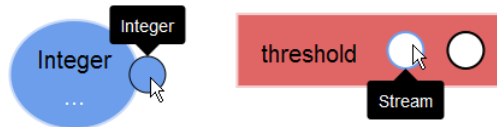
- **Name contains:** a string can be specified, and only Functions matching that string are shown.
- **Attributes of type [type]:** only attributes of selected Object are shown (an attribute is a Function which accepts only selected Object's Type as parameter)..
- **Accepts [type] as parameter:** selecting an Object, only Functions which have at least one Parameter which accepts selected Object's Type are shown.
- **Returns parameter(s) type(s) as result:** when selecting a Function/Select/conditions or a Parameter, only elements which returns accepted Type are shown.

Some filters can be manually activated ("Name contains", for example), others are automatically activated when a VQL element is selected. These latter are activated depending on the selected element itself. Following table summarizes this behaviour:

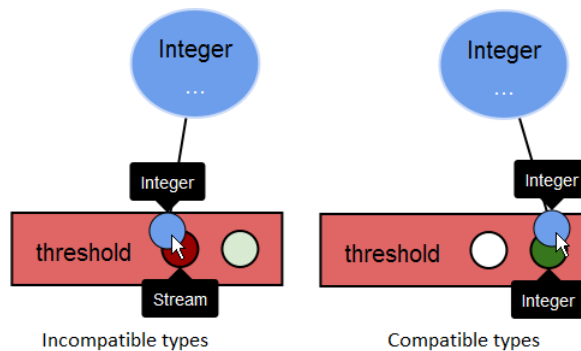
	Types	Functions
Object	-	Functions whose parameters accept selected object type
Function, Select, Conditions	Types accepted by selected element parameters	Functions which accept returned type as parameter, Functions whose parameters accept returned type
Parameters	Types accepted by selected parameter	Functions which return accepted type

Filters can be composed, so for example, selecting an Object, and specifying a Function name (or only a piece of it), only functions accepting that Object type and matching that string are shown.

Another facilitation is represented by tooltips over Parameters and the Handle, used to indicate its Types; they are shown on mouse hover. Tooltips on the Handle is always shown above it, while tooltips on Parameters are always shown below them. An example is shown in the following figure:



When trying to connect an Object with a Parameter, firstly there will be a Type compatibility check, and only if Types are compatibles the connection is created. This check is performed when the Handle enters the target Parameter, and a visual representation of check result is shown: the Parameter becomes red if Types are incompatible, green otherwise:



Finally, when selecting an element, connectable elements are highlighted in green, therefore they can be easily found. figure 9.9 shows an example where an Integer Object is selected, and the second Parameter of THRESHOLD Function is highlighted in green, showing that the Integer Object can be connected with it.

Implementation

Each VQL element has a graphical representation on the Canvas and a JavaScript object describing its attributes and methods. These objects have the following common properties:

- *type*. It denotes the VQL element's type, can be "object", "function", "select", "conditions" or "parameter".
- *objectType*. It is the FDSMS type associated with the element. For Object elements, it represents the Object Type itself. For Function, Select and Conditions, it represents the result's Type. For Parameters, it represents the accepted Type.
- *connections*. An array of Connections created on the element.
- A set of graphical properties, used to draw the element itself on Canvas.

The elements also share some common methods:

- *highlight()/unhighlight()*. These methods are used to emphasize selected elements.
- *serialize()*. It creates a textual representation of the Visual Query, which can then be saved on the User DataBase.

Depending from the element, other specific attributes or methods are provided, used, for instances, to set the Object Value or the DISTINCT property for a Select element.

Each element is created by a specific function (*createObject()*, *createFunction()*, *createSelect()*, *createConditions()*), and each of these functions, after creating the corresponding element, calls the *setupElement()* function which sets common attributes and methods.

When an element is removed from the Canvas, the *removeSelectedElement()* function is called, which operates differently based on selected element's type. However, the main pattern is:

- remove connections from that element;
- remove the shape from the Canvas;
- remove the element from elements arrays.

The two latter points are not performed for Parameters.

If a click is performed over an element, firstly there is a check to determine if the click is a drag and drop or a click-and-release. Then, if verified the intention for a single click, there are three different cases:

- **no other elements were selected.** The *selectElement()* function is called which, from other things, creates the Handle for elements that are not Parameters.
- **Same element is clicked.** This operation corresponds to unselect the element. The *unselectElement()* function is called, which sets currently selected element as unselected (removing the highlight from it), removes the Handle if present, and removes the green highlight from connectable elements.
- **Another element was already selected.** The function *changeSelectedElement()* is called, which changes currently selected element with the new one.

Connection of two elements starts by dragging the Handle. Every time the Handle moves over the Canvas, all Parameters are checked in order to find if the Handle hovers one of them:

```
dojo.connect(handle.moveable, "onMoved", function(mover,
    shift) {
    ...
    var handleBB = handle.body.getTransformedBoundingBox();
    var handleX = handleBB[0].x + (handleBB[1].x - handleBB
        [0].x) / 2;
    var handleY = handleBB[0].y + (handleBB[3].y - handleBB
        [0].y) / 2;
    var parameterTargetFound = false;
    dojo.forEach(parameters, function(parameter, i) {
        var parameterBB = parameter.body.
            getTransformedBoundingBox();
        var parameterX = parameterBB[0].x + (parameterBB
            [1].x - parameterBB[0].x) / 2;
```

```

var parameterY = parameterBB[0].y + (parameterBB
    [3].y - parameterBB[0].y) / 2;
var distanceX = Math.pow(handleX - parameterX, 2);
var distanceY = Math.pow(handleY - parameterY, 2);
var distance = Math.sqrt(distanceX + distanceY);
    if (distance < parameterRadius +
        parameterRadius
        && !parameter.connected
        && parameter.element !== handle.element
        && (handle.element.type === "object"
            || handle.element.type === "function"
            || (handle.element.type === "select"
                && parameter.element.type === "function"
                ))
            || (handle.element.type === "conditions"
                && parameter.element.type === "
                conditions")) {
        parameterTargetFound = true;

```

If a Parameter has been found, it is checked if it can be connected with the selected one:

```

...
else if (parameterTargetFound) {
    parameterTarget.conform(handle.objectType);
}
...
parameter.conform = function(objectType) {
    this.conforming = true;
    ...
    this.isConform = isSameTypeOrSubtype(objectType, this.
        objectType);
    if (this.isConform) {
        this.body.setFill(greenDark);
    } else {
        this.body.setFill(redDark);
    }
};

```

In order to connect an element with a Parameter, it has to accept the element's SCSQ type. This can be checked by the function *isSameTypeOrSubtype()*:

```

function isSameTypeOrSubtype(type, typeOrSubtype) {
    // same SCSQ type

```

```

    if (type === typeOrSubtype) {
        return true;
    }
    // check if type is a subtype of typeOrSubtype
    for (var j = 0; j < allTypes.length; j++) {
        if (typeOrSubtype === allTypes[j].type) {
            if (dojo.indexOf(allTypes[j].subtypes, type) >=
                0) {
                return true;
            }
        }
    }
    return false;
}

```

If target Parameter accepts selected element's SCSQ type, then they can be connected when mouse is released. Connection is created by the function *createConnection()*:

```

function createConnection(source, destination) {
    // store connection properties
    var connection = new Object();
    // add this connection to array of created connections
    connections.push(connection);
    ...
    source.connections.push(connection);
    destination.connection = connection;
    connection.source = source;
    connection.destination = destination;
    ...
}

```

It has two parameters, *source* and *destination*. *Source* can be an Object, a Function, a Select or a Conditions element, *destination* is always a Parameter. Moreover, the connection object is stored also in *source* and *destination* objects representations, therefore the connection can be easily retrieved later. Two arrays are adopted to store Types and Functions in the ET, respectively *allTypes* and *allFunctions*.

Types are represented with JavaScript objects, which have two properties:

- *type*: the FDSMS type name;
- *subtypes*: an array of FDSMS types, subtypes of corresponding type.

Example of NUMBER FDSMS type: *{type: "Number", subtypes: ["Real", "Integer"]}*.

Functions are represented with JavaScript objects, which have four properties:

- *func*: the Function name;
- *description*: what this Function does and/or how it operates;
- *parameters*: Function Parameters;
- *results*: returned Types.

Example of PLUS FDSMS function: *{func: "plus", description: "sums two numbers", parameters: ["Integer", "Integer"], results: ["Integer"]}*.

When a filter changes its state, corresponding elements are sifted, and only elements corresponding to the activated filters are shown. This is performed by two functions: *updateTypesFilter()* for Types, and *updateFunctionsFilter()* for Functions. These functions have a similar behaviour:

- clear the elements list;
- for each element, if it fits the filters and their options, then it is added to the elements list.

As already said, there are filters on Types and Functions. For each filter there is a JavaScript function in charge of verifying if an element fits it or no. Following, the list of these functions with their implementation details (squared brackets represents selected item).

Filters for Types are:

- *Name contains*: only types containing given string are shown.

```
function typeNameContains(type, text) {
    text = text.split(" ");
    var re = ".*";
    for (var i = 0; i < text.length; i++) {
        var w = text[i];
        re += w + ".*";
    }
    var typeName = type.type;
    return typeName.match(new RegExp(re, "i")) !== null
    ;
}
```

- *Accepted by [function]*: when selecting a Function or a Parameter, only accepted Types are shown.

```
function isTypeAcceptedBySelectedElement(type) {
    var accepted = false;
    if (selectedElement.type === "parameter") {
        accepted = isSameTypeOrSubtype(type.type,
            selectedElement.objectType);
    } else {
        for (var k = 0; k < selectedElement.parameters
            .length; k++) {
            var parameter = selectedElement.parameters[
                k];
            accepted = isSameTypeOrSubtype(type.type,
                parameter.objectType);
            if (accepted) {
                break;
            }
        }
    }
    return accepted;
}
```

Filters for Functions are:

- *Name or description contains*: only Functions containing specified string are shown.
- *Attributes of type [type]*: only attributes of selected Object are shown (an attribute is a function which accepts only selected object's type as parameter).

```
function isFunctionAttributeOfSelectedObject(f) {
    if (f.parameters.length === 1) {
        var type = selectedElement.objectType;
        return isSameTypeOrSubtype(type, f.parameters
            [0]);
    }
    return false;
}
```

- *Accepts [type] as parameter*: selecting an Object, only Functions which have at least one Parameter which accepts selected Object's Type are shown.

```
function isFunctionAcceptsSelectedObject(f) {
    var type = selectedElement.objectType;
    var accepted = false;
    for (var k = 0; k < f.parameters.length; k++) {
        var parameter = f.parameters[k];
        accepted = isSameTypeOrSubtype(type, parameter)
            ;
        if (accepted) {
            break;
        }
    }
    return accepted;
}
```

- *Returns parameter(s) type(s) as result*: when selecting a Function/Select/Conditions or a Parameter, only elements which returns accepted Type are shown.

```
function isReturnsTypeAcceptedBySelectedElement(f) {
    if (f.results.length == 0) {
        return false;
    }
    var type = f.results[0];
    var accepted = false;
    if (selectedElement.type === "parameter") {
        // check if type is accepted by selected
        parameter
        accepted = isSameTypeOrSubtype(type,
            selectedElement.objectType);
    } else {
        // check if type is accepted by at least one
        parameter
        for (var k = 0; k < selectedElement.parameters
            .length; k++) {
            var parameter = selectedElement.parameters[
                k];
            accepted = isSameTypeOrSubtype(type,
                parameter.objectType);
        }
    }
}
```

```

                if (accepted) {
                    break;
                }
            }
        }
        return accepted;
    }

```

9.3.2 Data visualization

A Visual Query, as being a SELECT-FROM-WHERE query, returns back results from its execution, depending on the query itself. The user has the ability to choose how to visualize these results, to better exploit the information potential of these data.

The number of results, and the Type of each of them, is related to the Functions connected to the Select element (or, as said in other words, on results returned from the SELECT statement). In figure 9.10, the composed Visual Query returns two results, and in particular two Numbers. These results may come, for example, from a query like SELECT POWER(“SENSOR 1A”) FROM “MILLING MACHINE C” WHERE ..., which returns power consumption measured on a certain sensor located on a certain machine; the function POWER(“...”), in this example, returns a stream of tuples composed of two values: power (result 2) and corresponding time stamp (result 1).

Each of these results can be associated with a visualization through the properties described later. Taking back the example, as shown in figure below, the user has chosen to visualize these results with a line chart, associating result 1 (time stamp) with the X axis, and result 2 (power) with the Y axis, both on the same data series.

The application provides different kinds of visualizations to choose from: lines, bars, points, gauges and semaphores. By now, a single visualization of each type can be used, and is activated through a dedicated checkbox.

Each visualization has different properties associated, depending on visualization itself, but they all share some common ones. These properties includes:

- Chart name: a visualization can have a meaningful name, describing depicted data.
- Associated results: each activated visualization has assigned one or more results to show up. How to assign results to visualizations depends

on visualization itself, so for example on lines, bars and point charts, which are Cartesian charts, results are associated with axis and series, while on gauges and semaphores are simple values.

- **Threshold over/under:** a threshold is useful to depict errors on data, colouring in a different way values which are above/below the threshold (red, for example).

Lines, bars and points charts share some other properties:

- **X and Y axis names:** as for chart name, axis name can be used to label data on each axis, even providing a measurement unit.
- **Series:** a series is a sequence of data points, typically measured at successive time intervals. As series are associated with Cartesian charts, each data point has an X and an Y value. Series can be added or removed at will on each chart, and each one can have a different meaningful name.
- **Stream:** even gauges and semaphores can be used to show up data streams, but when this property is activated on lines, bars and points charts, they use a different visualization specific for flowing values.

Gauges have some own properties, like:

- **Units:** displayed value can has a label which shows measurement unit. This must be provided by the user.
- **Min/Max:** the user can choose the measurement scale at which data is displayed. If this option is omitted, the gauge will scale automatically.

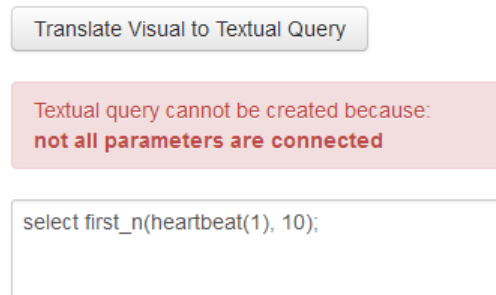
Following figure shows how chosen properties are reflected on the final visualization:



9.3.3 Query execution

A Visual Query have to be translated into a textual query written in SC-SQL language, in order to be executed on FDSMS and get results back. In chapter 8 has been already presented how this is done from a mathematical point of view, and this section explains how this is integrated with the user interface.

The translation is automatically performed by the application, but it must be manually activated by the user, before executing it. This can be done by pressing on “Translate Visual to Textual Query” button. As already explained in chapter 8, the query must pass some correctness tests before being translated; these tests include a check if all elements are connected, if the query ends with a Select, and if there are closed loops composed of Functions and Selects. Check errors appear in a red alert panel (as seen in figure below) with a description of the error, so that the user knows immediately what was wrong; this way, the user is aided in recognizing errors made during query composition.



If the visual query passes all tests, it is translated into a textual query, and then it can be executed. The query is executed by clicking on the “Execute query” button as shown in figure 9.11.

Once the query is executed on the FDSMS, results are received and visualized on chosen charts. Query execution can even be stopped, which means that results are no more retrieved from the FDSMS bridge; query execution can be then resumed, and new results are visualized. The system also offers the possibility to specify how many items a time are shown from the results, selecting the value in the “Get elements a time” field. Doing this, the user has the opportunity to simulate tuple based sliding window.

Implementation

figure 9.12 summarizes, with a sequence diagram, how the query translation and execution is done, and how different resources are called and interact. The `FDSMSClient.java` class maintains a data structure called *dataPool*, where results returned from queries execution are stored. This data structure is a map, which associates each executed query with its results, therefore they can be easily retrieved for visualization purposes. With this approach, data received from streams can be retrieved with consecutive asynchronous requests, without interfering with other operations. Furthermore, the data structure can be concurrently accessed, and updated with new data while other threads reads stored values.

The JavaScript code in *query.js* file sends an *executeQuery* asynchronous request to *exec.jsp*, on the web server, with the query to execute as parameter (among other properties). Then, it starts a timer, which asks *exec.jsp* for new query results every one second. Every time new results are returned, they are displayed through chosen visualizations, until the end-of-data attribute is received.

```

// send query to execute to SCSQ server
request.get("exec.jsp", {
  query : {
    func : "executeQuery",
    queryID : queryID,
    query : query,
    bufferSize : bufferSize
  }
}).then(function() {
  // start a timer, used to retrieve new results
  // from server every 1 second
  queryDataRetrieving = setInterval(function() {
    // retrieve new results from server
    request.get("exec.jsp", {
      query : {
        func : "getData",
        queryID : queryID
      },
      handleAs : "json"
    }).then(function(data) {
      // query execution ends
      if (data.hasMoreData) {
        // stop timer
        clearInterval(queryDataRetrieving);
        return;
      }
      if (!executionStopped) {
        // for each result
        for ( var i = 0; i < data.length; i++) {
          var result = data[i];
          // if an execution error occurs
          if (result.error) {
            // show it
            errorPanel.innerHTML = result.error
            ;
            domClass.remove(errorPanel, "hide")
            ;
            return;
          }
        }
        var results = result.results;
        // display results based on
        // chosen visualizations
        if (visualizationConfiguration.visLines

```

```

        )
        ...
        if (visualizationConfiguration.visBars)
        ...
        if (visualizationConfiguration.visMarks)
        )
        ...
        if (visualizationConfiguration.visGauge)
        )
        ...
        if (visualizationConfiguration.visSemaph)
        ...
    }
}
});
}, 1000);
});

```

exec.jsp, from its side, simply forwards these requests to the FDSMS client. How the FDSMS client interacts with the FDSMS bridge to execute queries is explained in section §9.4.

```

else if (function.equals("executeQuery")) {
    Integer queryID = Integer.parseInt(request.getParameter("queryID"));
    String query = request.getParameter("query");
    Integer bufferSize = Integer.parseInt(request.getParameter("bufferSize"));

    SCSQclient.executeQuery(queryID, query, bufferSize);
}

else if (function.equals("getData")) {
    Integer queryID = Integer.parseInt(request.getParameter("queryID"));

    out.print(SCSQclient.getQueryData(queryID));
}

```

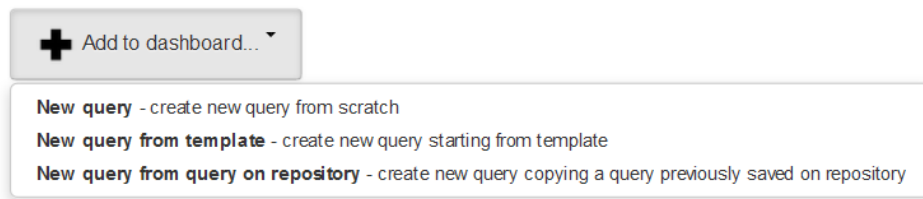
9.3.4 Dashboard

A dashboard is a collection of queries and their visualizations. As the name suggests, its function is to aggregate different queries' results, and display

them through associated visualizations, with a monitoring aim. In fact, one of the main objectives of the Smart Vortex project is to facilitate data analysis and supervision over time, and a dashboard is intended for this use.

As said, a dashboard is a collection of queries, which can be added or removed at will. These queries are associated only to the dashboard on where they are created, and the user can choose to create a new query from scratch, meaning that the VQE will open with an empty Canvas, or to start from an existing template or previously created query (see figure below); these two latter options are explained in details later.

Dashboard title



When a query is added to a dashboard, it is displayed using its associated visualization, and directly executed, so the visualization will show up its results. figure 9.13 shows a dashboard example where three different queries have been added to it, and executed in parallel. In this case, the aim of the dashboard is to monitor power deviations from an expected value, measured from different sensors on different machines.

A dashboard can also has a meaningful name, given by the user, explaining its purpose and easing its finding.

9.3.5 Saved queries

As said, each query can be associated to a single dashboard, and it cannot be shared between different dashboards. There are cases, though, where it would be useful to have the same query visualized on different dashboards, or to have the same query visualized with different parameters. In these cases, an already composed query can be saved on a common query repository, by a dedicated action of the VQE (see figure below). Saved queries can then be reused on other dashboards, as explained above.

Query title

The screenshot shows a user interface for editing a query. At the top, there is a section titled "Query title". Below this, there is a "Query" dropdown menu. A tooltip or dropdown menu is visible below the "Query" dropdown, containing the text "Save on queries repository". Below the "Query" dropdown, there are three tabs: "Types", "Functions", and "Data Sources". Below the tabs, there is a "Filter" section. The "Filter" section has a checkbox labeled "Name contains:" followed by an empty input field.

9.3.6 Templates

Templates, as the name suggests, are particular kinds of query models, a starting point for the creation of other queries. Templates cannot have any visualization associated, and they cannot be executed. They still are Visual Queries composed through the VQE, and can be used when new queries are added to a dashboard, as seen above.

When the same query is going to be utilized on different (or the same) dashboards, but with different parameters, it would be useful to start from a common template, previously created, and then instantiate each query with proper values.

9.3.7 Comments

Users can comment on dashboards and queries, about everything concerning them (figure below). For example, a user can point out that a query has a problem and should be changed, or can comment on queries results.

The screenshot shows a comment interface. It features a user profile for "Alberto" with a silhouette icon. Below the profile, there is a comment text: "Milling machine A has a problem..". Below the comment text, there is another user profile for "Alberto" with a silhouette icon and an empty input field. At the bottom right of the comment box, there is a blue button labeled "Comments" with a downward arrow.

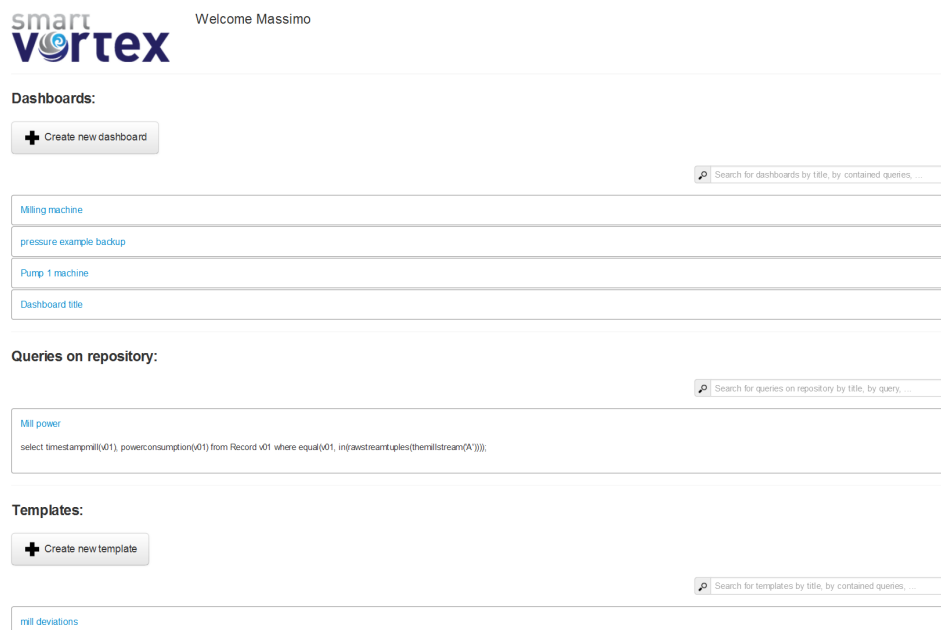
Even if the objectives of the Smart Vortex project for this year do not

complement the collaboration between users, this feature has been introduced anyway, and it will be improved in future researches.

9.3.8 Overall view

By now, each element composing the web application has been presented and illustrated, giving an explanation about how to interact with each component, and how they operate. This section gives an overview of the entire application, showing how these elements are related.

The main view of the web application is the home page (figure below). Here, an overall view of all of the elements created by the user is given, in particular the list of created dashboards, the list of queries saved on repository and the list of created templates are presented. These elements are grouped by typology, and each group is searchable, therefore a corresponding element can be easily found. From here, new dashboards and new templates can be created. Clicking on an element shows up further information about it, and clicking on its name opens it up.



The screenshot displays the 'smart vortex' web application interface. At the top left is the logo, and at the top right is the text 'Welcome Massimo'. Below this, there are three main sections:

- Dashboards:** A button labeled '+ Create new dashboard' is on the left. On the right is a search bar: 'Search for dashboards by title, by contained queries, ...'. Below these are four list items: 'Milling machine', 'pressure example backup', 'Pump 1 machine', and 'Dashboard title'.
- Queries on repository:** A search bar: 'Search for queries on repository by title, by query, ...'. Below it is a query example: 'Mill power' followed by the SQL query: 'select timestamp(v01), powerconsumption(v01) from Record v01 where equal(v01, in(rawstreamuples(thermillstream(A))))'.
- Templates:** A button labeled '+ Create new template' is on the left. On the right is a search bar: 'Search for templates by title, by contained queries, ...'. Below it is one list item: 'mill deviations'.

When clicking on a dashboard name, the corresponding dashboard is opened. As already explained, a list of associated queries is presented, with their visualizations. Dashboard name can be changed by clicking on it, and clicking on each query name opens the VQE with the corresponding query. Even

when adding a new query to the dashboard opens the VQE, and which query is opened depends on the corresponding selected action. For example, if the action “Create new query from scratch” is selected, the VQE will open with an empty Canvas, or, if the selected action is “Create new query from template”, the user is prompted to choose which template to use, and then the VQE will open with chosen template on Canvas.

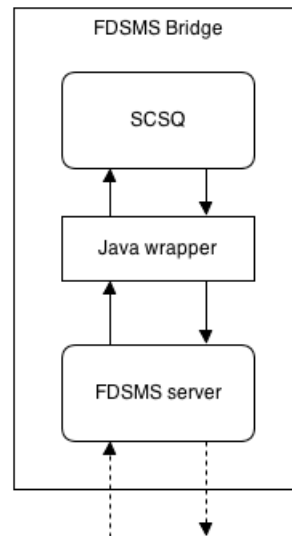
When a query is edited, the user has the ability to modify the Visual Query through the VQE (figure 9.2), choose the results visualizations (figure 9.10), or execute it (figure 9.11), all on the same page. This way, the working flow of a query creation is never broken, and the user can build the query and choose its visualization incrementally, directly displaying results and correcting possible issues.

The same approach is followed when the name of a query saved on repository, from the home page, is clicked. The VQE will open with the corresponding query shown on Canvas, and the user can modify the visualizations properties or execute the query.

When editing a template however, as explained, only the Visual Query can be modified, therefore, when clicking on the template name from the home page, only the VQE will open, without visualization options or the possibility to execute the template (in fact, a template cannot be executed by definition).

9.4 FDSMS Bridge

The FDSMS Bridge is a stand-alone server, used to execute queries on SCSQ and get results back. It is a multithread TCP server that forwards request to SCSQ and returns results properly formatted in JSON. It is also responsible for retrieving all types and functions declared in the SCSQ FDSMS, during the web application user interface start up. The FDSMS Bridge can be divided in two blocks: the FDSMS server and SCSQ.



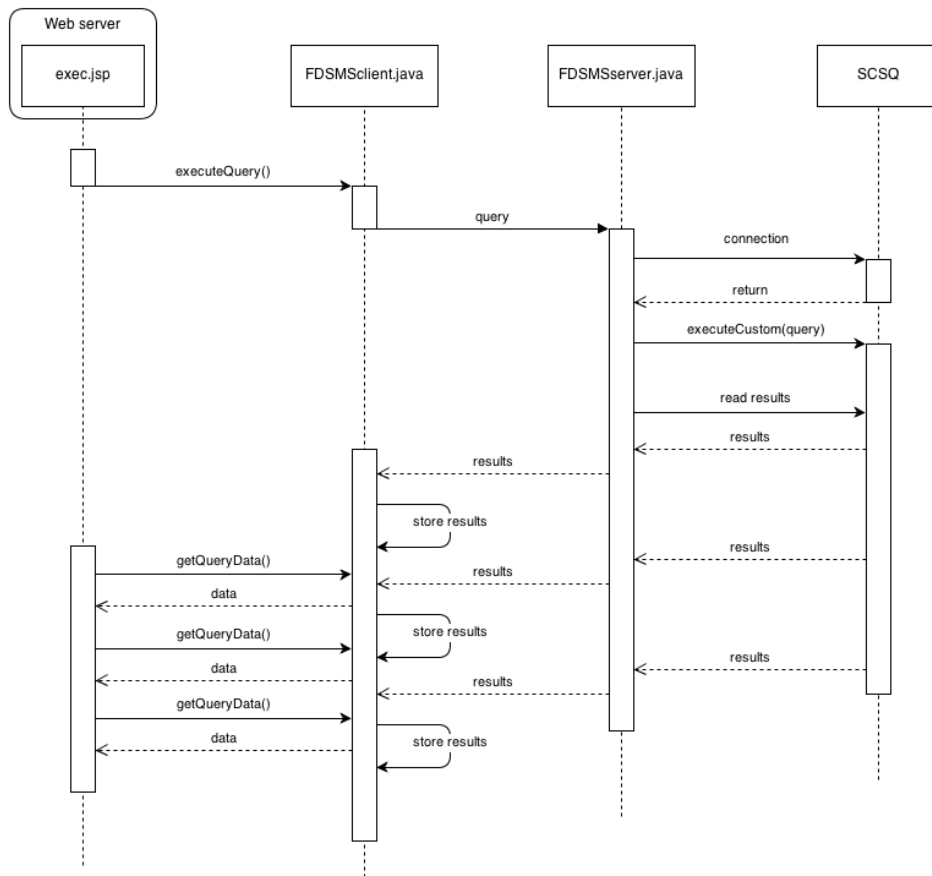
SCSQ is the backend where queries written in SCSQL are executed; even continuous queries can be executed. A Java wrapper over SCSQ is available, and it exposes an API which allows to communicate with SCSQ from external applications. The Java wrapper sends queries and commands as text strings to SCSQ, and gets results back.

The FDSMS server is the real communication interface between the web server and SCSQ. It creates a TCP server on its initialization, used to exchange messages with the web server. Each time a new request arrives from the web server, a new execution thread is created, which opens a new connection with SCSQ through the Java wrapper, executes the requested query, and then returns results back, properly formatted in JSON format.

The web server sends requests formatted as simple text strings, and they could be of two kinds: a request for query execution, or a request to retrieve all of the types and functions declared on SCSQ.

9.4.1 Query execution

A query execution request is composed of different phases, depicted in the following figure. When the executed query is a Continuous Query, the execution result is a stream, retrieved from the web server in different data chunks. While new data is available, the web server continues to reclaim it, until a notification of query execution termination is provided.



The FDSMS client, from the web server, opens a TCP connection with the FDSMS server, sends the query to execute, then wait for results. When results arrive, they are stored in the *dataPool*, so the FDSMS client reads them and sends them back to the web server.

```

// create new client socket on local host
Socket socket = new Socket(InetAddress.getLocalHost(),
    55555);
// output stream
PrintWriter os = new PrintWriter(socket.getOutputStream(),
    true);
// input stream
BufferedReader is = new BufferedReader(new
    InputStreamReader(socket.getInputStream()));
// write command on output stream
os.println(command.getName());
if (command.getName().equals("query")) {
  
```

```

        os.println(command.getQuery());
        os.println(command.getBufferSize());
    }
    // wait for server response and read it
    String result;
    do {
        result = is.readLine();
        if (result == null) {
            // no more results to read
            SCSQclient.dataPool.get(command.getQueryId()).
                noMoreData();
        }
        else {
            // read results from data pool
            List<String> data = SCSQclient.dataPool.get(command
                .getQueryId()).getData();
            data.add(result);
        }
    } while (result != null);
    // close socket
    os.close();
    is.close();
    socket.close();

```

The FDSMS server, from its side, has created the TCP server during startup, and waits for requests from the web server.

```

// start TCP server
ServerSocket server;
try {
    server = new ServerSocket(55555);
} catch (IOException ex) {
    System.err.println("Server creation error");
    ex.printStackTrace();
    return;
}
// initialize SCSQ client connection
try {
    Connection.initializeClient();
} catch (AmosException e) {
    System.out.println("Cannot initialize connection with
        SCSQ");
    e.printStackTrace();
    return;
}

```

```

}
// persistent connection
while (true) {
    // accept new client connection
    Socket socket;
    try {
        socket = server.accept();
        new ServerConnection(socket).start();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

When a new request arrives, a new execution thread is created, and the connection accepted. The “query” message sent from the client means that it wants to execute a query, subsequently written on the communication pipe, and followed by the buffer size used to retrieve the results. A new connection with SCSQ is created, the query executed, and results read, one tuple a time. Every tuple is converted in a JSON format, and then sent back to the client.

```

// connection output stream
PrintWriter os = new PrintWriter(socket.getOutputStream(),
    true);
// connection input stream
BufferedReader is = new BufferedReader(new
    InputStreamReader(socket.getInputStream()));
// read request type from client
String requestType = is.readLine();

...

// execute query
if (requestType.equals("query")) {
    // read query to execute from client
    String query = is.readLine();
    // read buffer size
    String bufferSize = is.readLine();
    // new SCSQ connection
    Connection conn = null;
    try {
        conn = new Connection("a");
    } catch (AmosException e) {
        System.out.println("Error on connecting to SCSQ");
        e.printStackTrace();
    }
}
}

```

```

        return;
    }
    // execute query with given buffer size
    Scan scan = conn.executeCustom(query, "(:bufferSize " +
        bufferSize + ")");
    // for each result
    while (!scan.eos()) {
        // get tuple
        Tuple tuple = scan.getRow();
        // JSON representation of results
        JSONObject resultJsonObject = new JSONObject();
        JSONArray valuesJsonArray = new JSONArray();
        for (int i = 0; i < tuple.getAriety(); i++) {
            valuesJsonArray.put(tuple.getElem(i));
        }
        resultJsonObject.put("results", valuesJsonArray);
        // convert JSON representation to string and write
        // it on
        // output stream
        os.println(resultJsonObject.toString());
        // read next scan
        scan.nextRow();
    }
}

```

9.4.2 Types and functions retrieving

Since types and functions declared in SCSQ are used in the VQE to compose Visual Queries, they must be retrieved from it, represented in an exchangeable format, and then sent to the web application to be shown up. The task of retrieving this data from SCSQ is complex and time expensive, therefore an intermediate solution has been adopted. In fact, a batch script which retrieves all the types and functions, represents them with their information in a JSON format, and stores this data in a proper file, has been created. This script can be run manually at need, and a proper DataBase dump, called *db_elements*, is generated. This file is then accessed whenever the client requests the SCSQ elements: types and functions are read, and then sent back to the client.

```

// connection output stream
PrintWriter os = new PrintWriter(socket.getOutputStream(),
    true);

```

```
// connection input stream
BufferedReader is = new BufferedReader(new
    InputStreamReader(socket.getInputStream()));
// read request type from client
String requestType = is.readLine();

...

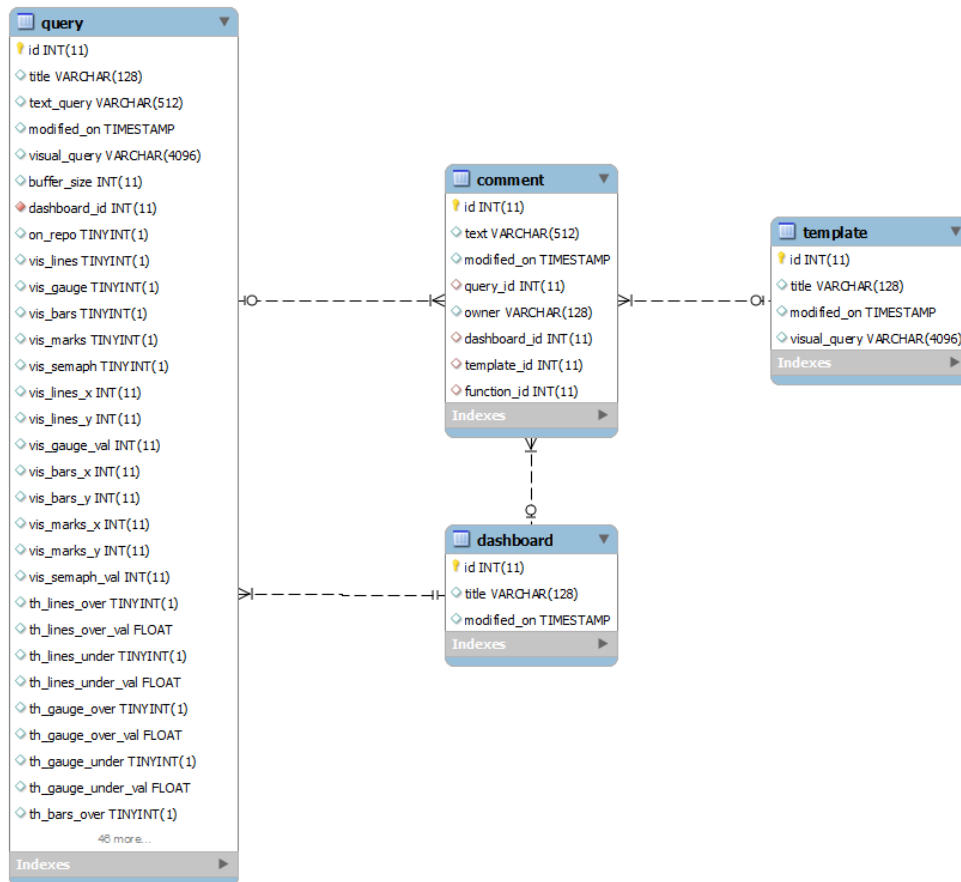
// get all types and functions from DB
if (requestType.equals("retrieveDBData")) {
    // open db_elements file
    BufferedReader fis = new BufferedReader(new FileReader
        ("db_elements"));
    // read each line from db_elements file ...
    String line;
    while ((line = fis.readLine()) != null) {
        // ... and write it on output stream
        os.println(line);
    }
    // close db_elements file
    fis.close();
}
```

This way, the operation cost, in terms of time, is considerably reduced.

9.5 User DataBase

As already introduced, the User DataBase stores user related data, which is used to build web pages, creating a different environment for each user.

The following figure shows the DataBase schema, depicting how the DataBase is structured. Basically, four tables are used to store corresponding entities: dashboards, queries, templates and comments. Each entity has a unique surrogate key, and a set of attributes describing it.



The DASHBOARD entity stores dashboards attributes, like TITLE (dashboard name) and MODIFIED_ON (last modified timestamp). It has from 0 to N associated queries (queries added to the dashboard), and from 0 to N associated comments.

The TEMPLATE entity stores templates attributes, like TITLE (template name), MODIFIED_ON (last modified timestamp) and VISUAL_QUERY, a textual representation of the Visual Query, therefore it can be restored on the Canvas of the VQE. It also has from 0 to N associated comments.

The QUERY entity represents queries with their attributes: TITLE is the displayed name of the query, MODIFIED_ON is a timestamp of the last modified moment, VISUAL_QUERY and TEXT_QUERY are, respectively, a textual representation of the Visual Query, and the textual SCSQL query translation. Other attributes, like the ones starting with VIS_ or TH_ represents visualization properties, so for example which kind of graph has been chosen to display query results, or if a chart shows up a threshold on it. The ON_REPO attribute stores a Boolean value, TRUE when the query

has been saved on the queries repository. A query has from 0 to N associated comments.

The COMMENT entity represents comments associated with other entities, and their attributes, like the OWNER of the comment (the name of the user who wrote the comment), the TEXT (what the user has written), and the ID of the associated entity. Since a comment can be associated with only one entity, this is a mutually exclusive value, therefore, if for example a comment is associated with a dashboard, the DASHBOARD_ID attribute will have a positive value (the ID of the associated dashboard), while the other IDs will have -1 as value.

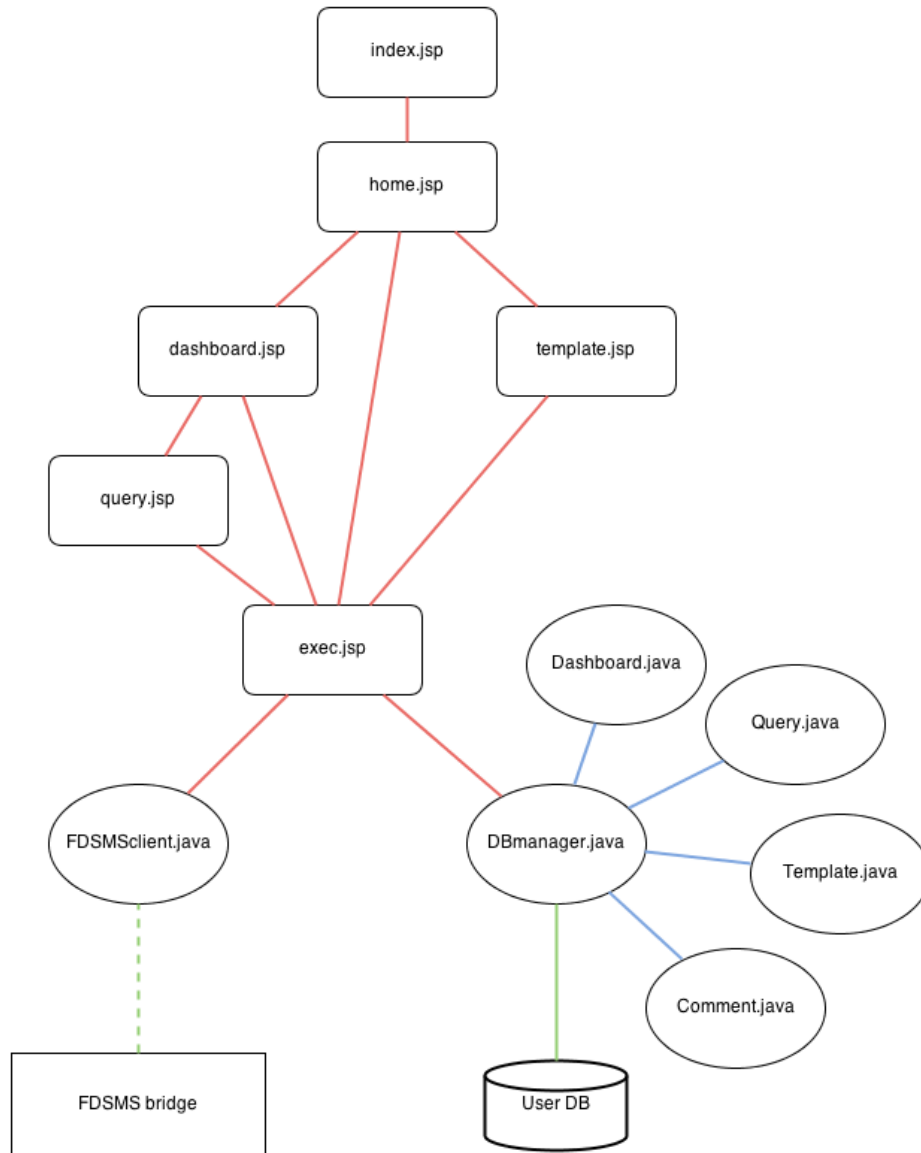


Figure 9.1: Web application architecture.

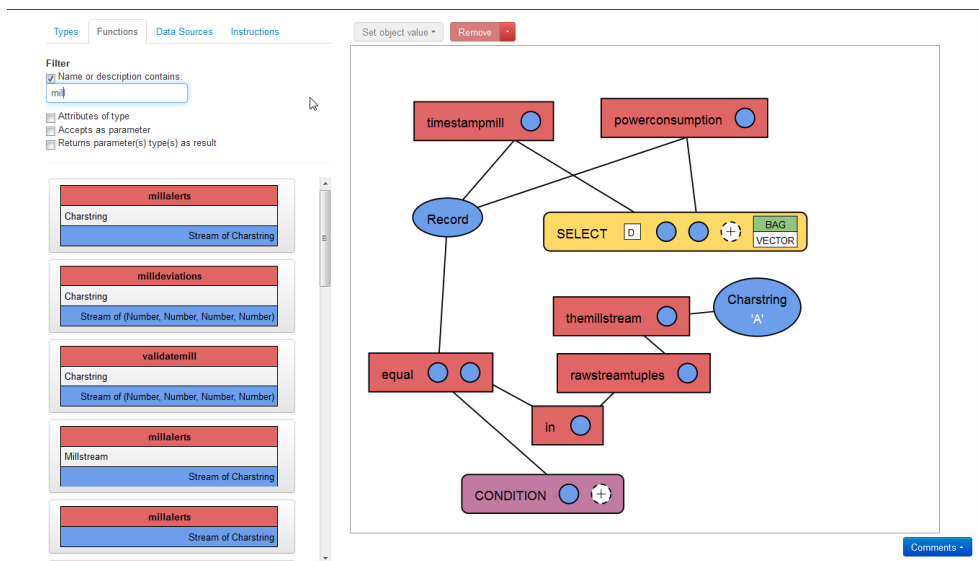


Figure 9.2: The Visual Query Editor.

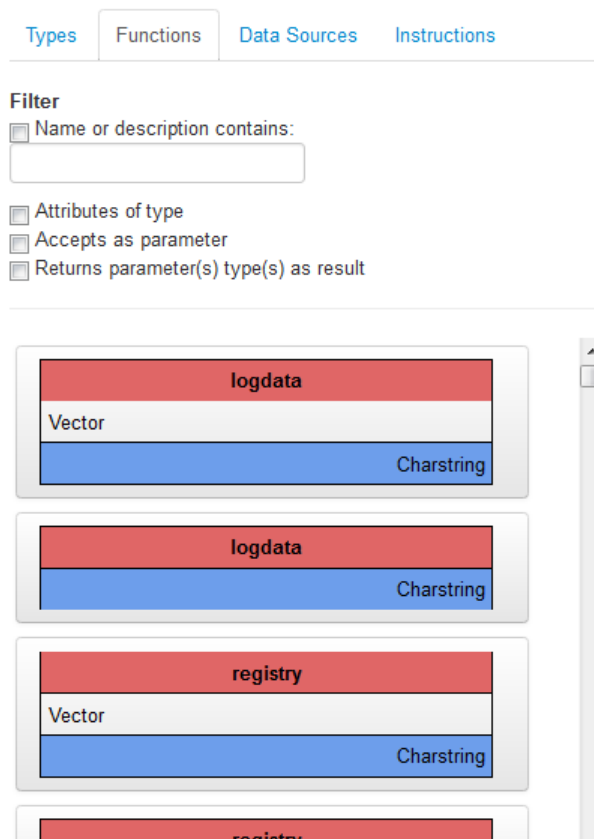


Figure 9.3: Functions representation inside the Elements Toolbox.

Types	Functions	Data Sources	Instructions
		Mill A	
		Mill B	
		Mill C	
		Mill D	
		E01PS10-2991-20120511114235.CSV	
		E01PS10-2991-20120511114922.CSV	
		REG407	
		REG403	
		REG409	

Figure 9.4: Data Sources group.

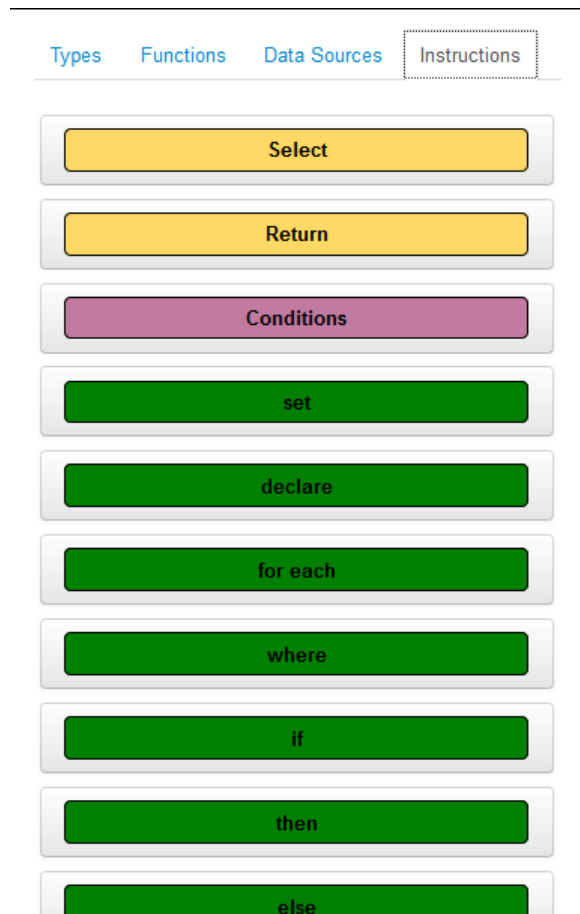


Figure 9.5: Instructions group.

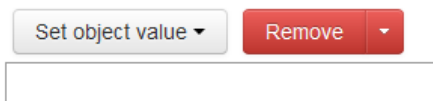


Figure 9.6: The Toolbar.

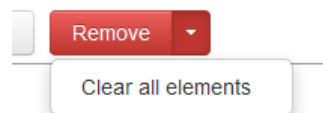


Figure 9.7: "Clear all elements" Toolbar action.

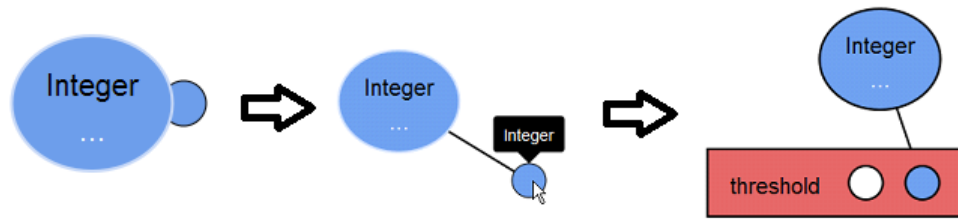


Figure 9.8: The connection steps.

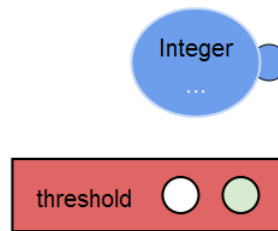


Figure 9.9: Connectable elements are highlighted in green.

Choose visualization...

Query results: result 1 (Number), result 2 (Number)

Lines

Chart name: Mill power X axis name: Time stamp Y axis name: Power

Series f:

Name: Series name

X: result 1 (Number)

Y: result 2 (Number)

Add series

Threshold over 0.00

Threshold under 0.00

Stream

Bars

Chart name: Chart title X axis name: X axis Y axis name: Y axis

Series f:

Name: Series name

X: result 1 (Number)

Y: result 1 (Number)

Add series

Threshold over 0.00

Threshold under 0.00

Stream

Points

Chart name: Chart title X axis name: X axis Y axis name: Y axis

Series f:

Figure 9.10: Visualization customization.

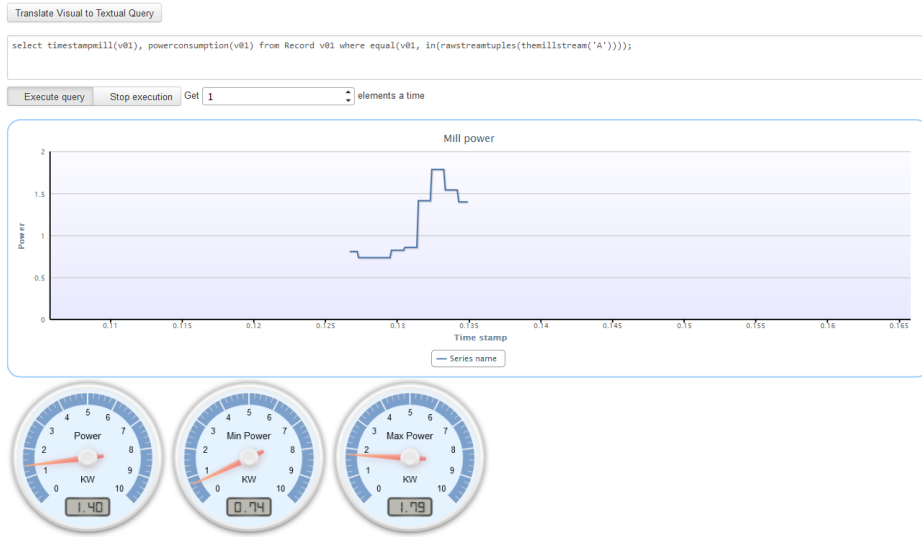


Figure 9.11: Query execution.

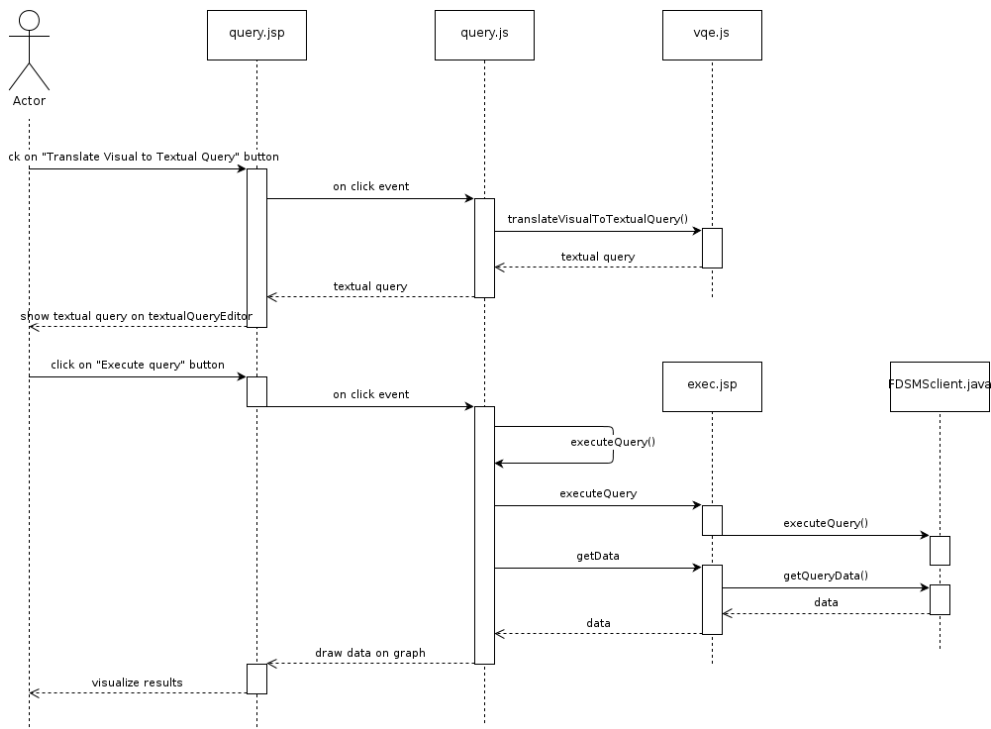


Figure 9.12: Query translation and execution sequence diagram.

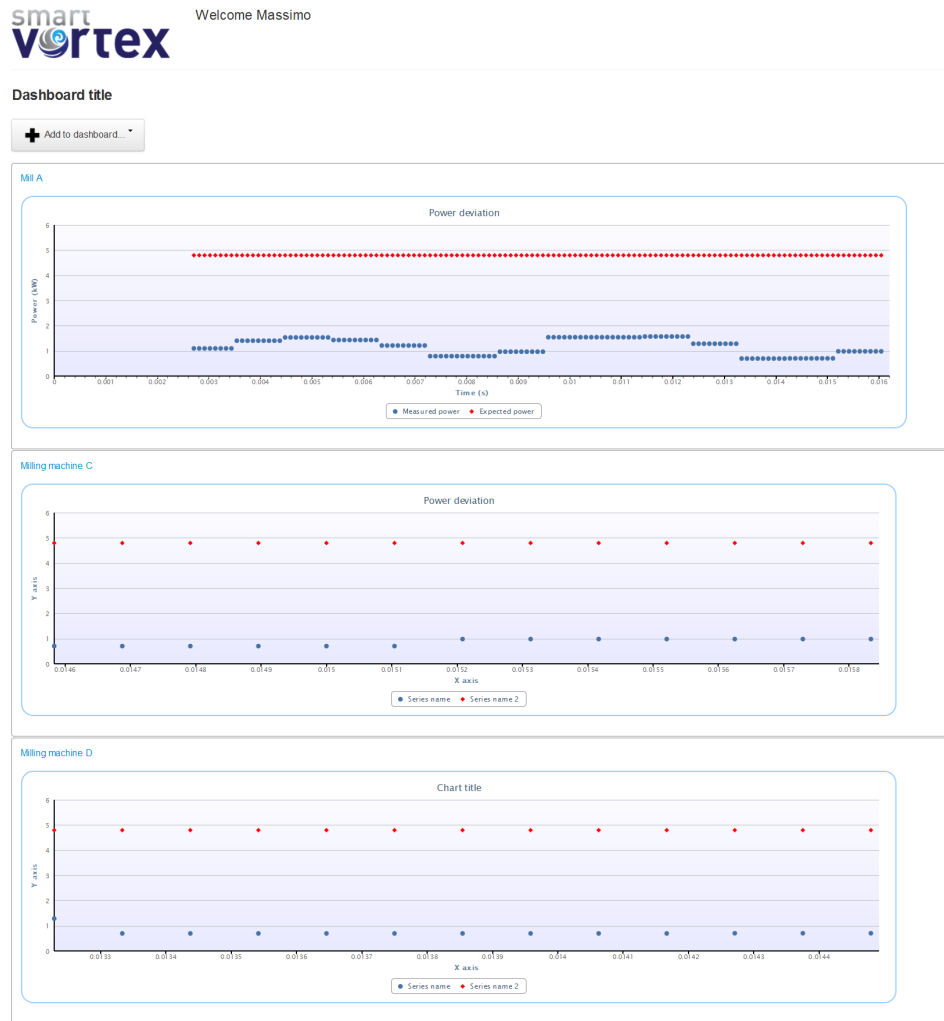


Figure 9.13: A dashboard example.

Part VI

Conclusions and Future Works

Chapter 10

Implemented requirements

section §4.2 presented an analysis of the different ISPs and provided use cases, thus identifying all the functionalities that the User Layer of the Smart Vortex system needs to provide, detailed in a proper list. In such a list of functionalities, different actors of the system (administrators vs. end-users/engineers) have been considered, as well as different categories of available functionalities, divided into querying data and visualizing data. Moreover, a kind of “overall” use-case (called meta use case) has been defined, as a demonstration of what has been envisioned as applicable to the User Layer.

During the 2nd year of the project, the design and realization of the components of the User Layer has been carried out, and has been presented in this thesis. In particular, the effort has been addressed in the definition of a Visual Query Language over SCSQL, the adopted Query Language on Data Streams, and in the development of a Visual Query Editor, to build Visual Queries and to assist the user with graphical aids during this task. Another task has been the implementation of a translation algorithm between Visual Queries and textual queries written in the SCSQL language, and the design of proper visualizations to display queries execution results.

The tool developed constitutes the first version of the prototype; so far, it is available in “single-use mode” i.e., with no collaboration/sharing features. The targeted environment is a web application.

Looking at the Smart Vortex Framework architecture (figure 4.2), and in regards to the Visual and Multimodal Query and Data Presentation macro-component, the forementioned implemented components corresponds to:

- the GUI Container, composed of various visual query widgets used to implement the VQE and the Query Repository where the user queries are stored;

- the Query Manager, containing all the necessary logic to translate a Visual Query into an SCSQL textual query.

The integration level with the other Smart Vortex components is still low, in particular with the multimodal framework and with the rule and policy framework.

The following tables summarize the list of requirements, as presented in section §4.2, with their current status, highlighting the implemented ones. All of the Query Functionalities have been implemented; in particular, requirement PCA-3_QF015 is fulfilled thanks to the filter functionalities of the VQE (9.3.1), PCA-3_QF014 is fulfilled by tooltips over Visual Query elements, depicting associated Types, and graphic metaphors during the Parameter's Type check (9.3.1); check errors notification fulfill also requirement PCA-3_QF016. Query execution results are displayed with a proper chart (PCA-3_VF01, PCA-3_VF02 and PCA-3_VF03) or visualization, changed at need, as for requirement PCA-3_VF08. A Dashboard (section 9.3.4) as a set of queries is implemented, as for PCA-3_VF06_v2.

PCA-3 Visualization Functionalities (PCA-3_VF)

Requirement ID	Requirement Title	Current Status
PCA-3_VF01	Line Chart Visualization	✓
PCA-3_VF02	Histogram Visualization	✓
PCA-3_VF03	Bar Chart Visualization	✓
PCA-3_VF04_v2	Browse Data Source	✗
PCA-3_VF05	Zoom Visualization	✗
PCA-3_VF06_v2	Configure Dashboard	✓
PCA-3_VF07	Possible visualization styles	✗
PCA-3_VF08	Choice of visualization	✓
PCA-3_VF09	Annotate data views	✓
PCA-3_VF10	Visualize 3D model	✗

PCA-3 Query Functionalities (PCA-3_QF)

Requirement ID	Requirement Title	Current Status
PCA-3_QF01/02	SCSQL Visual Query Language	✓
PCA-3_QF03	Query Repository	✓
PCA-3_QF04_v2	Visual Query Editor	✓
PCA-3_QF05	Visual Spatial Query Operators	✓
PCA-3_QF06	Visual Time Query Operators	✓
PCA-3_QF07	Visual Logical Query Operators	✓
PCA-3_QF08	Visual Arithmetic Query Operators	✓
PCA-3_QF09	Visual Comparison Query Operators	✓
PCA-3_QF010	Create visual query template	✓
PCA-3_QF011	Edit/Modify visual queries/query templates	✓
PCA-3_QF012	Refine query and view update	✓
PCA-3_QF013	Facility of searching	✓
PCA-3_QF014	Graphics facility for query composition	✓
PCA-3_QF015	Functions filter	✓
PCA-3_QF016	Assist the end user	✓
PCA-3_QF017	Different modalities of query building	✓

PCA-3 Administrator Functionalities (PCA-3_AF)

Requirement ID	Requirement Title	Current Status
PCA-3_AF01	Create user	✗
PCA-3_AF02	Manage user	✗
PCA-3_AF03	Write new visual query	✓

Chapter 11

Considerations and limitations

Even if most of the required functionalities of this project phase have been implemented, the designed VQL and the implemented prototype present some limitations.

11.1 Visual Query Language

Currently, only SELECT-FROM-WHERE queries can be created through the VQL, but this does not represent a serious restriction, as it provides the ability to cover the most part of the needed queries. The VQL is not intended as a data manipulation language, therefore Visual Queries regarding the interaction with the DataBase schema, like type/object creation/removal/updating, are not foreseen. However, other essential AmosQL constructs have not been taken into consideration, comprehending:

- declaration of local variables, through commands SET and DECLARE;
- declaration of new functions, through the command CREATE FUNCTION <NAME>(<ARGS>) -> <QUERY>;
- procedural commands, like FOR...EACH, IF-THEN-ELSE, WHILE(<ARGS>)...DO..., RETURN, and so on;
- Cursors, through commands OPEN, FETCH, CLOSE,

Additionally, some AmosQL types, like Vector, are not completely supported. A Vector like {1,2,3} cannot be directly created using the Visual Query Language constructs, but it can be reproduced using the statement *vectorize(iota(1,3))*. Indexes on Vectors are not available as well. In this case,

a possible workaround is to manually create a user function like *vectorelement(Vector v, Integer index)->Object* (that given a Vector *v* and an Integer *index* returns the object at the *index* position of the array), and use it through the Visual Query Editor.

11.2 Query translation algorithm

The presented query translation algorithm has some limitations. By now, it is intended for the translation of Visual Queries composed with the current VQL version, therefore the other forementioned constructs are not supported. However, the translation algorithm can be easily extended, in order to support constructs like functions declaration, procedural commands, and so on. Performances of this algorithm are good for most of the queries, even because composed queries are in general quite simple. The AmosQL user manual suggests to decompose complex queries into simpler ones, by declaring for example subfunctions (which are even reusable into other parts of the code). Furthermore, current drawing capabilities of the VQE are limited, and Visual Queries bigger than a certain size cannot be created.

Performances could be increased by using other intermediate data structures, and thus generating a graph with double oriented arcs. This way, the graph can be walked in both directions, speeding up some parts of the algorithm, especially when searching for connected Conditions. On the other hand, this approach would also increase the need of resources, and most important, the complexity of the algorithm, and the code in general. The algorithm proposed here is a balance between performances and complexity.

Nevertheless, some methods of graph optimization and graph traversal could be adopted, in order to find faster translation algorithms.

11.3 Visual Query Editor

As already mentioned, The VQE Canvas has a fixed size, therefore the area dedicated to Visual Queries composition is limited, and queries bigger than a certain size cannot be drawn. Two solutions to this limitation would be a pan and a zoom capability: the user would have, this way, a possibly infinite Canvas to build the queries, and to have an overall view of them.

A better representation for Types and Data Sources, in the Elements Toolbox, should be adopted. For example, Types could be organized and depicted with their hierarchy, in order to have a faster recognition, and Data Sources

should have at least a filter functionality to search for Data Sources by location; even better, Data Sources could be represented on a map, to better exploit how to locate them.

The visualization choice, to display queries results, is now limited to a single chart for each visualization's type. Having more charts of the same type on a dashboard is useful in monitoring or analyzing scenarios, where different data streams from different data sources are compared each other using the same visualization. Furthermore, queries on dashboards cannot be rearranged by now, and complex arrangements are not supported. Even this feature would be useful in a monitoring scenario, to display, for instance, data streaming from machines sensors, capturing anomalies or malfunctionings.

Chapter 12

Future Works

Since current prototype has been designed and implemented during year 2 of the Smart Vortex project cycle, year 3 is dedicated to further improve it. A first effort will be on the implementation of requirements regarding the functionalities that are still missing. In particular:

- Requirement PCA-3_VF04_v2. During year 3, the ability to dynamically access data sources, data streams and related metadata (e.g., via a repository associated with the FDSMS, that provides specific APIs for accessing data sources, schemas, metadata, etc.) will be added. If this information cannot be dynamically obtained, all data sources will be hardcoded.
- Requirement PCA-3_VF05. For subsequent prototype versions, functionalities that allow the user to establish “interactive sessions” with the visualization obtained from a given query (e.g. overlapping of different visualization, tooltip on charts showing their current values etc.) are envisioned. There will be also an investigation about the possibility to refine queries by acting directly on the data visualizations (e.g., exploiting the so called “semantic zooming” capabilities).
- PCA-3_VF10. An integration with the 3D visualization plugin, coming from the ISP-1.1 scenario, is also planned.
- Expansion of the VQL to include AmosQL constructs lefted out, and implementation of the newly introduced constructs in the translation algorithm.

In addition to completely fulfil the list of individuated requirements, the designing and implementation of the collaboration features, transitioning from

the current “single-use mode” to a “multiple-use mode”, is planned. In parallel, since in year 2 the main focus has been on developing a Web-based prototype, a porting to mobile devices, through native applications, is planned. The decision about which platforms to support (iOS, Android, both) will be taken during Y3, also on the basis of users and project needs.

Year 3 will also be the project cycle where the extensive system validation with real users is started. Interviews and real use tests will be prepared, and different kinds of users will be chosen, in order to find usability weaknesses. An initial set of Key Performance Indicators (KPIs) [46] measures will be calculated. This work will last for the entire duration of the project, with the intent of validating each prototype release.

The detailed list of KPIs and their current status follows:

- Effectiveness of usage and visualization – not yet measured.
- Effectiveness related to resource consumption – not yet measured.
- User satisfaction – not yet measured.
- Conformity to existent standard (ISO9241-11) – ongoing, meaning that the design of the software modules has been carried out according to the best practices in Human-Computer Interaction, in order to be able, in the next releases, to fully adhere to the standard.

Even if no tests have been already done, an initial methodology to use has been defined. Users are actively involved throughout the whole software development process. For the evaluation activities qualitative usability evaluation methods like feature inspection, observation of users while they perform different tasks, cooperative evaluation and questionnaires (which will give details about the user satisfaction) are to be used.

Concerning the number of people to interview during these tests, Robert Virzi claimed that about 80% of the known usability problems could be discovered with 5 testers. 3 testers would find the most severe problems [44]. Also Nielsen and Landauer confirmed that 5 users discover approx. 80% of usability problems. With 3 testers they showed that 70% of usability problems could be found [45]. In accordance to this result, several investigations will be carried out, in order to discover usability problems and give improvement recommendations to the developers.

This set of tests could be accomplished:

- Controlled Experiments – to be held in lab environment under controlled conditions. Each partner could presents its own component,

explaining the functionalities and several specific details to other partners. After that, users have to look carefully through the components, test them, and provide comments about how to improve the prototype.

- Cooperative evaluation – done with expert users, completely aware about the Smart Vortex project and the functionalities of the components to test. These tests are designed to estimate the performance of specific tasks (see KPIs) in order to gather useful data on usability issues of the system.
- With external users - After the performance of the user tests with expert users, another usability test with non-expert users will be accomplished. The purpose is to get a feedback on the user interface also from users who are not familiar with the data stream topic.

Another area in need of further work is the communication technology between the web application and the FDSMS bridge, and between the web server and the presentation layer. By now, as presented, a JSP-over-Tomcat technology has been used to retrieve queries results, exchanging data in a textual format, with a pull approach (from client to server). This is not the best approach, since the visualization of data streams demands for a push technique, where data is sent from the server to the client, and not vice-versa. This way, as soon as new results are available from a stream execution, they are directly sent to the client, and visualized in (almost) real time. A possible technology adopting this approach is CometD, which uses persistent HTTP connections to communicate with a web client, in both directions.

Even the web application would take advantage of the introduction of this technology. Since one of the objectives for year 3 is to implement a collaborative version of the prototype, typical complications of collaborative environments will have to be taken into consideration. One of these issues is, for example, the synchronization of shared resources. A typical scenario could be a dashboard shared between different users, so they can monitor the same data sources from different access points; all of the visualized queries must receive the same data on all of the shared dashboards, updated in real time, and if a user operates a modification on the dashboard (like adding or removing a query visualization, or commenting on it), this must be propagated on the other dashboards, so they reflect this change. These complications could be overtaken by adopting a push technique, where changes operated by a user on a shared resource are sent to the other clients.

Part VII

References

Bibliography

- [1] Smart Vortex Document of Work. Scalable Semantic Product Data Stream Management for Collaboration and Decision Making in Engineering (2011)
- [2] D.J.Abadi, D.Carney, U.Cetintemel, M.Cherniack, C.Convey, S.Lee, M.Stonebraker, N.Tatbul, and S.Zdonik: Aurora: a new model and architecture for data stream management, *The VLDB Journal*, 12(2):120-139, 2003.
- [3] A.Arasu, S.Babu, and J.Widom: The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2):121-142, 2006.
- [4] <http://www.streambase.com/>
- [5] E.Zeitler and T.Risch: Massive scale-out of expensive continuous queries, presented at 37th International Conference on Very Large Databases, VLDB 2011, in Proceedings of the VLDB Endowment, Vol. 4, No. 11, 2011.
- [6] E.Zeitler: Scalable Parallelization of Expensive Continuous Queries over Massive Data Streams, Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 836 ISSN 1651-6214, ISBN 978-91-554-8095-0, Acta Universitatis Upsaliensis, 2011 (<http://www.it.uu.se/research/group/udbl/Theses/ErikZeitlerPhD.pdf>).
- [7] T. Risch, and V. Josifovski: “Distributed Data Integration by Object-Oriented Mediator Servers”, in *Concurrency and Computation: Practice and Experience* J. 13(11), John Wiley & Sons, September 2001, pp 933–953.
- [8] Zeitler, E. 2011. Scalable Parallelization of Expensive Continuous Queries over Massive Data Streams. Acta Universitatis Upsaliensis. Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty

of Science and Technology 836. 35 pp. Uppsala. ISBN 978-91-554-8095-0.

- [9] http://www.it.uu.se/research/group/udbl/amos/doc/amos_users_guide.html
- [10] E.Zeitler: SCSQ user's guide:, UDBL, Dept. Of Information Technology, Uppsala University, 11 Aug. 2011, <http://www.it.uu.se/research/group/udbl/publ/scsql.pdf>.
- [11] M. Yen, R. Scammel. A Human Factors Experimental Comparison of SQL and QBE. IEEE Transactions on Software Engineering Volume 19 Issue 4. (1993).
- [12] Tiziana Catarci, Maria F. Costabile, Stefano Levialdi, Carlo Batini. Visual Query Systems for Databases: A Survey. Journal of Visual Languages and Computing. (1997).
- [13] M. Angelaccio, T. Catarci, G. Santucci. QBD*: A Graphical Query Language with Recursion. IEEE Transactions on Software Engineering, 16, 1150-1163. (1990).
- [14] Steven P. Reiss. A Visual Query Language for Software Visualization. Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments. (2002).
- [15] Stavros Polyviou, Paraskevas Evripidou, George Samaras. Query by Browsing: A Visual Query Language Based on the Relational Model and the Desktop User Interface Paradigm. The 3rd Hellenic Symposium on Data Management (2004).
- [16] Antonio Massari, Stefano Pavani, Lorenzo Saladini, Panos K. Chrysanthis. QBI: Query By Icons. In Proceedings of SIGMOD Conference (1995).
- [17] A.J. Morris, A.I. Abdelmoty, B.A. El-Geresy. Design and Implementation of a Visual Query Language for Spatial Databases. Proceedings 6th Int. Working Conference on Advanced Visual Interfaces. (2002).
- [18] Norman Murray, Norman Paton, Carole Goble. Kaleidoquery: a visual query language for object DataBases. Working Conference on Advanced Visual Interfaces - AVI , pp. 247-257. (1998).
- [19] A.J. Morris, A.I. Abdelmoty, B.A. El-Geresy. Design and Implementation of a Visual Query Language for Spatial Databases. Proceedings 6th Int. Working Conference on Advanced Visual Interfaces. (2002).

- [20] C.J. Kacmar, J.M. Carey. Assessing the Usability of Icons in User Interfaces. *Behaviour and Information Technology*, 10. (1991).
- [21] Balkir, N.H. Ozsoyoglu, G. Ozsoyoglu, Z.M. A graphical query language: VISUAL and its query processing. . *Knowledge and Data Engineering*, IEEE Transactions. (2002).
- [22] Albert N. Badre , Tiziana Catarci, Antonio Massari , Giuseppe Santucci. Comparative Ease of Use of a Diagrammatic Vs. an Iconic Query Language In Interfaces to Databases. *Electronic Series Workshop in Computing*, Springer, pages 1-14. (1996).
- [23] Card, S. K., Mackinlay, J., Shneiderman, B. 1999. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann.
- [24] Becker, R. A., Cleveland, W. S., Shyu, M.-J. 1996. The visual design and control of trellis display. *Journal of Computational and Graphical Statistics* 5(2): 123-155.
- [25] Becker, R. A., Cleveland, W. S. 1987. Brushing scatterplots. *Technometrics* 29(2): 127-142.
- [26] Grossman, T., Balakrishnan, R. 2005. The bubble cursor: enhancing target acquisition by dynamic resizing of the cursor's activation area. *Proceedings of the ACM Conference on Human Factors in Computing Systems*: 281-290; <http://doi.acm.org/10.1145/1054972.1055012>.
- [27] Shneiderman, B. 1996. The eyes have it: a task by data type taxonomy for information visualizations. *Proceedings of the IEEE Symposium on Visual Languages*; <http://portal.acm.org/citation.cfm?id=832277.834354>.
- [28] van Ham, F., Perer, A. 2009. Search, show context, expand on demand: supporting large graph exploration with degree-of-interest. *IEEE Transactions on Visualization and Computer Graphics* 15(6): 953-960; <http://dx.doi.org/10.1109/TVCG.2009.108>.
- [29] Bederson, B. B., Hollan, J. D. 1994. Pad++: a zooming graphical interface for exploring alternate interface physics. *Proceedings of the ACM Symposium on User Interface Software and Technology*: 17-26; <http://doi.acm.org/10.1145/192426.192435>.

- [30] Zellweger, P. T., Mackinlay, J. D., Good, L., Stefik, M., Baudisch, P. 2003. City lights: contextual views in minimal space. Extended Abstracts of the ACM Conference on Human Factors in Computing Systems: 838-839; <http://doi.acm.org/10.1145/765891.766022>.
- [31] Tufte, E. 1983. *The Visual Display of Quantitative Information*. Cheshire, CT: Graphics Press.
- [32] Riche, N. H., Lee, B., Plaisant, C. 2010. Understanding interactive legends: a comparative evaluation with standard widgets. *Computer Graphics Forum* 29(3): 1193-1202.
- [33] <http://www.lesscss.org/>
- [34] <http://twitter.github.com/bootstrap/>
- [35] <https://dojotoolkit.org>
- [36] <http://jquery.com>
- [37] <http://jqueryui.com/>
- [38] <http://trentrichardson.com/examples/timepicker/>
- [39] <http://www.highcharts.com/>
- [40] <https://github.com/Mikhus/canv-gauge>
- [41] Report McKinsey & Company: Big Data: The next frontier for innovation, competition, and productivity. (May 2011).
- [42] H.U. Heidbrink et al.: D1.1 – Survey of users in Europe and Smart Vortex usage scenarios and cooperation support. Smart Vortex Consortium, 2011.
- [43] H.U. Heidbrink et al.: D1.2 – Requirement Analysis Report and Specification of the Smart Vortex Assessment Framework. Smart Vortex Consortium, 2011.
- [44] R.A. Virzi. Refining the test phase of usability evaluation: How many subjects is enough? *Human Factors*, 34:457–468, 1992.
- [45] J. Nielsen. *Usability Engineering*. Academic Press, 1993.
- [46] http://en.wikipedia.org/wiki/Performance_indicator