

Interfaces for Visualizing Multi-Valued Attributes: Design and Implementation Using Starfield Displays

Laurent Cailleteau

*Human-Computer Interaction Laboratory
University of Maryland
Institute for Advanced Computer Studies
College Park, MD 20742
USA
caillete@cs.umd.edu*

and

*Ecole des Mines de Nantes
4 rue Alfred Kastler
B.P. 20722
44307 Nantes Cedex 03
FRANCE*

Partial requirement for the Diplôme d'Etudes Approfondies

Final version, August 26, 1999

Contact:

*Catherine Plaisant
Human-Computer Interaction Laboratory
A.V. Williams Building
University of Maryland
College Park, MD 20742, U.S.A.
Plaisant@cs.umd.edu
<http://www.cs.umd.edu/hcil>*

CONTENTS

ABSTRACT	4
ACKNOWLEDGMENTS	5
1. INTRODUCTION	6
2. REVIEW OF PREVIOUS WORK	8
2.1. DYNAMIC QUERY INTERFACES.....	9
2.2. QUERY PREVIEW AND QUERY REFINEMENT, THE CASE OF NASA EARTH SCIENCE	16
2.3. DYNAMIC QUERY DATA STRUCTURES, A NEED FOR SPEED	31
2.4. DATA VISUALIZATION ISSUES	38
3. STARFIELD DYNAMIC OBJECT MINER (STARDOM), SOFTWARE AND USER INTERFACE DESCRIPTION	45
3.1. INTRODUCTION	46
3.2. DESCRIPTION OF THE INTERFACE	47
3.3. INTERNAL REPRESENTATION, OR BACK-END STRUCTURE.....	56
3.4. RANGE AND MULTI-VALUED ATTRIBUTE DATA	66
3.5. GRAPHICAL STRUCTURE	72
4. EVALUATION OF THE CURRENT VERSION OF STARDOM, FURTHER IMPROVEMENTS	76
4.1. TIME EXPERIMENTS AND SEARCH OF A SMOOTH, CONTINUOUS DYNAMIC FEEDBACK	77
4.2. DISCUSSION AND FUTURE WORK.....	88
5. CONCLUSION	94
REFERENCES	95

Abstract

Dynamic queries and starfield displays have been developed at the University of Maryland to facilitate access to database information. Dynamic queries apply direct manipulation to querying. The world of objects and the world of actions are visible on the screen and controls provide immediate, continuous and reversible feedback to users as they formulate their query.

My six-month project had two components: (1) the development of a Java implementation of a dynamic query and starfield display interface, and (2) the investigation of the visualization of multi-valued and range-valued attribute data. This was the most novel aspect of this project. This work was conducted at the Human Computer Interaction Laboratory at the University of Maryland. It was part of a larger project sponsored by NASA whose goal was to design and develop user interfaces helping scientists locate data of interest in the large NASA archives.

After the review of previous work, the user interface and data architecture of StarDOM (Starfield Dynamic Object Miner) is presented. Time experiments are described and possible improvements are discussed.

Acknowledgments

I would like to thank my advisors, Catherine Plaisant, David Doermann, Ben Shneiderman and Jean-Daniel Fekete for their encouragement and suggestions throughout the research, as well as all Human Computer Interaction Laboratory (HCIL) and Language And Media Processing (LAMP) members, for their friendship and encouragement.

1. Introduction

In the past few years, dynamic queries [1, 4] and starfield displays [2] have been developed as an easier access to database information. They offer an alternate to command line or form fill-in interfaces.

At the University of Maryland's Human Computer Interaction Laboratory, a chemical table of elements was the first realization of dynamic queries. Then, a Home finder was built [1] (Figure 1-1). This system shows a map of Washington, DC illuminated by more than a thousand points of light indicating homes for sale. It is then possible for a user to move sliders in order to match certain criteria, like the minimum and maximum number of bedrooms, price of the houses, and click on some buttons to ask for air conditioning, garage, etc. Dynamic queries apply direct manipulation to querying. The world of objects (houses for sale) and the world of actions are visible. Controls provide immediate, continuous and reversible feedback to users.

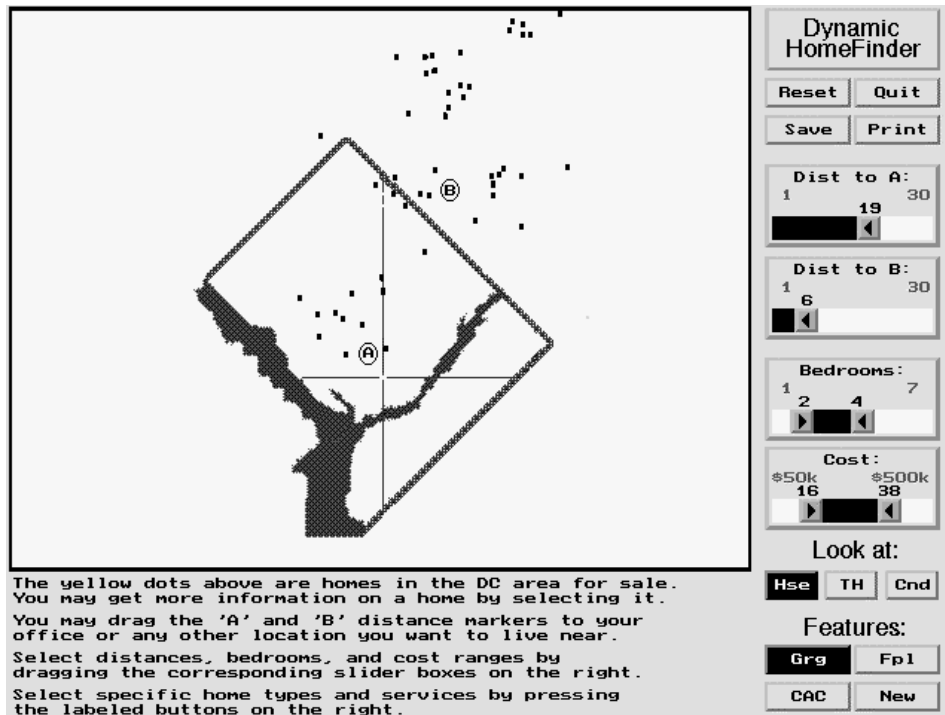


Figure 1-1: In this dynamic query interface, users can search for houses for sale in the Washington DC area. As users formulate their query using sliders and buttons, the display is updated in less than 100ms to reflect the answer to the query, providing reversible and continuous feedback.

The concept of a generic two-dimensional graphic with zooming, colors and filters was applied for the first time in the FilmFinder [2], and the commercial version, called Spotfire, appeared a few years later, in mid-1996, with increased flexibility and user control. On this basis, several prototypes have been developed since, like a Java-based visual tool for high school teachers, for the Baltimore Learning Community Project (www.learn.umd.edu), or a collection finder for the Library of Congress.

To guarantee rapid updates of the display (less than 100 msec.) dynamic query interfaces require the data to be in memory. To allow larger number of records, another paradigm called Query Previews was proposed. In a two-step approach, the query preview allows users to rapidly narrow their search by selecting rough values over only a few attributes. A result bar shows the size of the result set which to a smaller size that can be managed in the second refinement step, which consists of a classic dynamic query interface. This approach has been implemented on NASA web sites for searching remote sensing environmental databases.

My six-month project had two components.

- (1) First I participated in the development of a Java implementation of a dynamic query and starfield display interface. This application resembles Spotfire in many aspects, but Spotfire had the disadvantage to be limited to the Windows platform, and since it is a commercial product, we could not adapt it to our specific needs. In particular, NASA's applications have to be available on the Web and capable of dealing with special characteristics of the NASA data. This new application was named StarDOM for Starfield Dynamic Object Miner.*
- (2) The second and most novel aspect of this project was to investigate the visualization of multi-valued and range-valued attribute data, which has not been considered by other similar visualization systems. Examples of range attributes include time and space coverage for earth science datasets. Examples of multi-valued attributes include the names of the scientists who produced the dataset. This work was implemented in StarDOM, and tested with real NASA data.*

This report first describes related previous work, then presents the user interface of StarDOM. A detailed description of the data architecture is presented, with special attention given to the multi-valued and range-valued attribute handling. A section describes time experiments conducted to optimize the speed of StarDOM queries. Finally a discussion of possible future work is presented.

2. Review of previous work

The first section of this chapter introduces the notion of dynamic querying. Then, in the second section will be developed an historical point about NASA EOSDIS data projects in HCIL lab, with a small presentation of the previous and current research on this subject (explanations of the two-step approach: Query Preview and Query Refinement). The last part is more technical, and describes previous work relevant to the development of the StarDOM prototype, in terms of data structures and in terms of user interface.

2.1. Dynamic query interfaces

Let's have first a small introduction to dynamic query interfaces used in data visualization. The paper [2], *Tight Coupling of Dynamic Query Filters with Starfield Display*, describes the new principles of visual information seeking through the use of dynamic query filters and starfield displays. The aim of this approach is to quickly reduce the number of records of a database.

2.1.1.1. Dynamic Query Filters

The dynamic query filters are different kind of widgets (range sliders, alpha sliders, etc.) used to create requests, replacing the usual line commands of SQL. The previous work of Christopher Ahlberg and Ben Shneiderman [4], [5], [6] demonstrated dramatic performance improvements and high levels of user satisfaction. Dynamic query filters allow the visual interpretation of many types of queries and prevent the possible typing errors. With dynamic queries, the query is represented by a number of widgets. Each slider consists of a label, a field indicating its current value, a slider bar with a drag box and a value at each end of the slider bar indicating minimum and maximum values. A good example in this case is the Alphaslider [3, 7]. The Alphaslider is used to rapidly scan through and select from a list of alphanumeric values.

2.1.1.2. Starfield display

Starfield displays map multi-dimensional data into a two-dimensional screen. Most of the time, it is possible by selecting ordinal attributes of the items and using them as the axes. The starfield displays can be seen as a scatterplot with additional features to support selection and zooming.

The tight coupling between starfield display and dynamic query filters should allow the direct visualization of the modifications. The outputs-are-inputs principle allows the users to create a new request over the results of the previous ones.

2.1.1.3. Example of a dynamic query interface system: the HomeFinder

[1], *The Dynamic HomeFinder: Evaluating Dynamic Queries in a Real Estate Information Exploration System*, gives a good example of use of starfield displays and dynamic queries. Even if this program is rather old, it remains the preferred demonstration of dynamic interfaces for HCIL. Let's also use this demonstration as an example of use of dynamic interfaces.

The program used for the experiment in [1] applied dynamic queries to a real estate. Finding a home is a laborious task. The two most common methods currently used are paper and relational databases with SQL based access capabilities: slow, difficult to use, little feedback, irreversible.

The dynamic query interface (see Figure 2-1) provides a visualization of both the query formulation process and corresponding results. This application was built using the C programming language. A map of Washington, DC metropolitan area, is displayed on the left. The homes that fulfill the criteria set by the user's current query are shown with yellow dots on the map. Users perform queries, using the mouse, by setting the values of the sliders and buttons in the control panel to the right. The query result is determined by ANDing all sliders and buttons.



Figure 2-1: Dynamic HomeFinder (DQ interface) with some homes displayed and A&B marker set.

Now, let's consider the simple example described in [1] to better understand the advantages of dynamic query interfaces like the HomeFinder compared to SQL-based form fill-in interfaces.

Take a hypothetical situation where a new professor, Dr. Alison Jones, has just been hired by the University of Maryland. She might encounter this tool in a touchscreen kiosk at a real-estate office or at the student union. She selects the location where she will be working by dragging the 'A' on the map. Next, she selects where her husband will be

working, downtown, near the capitol, by dragging the 'B'. Figure 2-1 shows the interface after Dr. Jones has dragged the 'A' and 'B' indicators to her desired locations.

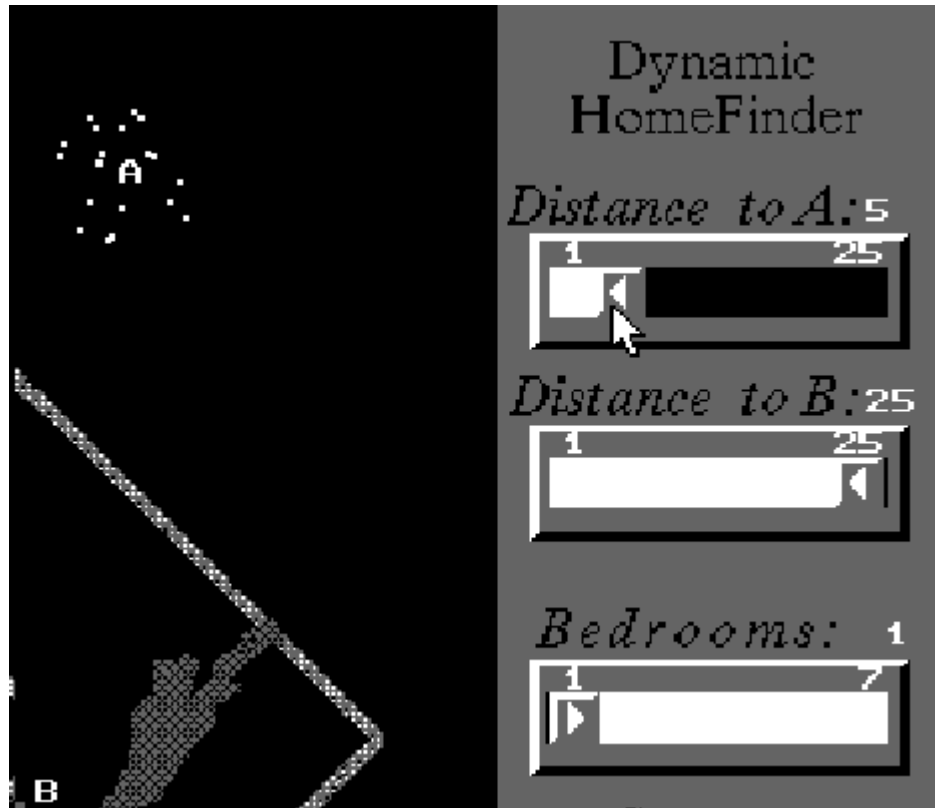


Figure 2-2: HomeFinder with all homes within a 5-mile radius of the 'A' marker displayed.

Dr. Jones would like to ride her bicycle to work, if possible, so she sets the 'Distance to A' slider on the right to 5 miles or less. This is indicated by the highlighted region of the slider now indicating from 0-5 miles. Her husband is taking the Metro, so the distance to his office is not very important. Figure 2-2 shows how the screen looks after she has adjusted the 'Distance to A' slider. Note that this is done instantaneously as she moves which cannot be captured with snapshots, but which enables her to quickly see how homes are eliminated as she narrows the distance requirement.

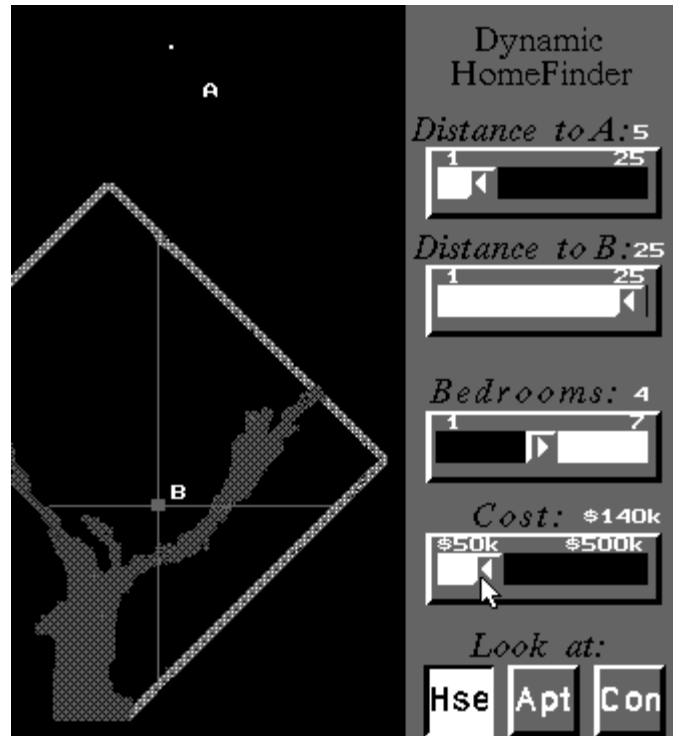


Figure 2-3: HomeFinder with all houses within a 5-mile radius of the ‘A’ marker AND have 4 or more bedrooms AND cost less than or equal to \$140,000.

Dr. Jones is only interested in houses, not in apartments or condominiums so she toggles those buttons off. Finally, she drags the bedrooms slider up to 4, since she needs at least four bedrooms, again indicated by the highlighting in Figure 2-3 showing that houses with 4-7 bedrooms are now being displayed. In Figure 2-3, she also drags the cost slider to \$140,000, a modestly priced home where she used to live. Here we encounter the all-or-nothing phenomena as Dr. Jones has eliminated too many houses with her query. This is easily solved as she realizes that houses must be more expensive in this area. Dr. Jones drags the cost slider up to \$220,000 in Figure 2-4, a price that many more houses in the area fulfill.



Figure 2-4: Close-up of Dynamic HomeFinder with all houses within a 5-mile radius of the 'A' marker AND have 4 or more bedrooms and cost less than or equal to \$220,000.

Finally, just out of curiosity, Dr. Jones clicks on the 'Garage' button in Figure 2-5 only to find that few houses have a garage in the price range and area she is looking at. Once she has narrowed her query, it is easy for Dr. Jones to experiment, seeing what services the homes offer, or what is available if she was willing to pay a little more, and so on. In this way the interface encourages exploration and bolsters user confidence.

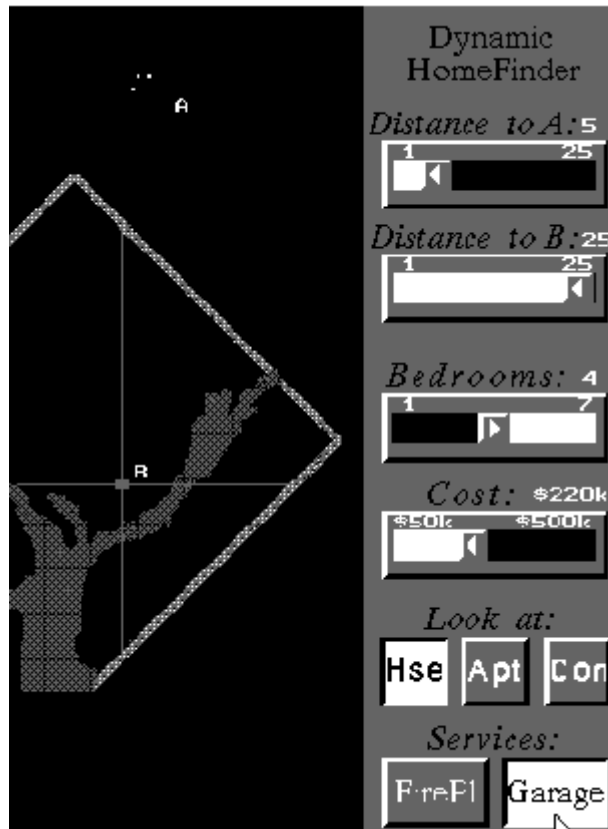


Figure 2-5: Dynamic HomeFinder with all houses within a 5-mile radius of the ‘A’ marker AND have 4 or more bedrooms AND cost less than or equal to \$220,000 AND have a garage.

2.1.1.4. Spotfire, a powerful information retrieval system

Spotfire (<http://www.spotfire.com>), a commercial product first born in the HCIL laboratory, is certainly one of the best examples of dynamic query interfaces. Its information retrieval capabilities are appreciated by more than 150 customers. Among these customers, we find the majority of the world's largest pharmaceutical, specialty chemical and biotechnology firms. The company was founded in 1996 and maintains its European headquarters and development center in Göteborg, Sweden, and its U.S. headquarters in Cambridge, Mass.

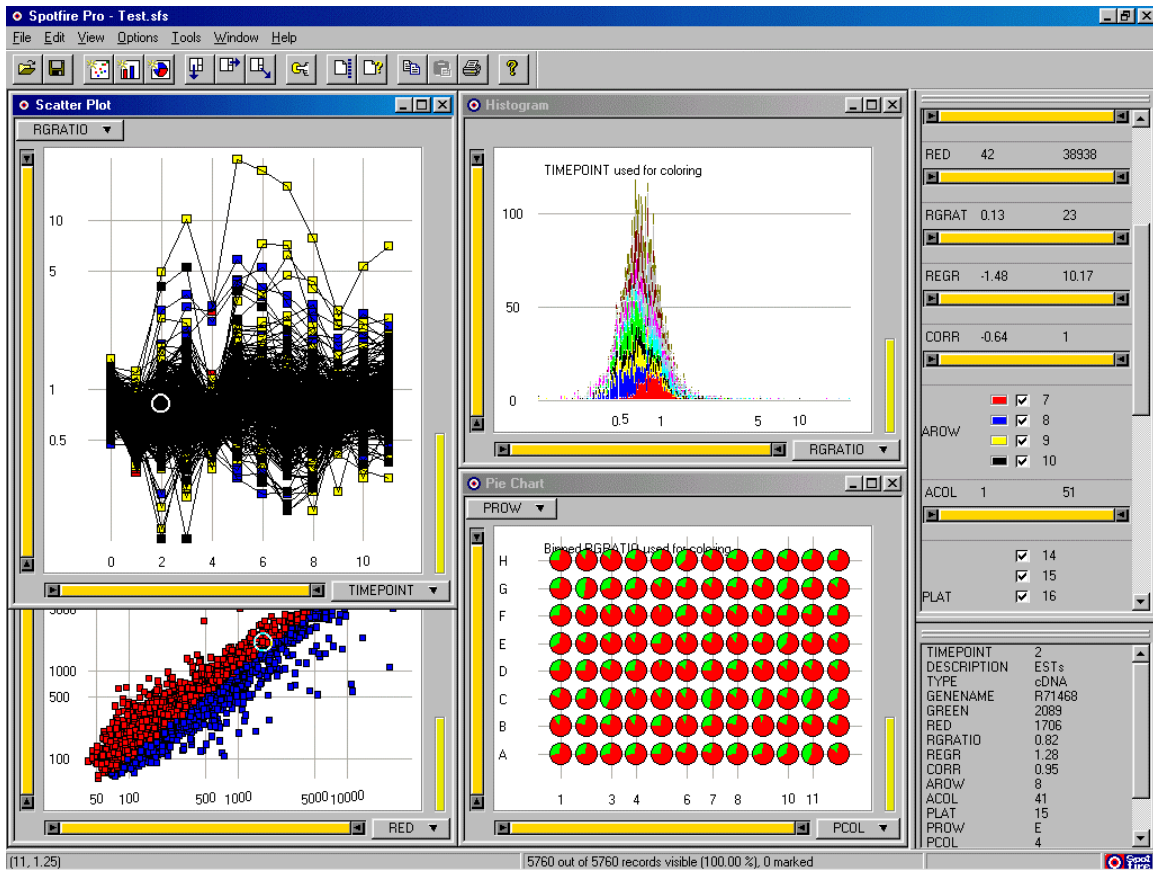


Figure 2-6: Example of information data visualized through Spotfire.

Spotfire provides a wide range of tools to visualize many different kinds of information. Figure 2-6 presents an example, with a multi-window representation, and different features of Spotfire, like the color coding option, pie-charts, a “group-by” representation at the top left, and so on.

Spotfire is a powerful visualization system allowing users to employ it for many different purposes. Its API functions can also be used for further development.

2.2. Query Preview and Query Refinement, the case of NASA Earth Science

Let's now study the history of the work for NASA EOSDIS in HCIL. The paper [8], *Refining Query Previews Techniques for data with Multi-valued Attributes (the case of NASA EOSDIS)*, combined with [22], *Interface and Data Architecture for Query Preview in Networked Information Systems*, give a good overview of the existing systems, and of the problem of multi-valued attributes. The aim of the system EOSDIS of NASA (Earth Observing System Data and Information System) is to help scientists find datasets of interest in large distributed NASA archives.

EOSDIS (Earth Observing System Data and Information System) is a project lead by the U.S. National Aeronautics and Space Administration (NASA). Its goal is to provide online access to a rapidly growing archive of scientific data about the earth's land, water and air. Data is collected from satellites, ships, aircraft, and ground crews, and stored in designated archive centers. Scientists, teachers, and the general public are given access to this data via the Internet.

HCIL is developing user interfaces that use graphical dynamic queries. In the case of NASA EOSDIS databases, two different phases have been created: the "Query Preview" paradigm, and the "Query Refinement". The aim of these graphical systems is to facilitate the browsing and retrieval of data from very large archives. The next part of this report (2.2) presents in detail the current functioning of NASA EOSDIS data visualization.

A good number of users will be using the NASA EOSDIS information systems to locate earth science data of interest. The traditional approach of such a system is to propose a form fill-in interface to allow users to query the databases. Of course, such interfaces are frustrating for the users when the query returns either zero-hit or mega-hit. In the case of a zero-hit result, the user had waited a certain time (the time for the request to be solved) for a useless answer, and in the case of mega-hit query, the download phase of the data would be phenomenal.

The concept of Query Previews and Query Refinement has been developed in the way to solve this problem. Query Previews allow users to see the number of data returned by a specific request, and the user is allowed to modify his request to obtain a manageable number of hits. The initial Query Preview concept appeared in 1995, and the first operational Query Preview in NASA Global Change Master Directory (GCMD) has been installed in 1998 (<http://gcmd.nasa.gov>). The Query Refinement phase is the next phase after the Query Preview. Once the number of records becomes small enough, Dynamic Querying can be used again. Even if the information is still at the level of meta-data, it is possible to filter according to all the existing attributes of the database.

2.2.1. The operational Query Preview interface for the NASA GCMD

2.2.1.1. Description of the system

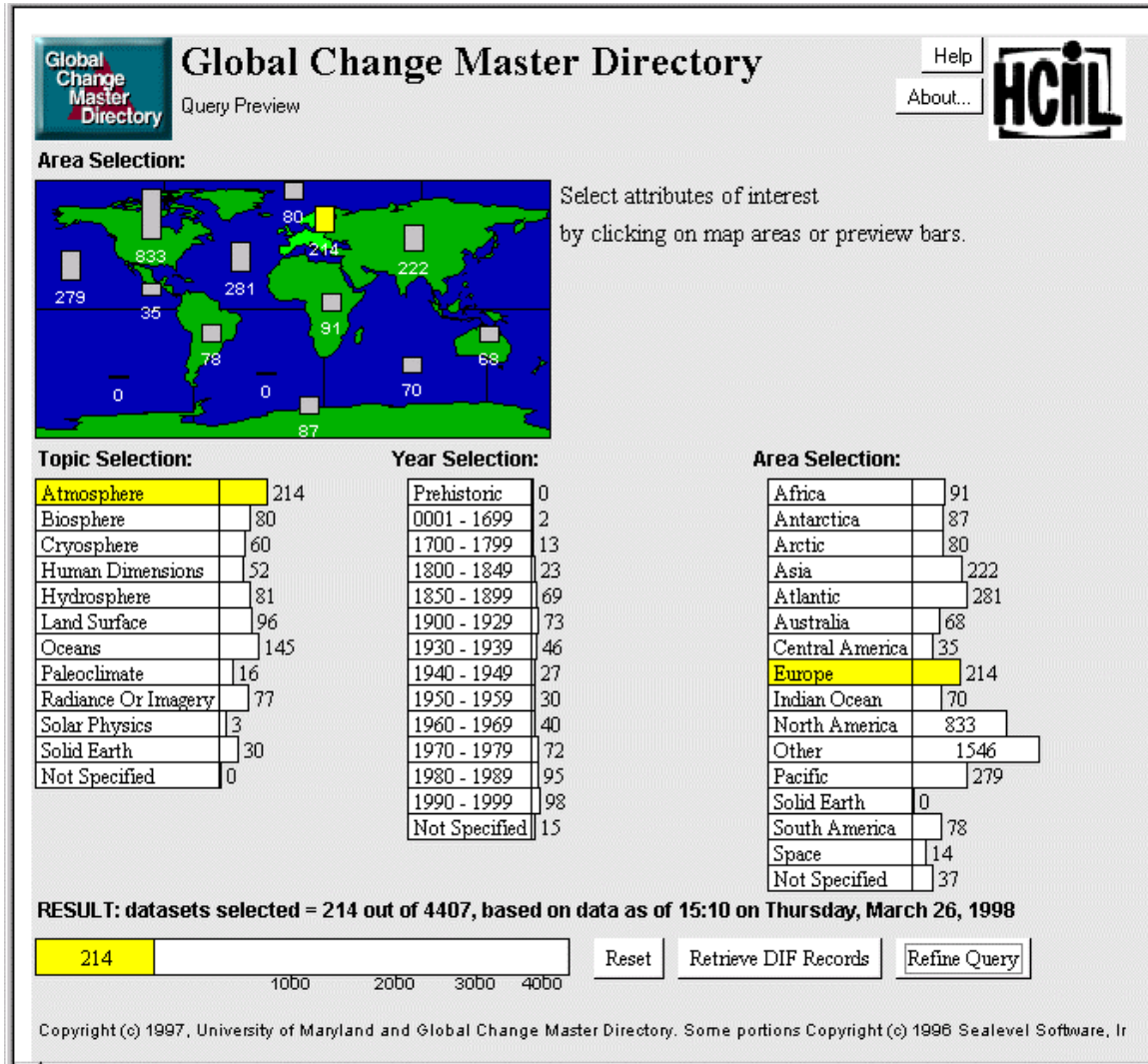


Figure 2-7: The query preview system gives users an idea of the size and distribution of the result set before the query is submitted, and addresses the zero-hit or mega-hit query problem. Counts are updated using pre-computed preview tables.

GCMD uses a query preview system developed in HCIL to search datasets (Figure 2-7). The records being searched consist of the metadata (i.e., data about data) for each dataset of NASA. These datasets can be considered as meta data, and not real data which are too large to be searched or downloaded directly. These datasets have a reduced

number of attributes, for instance, years, areas over the planet, topics (atmosphere, oceans, land surfaces, etc.). When users select attribute values, the number of available data sets is displayed, allowing the user to improve his request to obtain a different number of datasets, or to accept it.

The three possibilities, visible above, are *Reset*, that would make the user to start a new query, *Refine Query*, that would give extra available attributes to help the user to improve his request, and *Retrieve DIF Records*, to receive a list of results. If the user chooses the option *Refine Query*, the system goes to the second phase called Query Refinement. This possibility can be offered when the number of datasets selected in the Query Preview is inferior to a certain threshold.

2.2.1.2. Implementation of this Query Preview system

To be effective, this query preview system only needs preview tables of counts (that can be updated every night for example). The real implementation revealed a problem when it is used with multi-valued attributes. Indeed, the preview table is just a simple N dimensional array for an N attribute preview table. This technique works well for data having a single value for each attribute (e.g., the case where each record only has one topic, covers only one time period, and one area of the globe) but becomes a problem with multi-valued data. The duplication factor induced by multi-valued data (data covering more than one region of the globe for example) becomes a major restriction.

A detailed implementation of this system is explained in [10], “Interface and Data Architecture for Query Preview in Networked Information Systems”. Let’s see the example given in this paper [10], with tables used for the case of a Restaurant Finder, displayed on Figure 2-8 and Figure 2-9.



Figure 2-8: An implementation of the Query Preview system in the case of a Restaurant Finder. It is here possible to dynamically query restaurants according to its type, location, quality, etc.

	None	Visa	MasterCard	Visa & MC
Italian restaurants	8	23	12	160
French restaurants	18	45	12	80
Indian restaurants	6	34	23	98

Figure 2-9: Extract from the preview tables projected on two attributes (type of restaurants, and accepted credit cards), case of a Restaurant Finder. Here, the type of card is multi-valued, requiring extra counts (extra columns).

This table results from a query where a specific attribute (the rating) has been specified. The two other attributes are the kind of restaurant (French, Italian or Indian), and the type of cards they accept. This last attribute has two possible values, i.e.,, restaurants can accept Visa, MasterCard, but they are also able to accept both of them, or none of them.

In consequence, in the given case of 12 possible values for an attribute, 12 for a second attribute, and 10 for a third one, the size of the preview table if all these data are single-valued is $12 \times 12 \times 10$. In the case of these data are multi-valued, and if all the combinations are conceivable (that would correspond to the worst option), the size of the preview table rises to $2^{12} \times 2^{12} \times 2^{10}$.

Of course, an alternative technique has to be used in the case of large number of records with multiple attribute values. In order to control the Preview Table size, many practical solutions can be used, e.g., reduce the number of attributes or values, or keep this large table on the server. A simple solution is just to ignore the possible combinations of data. As a result, the datasets that have n parameters will be counted n times. This is possible if the combinations are a small proportion of the data, but unacceptable otherwise, because the errors introduced by the wrong count of data would become too visible.

Nevertheless, two approaches have been developed to try and solve this hard problem.

2.2.1.3. Single-attribute, range queries

In the particular case of range queries, on range variables, multi-valued data are frequent, especially for EOSDIS queries (e.g., temporal or geographical coverage). An algorithm was devised by Richard Beigel and Egemen Tanin at HCIL, to make the preview tables independent of the number of records (see [12] for details, and Figure 2-10 for a presentation).

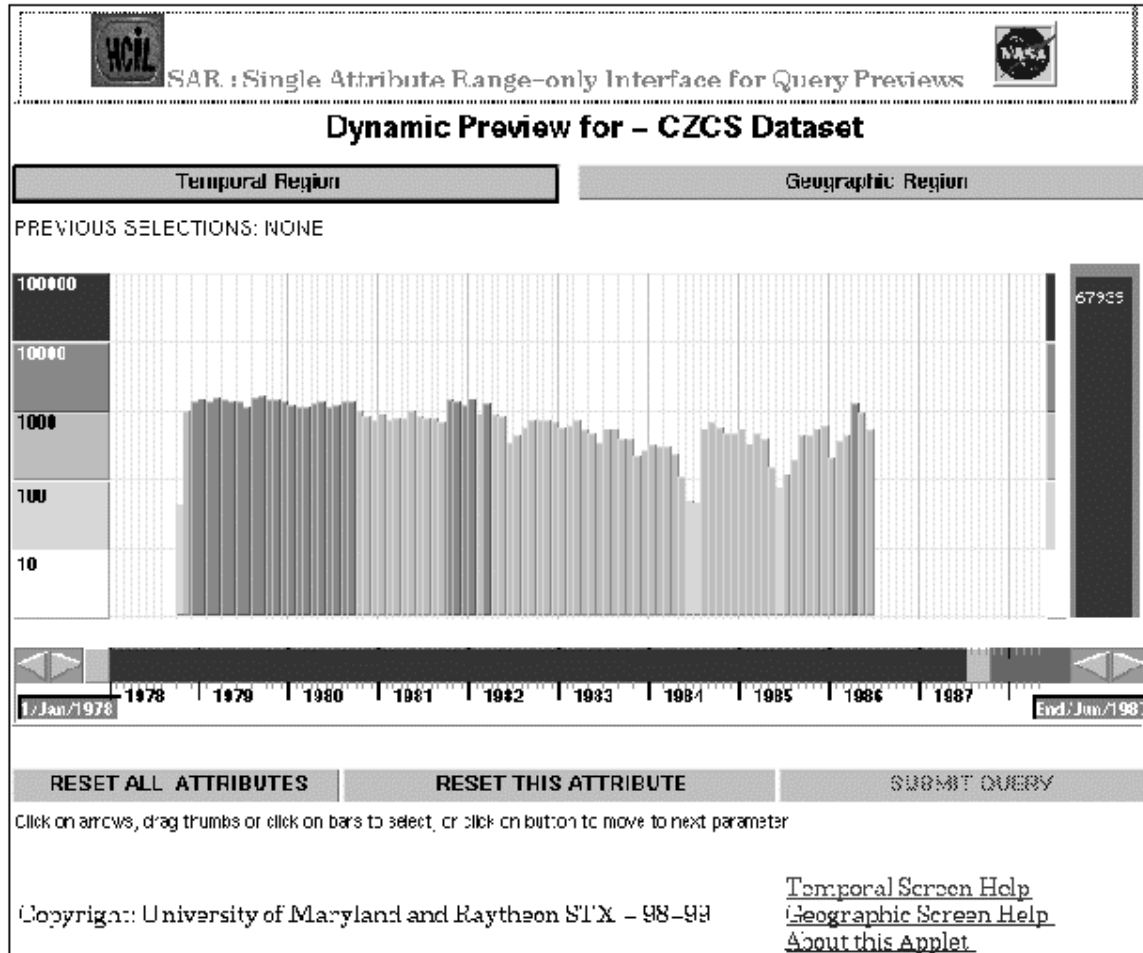


Figure 2-10: This alternative approach presents a series of single attributes, range only, query previews. The first attribute shown here is time. Each month has a bar showing the total number of records for that month. We can see that data is available from Jan 1978 to Jun 1987. Around May 1984, the instrument must have had problems and therefore the number of granules is much smaller than usual. Using the sliders, users can select a time range, and see the total number of granules selected on the large bar on the right. This approach scales up because the preview table size is independent of the number of records. This approach is also able to handle multi-valued attribute data.

2.2.1.4. The binary preview

The second approach, the binary preview, only indicates if there is data or not, no count of the data is made. Thus, the size of the table won't increase drastically with the number of multi-valued attributes. Let's see what happens with the previous example, tables are not as large in this case (see Figure 2-11).

	Visa	MasterCard
Italian restaurants	0	1
French restaurants	1	0
Indian restaurants	1	1

Figure 2-11: Extract from the Preview table projected on two attributes (kind of restaurants, and accepted credit cards), in the case of binary Preview. A simple boolean operation gives the result for the “None”, and “Visa & MasterCard” cases.

If a user wants to know if there are Italian restaurants that accept MasterCard or Visa, the request will compute the result by a simple Boolean operation, and the result will be ‘true’ in this given example. Figure 2-12 presents a screenshot of the binary preview system.

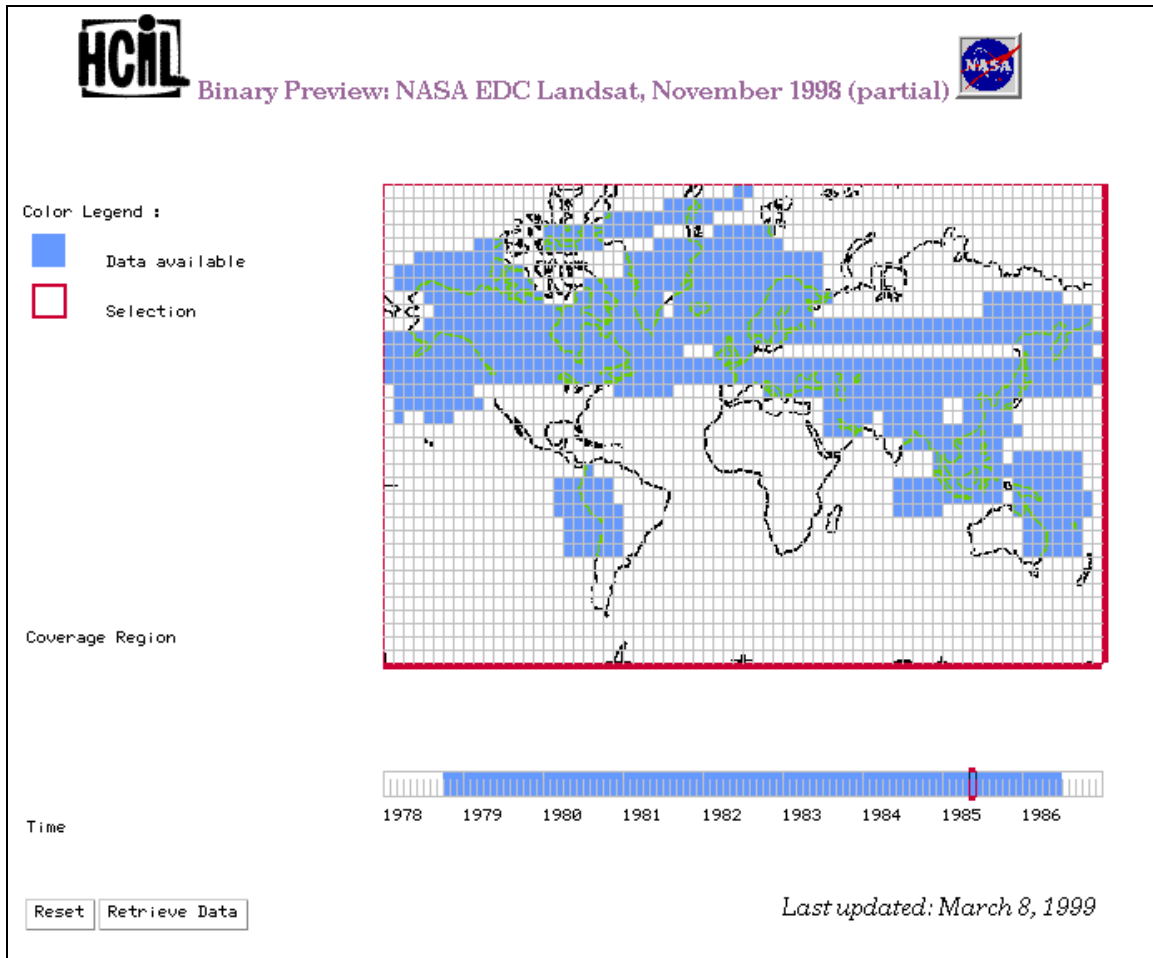


Figure 2-12: As users select one month, the binary previews on the map display the data available for those months, revealing that only a subset of the globe was actually observed during that month. Users can select more than one month at a time. Similarly, users can select an area(s) on the map and see during which year and months the data was actually collected for that area.

2.2.2. The Query Refinement phase

The Query Refinement Phase is the next logical phase after the Query Preview. The aim of the Query Preview is to reduce the large amount of available data to a manageable stack, thanks to tables containing numbers of available data.

Then, the data of the individual records are passed to the Query Refinement phase. This data is still meta-data, i.e., not real data, but only a representation of them through a certain number of attributes. The purpose of the Query Refinement phase is to finally refine the queries before downloading or sending the real data. This time, all the possible available attributes concerning the data are used and can be filtered by the user.

The difficulty of the Query Refinement phase is not in the back-end anymore, but in the front-end. The multi-valued attributes are not linked to the same problem as before. In this phase, there is no need of tables keeping track of the number of data according to different attributes, because the real meta-data is downloaded. The problem here is how to display them and allow the user to filter them according to specific attributes.

In the case of NASA EOSDIS data, the possible attributes of the datasets are:

- the detailed space extent and temporal interval (range data),
- parameters measured in the dataset (one or more values for a data),
- the sensor used to generate the dataset (one or more values for a data),
- the platform on which the sensor resides (one or more values for a data),
- the project with which the platform is associated (one or more values for a data),
- the data archive center where the data is stored (one or more values for a data),
- data processing level which indicates raw sensor data (level 0) to highly processed data (level 4).

Figure 2-13 presents the query refinement phase system originally planned for NASA. Users can browse all the information about individual datasets. Thus, the result set can be narrowed again to make more precise selections on more attributes:

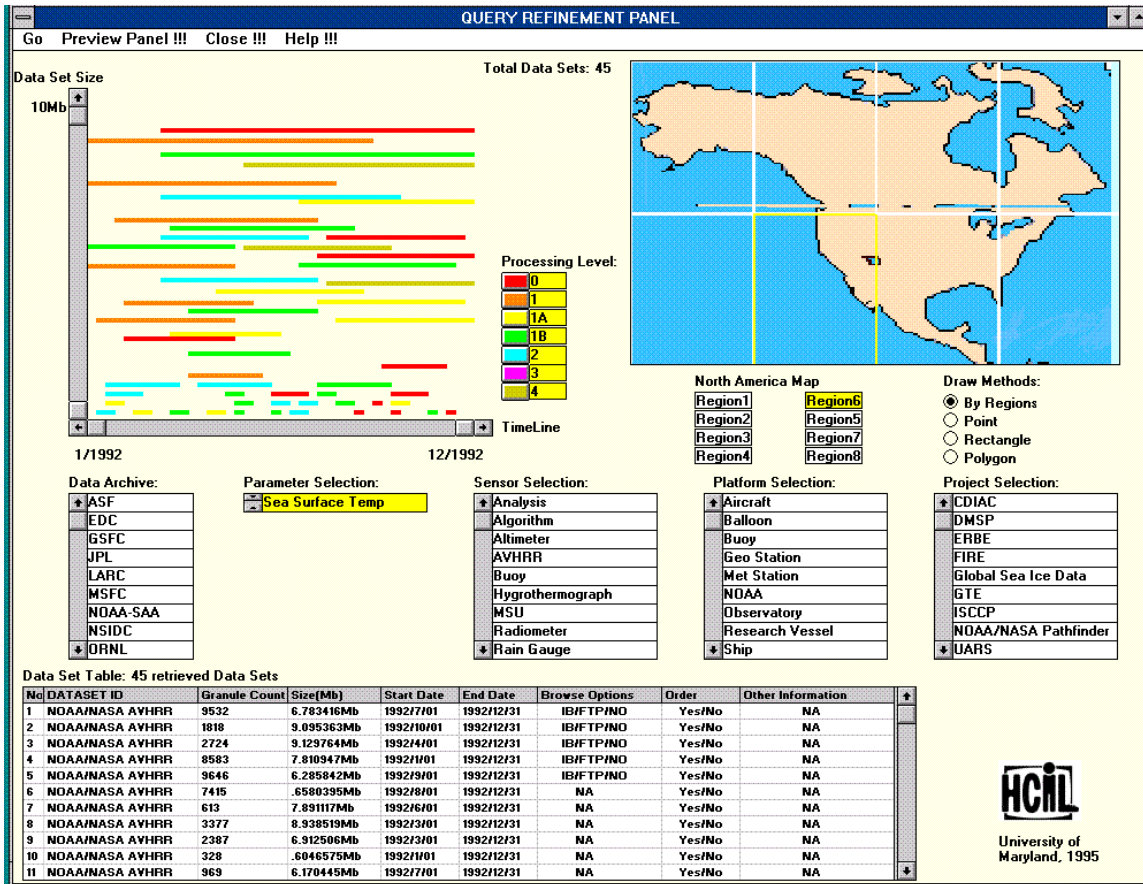


Figure 2-13: In the Query Refinement, users can browse all the information about individual datasets. The result set can be narrowed again by making more precise selections on more attributes than in the Query Preview phase.

2.2.3. Other research in this field

2.2.3.1. VQE - Visage, an Interactive Visual Query Environment for Exploring Data

A visual query environment has been developed at the Carnegie Mellon University Robotics Institute, Pittsburgh. [19] and [20] give a detailed explanation of this system, which is based on two tools. The first one, called Visage, is a direct manipulation exploration environment, the other one, VQE, is a visual query language that allows the reuse or modification of exploration sessions.

There are three main domains of complexity, which have been attempted to address in the Visage/VQE environment.

The first one concerns the browsing to locate information. It is difficult for people to locate information when it is distributed across multiple objects. For example, on the database scheme given on Figure 2-14, locating the current owners of a set of houses

requires knowing that there is no simple attribute of houses that tells this. Rather, one must find the most recent sale object associated with each house and then retrieve the buyer related to the sale. In other words, users must be aware of the database schema in order to use the correct terms for objects, relationships, and attributes to find relevant information. A graphical representation of the schema that can be directed manipulated is the solution.

Second difficulty: creating visualizations that present the relationships among attributes from multiple objects. For example, Figure 2-14 (bottom) is a chart of the `lot_size` of houses versus the `selling_price` of the corresponding sales of those houses. A single point encodes attributes of two objects: a house and sale. No previous interactive visualization system directly supports the construction of visualizations that are derived from multiple objects. Typically, a relational database query must be constructed using a separate interface that joins multiple tables into one containing attributes from both houses and sales.

The third complexity domain concerns the expression of queries. Two possibilities exist to specify a set of objects. Either it is possible to emit an **intentional query**, that would, for instance, select “all the houses within 100 yards of a house, with a different value for neighborhood attribute”. Either it is easier to indicate sets of objects via direct manipulation than to create an expression defining the set (**extensional query**).

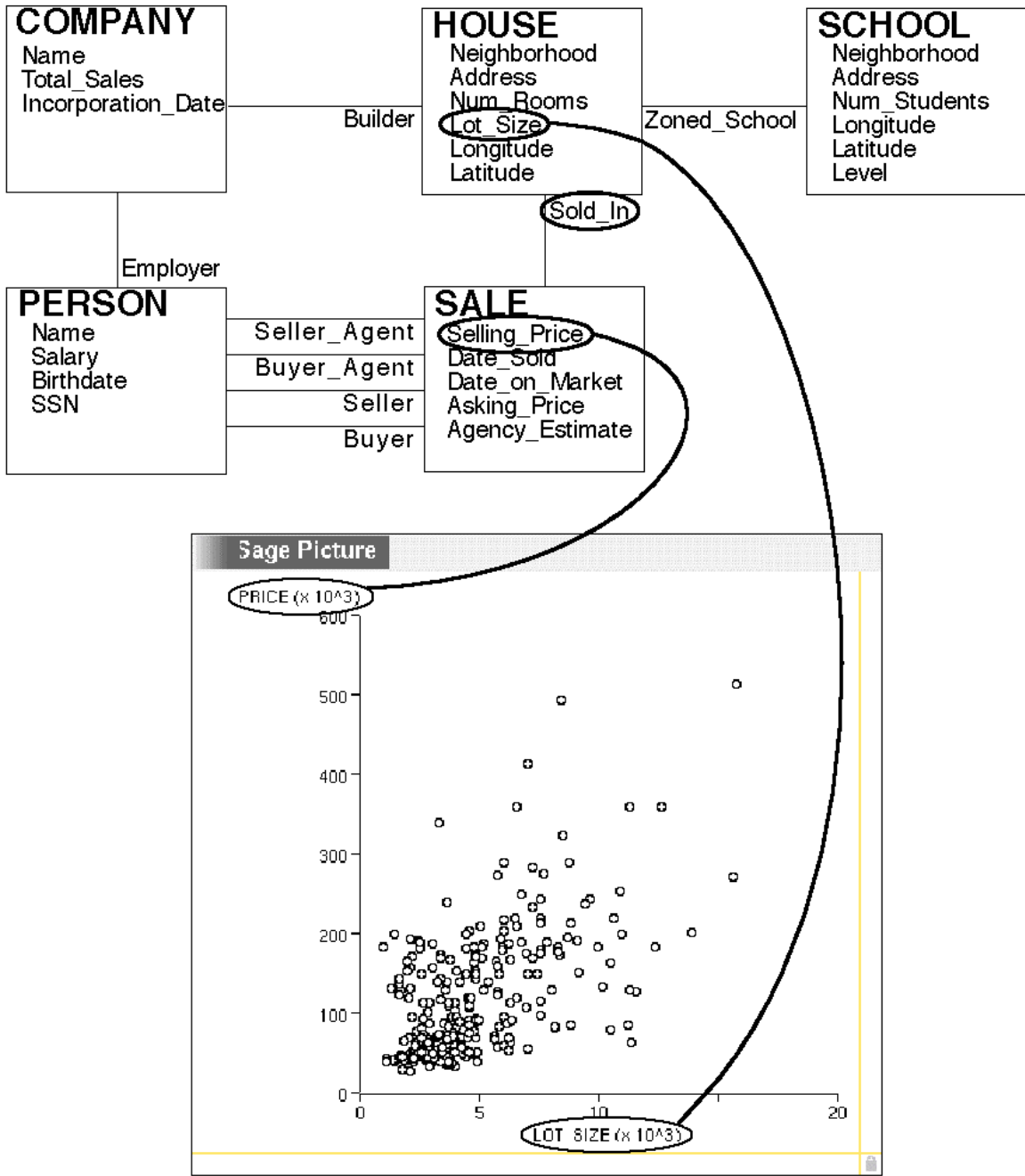


Figure 2-14: Top: the database schema. Bottom: chart comparing attributes from two different object types. All <house, sale> pairs related by the sold_in relationship generate a point on the chart.

As an answer to these problems, the combination of the systems Visage and VQE provide:

- A database schema browser (see Figure 2-14), and graphical representation of the queries (VQE),
- An automated design of visualizations that integrate many attributes,

- Direct manipulation operations, either for making extensional queries (Visage), either for making intentional queries (VQE).

This system provides many powerful browsing tools. For instance, data objects are represented as first class interface objects that can be manipulated using a common set of basic operations, such as drill-down and roll-up, drag-and-drop, copy, and dynamic scaling. These operations are universally applicable across the environment, whether graphical objects appear in a hierarchical table, a map, a slide show, a query, or other application user interface. Furthermore, graphical objects can be dragged across application UI boundaries.

A user can navigate from the visual representation of any database object to other related objects. For instance, from a graphical object representing a real estate sales agent, one can navigate to all the houses listed by that agent. It is also possible to aggregate database objects into a new object, which will have attributes derived from its elements. The Figure 2-15 shows this possible navigation to other objects in the database.

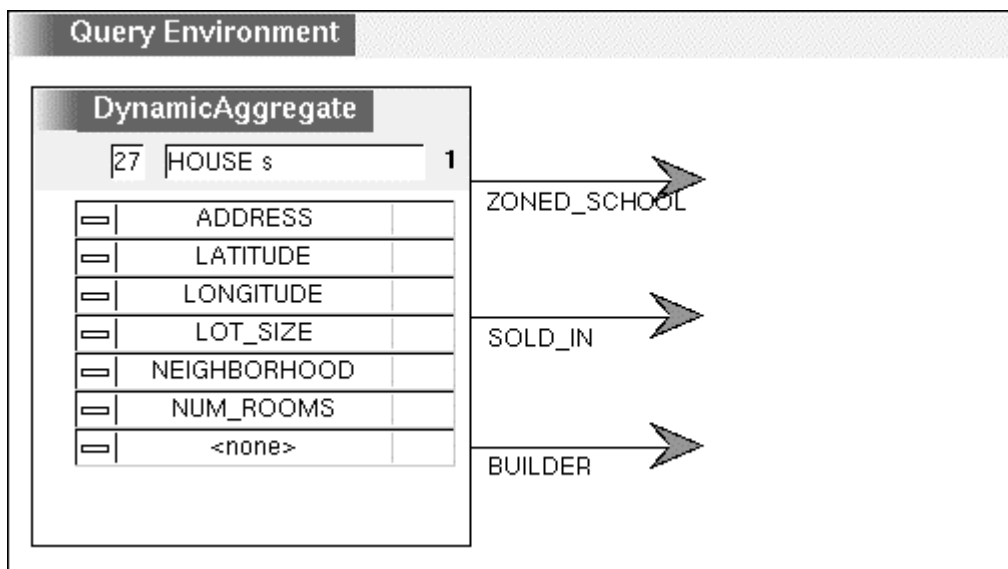


Figure 2-15: Houses have been selected on a starfield display (for instance), and dragged to this VQE environment, where they become a dynamic aggregate. The arrows on the right constitute a menu for parallel navigation.

Good examples of Visage/VQE navigation are given in [19] and [20]. Let's have a look at one of them, represented on Figure 2-16. Here, a sale agent wants first to link the houses with their sales. He will select the sold_in relationship and drag its arrowhead to a screen location where a second dynamic aggregate will be placed. We call this operation parallel navigation, because we are navigating across the sold_in relationship for each element of the aggregate. The new dynamic aggregate will be the aggregate of all the sales of these houses. Parallel navigation corresponds to a join in a relational query.

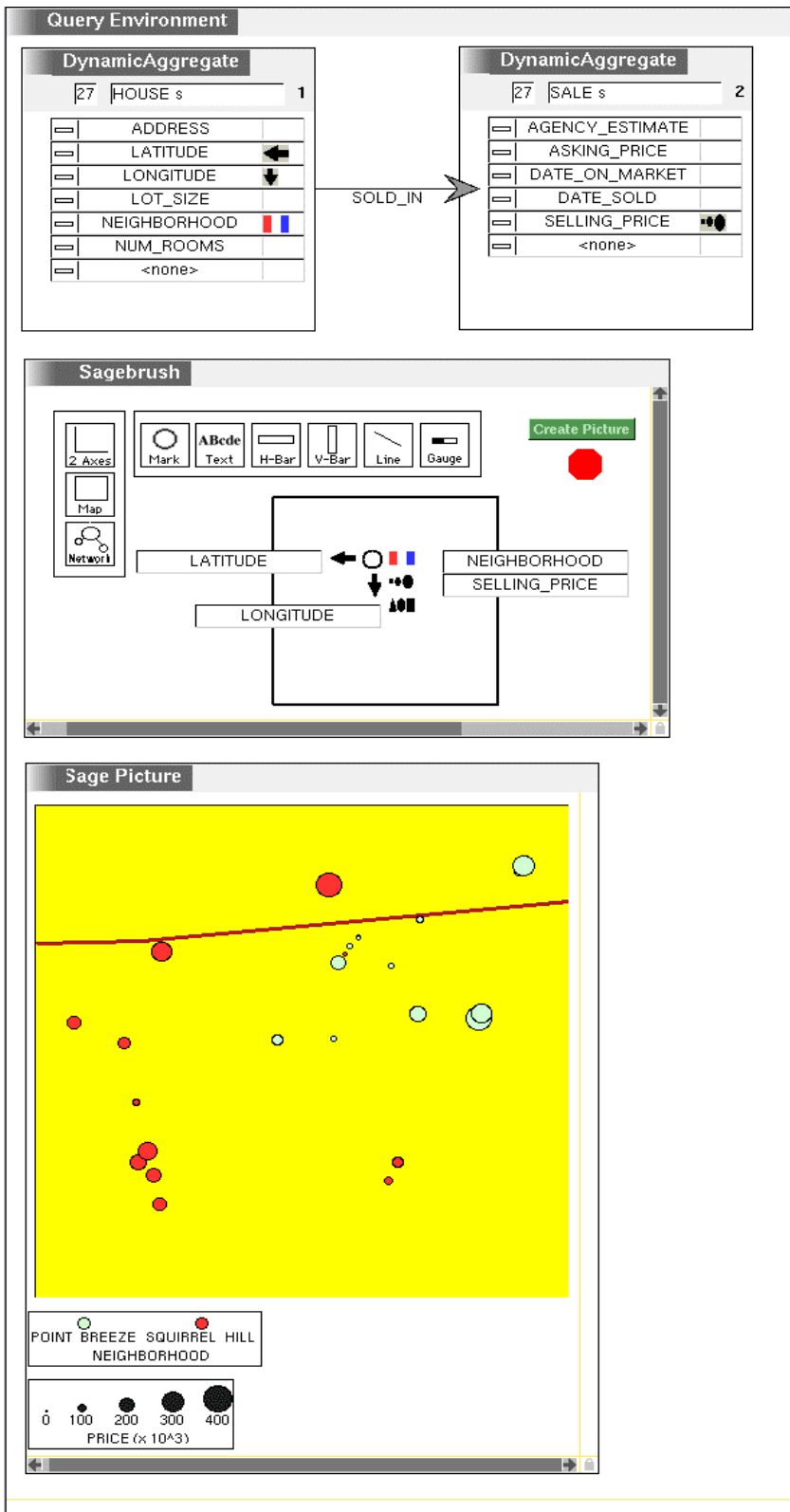


Figure 2-16: The sagebrush sketch will represent the resulting aggregate data with different graphical options (color for neighborhood, size for price).

2.2.3.2. DEVise: Integrated Querying and Visual Exploration of Large Datasets

The aim of DEVise is to propose an intuitive and powerful set of querying and visualization primitives. With DEVise, users can render their data in a flexible, easy-to-use manner. Rather than provide just a collection of presentation possibilities (e.g. piecharts, scatterplots, etc.), a mapping technique was developed to allow a remarkable variety of visual presentations to be developed easily through point-and-click interface. A special feature is that users can interactively drill down into a visual presentation, all the way down to retrieving an individual data record.

Many powerful paradigms were implemented on DEVise, all more powerful than the other! One of them allows users to handle large, distributed datasets. The tool is not limited by the amount of available main memory, and can access remote data stored on disk or tape. One other option enables several users to share visual presentations of the data, and to dynamically explore these presentations, independently or concurrently.

The DEVise exploration framework is extremely powerful, and the best way to appreciate it fully is to work with the system, or look at some applications in detail. Paper [21] describes some relevant examples. Some detailed applications, with colored DEVise screens are available at <http://www.cs.wisc.edu/~devise>.

2.3. Dynamic query data structures, a need for speed

This part describes some of the recent work on structures developed to improve database queries. The problem here is even more complex because of the graphical output feedback. Like any request in any other database, the answer needs to come as fast as possible, and needs to be displayed on the screen with a feedback as continuous as possible.

2.3.1. Incremental data structures for dynamic query interfaces

One of the most important features of StarDOM is its ability to provide graphical widgets to create dynamic queries. Of course, the structure used to support this dynamic querying has to be adapted, in order to provide a quick feedback, especially in the case of the range sliders of the starfield display.

In all the previous parts of the bibliography, different cases of dynamic query interfaces mechanisms were presented [1, 2, 3, 4, 5, 6, 7, 8, 10]. This recently developed mechanism has the great advantage to provide continuous feedback to the user as the query is being formulated.

Different kinds of widgets can be used for this purpose, like range sliders for continuous data attributes, alphanumeric sliders for textual attributes, toggles for binary attributes, and check boxes for discrete multi-valued attributes. These widgets are tightly coupled, and they can provide a kind of feedback as well. For instance, a bar drawn inside a range slider can indicate the bounding box of the data. When other filtering widgets are used, this bar can be updated to show the new bounding rectangle of the data, projected on the corresponding attribute.

Egemen Tanin and Richard Beigel developed a structure to handle dynamic query interfaces, and allow a fast feedback. The paper [13], Design and Evaluation of Incremental Data Structure and Algorithms for Dynamic Query Interfaces, describes the structure they developed, in the specific case of range sliders (used for continuous attributes). Of course, this kind of structure can be very interesting for a system like StarDOM.

This querying algorithm performs three major operations: setup, selection, and querying.

Setup

During this phase, all the different widgets are drawn on the screen, and the data structure is copied with a re-scale of every attribute to the range [1, p], where p is the number of

pixels in the attribute's range slider. This re-scaled information is not very often needed, this is the reason why it is possible to allow several seconds for it.

Selection

This step occurs when the user clicks on a range slider. During selection, the algorithm computes the auxiliary data structures, which depends on the currently selected attribute, and the ranges of the other attributes. The approximate allowed time for this operation is of 1 second, threshold beyond which users become annoyed.

Querying

This last step occurs when the user actually drags the mouse to update the selected range slider. This phase updates the output on the screen. Experiments show that every query should be processed in about 100 milliseconds in order to give a continuous response.

Now, let's have a look at the operations carried out during the selection phase, and the structures used here. All these structures won't be used in the case of StarDOM, because of the space required for storing them. Detailed explanations are given in the last part of this report. Still, this is an interesting case to study, especially because of the rather simple implementation (no need to build a whole tree structure for instance), and the constant time response to queries.

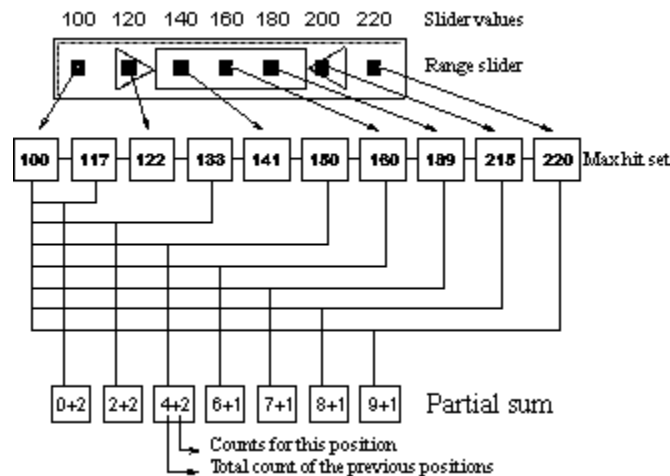


Figure 2-17: A sample range slider with bucket information. This example shows 10 records distributed on the range slider sorted in ascending order for that attribute. The arrows from the range slider to the hit set show the related positions in the sorted maximum hit set that each discrete pixel position of the range slider maps to.

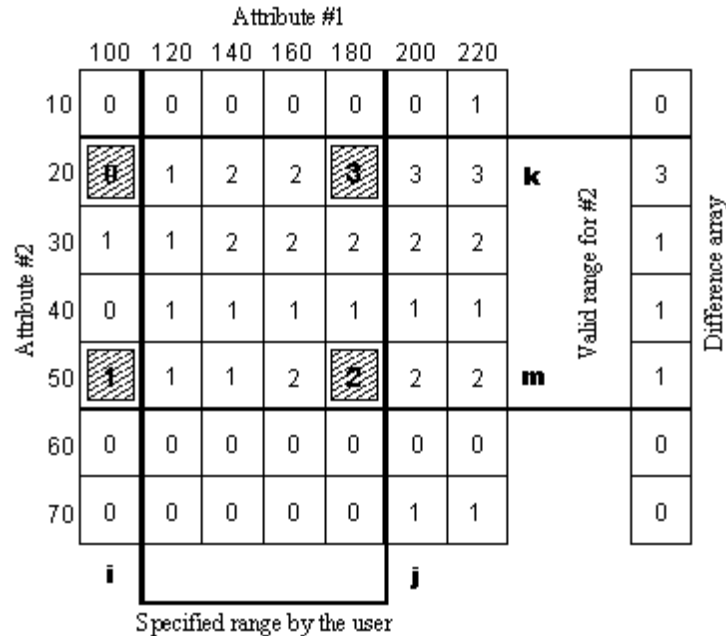


Figure 2-18: This figure uses the same example as the previous one. Each box in the square holds a count. For a given specified range of attribute 1, we can find the valid range of attribute 2 by projecting the hit set's bounding rectangle onto attribute 2.

Thanks to the bucket counts explained on Figure 2-17, it is possible to construct a table like on Figure 2-18. Each row of this figure is a prefix sum of counts from left to right. So if we subtract column j from column i , we can deal with the resulting difference array to find the valid range for attribute 2. The highest nonzero row (k) and the lowest nonzero row (m) give the valid range for attribute 2. With such a structure, the bounding boxes can be found easily, no matter which attributes are projected on the axes.

2.3.2. Trees structures

The study [23], *Data Structures for Dynamic Queries: An Analytical and Experimental Evaluation*, by V. Jain and Ben Shneiderman, describes and evaluates four different possible data structures:

- the multi-list,
- the grid file,
- the k-d tree,
- and the quad-tree.

The result of this study is that multi-list are suitable for small (few thousand points) data sets irrespective of the data distribution. For large data sets, the grid files are excellent for uniformly distributed data, and trees are good for skewed data distributions. There was no significant difference between the tree structures.

Even if the query computation depends on the hardware of the machine used, it can be optimized to a great extent by using suitable data structures. In the particular case of dynamic queries, the problem concerns a range that can be described as follows:

For a given data set, and a query that specifies a range for each attribute, find all records whose attributes lie in the given ranges.

The cost functions of various data structures, suitable for rectangular queries, are provided on Figure 2-19. Of course, many other complex structures exist, but they are mainly of theoretical interest only because of their high storage overhead.

Data structure	Storage cost $S(N, k)$	Search cost $Q(N, k)$
Sequential list	$O(N.k)$	$O(N.k)$
Multi list	$O(N.k)$	$O(N.k)$
k-d tree	$O(N.k)$	$O(N^{1-1/k} + F)$
Quad tree	$O(N.k)$	$O(N_1^{1-1/k} + F)$

Figure 2-19: Storage and search time overheads for various data structures. N is the number of records, k is the number of attributes and F is the number of records found. For the quad tree, N_1 is the number of nodes in the tree.

A few graphics explaining the implementation of some different structures are given below.

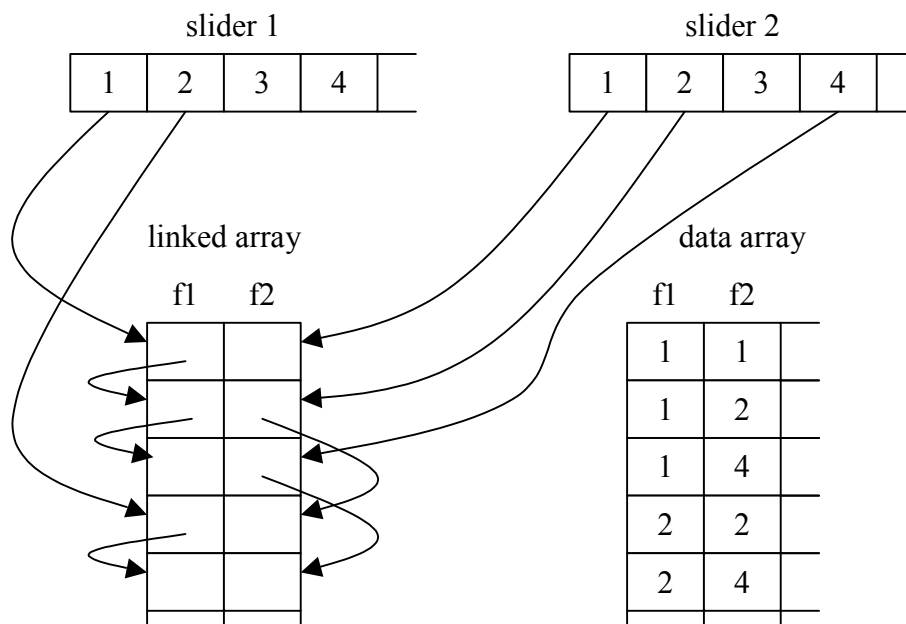


Figure 2-20: Case of a linked array used to index the data array.

Figure 2-20 presents the case of a linked array, with two attributes. For every value of every slider, the linked array table provides the list of data concerned. Besides, with every record is associated a flag that keeps count of the number of fields of the record which satisfy the region of interest. When this count becomes equal to the number of dimensions, then the record is displayed.

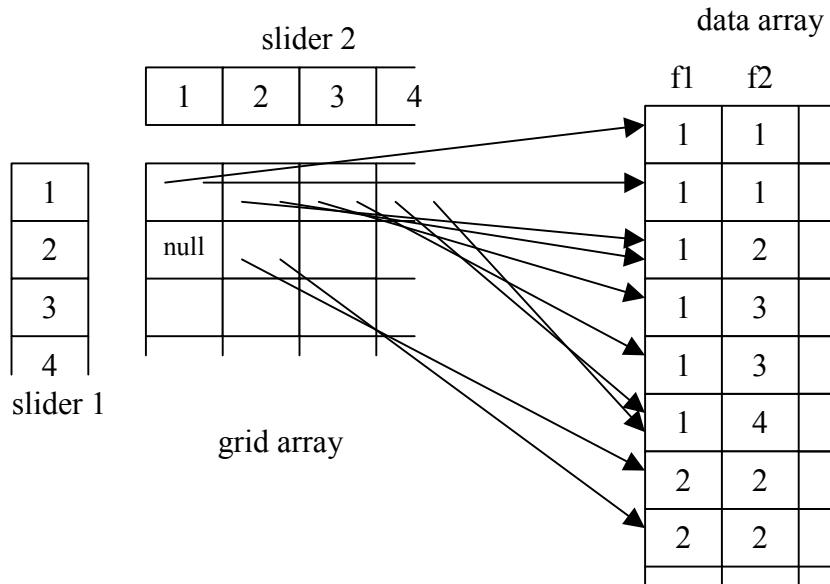


Figure 2-21: Case of a grid array used to index the data array.

On Figure 2-21, the method used by the grid array to index it is essentially a pair of indexed numbers or pointers which point to the first and to the last record in the data array that belong to the bucket.

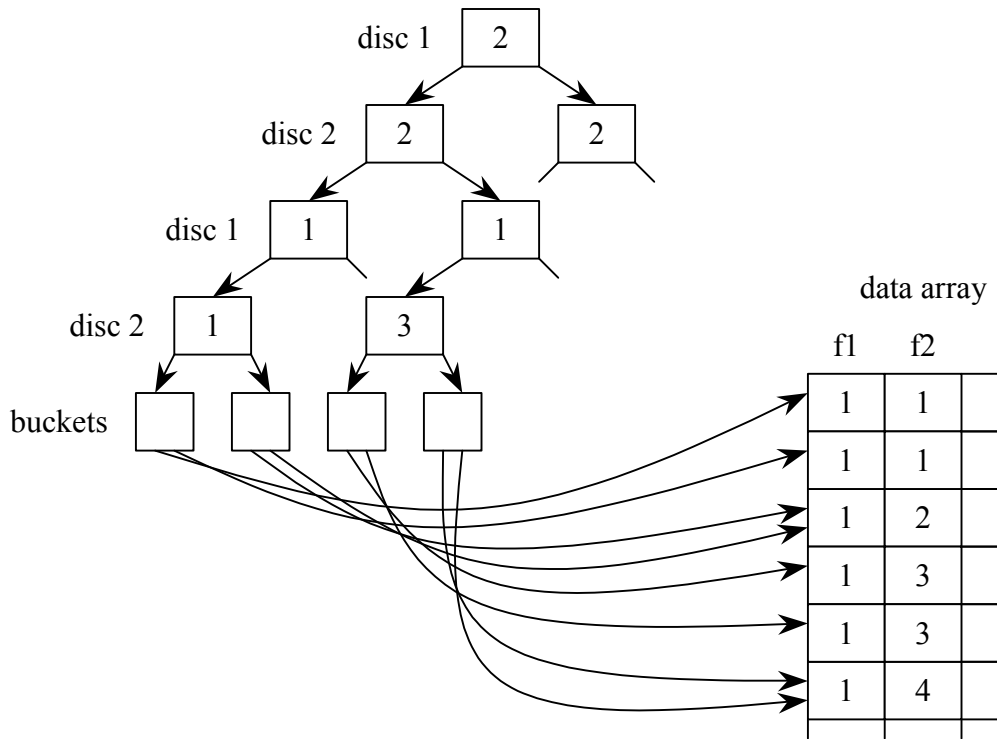


Figure 2-22: Case of a k-d tree used to index the data array.

Figure 2-22 shows a part of a k-d tree and the data array associated with it. It is possible to implement optimizations on this tree, in order to fit different kind of applications. The detailed explanations of this kind of tree are not given here, but can be found easily.

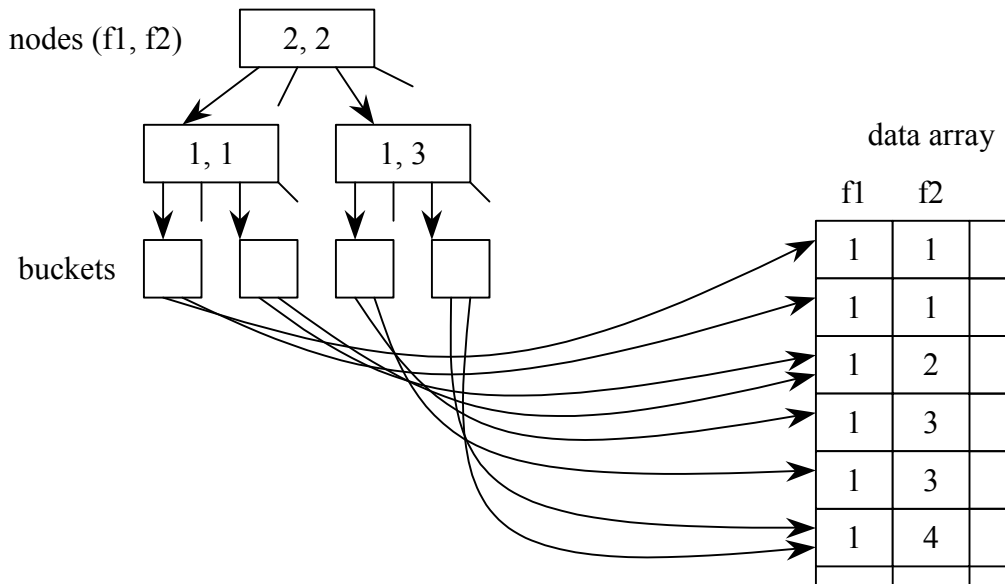


Figure 2-23: Case of a quad tree used to index the data array.

The last structure studied in the paper concerns quad trees (see Figure 2-23). The main difference between quad trees and k-d trees is that the separation in different slices is made dimension after dimension in the case of k-d trees, and all dimensions together for the quad trees. In the quad tree case, if a node has D discriminator keys, there is 2^D pointers for the children.

The article presents the total storage overhead for each of the four structures, as well as the storage and search costs. It is an article very interesting to consider if the structure of StarDOM has to be improved in a next version.

2.4. Data visualization issues

Another important feature of systems like StarDOM is their information retrieval characteristic. Users may want to see exceptions in the database, through a color difference, isolated points, or huge points. Though, a particularity of StarDOM is that users, except in the case of creation of profiles, are free to choose which attribute to map on which feature (color, size, shape, and so on).

Of course, many interfaces propose different features to retrieve significant data, and the number of different possibilities is very large.

2.4.1. Multi-windows

Multi-window prototypes allow users to see the effects of a modification of a query on an attribute on other sets of attributes. This is a very important feature that helps users to track more easily exceptional data.

Of course, this feature is already present in many other database visualizations, like in [19] and [20], where the dynamic aggregation of data points proposes much more interesting feature. With their Visage's "information-centric" approach, users can drag-and-drop objects between views and brush them. Its SAGE components overcome the problem of a limited set of visualizations by generating custom data visualization automatically.

Identically, the powerful DEVise system (see part 2.2.3.3.) coordinates region selections and axes of graphs to synchronize zoom and pan.

A recent development at the Human Computer Interaction Laboratory uses multiple view coordination in order to explore information. Chris North's system, called Snap-Together Visualization (see [11]), has already proved its usability. The multiple coordinated view approach is a powerful and increasingly employed user-interface technique for exploring information. Each view is a visualization of some part of the information, and views are coordinated ("tightly coupled", or "linked") so that they operate together as a unified interface.

Snap-Together Visualization (STV) has an architecture and system that allows users to coordinate visualization tools. Users can build coordinated multiple-view interfaces for information exploration that are customized to the specific needs of their data and tasks. Users query their data and load results into desired visualizations, then establish coordination between visualizations for selecting, navigating, or re-querying. Developers can make independent visualization tools 'snap-able' by simply including a few hooks with a small amount of code.

STV creates a new class of users called "coordination designers". Coordination designers know the data, have knowledge of good user interface design, and use STV to construct coordinated multiple-view interfaces. Coordination designers can be end-users who snap visualizations together for their own browsing needs. Or, coordination designers, like web designers, can build browsing interfaces for other less-knowledgeable end-users to explore information. Coordinated designs can be saved, distributed, and then broadly used by others.

Figure 2-24 presents an example of Snap-Together use. This complex scenario is used in the case of WestGroup which provides information support for legal professionals, including databases of millions of court cases from Federal and state jurisdictions. STV is used to quickly prototype multi-view visualizations to help WestGroup explore user interface alternatives for different "workbenches", for different types of users.

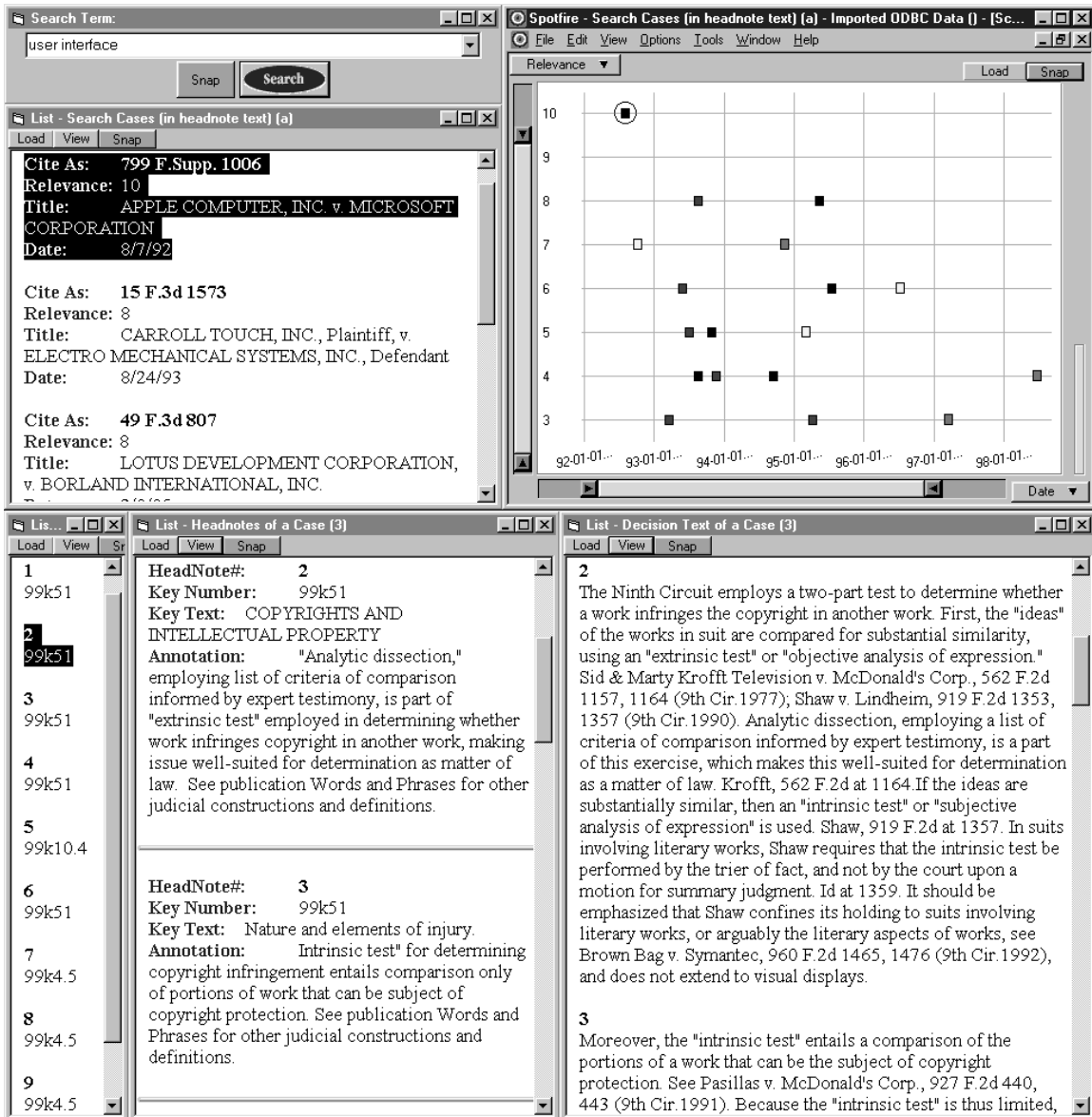


Figure 2-24: Exploring legal information with a searcher's workbench created using Snap-Together Visualization. The Search Term box executes a search query and loads the results into the text view and Spotfire. Selecting a case in Spotfire highlights the title in the text view and loads details of the case into the Case Viewer (bottom). The Case Viewer is composed of 3 views (left to right): case overview, headnotes, and decision text. Scrolling in the headnotes and decision text is synchronized, and selecting in the overview scrolls both.

The STV system (Snap-Together Visualization) can drop or paste any OLE compliant window and extract any fields from records. STV is currently developed on the Windows platform, using ODBC for database access, and COM for inter-process communication. Even if StarDOM is implemented in Java, and can use ODBC to provide database access, there is no possible inter-process communication, without restricting the system to the Windows platform.

2.4.2. Labeling of objects

Excentric labeling is a concept developed by Jean-Daniel Fekete, from the Ecole des Mines de Nantes, and Catherine Plaisant, from the University of Maryland (see [9]). These excentric labels are based on light graphical components used to inform the user of the data that is located under the mouse cursor (see Figure 2-25).

The Excentric Labeling is a dynamic technique used to give information about the neighborhood of the mouse cursor. This technique is useful to display graphically information about a large number of points, without compelling the user to see only a single piece of information at a time, or obscuring the window to such an extent that the labels get overlapped and unusable.

In the current version, all labels in the neighborhood of the cursor are shown when the cursor stays more than one second over an area. Either a circle or a rectangle defines the neighborhood, and lines connect each label to the corresponding object. Different layouts can then be applied over this excentric technique. For example, the non-crossing lines labeling allows a nicer displaying, and an easier tracking of the objects. Actually, an option allows the users to choose which attribute to display in the labels, whereas some others allow the user to control the display of the excentric labels, like the size of the cursor, or its shape.

The starfield representation of StarDOM is a typical example of application for the Excentric Labeling. Static techniques of labeling have been used for a long time in cartography, but in order to have a light graphic display, a dynamic labeling is more appreciated.

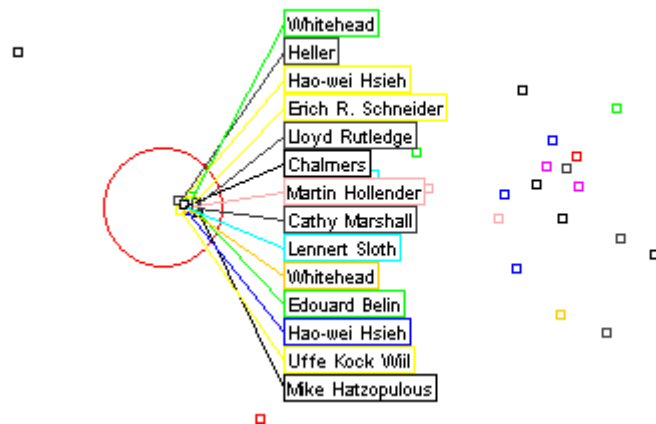


Figure 2-25: Example of use of the Excentric Labeling on a starfield display. All the data located under the circle are displayed, according to a given layout algorithm.

This visualization technique is very useful to track information quickly. It becomes even more powerful when it allows a selection of the data, by freezing the display, like implemented in the current version of StarDOM.

2.4.3. Geometric characteristics of objects

Researchers have proposed many approaches to visualizing arbitrary multidimensional data. The well-known books of Bertin (see [15]) and Tufte (see [16]) include many examples. The article [17], *Automating the Design of Graphical Presentations of Relational Information*, by Jock Mackinlay, from Stanford University, describes a semantic approach to this problem.

The goal of the research described in this paper [17] is to develop an application-independent presentation tool that automatically designs effective graphical presentations, such as bar charts, scatter plots, and connected graphs. Of course, a wide variety of presentations exist from a single set of data, and the tool may propose them all. As a consequence, precise definitions of graphical languages exist, that describe the syntactic and semantic properties of graphical presentations. An example of input of the system can be:

*Present the Price and Mileage relations,
The details about the set of Cars can be omitted.*

Graphic design issues are codified with expressiveness and effectiveness criteria. Expressiveness criteria identify graphical languages that express the desired information. Effectiveness criteria identify which of these graphical languages, in a given situation, is the most effective at exploiting the capabilities of the output medium and the human visual system.

I will not explain the system in detail, and I let the reader look at [17] for further explanations, especially in term of semantics employed. Good results are obtained from the research with two-dimension presentations (such as bar charts, scatter plots, and connected graphs).

A problem raised by Jock Mackinlay in his study concerns the effectiveness. Given two graphical languages that express some information, which language involves a design that specifies the more effective presentation? We have to consider many very specific features. For instance, a color medium makes it possible to use graphical techniques based on the fact that the human visual system is very effective at distinguishing a small amount of distinct colors (see [18]). The difficulty is that there does not exist yet an empirically verified theory of human perceptual capabilities that can be used to prove theorems about the effectiveness of graphical languages.

Figure 2-26 presents a ranking of tasks according to quantitative information, with different degrees of accuracy (Cleveland and McGill).

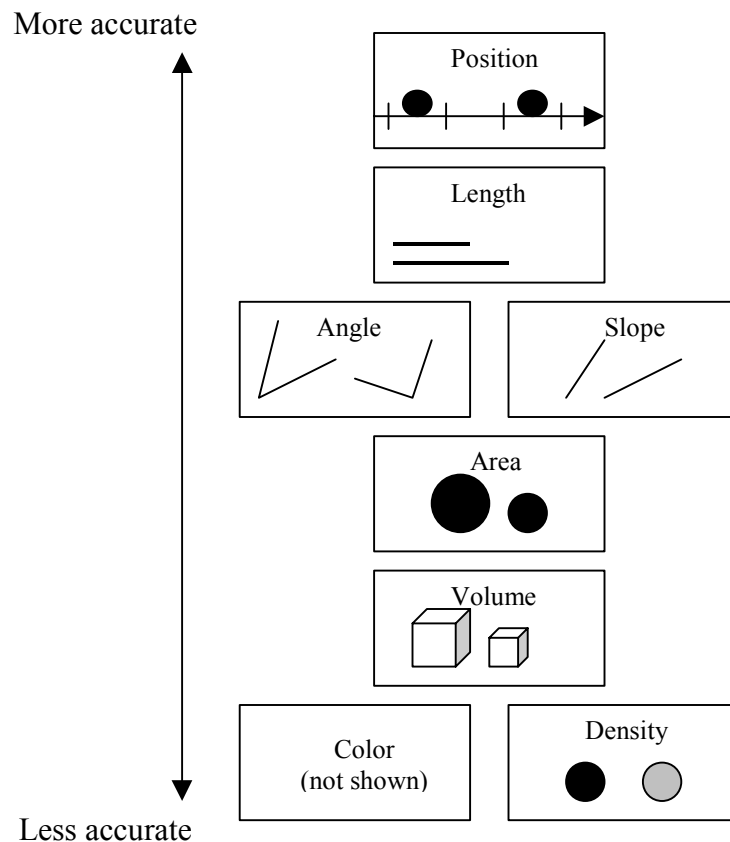


Figure 2-26: Accuracy ranking of quantitative perceptual tasks. Higher tasks are accomplished more accurately than lower tasks. Cleveland and McGill empirically verified the basic properties of this ranking.

Although the ranking in Figure 2-26 can be used to compare alternative graphical languages that encode quantitative information, it does not address the encoding of non-quantitative information, which involves additional perceptual tasks and different task rankings. For example, texture is not mentioned in Figure 2-26, and color, which is at the bottom of the quantitative ranking, is a very effective way of encoding nominal sets. As a consequence, the Cleveland and McGill's ranking has been extended, as shown in Figure 2-27. Although this extension was developed using existing psychophysical results and various analyses of the different perceptual tasks, it has not been empirically verified.

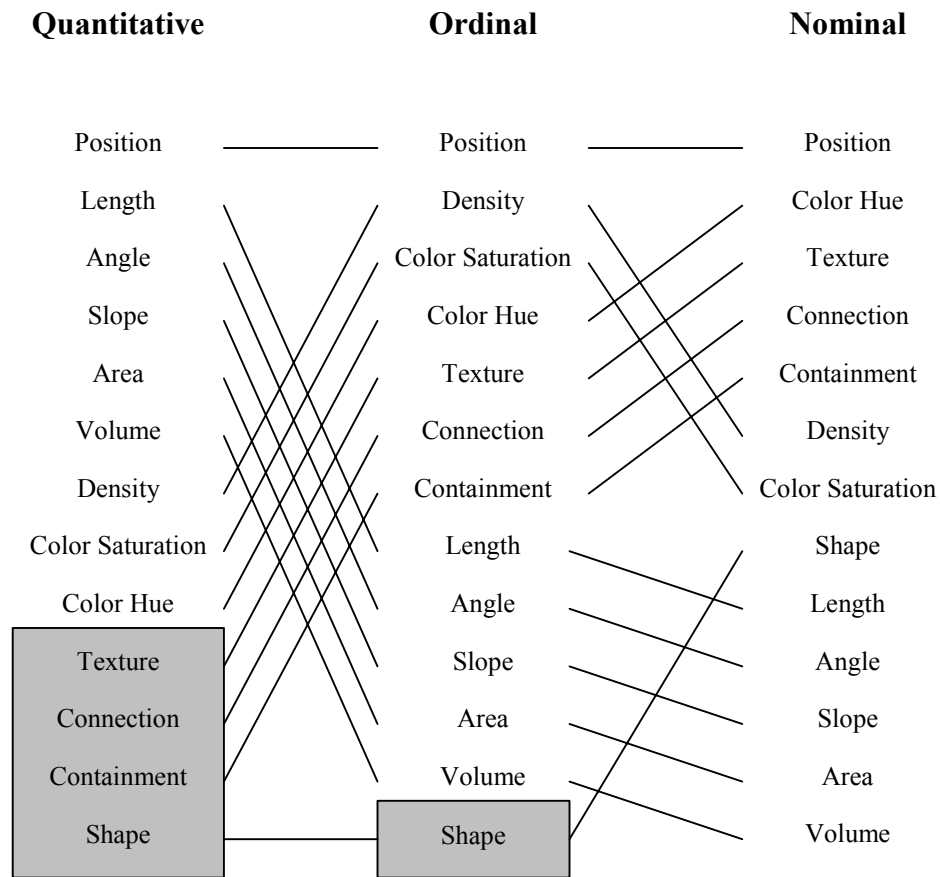


Figure 2-27: Ranking of perceptual tasks. The tasks shown in the gray boxes are not relevant to these types of data.

These different rankings can be taken into account in the case of graphical languages, but in most of the case, for many graphical data visualization, the choice between perceptual tasks is let to the users, who may not create the most effective presentation of his data. This is also the case of StarDOM, even if the number of possibilities of perceptual tasks is very low in the current version. This topic is discussed later, in the “StarDOM structure” and in the “possible evolutions” parts.

3. Starfield Dynamic Object Miner (StarDOM), software and user interface description

This part of the report explains the structure of the current JAVA implementation of StarDOM. It explains my contribution, during the six months of summer project. The first implementation of this prototype was developed in the LAMP lab, before my arrival, and was called Selector. I worked three months on the implementation with Felix Sudhenko, a Computer Science graduate student of the University of Maryland, and alone during the last three months. Catherine Plaisant (HCIL) and David Doermann (LAMP) supervised our work.

3.1. Introduction

The first goal of The Starfield Dynamic Object Miner (StarDOM) development is the achievement of a standard Dynamic Query interface in Java, for use in variety of web applications. The existing commercial product Spotfire, that has a lot more features, works only on PC, and only on the Windows platform. The second goal of StarDOM is its use for the specific needs of NASA, which consists of the development of a Query Refinement phase for searching EOSDIS data. This implementation of the Query Refinement phase allows a graphical visualization of NASA EOSDIS data, using a starfield display, and friendly graphical widgets to allow users to submit queries.

Though, StarDOM can still be useful as a graphical database visualization tool. Most of the features of StarDOM are of course already implemented in Spotfire, except the multi-valued data, and the adaptability to other platforms than Windows. The very first implementation of StarDOM, originally called “Selector”, was developed in the LAMP lab to visualize a database of videos. The new StarDOM system may be used by LAMP lab for this same purpose, even if the new features are not used.

3.2. Description of the interface

3.2.1. Overall

The next parts of this report will describe the user interface of StarDOM. As you can see on Figure 3-1 and Figure 3-2, the interface is composed of several windows. The main one is called “starfield display”, and presents the projection of two attributes of the database along the coordinates of a two-dimensional graph (e.g. the attributes title and actor on Figure 3-1). Thanks to the range sliders located on the left and on the bottom of the graph, the users can zoom in or zoom out to see areas of interest. Excentric labeling (visible on Figure 3-3 and Figure 3-6) allows the user to identify the objects under the cursor, according to one of the attributes of the database. An option allows users to create other starfield displays. Thus, it is possible to have more than one visualization. Selecting data or filtering data on a starfield will automatically update the other starfields as well.

The “filtering panel”, located on the right of starfield display, contains a list of all attributes of the database. It is then possible to filter data according to any attribute, either through a range slider, or through checkboxes. This panel offers more features as well. It is possible to link the color feature to an attribute, or the size feature. The shape feature is also available in the case of a limited number of values for a given attribute (10 in the current version). As a result, the starfield display will be updated, and the objects will be displayed with the correct size, shape and color. When checkboxes are used, options are provided, to allow the users either to select all the objects, to unselect them, or to toggle them.

The information panel is usually located in the bottom right corner, and gives details about the currently selected object. Selection is possible on the starfield display by clicking on the object, or dragging the mouse if the user wants to select more than one object. When the user clicks, if there is more than a single object under the cursor, the excentric labeling system is used to choose which object to really select. When more than one object are selected, the last selected one is displayed in the information panel.

The last information area, located at the very bottom of the main window, gives indication about the number of objects still visible (this is the blue bar). The green bar indicates how many objects are selected. Let’s call this part of the system the “state bar”.

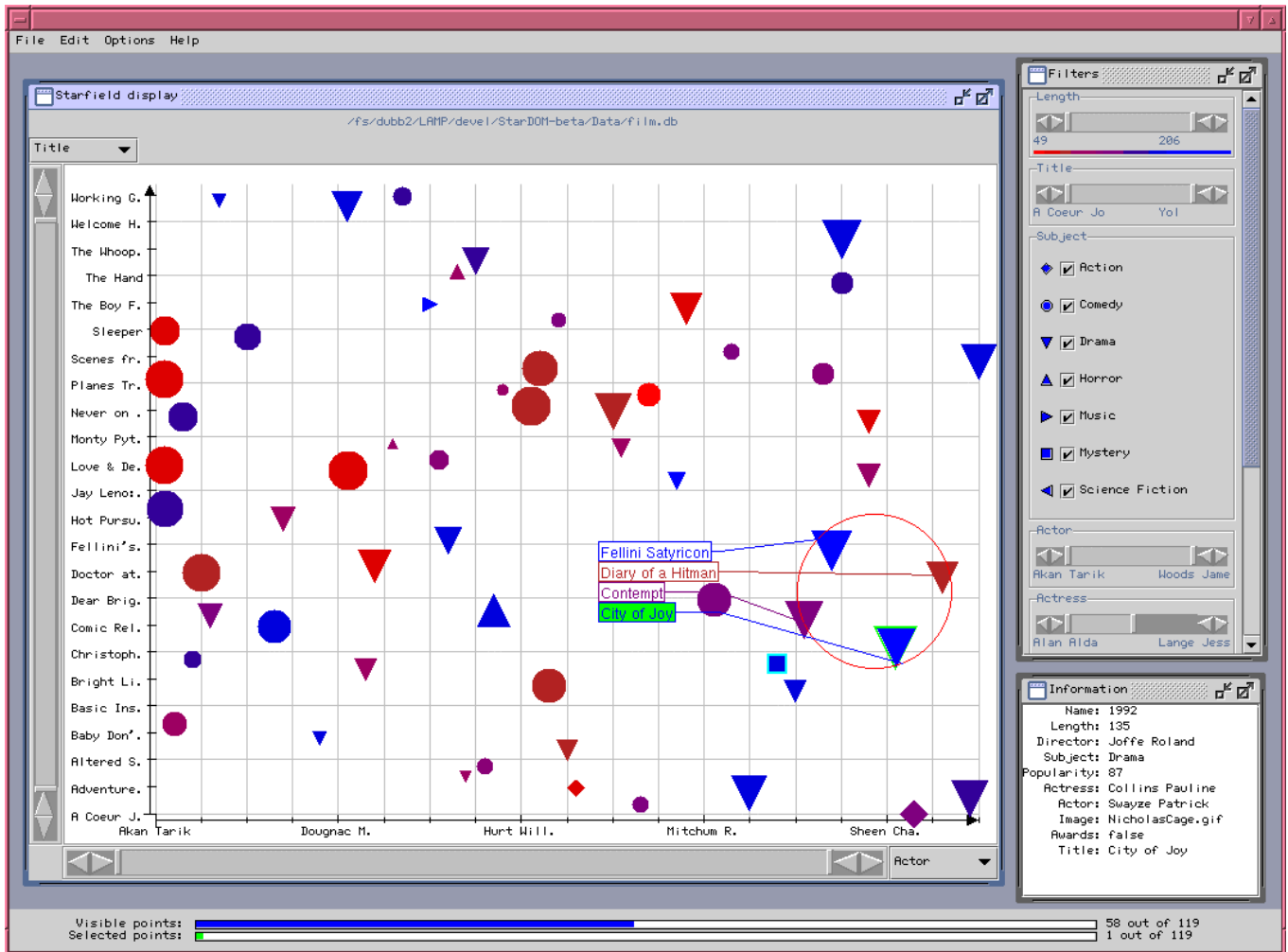


Figure 3-1: Example of use of StarDOM in the case of a database of videos. The different attributes are Title of the video, Name of the main actor, Main actress, Length of the video, Subject (e.g. action, comedy), and Popularity of the video. The color is used to spot the length of the video: blue corresponds to long videos. The shape differentiates the kind of videos, and the size measures their popularity. For instance, we see that the City of Joy is a famous long drama only through its display. The information panel gives more detail about this video.

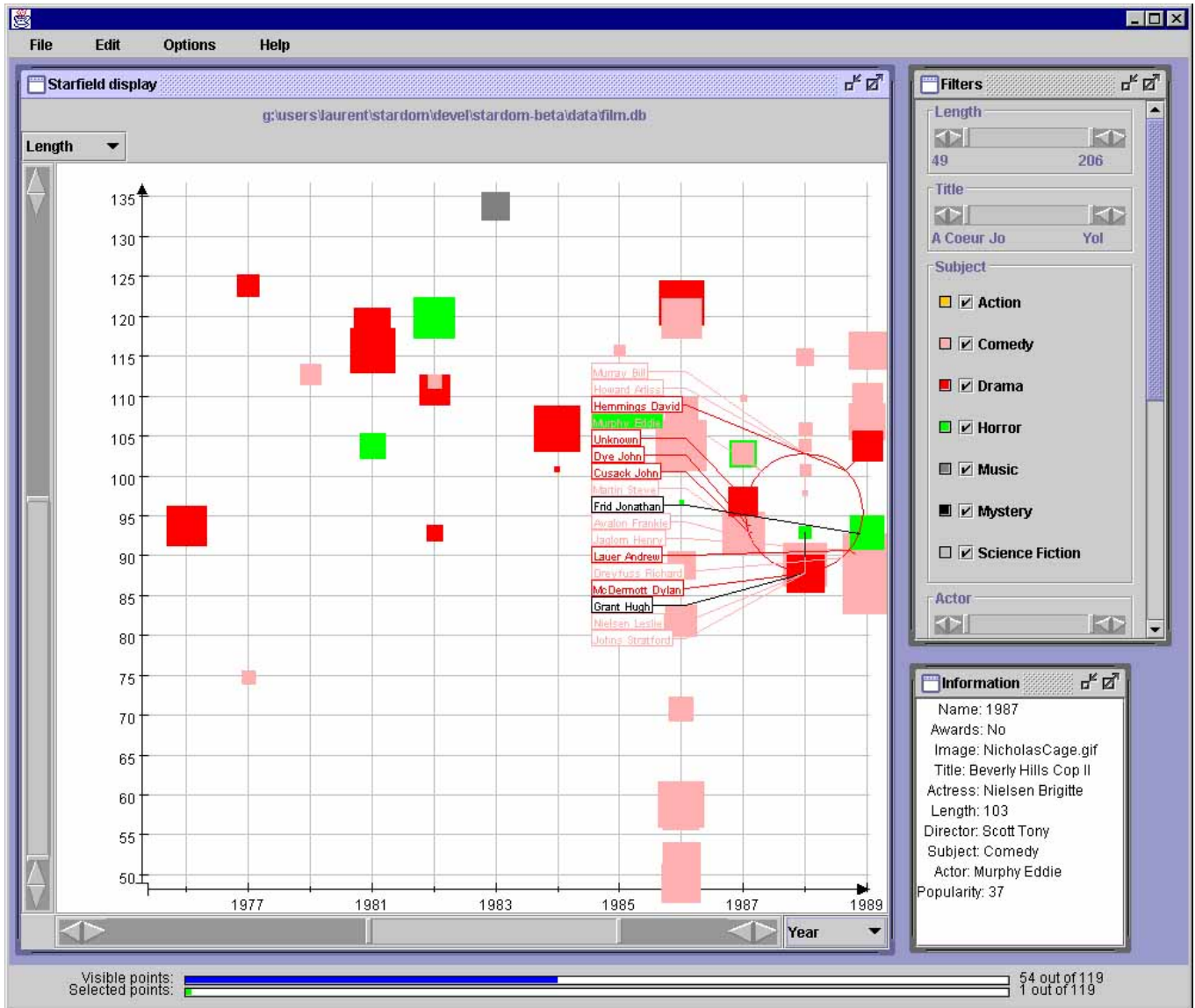


Figure 3-2: Second example of use on a Windows platform this time, with the same video database as in Figure 3-1. Here, the color is used to differentiate the types of videos. The size measures their popularity. The axes represent the year of realization (X-axis), and the length of the video (Y-axis). The information panel gives more detail about the selected video, which is Beverly Hills Cop II. For this example, the name of the main actor has been chosen for the excentric labeling, instead of the title of the video.

3.2.2. The challenge of the NASA data

The NASA data to be used in the user interface of StarDOM are objects regrouping a few numbers of attributes (8 in our example data from the GCMD).

Of course, it is possible to represent single-valued data of different kind on the interface (e.g. integers, strings, float), but most of the attributes are string multi-valued data. Multi-valued data means that an object can have more than a single value for one of its attributes. This can be represented by two distinct tables on a database schema. Only the title attribute identifies each data, and cannot be multi-valued. Multi-valued data are represented on starfield displays by using lines to link all the possible values of the object.

On the next screenshots (see Figure 3-3 to Figure 3-6), there are two range sliders for the same attribute called “temporal coverage”. This is the case for every range attribute. Thus, it is possible to filter the objects according to their beginning dates, or their end dates. Range data is another particular data attribute that is displayed using two shapes at the beginning and at the end of the attribute, linked together by a thick bar.

The last special data attribute concerns the spatial coverage. Most of the time, an object covers a very specific area on earth. As a consequence, a two-dimensional attribute representing a box on a world map is used. No filtering is allowed in the filtering panel for this kind of attribute, because the filtering is much easier through the starfield display.

The possible attributes of the datasets of NASA EOSDIS data are:

- the title of the dataset (“classic” data, i.e., single-valued data),
- the detailed space extent (two-dimensional range data, represented with a map of the world),
- the detailed temporal interval (range data),
- the sensors used to generate the dataset (one or more values for a data), e.g. cameras, barometers,
- the sources of the dataset, i.e., the platform on which the sensor resides (one or more values for a data), e.g. Skylab, Landsat,
- the project with which the platform is associated (one or more values for a data), e.g. EOSDIS, ERICA,
- the data archive center where the data is stored (one or more values for a data), e.g. EROS, EOAA.

3.2.3. Scenarios of use

The two examples given in this part use a NASA database with a sample of 200 points for a clear presentation. It is possible to use up to 500 datasets without too many problems in term of quick response, but then, the starfield displays become very cluttered and difficult to analyze.

3.2.3.1. Simple case example

This first example presented on Figure 3-3 describes a user interested only in data collected through cameras. Thus, he drags the range slider corresponding to the sensors attribute until the only value “cameras” remains. He could also have done this using checkboxes. The result is displayed in a short time, and the state bar indicates that 5 datasets match his query.

In order to have a quick view of the recent datasets (i.e., datasets representing recent NASA EOSDIS projects), the user chooses to map the size and the color property to the last temporal coverage range slider. Thus, EOSDIS projects still in use, or projects finished for a short time will be displayed as big blue squares.

The user can then go through the different datasets, either using the excentric labeling to have quick clear information, either using the information panel to have detailed indications about a specific dataset. The blue color of the points on the starfield display of Figure 3-3 indicates “deselected” points. If a dataset is selected, a green border appears around it, and when it becomes deselected, a blue border is drawn. This is a feature used to make the user remembers which datasets he has already looked at.

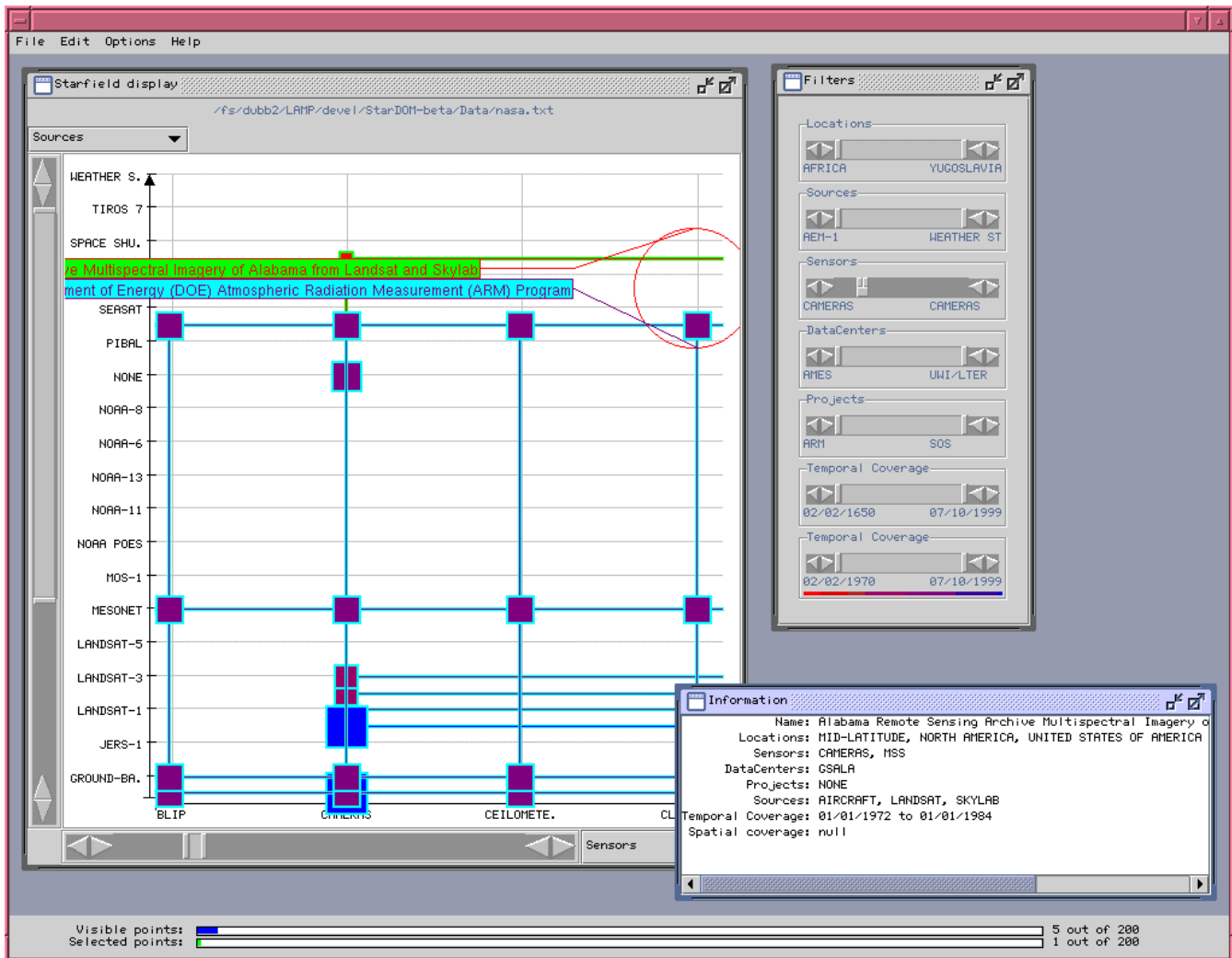


Figure 3-3: Example of use of StarDOM. The data represented on the starfield display are multi-valued on both axes (sensors on the X-axis, and sources on the Y-axis). This is the reason why a single dataset is represented by a group of squares linked together by horizontal and vertical lines.

3.2.3.2. Second example

This example is a little more complex than the first one described above. In this case, the user is a scientist interested by the more recent data captured by the different Landsat satellites. He chose to display the datasets with two windows, one representing the spatial coverage of the data, with a map of the world beneath it, and one representing the temporal coverage for every dataset.

First, the scientist is interested mostly in recent data. He will then choose to map the color and size property to the end date of the temporal coverage. The more blue the data displays are (and the bigger too), the more recent the information is. The resulting display is presented on Figure 3-4.

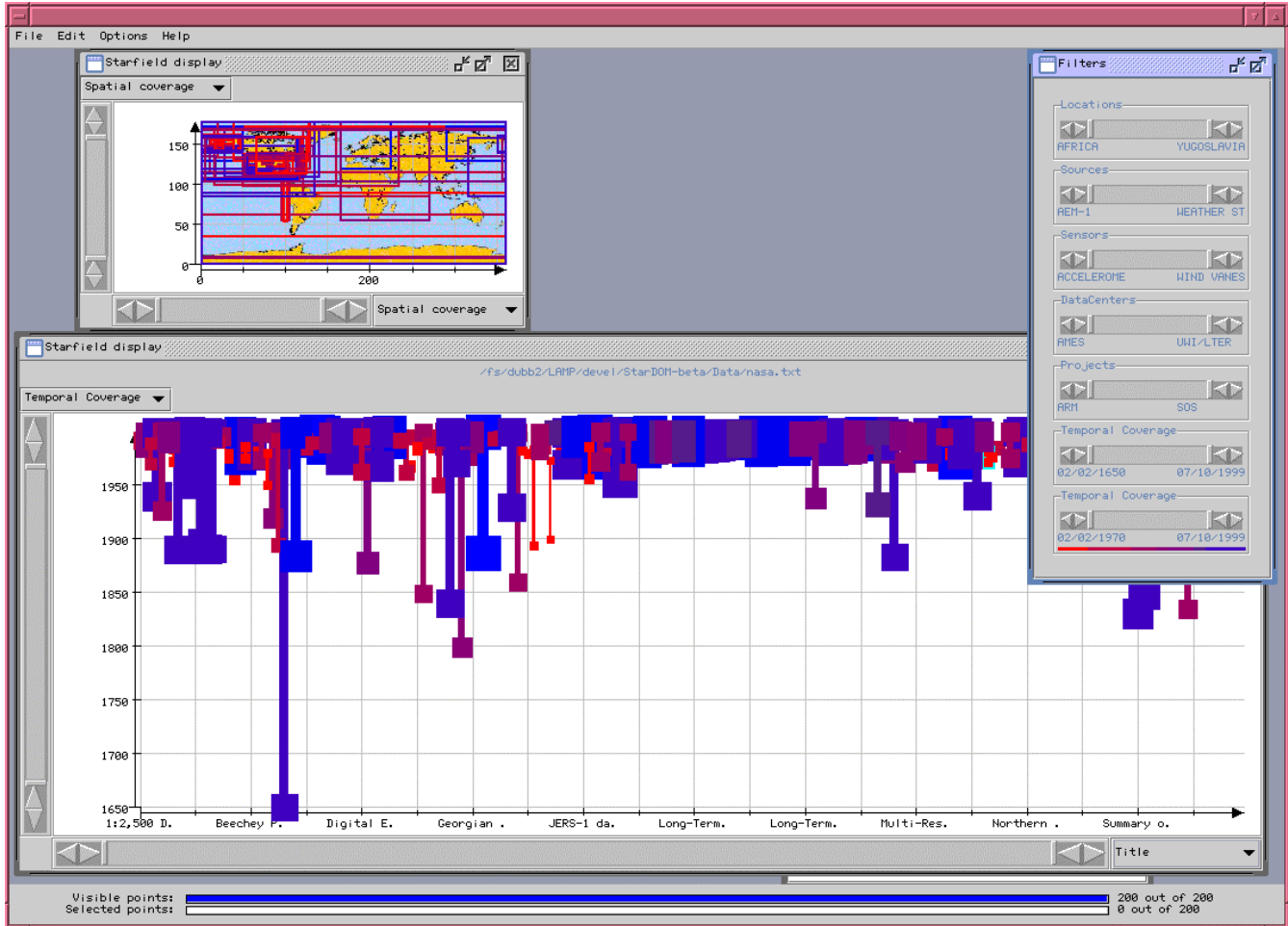


Figure 3-4: Second example of use. Here, two windows are used to see the possible interactions. A tool like StarDOM is also useful to spot exceptions in the database. Here for instance, the user can notice an exceptional data that goes far in the past (down to 1650). He can then select it easily to have more information about it.

As the scientist is only interested in information captured by Landsat, he will select the “checkbox” option for the attribute “sources” by right clicking on the attribute position in the filtering panel. He will choose the option “deselect all” in the same popup menu. Then, he will go through the different values in order to select only the Landsat satellites (Landsat, Landsat-1, Landsat-2, Landsat-3, Landsat-4 and Landsat-5). The result is presented on Figure 3-5. The scientist notices that 24 datasets are available in this sample, as displayed in the state bar. He also notices that the oldest datasets are from 1972, and most of the available datasets cover the United States and Alaska.

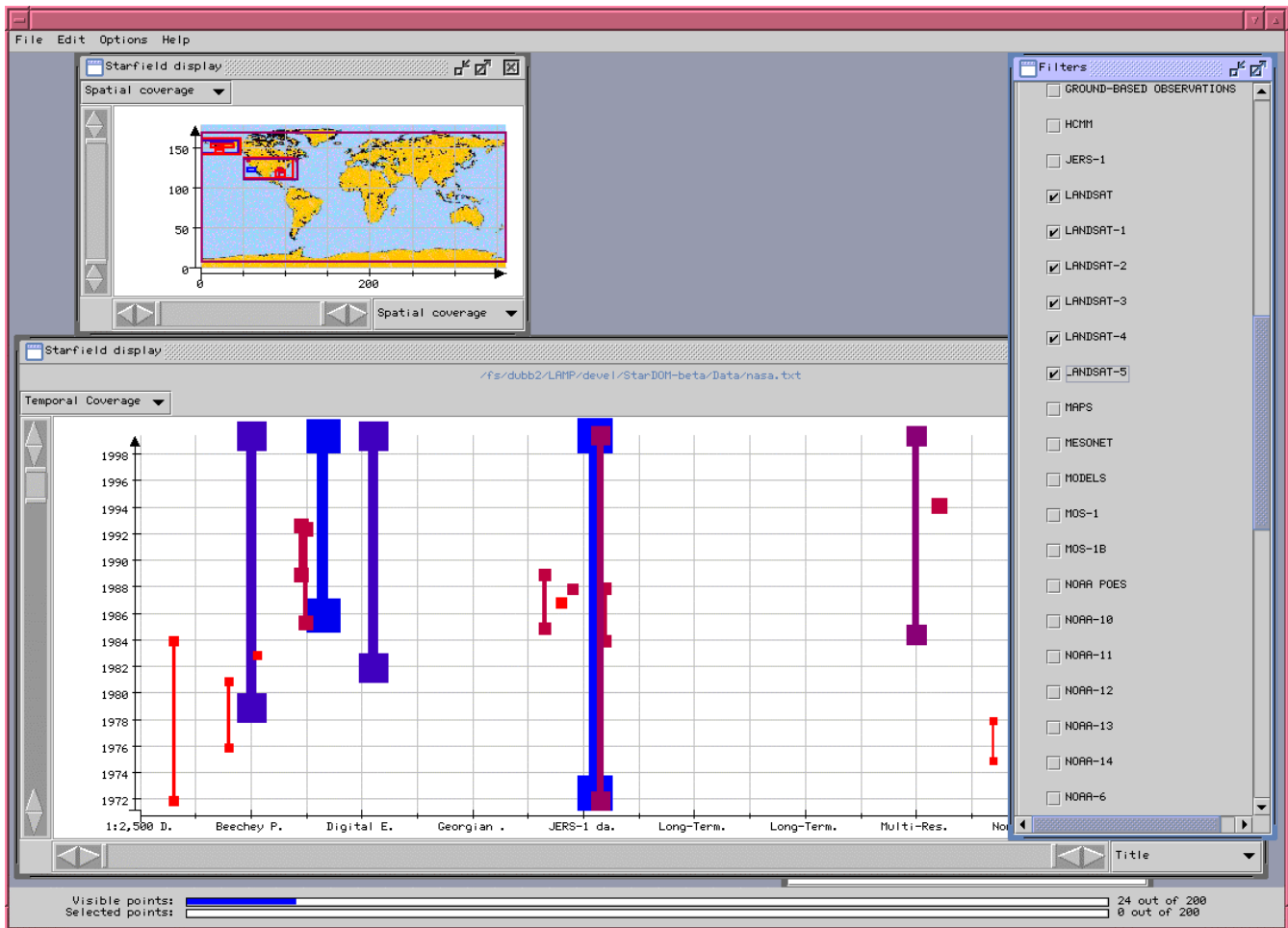


Figure 3-5: This screenshot shows that 24 datasets of the sample match the query. They represent data collected through Landsat satellites.

Now, let's imagine that our scientist is more interested in datasets concerning United States only, Alaska excluded. He will use the small window displaying the spatial coverage of the data, and drag the sliders to consider only this part of the world. Only 11 datasets match this new query. The scientist can then explore the data more in detail by using the excentric labeling, and the information panel (see Figure 3-6).

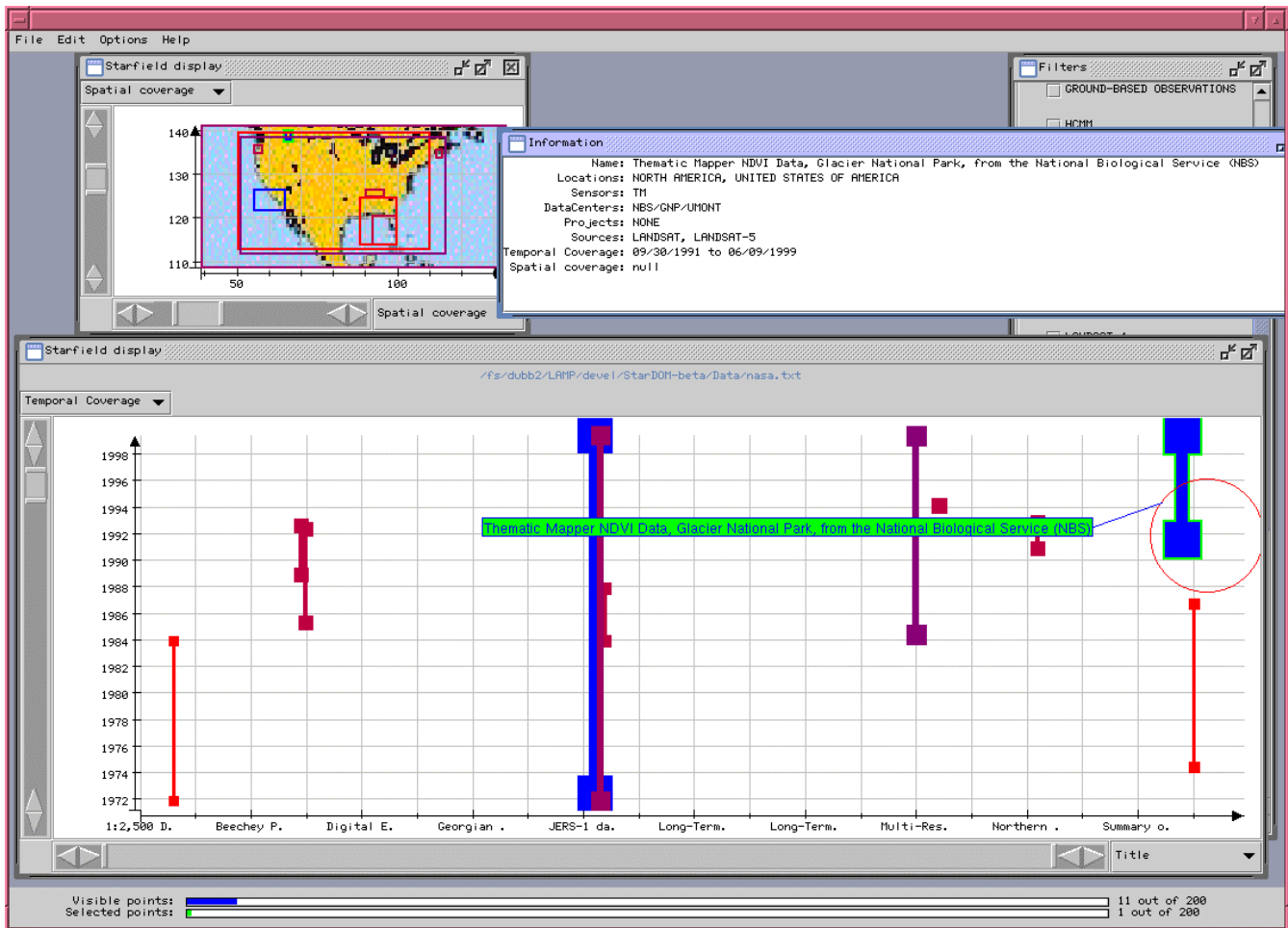


Figure 3-6: In this last step, the scientist zoomed on North America. The scientist can now go through each dataset, and eventually get the real data from the corresponding NASA data center.

3.3. Internal representation, or back-end structure

The current data structure of StarDOM is articulated on a table loaded in memory, and containing all the elements of the database.

3.3.1. Structure supporting the database in memory

Every single DataPoint of the structure contains a certain number of attributes. Each of these attributes gives the path to a DataCapsule that corresponds to the real data element. This system is represented on Figure 3-7 in UML.

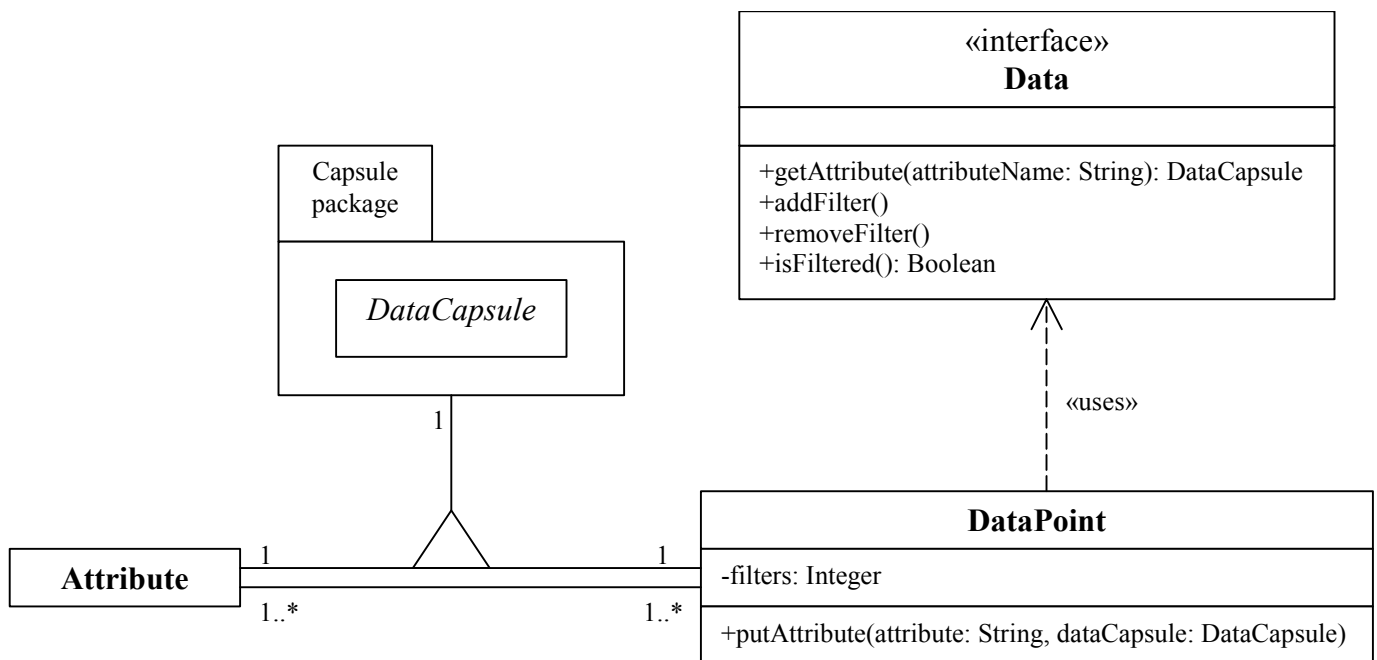


Figure 3-7: UML representation of the structure supporting the database in memory.

A more precise idea of the implementation can be obtained through the schema of Figure 3-8.

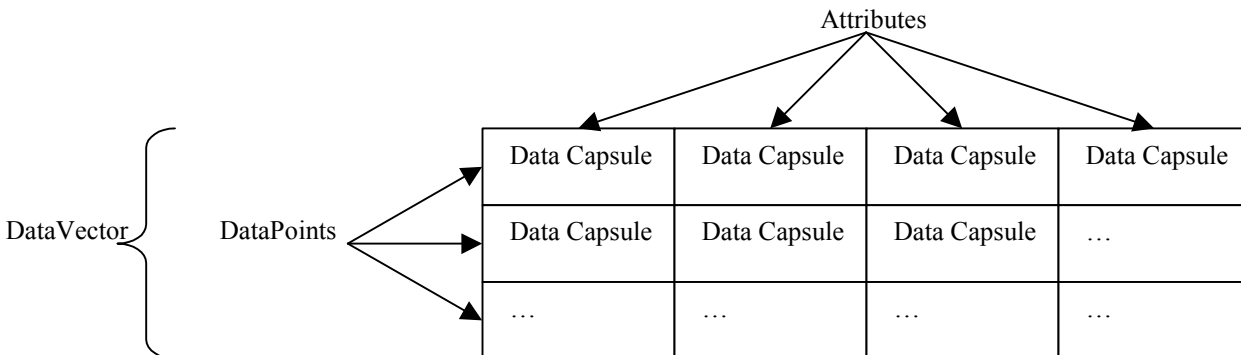


Figure 3-8: Every piece of data is represented through a data capsule, and the aggregations of data capsules give complete data. The vector DataVector stores all the datapoints of the database.

A DataPoint corresponds to a precise Data, and has a certain number of DataCapsules, accessed through the attributes values. The dimension of the data is equal to the number of attributes it has.

3.3.2. The capsules

According to the type of the data, different kinds of capsules are used. When the database is first launched, if integers are found, an IntegerCapsule is created, for float, it would be FloatCapsule, for double, DoubleCapsule, for string, StringCapsule, etc. All the different kinds of capsules provide basic elementary functions, like adding, retrieving, multiplying, and dividing.

Figure 3-9 gives a UML representation of the capsule system used in the prototype.

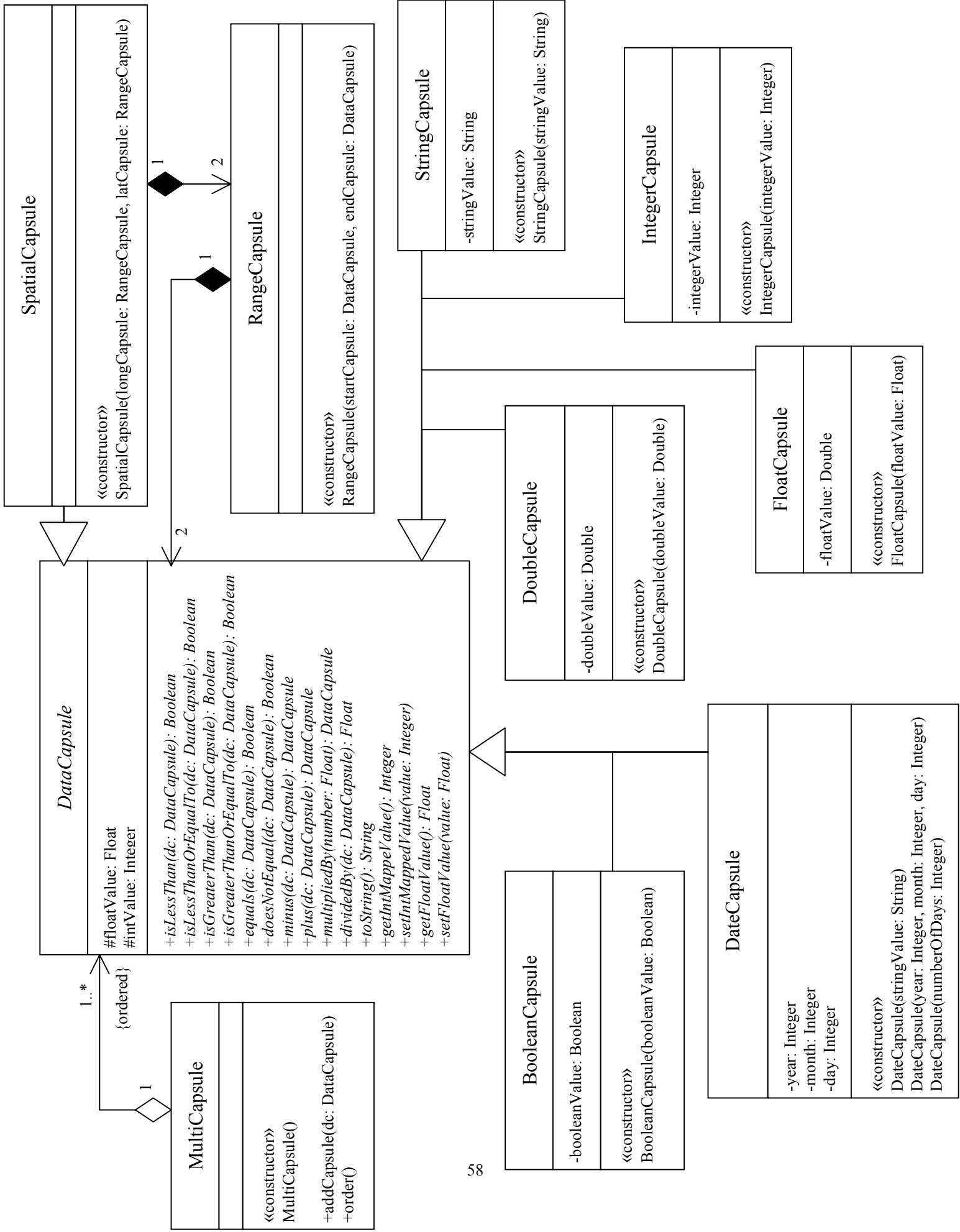


Figure 3-9: The capsule structure is a representation of the database in memory.

Two Java hashtables are used to link the “calculation” structure with the “display” structure (see Figure 3-10). Thus, it is possible to get the DisplayPoint from the DataPoint, and the DataPoint from the DisplayPoint. These close links are useful to fix values for the display of the data, knowing that those values are located in the LitePoint (or DisplayPoint) class.

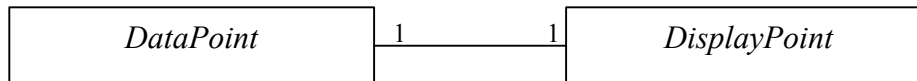


Figure 3-10: For a given starfield representation, a DataPoint, corresponding to the back-end structure, is linked to a DisplayPoint, for the front-end structure. All these relations are stored in a Java hashtable. Of course, there are as many hashtables as starfield windows.

One of the hashtables links DataPoints to DisplayPoints, whereas the other links DisplayPoints to DataPoints. The DisplayPoint structure will be explained in details below.

3.3.3. The dynamic querying structure supported by the starfield range sliders

Of course, one of the most important features of StarDOM is that it offers dynamic querying. As a consequence, every query has to be computed rapidly (usually 100 milliseconds), in order not to bother the user. Different time experiments (see further down) on the prototype in development showed that the time-consuming process was not the amount of time needed to answer a request, but the time needed to update the starfield display. Egemin Tanin and Richard Beigel also noticed this during their experiments (see [13], and their previous work [14]).

These time tests showed that the really time-consuming feature was the calculation of the coordinates of every point on the starfield display, when one of the two corresponding range sliders is modified. Of course, the modification the other range sliders, alphanumeric sliders, or check boxes located on the filtering panel, has no influence, because there is no need to recalculate the coordinates of every visible points, but only points that are changed on the screen.

Thus, the structure is implemented in this direction. More than allowing very fast determination of queries, it has to provide and maintain a specific structure of the visible data on the starfield display, i.e.,. to keep track of the results of the previous queries. The recalculation of the coordinates of points that have been filtered can become a burden.

When the database is launched into the system, as many Axis classes as attributes are created. The Java class Axis contains all the necessary information to represent an attribute (minimum and maximum values taken by the whole data sets for a given attribute is among this information). The element dataArray, on the Figure 3-11, is a Vector containing all the dataPoints. All the elements of the database are contained in this single vector. The variables nonMappedMin, nonMappedMax, and nonMappedRange are three variables used for displaying correctly the legends of the axis, and are calculated in the function nonMappedValues.

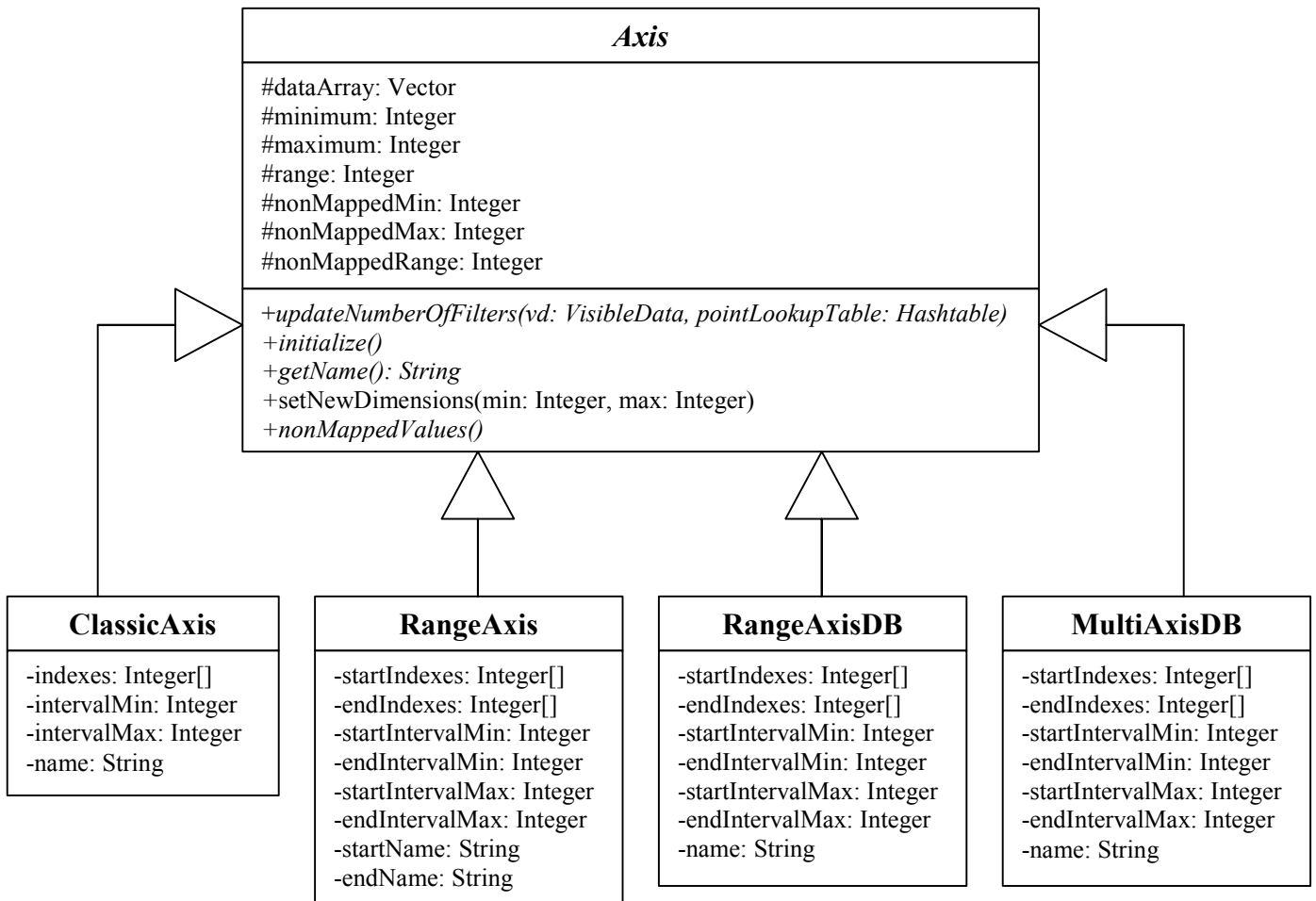


Figure 3-11: The different Axis classes. They keep track of the minimum, maximum, and range values for every attribute of the database. The explanation for the “intervalMin” and “intervalMax” values is given in the next part.

The different variables and functions in the different Axis classes seem to be identical, but they have a different implementation from one class to another. These different classes Axis are used both by the starfield display, and by the filtering panel. The only difference is that this is not sufficient in the case of the use of checkboxes in the filtering panel.

3.3.3.1. ClassicAxis class

ClassicAxis has a variable called indexes[]. This is a table of integers representing indexes to the table of DataPoints. When a new database is launched, as many integer tables as attributes are created, and ordered according to the attribute. This requires a certain amount of time at the beginning to launch the sorting algorithm, but this allows the system to be fast afterwards. It is possible to create them as needed, i.e., when users select a new axis, but this implies many identical recalculations.

The index table is used in combination with intervalMin and intervalMax variables. These two values indicate begin and end values of the index tables, in order to delimit the points to take into account. Let's take look at an example in Figure 3-12.

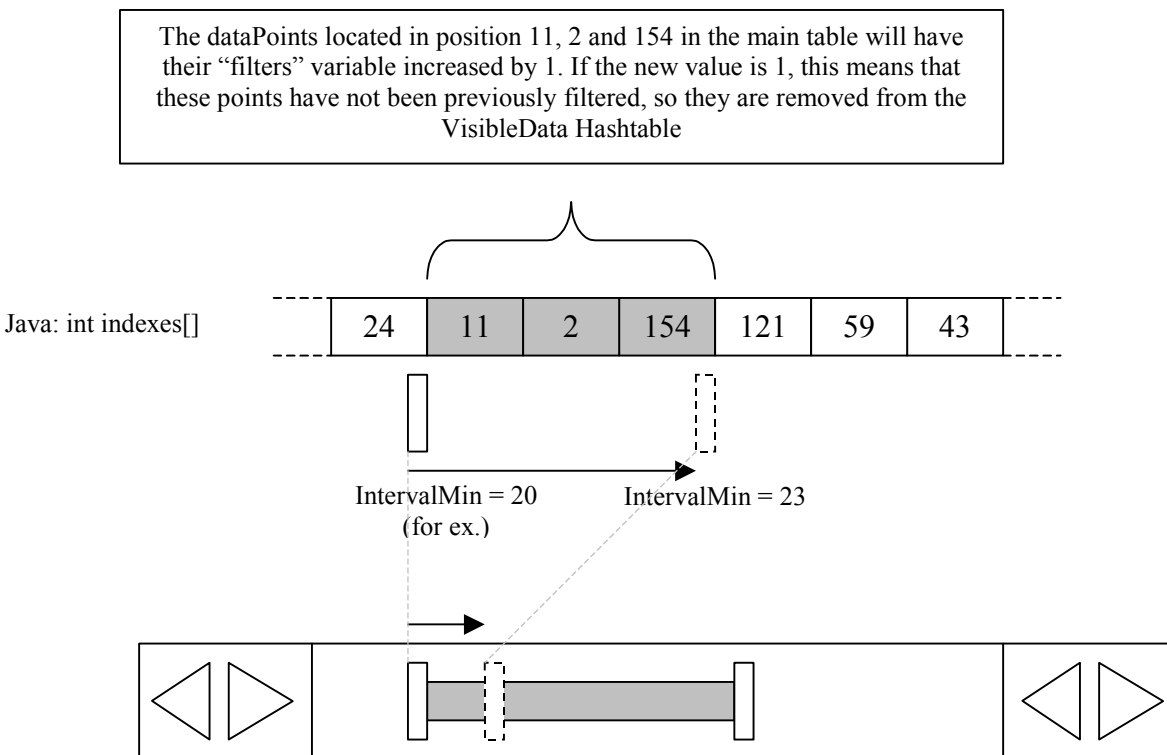


Figure 3-12: Example of update of the internal structure of an Axis when a range slider is modified. When the range slider is manipulated by the user, in a given direction (the left thumb to the right on the example), the intervalMin and intervalMax values are modified (here, intervalMin would be increased from 20 to 23). This structure is useful when a lot of points are considered on the screen. It is a

waste of time to calculate the values of the displayPoints that would be located outside of the screen.

3.3.3.2. Calculation of the answer to a query, represented by the visible data structure

How does the calculation of the visible data work?

The visible data structure implemented is inspired from the structure called “linked array”. A study about this structure has been done in [23], as explained in a previous part of this report. The detailed explanations of its workings are given further down.

The DataPoint class has a “filters” variable, that is a simple integer. If a specific DataPoint get filtered, either through checkboxes and range sliders on the filtering panel, either through range sliders on the axis, this variable will be increased by one. If the value of “integers” is 2, this means that the corresponding DataPoint is filtered through two widgets. If the value is 0, then the DataPoint is not filtered, and has to be displayed on the screen. All the visible points are stored in a Java Hashtable (that links DataPoints to DisplayPoints) located in VisibleData class.

If we come back to our example on Figure 3-12, a small move of the left thumb to the right direction will increase the intervalMin variable value. The DataPoints concerned will have their filters variable decreased by one. If its value becomes equal to 0, then the DataPoint and its corresponding LiteDisplay is added to the VisibleData structure. In the same way, if its value changes from 0 to 1, the DataPoint and its DisplayPoint will be removed from the structure. When the new values of the DisplayPoint have to be calculated (on the horizontal and vertical axis), all the points located in VisibleData are considered. All the points visible on the screen are contained in the table, and the table contains only these points.

Thus, the display will be much faster when only 500 points (from a database of 5000 for instance) are visible, instead of 3000 points. Of course, this may not be really satisfying if a great number of points remain on the screen, as shown on the time tests further in the report. The best solution at that moment is to display only a well-chosen sample of data, and get rid of the rest. The idea is that the user won’t be able to select or to visualize any data, because there may be too many points. A flag (combined with the number of visible data and the number of data to reach) can be used to indicate to the user that all the data elements are visible, or only a sample of the data.

The function updateNumberOfFilters is launched when the user changes one of the axes; i.e., the display points of the data will be projected according to another attribute along one of the axis. Indeed, the range slider can have previously been adapted to a specific representation, allowing a filtering of a certain number of points. But when the new axis is set, the range sliders comes back to their original position, with the lowest possible value for the minimum, and the largest possible value for the maximum. Thus, the points

that were previously filtered become visible once again, and the function `updateNumberOfFilters` is in charge of this computation.

3.3.3.3. Other axis classes

The other axes classes function similarly to the `ClassicAxis`. `ClassicAxis` has been designed to deal with data having a single value on a precise attribute. `RangeAxis` deals with range values, but range created by the user, in the properties panel, and not range loaded from the database.

What is the difference between those two range axes?

Well, considering a range axis created by the user from two existing attributes of the database means that the user wants to have a supplementary attribute to visualize two attributes at the same time thanks to ranges. In this case, there is no creation of a new indexes table, because the data can be filtered according to the existing “beginning” and “ending” attributes of the range. The axis class used in this case is `RangeAxis`, which uses the indexes tables of the two previously defined axes (`startIndexes` and `endIndexes` variables).

But in the case of range axis already existing in the database, the problem is different. This means that the information represented is best visualized through ranges (like periods of time), and there is no need to use two separate attribute values for begin and end.

In these two range cases, four different possibilities are considered. As an example, let’s take the horizontal axis.

- a) When the left thumb of the range slider is moved in the left direction, then certain data points indexed in the end table are “de-filtered”.
- b) When the left thumb of the range slider is moved in the right direction, then certain data points indexed in the end table are “filtered”.
- c) When the right thumb of the range slider is moved in the left direction, then certain data points indexed in the start table are “filtered”.
- d) When the right thumb of the range slider is moved in the right direction, then certain data points indexed in the start table are “de-filtered”.

Multi-valued attributes are treated in the same way. There are two indexes tables, the first one representing the first value for each multi-valued data, and the other one for the last value. The relations with the `visibleData` structure are similar to the previous case of range attributes.

3.3.4. The dynamic query structure supported by the non-starfield widgets

The filtering panel is a panel containing all the attributes of the database. For each attribute, a new filter is created, and placed on the right panel. A range slider represents

this filter first, but it can be modified to checkboxes filter, at the condition that there are not more than a given number of possible values (10 for instance).

As explained previously, an index table is used in each “RangeSliderFilter” class to keep track of the points to consider. Range sliders in the filtering panel work in the same way as in the axes.

3.3.4.1. Alphanumeric sliders structure

As approached in the article [13], different implementing systems can be used to deal with different kinds of filtering widgets. In the specific case of textual values, either alphanumeric sliders or check boxes can be employed. Two different possibilities of implementation exist for the alphanumeric sliders, and the choice depends on the kind of data used.

The first solution consists of considering the alphanumeric slider like any other range sliders, with an ordered index table, and two values indicating the position of the sliders in the table (see the previously explained structure). In this case, textual information is considered like continuous information, every textual value can be mapped to an integer one.

In the second possibility, the different textual values of an attribute can be considered not like a set of infinite data that increases with the number of points into account, but like a finite set of values. Of course, every data can be linked either to a single textual value, either to more than one in the case of multi-valued attributes. In this case, it is better to implement a structure like the one used for the checkboxes, i.e., a table linking every check box to the corresponding list of data. This structure is not larger in memory than the ordered table used for range sliders.

After considering the data used in NASA EOSDIS project, the choice of implementing the second structure was made. This is especially useful during the change of the widget, from a range slider to check boxes, because the same structure was used, and there is no need to re-calculate the structure. Of course, it is possible, even if not yet implemented, to use either one or the other structure, according to the number of textual values for a specific attribute.

Here are two examples of NASA EOSDIS data attributes that confirm the use of a finite number of textual values:

Sources:

Aircraft, ground stations, landsat, NOAA-6, NOAA-7, NOAA-8, NOAA-9, none, weather, and a few others...

Sensors:

Avhrr, cameras, MSS, none, and a few others...

A few other attributes (e.g. *Locations*) are in the same case, but on the opposite, attributes like *Title* are most likely different for every data.

3.3.4.2. Bounding rectangles of the attributes

An interesting point that has been implemented in Spotfire concerns the feedback provided by the widgets themselves, i.e., checkboxes and range sliders. When a request is submitted, and when some points are filtered and disappear from the starfield, the different existing widgets are updated to display the bounding rectangle of the remaining data, with yellow-colored bands. The screenshot of Figure 2-6 gives an example of this in Spotfire, in the case where the bounding rectangle is at its maximum. Figure 3-13 gives another example of this feature.

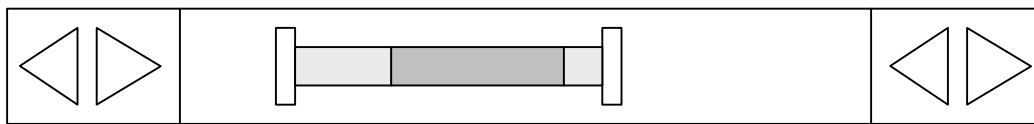


Figure 3-13: The darkest zone on the slider indicates where the still visible data points are located, according to this dimension. This informs the user that moving the thumbs of this slider would not be efficient before entering the dark zone.

This interesting feedback is valid not only on the non-starfield widgets, but also on the two main range sliders as well.

Egemin Tanin and Richard Beigel proposed a structure to calculate the bounding rectangle for every attribute in a constant time (independent of the number of points), see [13] and [14], and the presentation of these articles in the previous bibliography.

Another lighter solution to compute the bounding rectangles consists in keeping track of the borders of the box, for each attribute. As soon as a data becomes filtered, and is removed from these borders, the algorithm computes the closest visible point for the given widget, and updates its display. The implementation of this feature is not yet finished in StarDOM.

If we keep considering a single axis, it is thus possible to represent the data in memory. The problem of two-dimensional data (spatial coverage case) will be later.

3.4.1.3. How the filtering panel deals with range and multi-valued attribute data?

Let's now consider the filtering panel that contains all the widgets to filter the attributes of the database. Range data represent a group of elements (dates or any other type), situated between two extreme values. The best way to deal with them is to consider two possible filters, one according to the start value, the other according to the end value.

Now, filtering with multi-valued data is more complicate. Figure 3-15 proposes a simple example.

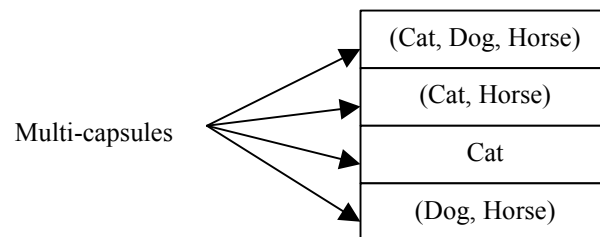


Figure 3-15: Example of multi-capsules, where four data records have different possible values on a given attribute.

In Figure 3-15, if the “Cat” value becomes filtered, then only the third element will be filtered, and not the first two, because they have other possible values. The choice of an “OR” query has been done, instead of an “AND” query, because it is always better to filter less than too much. In this case, each multi-capsule has an additional variable, called multiFilter. Its function is identical to the function of the variable filter, located in the data class. In the example of Figure 3-15, for the first multi-capsules, there are three possible values, ‘Cat’, ‘Dog’, or ‘Horse’. The variable multiFilter, which is first equal to 0, increases each time one of these three values becomes filtered by any widget. When the value of multiFilter becomes equal to the number of possible values (i.e., 3 in our example), then the multi-capsule becomes really filtered, and the filter value of the corresponding data is increased. The same thing happened on the other way. When one of the three values becomes unfiltered again, the value of multiFilter passes from 3 to 2, and the value of filter is decreased.

Two ways of implementing alphanumeric sliders were described previously, according to the kind of attribute (with a finite or an infinite set of textual values). Of course, these two ways remain possible in the case of multi-valued attribute data, with the use of the multiFilters variable.

3.4.2. Problem of the Spatial Coverage values

An important problem remains, linked with two-dimensional data. As long as we keep thinking in one dimension, the current implementation is sufficient. The starfield display canvas represents the combined representation of two distinct attributes. The user can thus filter the data along two attributes.

We have to think of the possibility to use the starfield display as a two-dimensional data representation. If we consider the case of NASA data covering a certain region of the globe, it is possible to use two “range attributes” to represent the region, one attribute representing the latitude, and the other attribute representing the longitude. The problem becomes more complex when we want to introduce more than a single rectangle in the same data.

From that point, two main directions exist.

3.4.2.1. Two distinct attributes possibility

One of them keeps with the previous implementation, i.e., two attributes are used to represent longitude and latitude, see Figure 3-17. This time, each attribute has to be a multi-valued attribute containing range values. The major inconvenience of this is the need to adapt a good part of the code in consequence. Indeed, we need a way to indicate that the two attributes are linked one to the other. When the database is loaded from the hard drive, an ordering algorithm is launched inside every multi capsule to order them. This algorithm must not be launched in the case of the multi-valued attributes, in order not to mix the values, and not to link a wrong longitude range to a latitude range, see Figure 3-16.

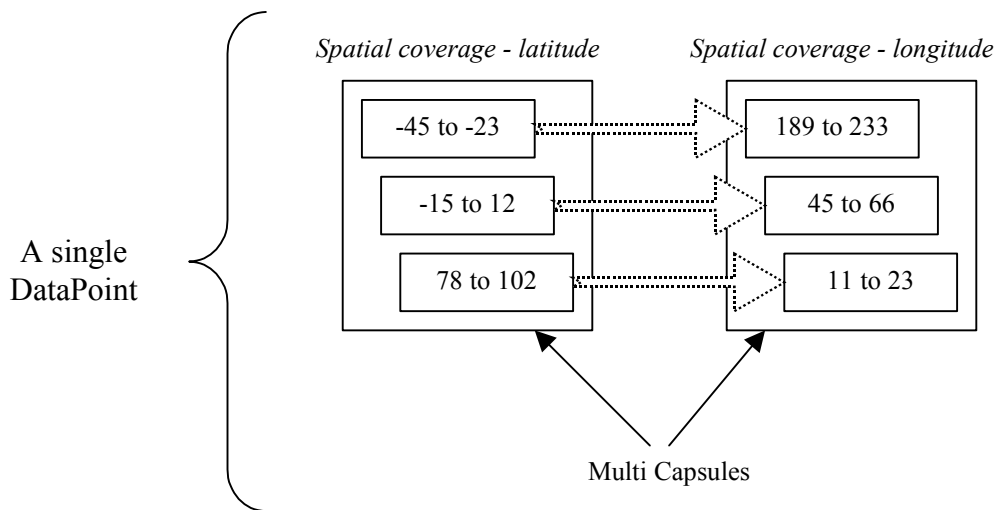


Figure 3-16: Here is an example of data where the spatial coverage corresponds to three geographical zones, each of them represented through a latitude interval and a longitude interval.

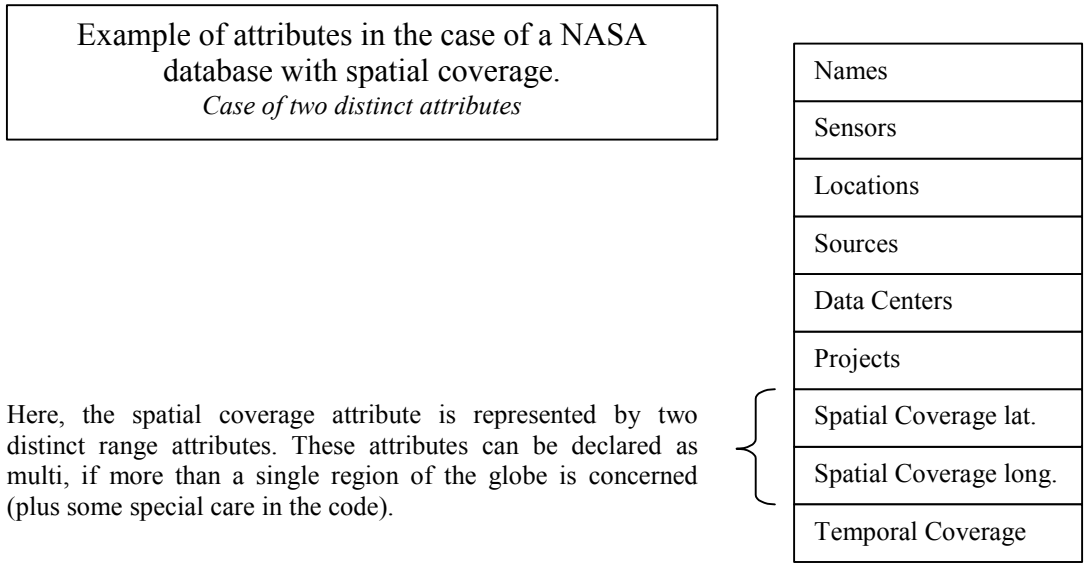


Figure 3-17: This figure describes the list of attributes in the case of a NASA EOSDIS database if we consider two distinct attributes for the spatial coverage.

3.4.2.2. Spatial attribute possibility

The second possibility is to create a new capsule specific to this kind of data, a “spatial capsule”. This capsule will be a two-dimension like capsule, with two double values, four values overall, defining a rectangle on a map world, see Figure 3-18. Of course, this kind of capsule forces addition of specific code as well. If more than a world region is covered by a data, the use of multi-capsules of spatial capsules is very easy in this case.

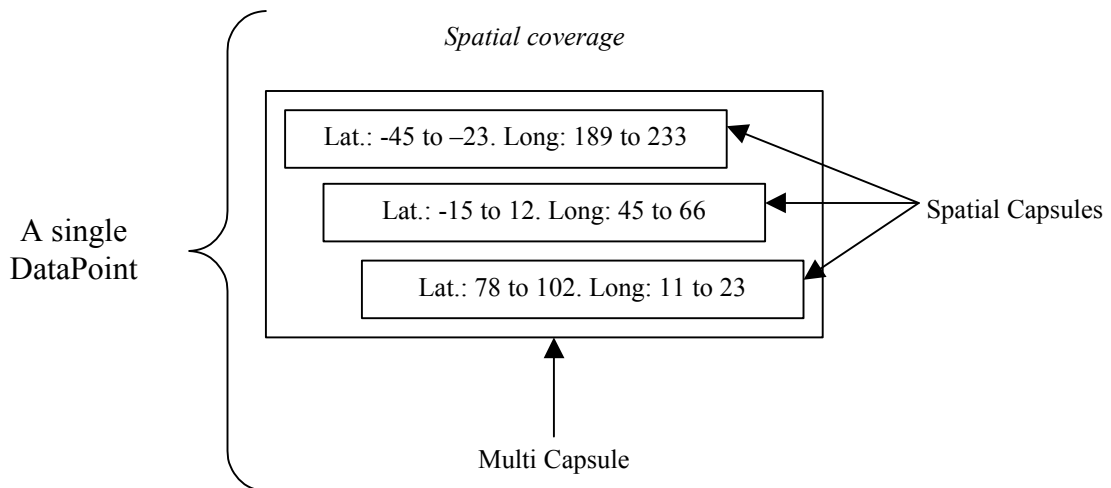


Figure 3-18: The three zones of the spatial coverage for this data are encapsulated in the same multi-capsule, which contained three spatial capsules. This is what is currently implemented in StarDOM.

The fact that the thus defined “spatial capsules” are two-dimensional creates some new constraints on the graphical display. We don’t have two distinct attributes like in the previous case, where the user had to select these two attributes to obtain the spatial representation. Of course, one of the attributes, the latitude or the longitude, could be combined with any other attribute to create a new representation. Though, the question is to know whether such a combination can be useful or not. In the case of a single attribute, the selection of the attribute in one of the axis (the horizontal axis for instance) automatically selects the same attribute on the vertical axis, because any other possible combination is forbidden (see Figure 3-19).

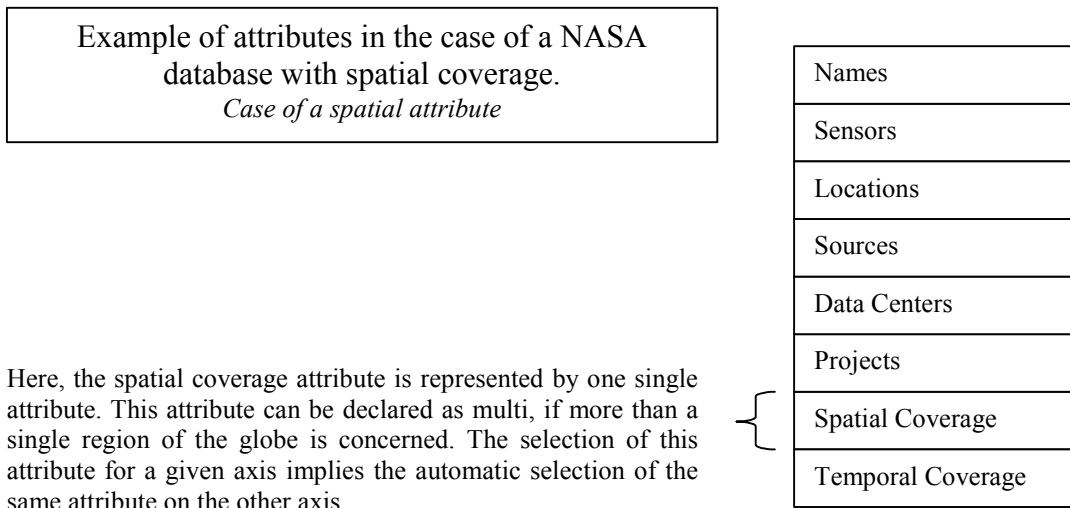


Figure 3-19: In the list of possible attributes, the Spatial Coverage has only one value, and represents a two-dimensional data. Its selection on one of the starfield axis will imply its selection on the other axis as well.

3.4.3. Problem of the “NONE” values

An important case to consider is the “NONE” data values for a specific attribute. Such cases happen very often in NASA databases, when an element of data is unknown. In the same manner as some data can have multi-valued attributes, more than a single piece of information for an attribute, it is also possible that some data don’t have value for certain attribute (no sensors, no project date, etc.).

Of course, missing information is not a reason to completely forget about the data. The question is “how to display NONE attributes values?” NONE values in the case of temporal information often means that the project (if it is a project) is not finished yet. A way to represent this information could be to replace these NONE values by the actual date and time values.

In the other cases, the problem is more complex. If the NONE-value attributes remain on the right panel filters, the best solution may be to simply ignore the value. It is then impossible to filter the NONE-value data along this attribute, which sounds logical.

Now, let's consider NONE-value attribute used in the starfield display. In this case, two attributes are considered, along each axis, and not displaying the point in case of a NONE-value is not a very good choice because the value of this point along the other axis may not be NONE. A possibility is then to consider NONE as a possible value on its own, and to display it like another attribute value. This is obviously the easier way, as presented on Figure 3-20.

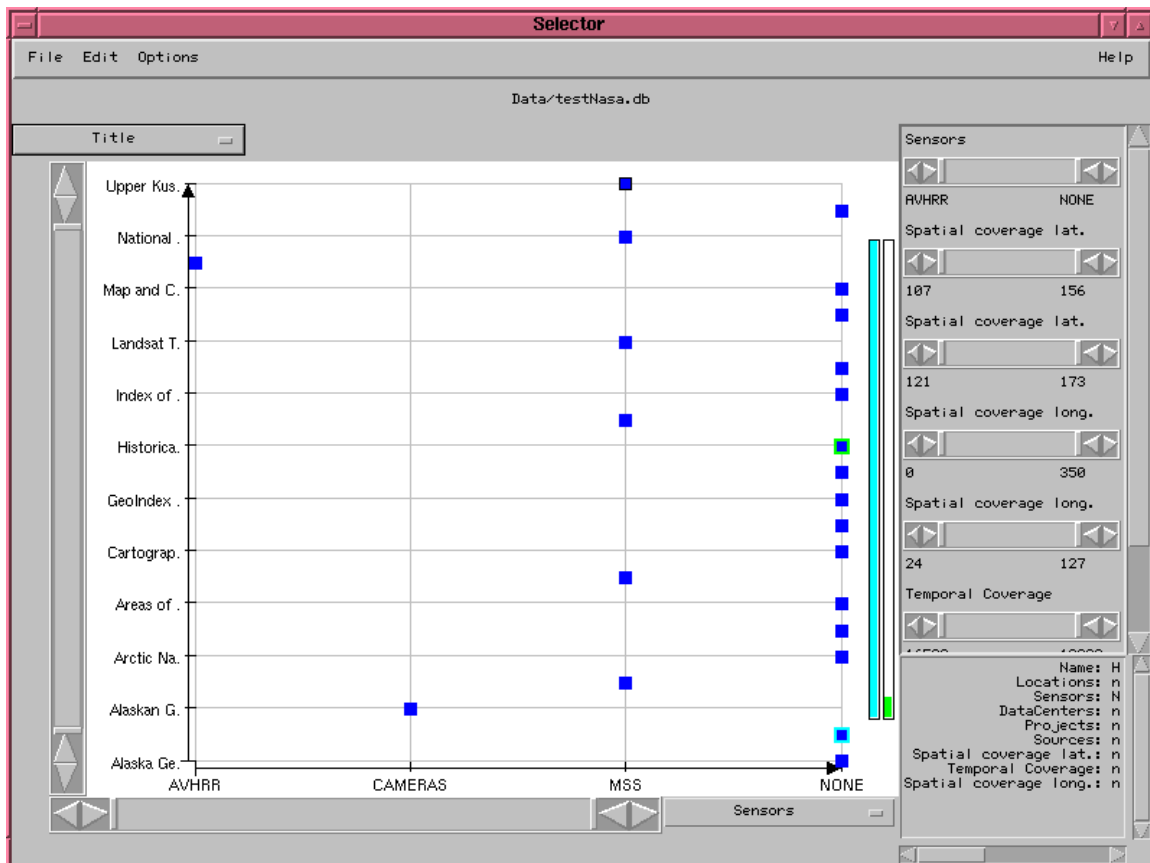


Figure 3-20: Example of a starfield representation with sensors values equal to “None” for many data points. “None” values are considered like any other possible values.

3.5. Graphical structure

3.5.1. Display of the data on the starfield

The graphical representation of the data is separated from the data structure implementation; these are two distinct phases. The only link that exists between the data structure and the graphical implementation are the two previous hashtables.

The first graphical items were represented by LitePoints, that are classes derived from Lite elements, used in Excentric Labeling. The advantage is that such elements have a very light structure in memory. As LitePoint is a derived class from Lite, it must define the abstract functions declared in Lite.java. Among these functions, there are:

```
void paint();  
Rectangle getBounds();  
int getPosition();  
void setPosition(int pos);
```

The getBounds() function is widely used by other applications, like the Excentric Labeling system, in order to know what are the points located under the cursor. It is also used for the selection system, when users click on a certain LitePoint on the screen.

The problem of this first implementation is that there is no possible representation of other kind of data than classic data, with a single value for each attribute. This class has been improved to allow the range visualization, by adding two other values in the class (startValue, endValue),

The new proposed display system has been presented previously in the data structure part. This kind of structure allows much more representation than the first one.

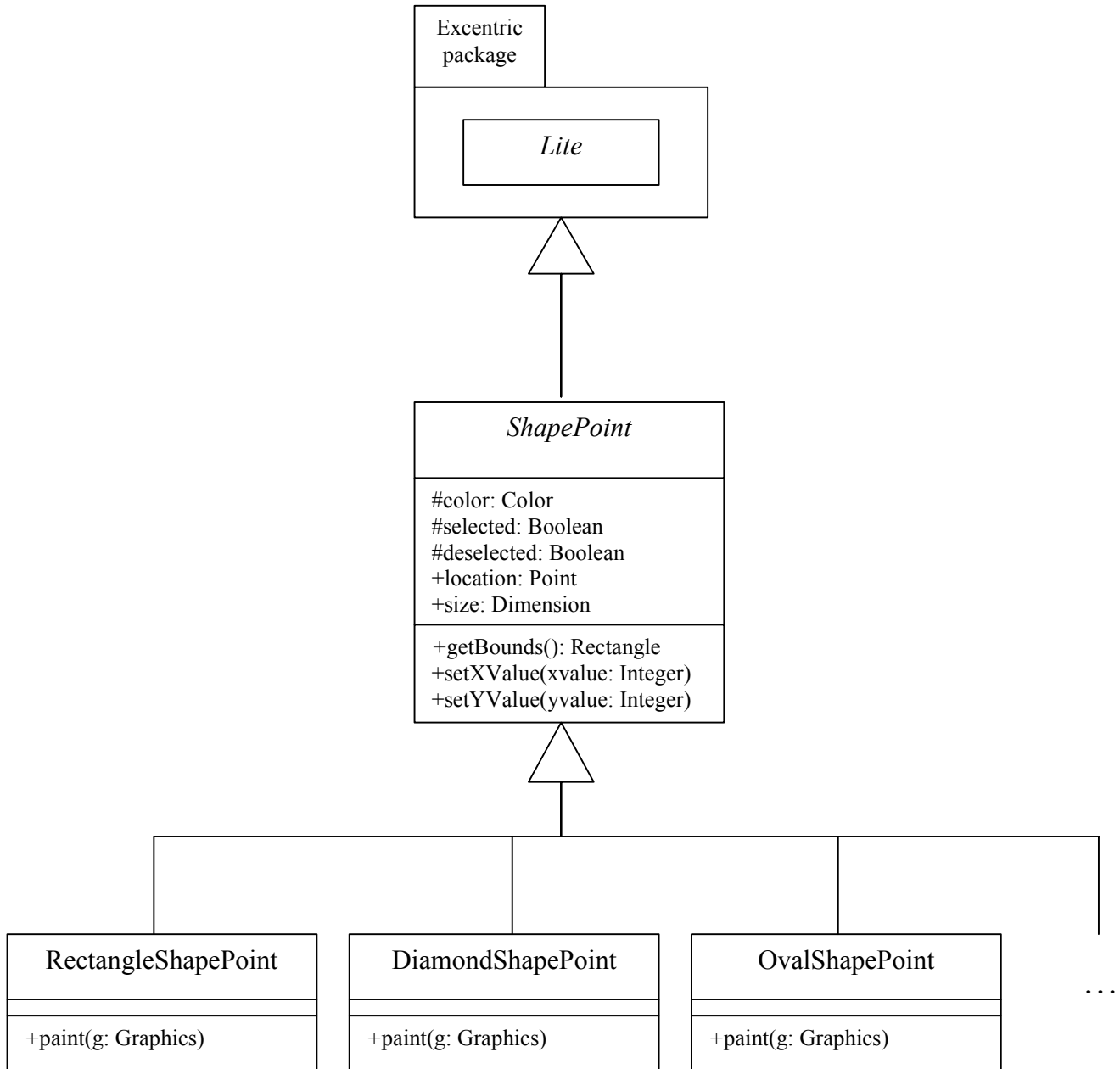


Figure 3-21: A ShapePoint is the basic structure of the graphical implementation. It represents a single piece of data on the starfield display. For instance, a multi-capsule containing two inner capsules would be represented using two different ShapePoint.

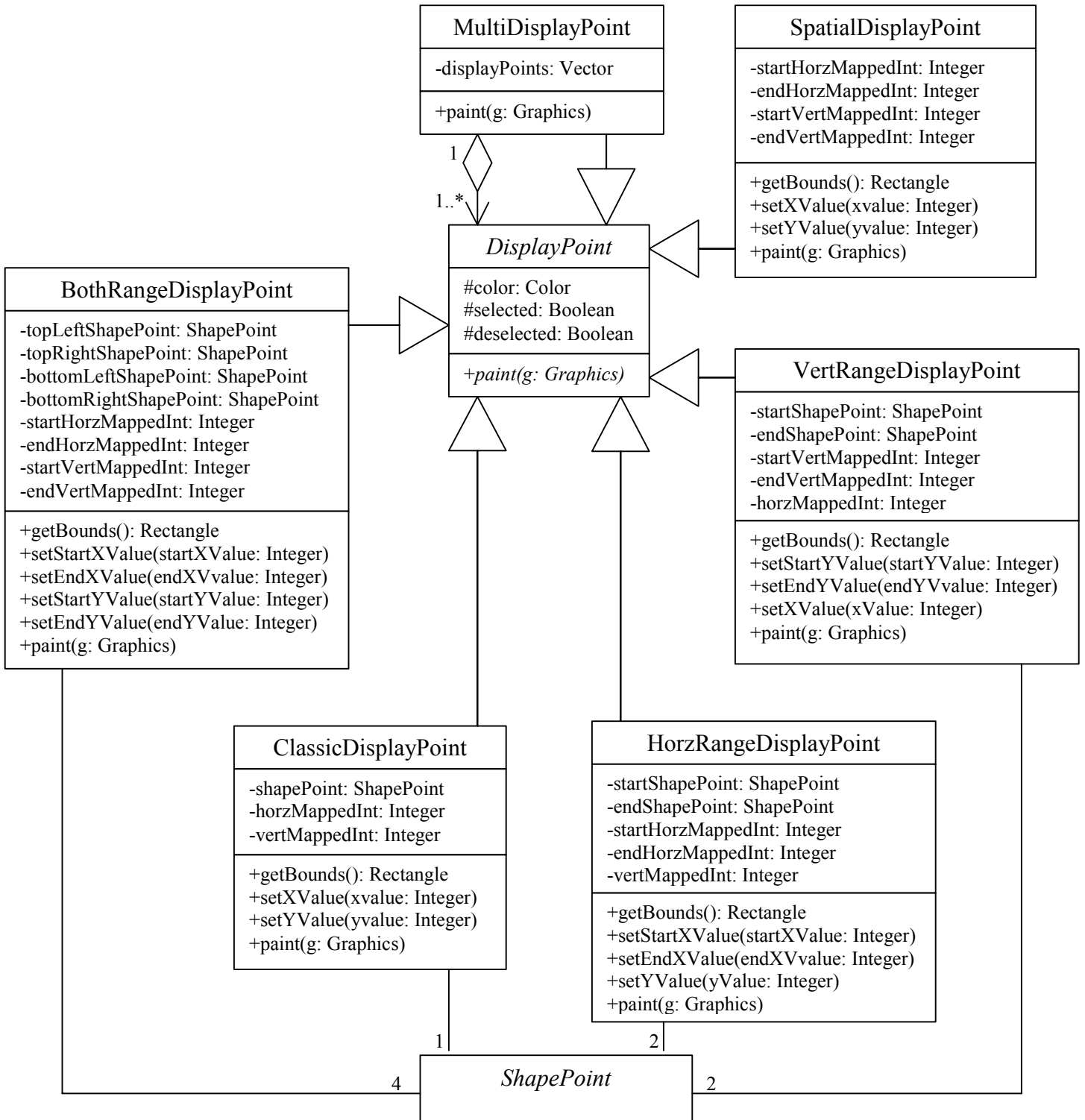


Figure 3-22: A DisplayPoint is the basic component for the representation of a data on a starfield display. There are different implementations of a DisplayPoint, according to the type of data to visualize (e.g. a range data, or a spatial data on a map).

The first UML representation on Figure 3-21 explains the structure of ShapePoints. It is a basic structure, based on the Lite elements of the Excentric Labels package. The aim of the ShapePoints is to display basic geometric figures, like rectangles and ovals, that will be used by more complex structure, like DisplayPoints. ShapePoints have a specific location, size, and color.

DisplayPoint (see Figure 3-22) is the structure used to represent any kind of capsules coming from the load of a database. ClassicDisplayPoint represents classic capsules, like integer capsules, or string capsules, and thus contains only a single ShapePoint element. HorzRangeDisplayPoint and VertRangeDisplayPoint have two ShapePoints, one for each extreme value of the range. BothRangeDisplayPoint can use four ShapePoint elements (one for each corner of the box) in its paint function, or it can have its own specific paint function displaying transparent boxes. All these functions have a certain number of “mappedInt” variables (horzMappedInt, vertMappedInt, startHorzMappedInt for example). These variables represent the value of the points on the screen, and are affected each time an axis is changed. “Mapped” means that every value is between 0 and a maximum value (5000 for example), in order to have only calculation between integers and gain some precious time.

A multiDisplayPoint has a vector of DisplayPoints. The paint function of this class launches the paint function of each DisplayPoint in the vector, and makes lines between them thanks to the getBounds() functions. Thus, it is possible to represent all kind of “multi-combinations” between different capsules, i.e.,. this is a multi-capsules representation.

4. Evaluation of the current version of StarDOM, further improvements

This last part of the report makes a general overview over the implemented prototype of StarDOM. Time tests evaluate the different times needed for the display of data, which remains a serious problem in the current version. Features of StarDOM are discussed, with some propositions for improvements, and further implementations.

4.1. Time experiments and search of a smooth, continuous dynamic feedback

4.1.1. Introduction

In order to have a smooth, continuous feedback, the starfield display should be repainted in less than 100 ms.

4.1.1.1. Context of the tests

All the test experiments have been made on SUN ULTRASPARC ultra1 machines, with a standard UNIX environment.

The sample example that will be consider in the next tests is a set of 5000 randomly generated data points, uniformly distributed, with only single-valued attributes. The load of the database and the calculation of all the ordered tables of indexes take, for our example of 5000 non multi-valued data points, takes about 25 seconds. The ordering method used is quicksort, which is in $O(n \cdot \log(n))$ in most cases, but could be in $O(n^2)$ in some particular cases. Some other existing ordering algorithms provide $O(n \cdot \log(n))$ complexity in all cases, but their average ordering time remains less efficient than quicksort.

All the tests have been made in the specific case of a single window, a case that may be unusual in practice, the user may prefer seeing the feedback of a query on a few windows. Multi-window systems slow down the process considerably, especially in terms of calculation of the coordinates of the visible points, and repainting of them. The larger the number of windows in the system, the longer the re-display of points becomes.

4.1.1.2. The three steps of the display of points

The time-consuming processes of the update of the display following a dynamic query can be divided into three main elements.

The first time, called **calculation time of the structure**, corresponds to the amount of time needed by the system to update the visible data structure. This time is more or less a function of the number of data that have to be added or removed from the structure. Of course, this is not always true, because the system may need to go through many points already removed, i.e., it can spend a lot of time to remove only a few data (see the explanation of the structure used above). Now, if the user makes large “jumps” with a given range sliders, the number of points to remove or to add increases (either the points disappear from the screen, either they reappear).

The second time, **calculation time of the coordinates of the points**, concerns a phase just before the repainting of the points. During this time, all the points that are known to be visible on the starfield display have their coordinates recalculated. This operation is compulsory after the user moves one of the starfield display range sliders (i.e., zooming), and before the points are repainted at the correct location on the screen.

The last time, **repaint time of the points**, corresponds to the amount of time needed by the Java machine to update the display of the starfield. This time cannot be optimized easily because it is linked to Java graphic performances.

So, if a user moves one of the range sliders of the axes of the starfield display, the amount of time spent to redisplay the points will be the sum of:

- The calculation time of the structure,
- The calculation time of the coordinates of the points,
- And the repaint of the points.

If a user moves a widget placed on the filtering panel, the amount of time may be less than the previous one. Of course, the same three operations have to be performed, but the amount of time necessary to do the calculation of the coordinates of the points and even the repaint of the points will decrease. Since, only the new points that appear on the screen need to have their coordinates re-calculated. In the speed tests, this difference between the main range sliders, and the other query widgets is not made. The case of range sliders of the starfield display only is considered (because this is the worst case).

4.1.2. Time experiments

4.1.2.1. First StarDOM version

Our first implementation of the prototype did not use any structure at all to store the results to a query. The coordinates of all the points were recalculated. The test showed that among the three processes, the most time-consuming one was the calculation time of the coordinates of the points. The calculation of the time is represented in Figure 4-1.

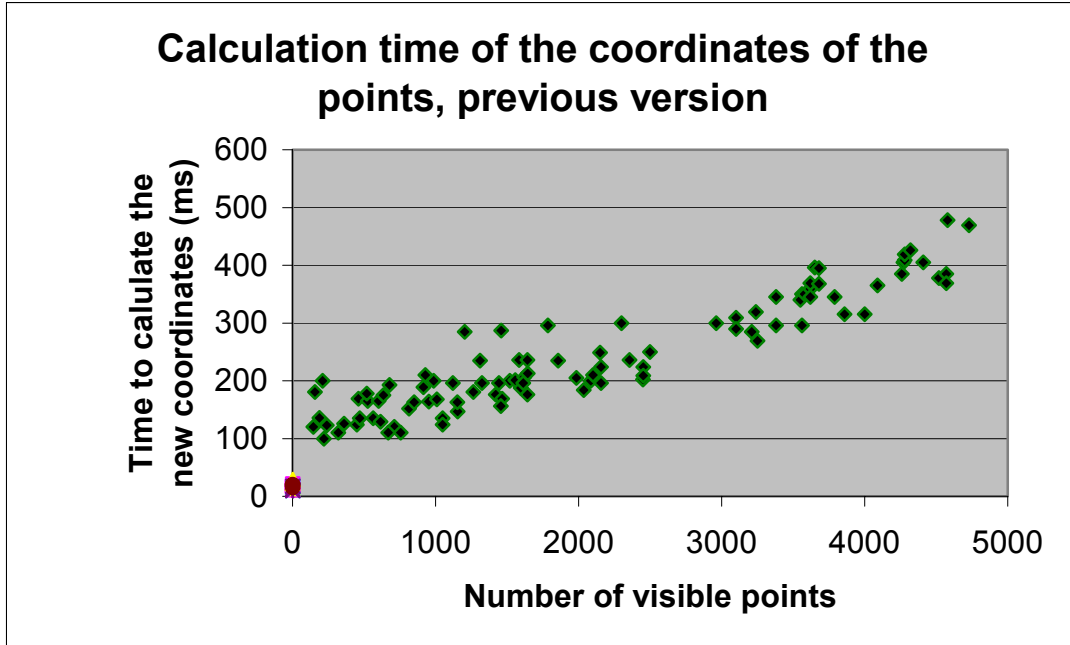


Figure 4-1: Time needed to calculate the coordinates of the points, with the first version of StarDOM, on a sample of 5000 points. We can see that even with a small number of points, the time needed was generally more than 100 milliseconds.

All the data that have to be displayed on the starfield display (i.e., the answer to the dynamic query) are stored in a structure represented by the Java class “VisibleData”. This class is simply a table of each data, and each display point.

Indeed, it appeared that the time needed to compute the correct visibleData structure was really negligible compared to the time needed to update the coordinates of the display points. This time corresponds to the update of the points following a movement of one of the two starfield displays range sliders. The algorithm has to cover all the points to display in order to set their new values on both axes. When the database is large, and all the points still visible, because not filtered yet, the delay can become annoying to the user.

4.1.2.2. Current StarDOM version with single-valued attribute data

Calculation time of the structure

The structure used to compute the data solution of the dynamic query is a rather simple one, commonly used in other dynamic query interfaces (see its description in the previous part). Of course, it does not have the efficiency of special database structures where results to any query can be made independent of the number of attributes, or even independent (or nearly!) the number of points in the database. Some examples of such structures have been discussed in the previous work section.

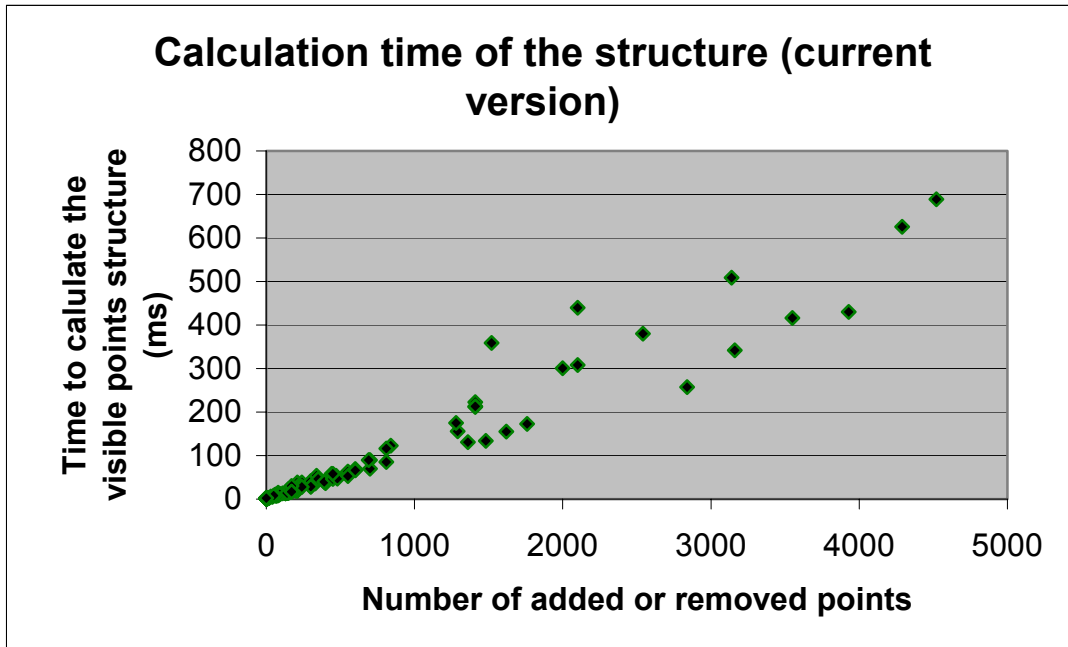


Figure 4-2: For a small number of added or removed points, the time needed to calculate the structure (i.e., to answer a query) is manageable. Even if this time is function of the number of added or removed points, and not of the total number of points, it becomes a problem with more than 1000 points updated.

The complexity of the calculation of the structure is easy to compute. Let's adopt these notations:

- t_{α} : Time needed to go through a data point without integration,
- t_{β} : Time needed to go through a data point that would be integrated or removed from the visible data structure,
- N_{α} : The average number of points to go through without integration, for a given move of the range slider,
- N_{β} : The average number of points to go through with integration or remove from the visible data structure, for a given move of the range slider.

The algorithm performing this calculation of the structure is located in the Java class "PointsCoords.java". Basically, as explained previously, this class has a index table storing indexes of points ordered according to the appropriate attribute, and two indexes giving the bounding box mapping with the position of the range slider (see Figure 3-12).

For a given movement of the range slider, four steps are carried out:

- The first one will consider a move of the left thumb to the left,
- The second one, a move of the left thumb to the right,
- The third one, a move of the right thumb to the left,
- The last one, a move of the right thumb to the right.

The algorithm will increase or decrease the corresponding index in order to stick to the new position of the range slider. Of course, during this process, all the data points crossed see their filter variable changed (either increased, or decreased). For certain values of the filter variable, the corresponding point needs either to be integrated to the visible data structure (that corresponds to the answer of the new dynamic query), either to be removed from this structure. This time is usually much larger than a simple crossing of the data. As a consequence, the complexity will be in $O(N_{\alpha} \cdot t_{\alpha} + N_{\beta} \cdot t_{\beta})$.

Calculation time of the coordinates of the points

Figure 4-3 shows the calculation time of the coordinates of the points in the current version of StarDOM. All the heavy accesses to different functions have been removed, and we can see the difference of efficiency between Figure 4-1 and Figure 4-3. It becomes now difficult to clarify even more the calculation of the coordinates, because it is reduced to only two fast simple operations, a multiplication, and a division. This time, the time is always less than 100 milliseconds, even with 5000 points, but of course, there are other times to consider.

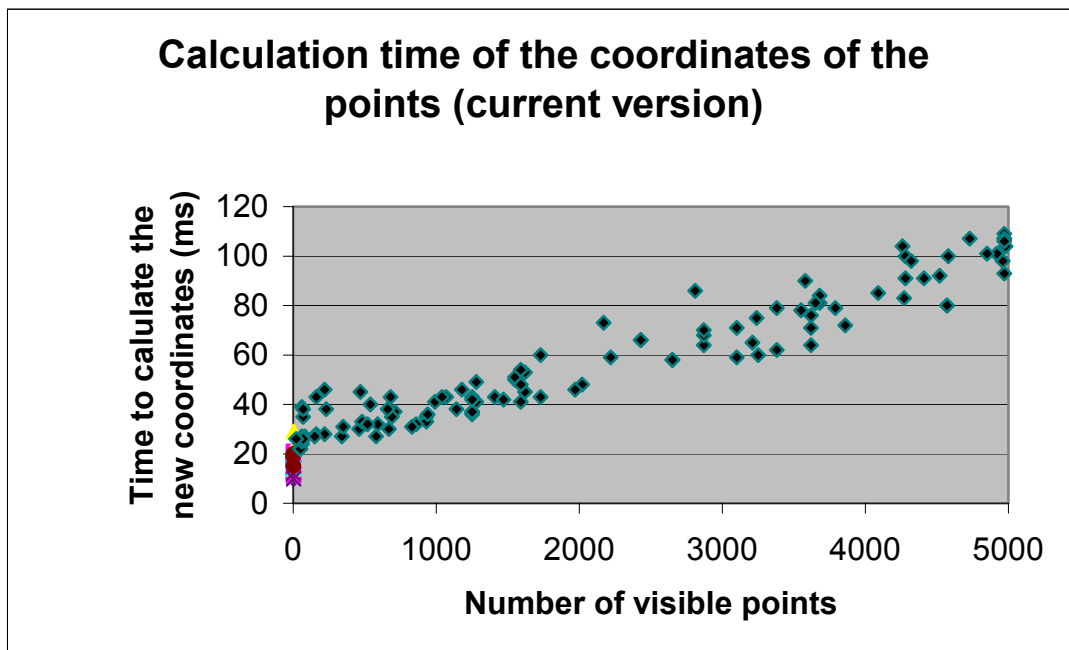


Figure 4-3: The calculation time of the coordinates of the points has been considerably improved, compared to Figure 4-1. It remained acceptable even in the case of the recalculation of the coordinates of 5000 points on the starfield.

Repaint time of the points

Figure 4-4 presents the last time-consuming process that corresponds to the time needed for repainting the points on the screen. This time is quite difficult to control, and may change for different reasons independent of the implementation of the system (e.g. if

users enlarge some windows, the repaint process will last longer). Still, it has been improved by creating lighter display classes, with as easy paint functions as possible.

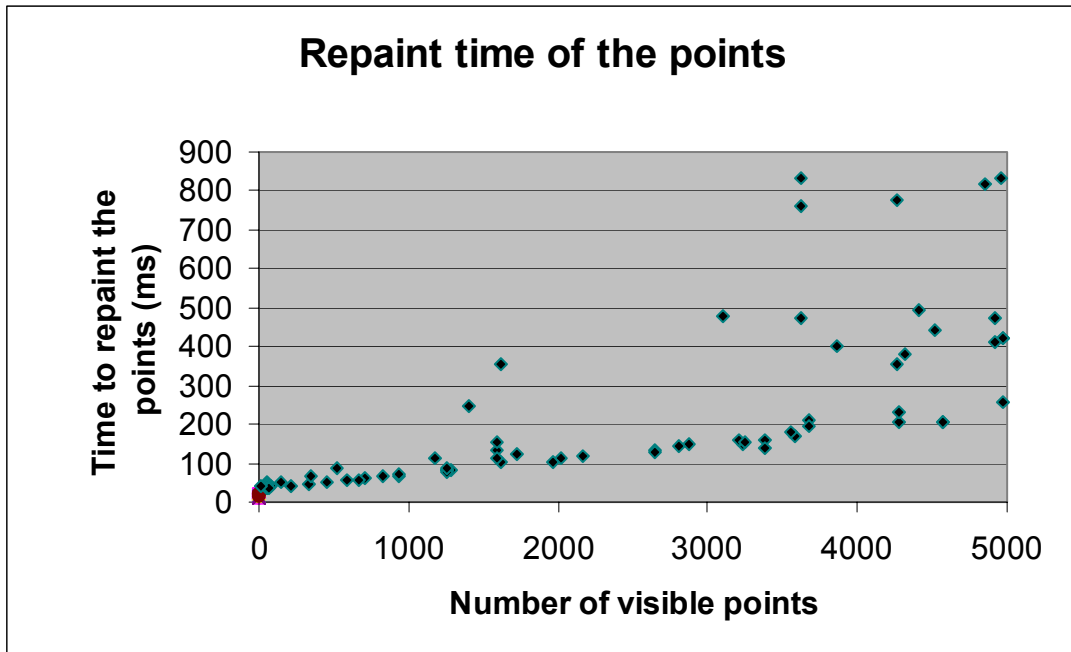


Figure 4-4: The repaint function is a process on its own, and it is difficult to measure. The figure shows that the time is acceptable for a small number of points (less than 1000 points). The high values from 3500 to 5000 points are due to the repaint of other components at the same time as the points. They may not be considered here.

4.1.2.3. Current StarDOM version with multi-valued attribute data

The previous set of time tests has been done with the same sample of 5000 points as before, with single-valued attribute data. This new set of tests has been done on a sample of 375 points, with two multi-valued attributes projected on the coordinates of the starfield. The proportion of the number of values for each of these two attributes is represented on the Figure 4-5. These tests give values less interesting than the previous ones, in the sense that every data is distinct one from another (different number of values for a multi-valued attribute), and the time needed for the few computations can be different.

This sample of data comes from NASA EOSDIS data. Even if it is very specific, it gives general idea of the time consumption.

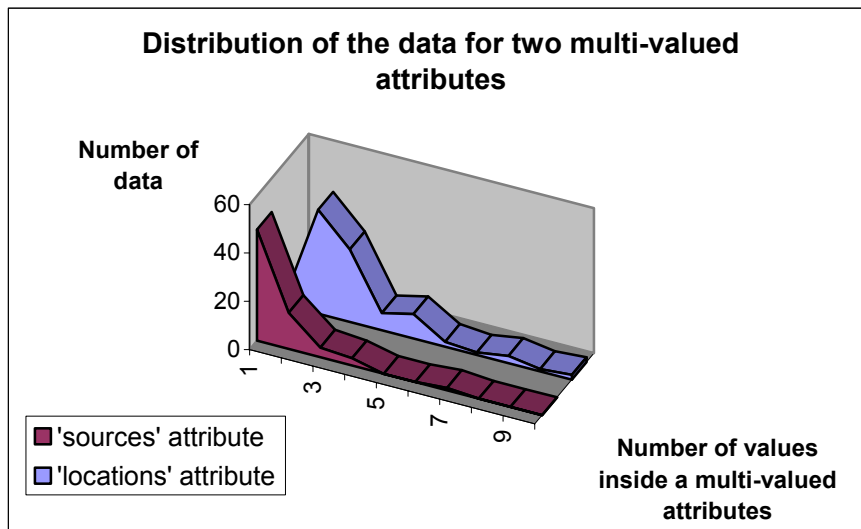


Figure 4-5: For this test on multi-valued attribute data, the attributes ‘sources’ and ‘locations’ have been chosen for coordinates of the starfield display. A data with two values on a given attribute may imply time values multiplied by two. In this case, the attribute ‘sources’ mostly has one or two values, whereas the attribute ‘locations’ has mostly two or three values, which means a increase time factor from two to six.

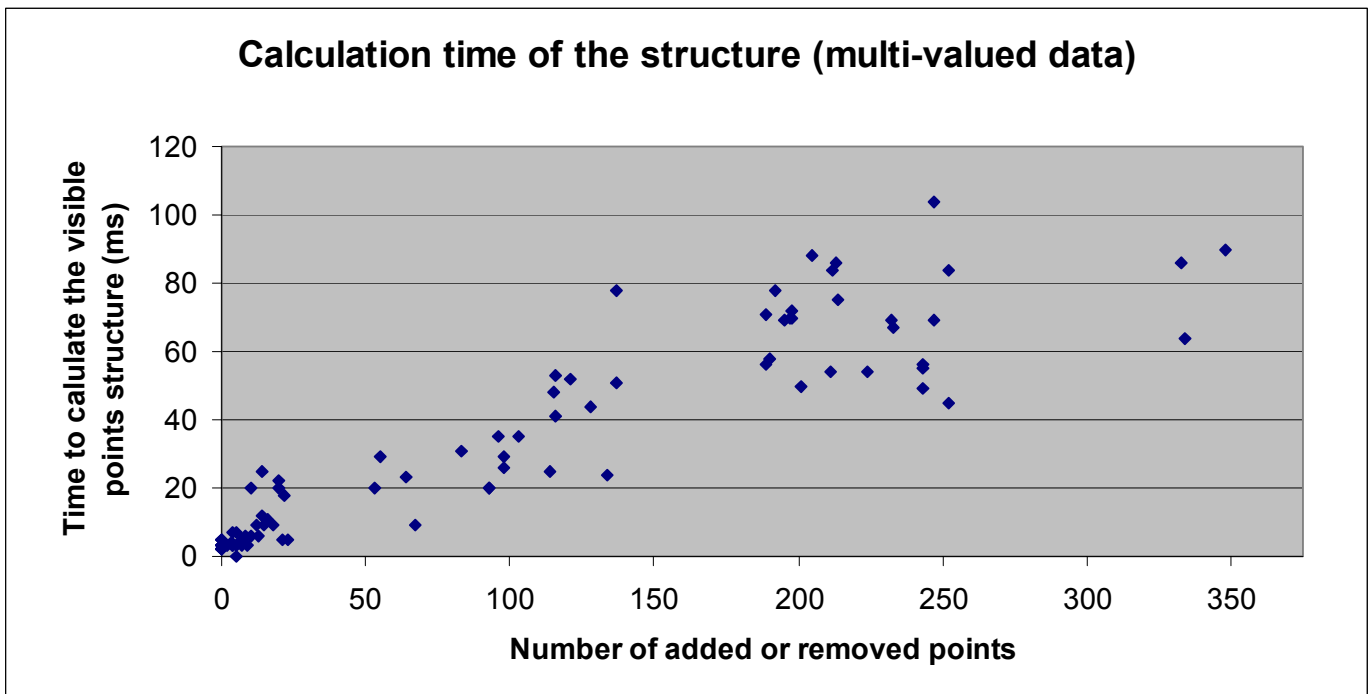


Figure 4-6: Calculation time of the structure (or querying time) in the case of a sample of 375 multi-valued data points. If the user uses smoothly the different sliders (removing or adding less than 100 points), the time remains less than 40 milliseconds.

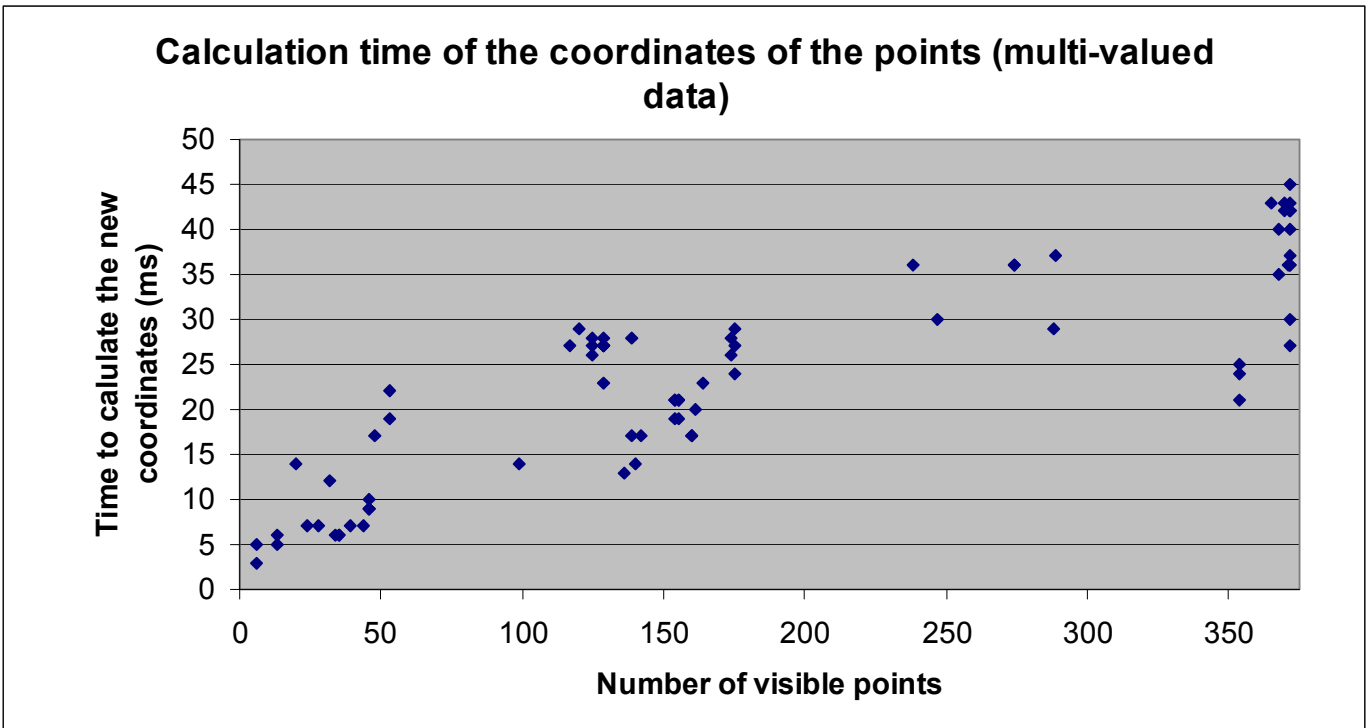


Figure 4-7: With the same sample of points, in the case of the coordinates of the points, the time remains acceptable.

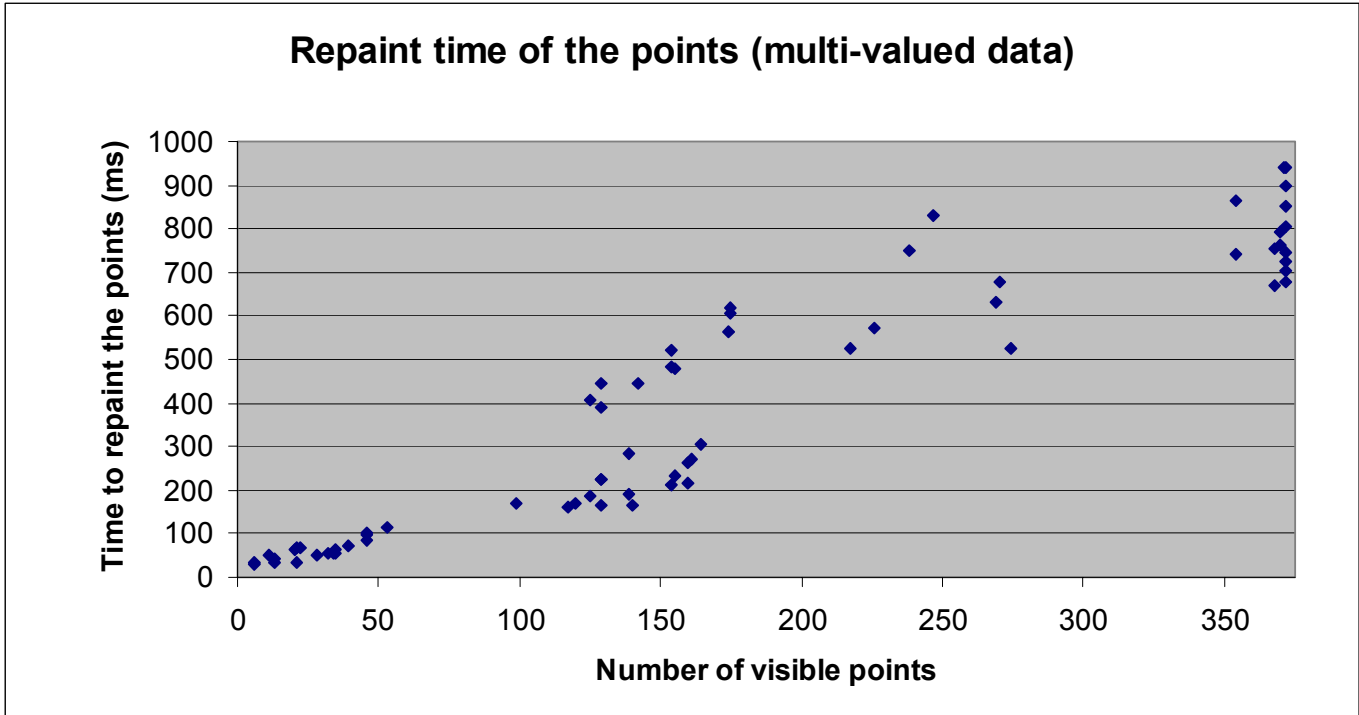


Figure 4-8: In the case of this sample of multi-valued data, the repaint function is the overwhelming process, and it has to be refined in future versions.

The three graphs (see Figure 4-6, Figure 4-7 and Figure 4-8) show that the optimization effort should be put on the repaint function first. Even if this is difficult to control, making the paint functions lighter may improve this time.

4.1.2.4. Synthesis of the results

Even if the same behavior has been observed on other machines, these results are valid on SUN ULTRASPARC ultra1 machines.

In the case of single-valued attributes, the update time remains less than 100 ms with a sample of 1000 points, for a small number of points added or removed at a time (i.e., 200 points). The repaint time of the points is the bottleneck, and it has to be improved. It is currently difficult to have a 100 ms time when more than 1000 points are considered. But in order to be able to update more points at once, the structure has to be changed as well. The time of structure computation increases sharply (even if linear) in function of the number of updated points, (e.g. 300 ms for 2000 points).

If we consider multi-valued attributes, the number of points to consider depends mostly on the average number of values in the multi-valued attributes. The calculation times of the structure and of the coordinates of the points can clearly be neglected in comparison of the time needed to repaint them. This becomes even worse when more than a single

starfield display are used. Of course, this time has to be improved drastically. Currently, even if 100 points sample already goes over 100 ms, it is really acceptable up to 300 points, with multi-windows, and with “highly” multi-valued data. The multi-windows option is not a real problem for the repaint function, because they often overlap each other, i.e., only small portions of them have to be updated.

4.2. Discussion and future work

4.2.1. Speed performance

4.2.1.1. Improve the “calculation time of the structure”

The relatively simple structure currently implemented in StarDOM (and detailed in the part 3 of the report) has the advantage to be light in memory, and easily manageable. Of course, in the case of large databases, when range sliders are moved very quickly, the time needed for its update would grow. A further implementation of the system may take this into account, in order to try and use another structure.

It may be a good idea to try to implement a new structure using trees (R-trees or Kd-trees for instance). They are the most popular structures used in databases nowadays. Some examples of tree implementations are given previously in the bibliography. Nevertheless, we have to keep in mind that we would like to keep the loading of the system fast enough, and the construction of the tree must not take too much time to avoid any inconvenience. The difficulty to deal with multi-valued attributes has to be considered as well. Besides, the result of the study [23], as explained previously, gives to the linked arrays a good comportment when confronted to small data sets of a few thousand points.

I don't think that it could be a good idea to implement the structure of [13] in the case of StarDOM. One of the restricting features of this structure is its space-consuming characteristic: the size of main memory can become a severe restriction. Moreover, a lot of preprocessing has to be done. This implies that the program can detect if the mouse is close to a widget, in order to launch the calculation of all the necessary tables for the selection phase. I found that the re-scaling feature was really interesting, and can be adapted to StarDOM. In this article, every data is re-scaled to the range $[1, p]$, where p is the number of pixels in the attribute's range slider. In StarDOM, all the data information is encapsulated in different classes called “capsules”. These capsules keep the information, that can be strings values, integers, floats, etc. The main aim of this is to provide different operations on capsules (e.g. addition, subtraction, multiplication, comparison) without taking care of the internal values. In a previous implementation of the prototype, the operations were also used to calculate the coordinates of points on the screen. That can reduce performance, especially in the case of string capsules (a lot of NASA EOSDIS information is represented with string values), because comparisons are made between string values. This remark is valid for other kind of data, like float encapsulations, date, etc.

Though, as noticed in the paper [13], the starfield update is a serious constraint when the database grows. With the incremental structure used in [13], the “pure” querying time is no more than 20 milliseconds (average of 10 milliseconds). This is negligible, with respect to the starfield display times obtained, especially when the number of records got bigger. As a result, there is no real need to implement this kind of structure if the starfield update time is the bottleneck.

4.2.1.2. Use of multi-thread

Some other possibilities to speed up the process remain, like the use of multi-threads for computing the data structure and coordinates, and displaying them. Spotfire uses this multi-thread possibility in its implementation, but not StarDOM. Because of a lack of time, such structures have not been tested yet. The result of such an implementation will be a smooth manipulation of the sliders at any time, but the update of the display would be delayed by a few seconds.

4.2.1.3. Display on a large canvas

The improvement research path should probably focus on the optimization of the starfield update times. An idea is to compute the whole starfield representation at the beginning, and store it on a large canvas. This canvas may take a few seconds to be modified each time an axis is changed, or a characteristic of the points (like color, shape or size). This idea may sound interesting, because the time of display would not be dependant upon the number of points to display. Nevertheless, it is difficult to implement this, first because the image would be really too large in memory: if a range slider has 500 pixels, this implies a 500 zoom factor, which becomes definitely too large if we consider both coordinate axes. Second, the zooming feature that this implementation creates on the display of points is not desired (the size of the shapes won't be appropriate anymore).

4.2.1.4. Display of a relevant sample of points

The display will be much faster when only 500 points (from a database of 5000 for instance) are visible, instead of 3000 points. Of course, this may not be really satisfying if a great number of points remain on the screen, as shown on the time tests. The best solution at that moment can be to display only a well-chosen sample of data, and getting rid of the rest. The idea is that the user won't be able to select or to visualize any data, because there may be too many points. A flag (combined with the number of visible data and the number of data to reach) can be used to indicate to the user that all the data elements are visible, or only a sample of the data. It is also possible to display only a relevant sample of the data while a slider is moved, and then display them all again when the slider is not used.

Another possibility is to use aggregation in order to display data more quickly. Of course in this case, other problems arise, like the update of dynamic aggregation. This corresponds to the problem of the Data Cube. Such problems are considered in many articles, like *CubeTree: Organization of and Bulk Incremental Updates on the Data Cube*, by Nick Rossopoulos, Yannis Kotidis (from the University of Maryland), and Mema Roussopoulos (from Stanford University).

4.2.2. Multi-window feature

The current multi-window system allows users to see the feedback of a query on different starfield displays. It is also possible to select a set of data on a window, and see the equivalent display points getting selected on the other windows. Though, it is possible to develop much better possibilities, like implemented in the Visage-VQE system (see [19] and [20]), or in DEVise (see [21]).

As presented in [11], an interesting feature of coordinated multi-window systems is that, since it receives notification of user actions in all visualizations, it can easily keep track of the history. It may be interesting to study if a history system can be useful in a prototype like StarDOM, and if so, try and implement it.

4.2.3. Mapping of graphical features to data points

Only shape, size and color are the available features to distinguish data. Of course, as described in the bibliography before, with the article [17], and as developed in Spotfire, many other graphical features can be used in the goal of information retrieval. The current implementation allows the use of color, shape and size, for classic data points. The case of multi-valued data is still to be implemented. It contains some unseen difficulty, where the use of color gradient seems the most appropriate. Some other graphical properties must be added as well (e.g. textures, orientation, angle).

4.2.4. Alternative with transparency

Figure 4-9 presents a screenshot of StarDOM when the transparency of Java 2.0 is used. Of course, the display is much nicer than simple rectangles, but it is also much slower. The installation of this feature has to be finished, in order to provide an option allowing users to have their displays with transparency. The system has to run a test to know if the Java version can support transparency (jdk version 1.2 or higher).

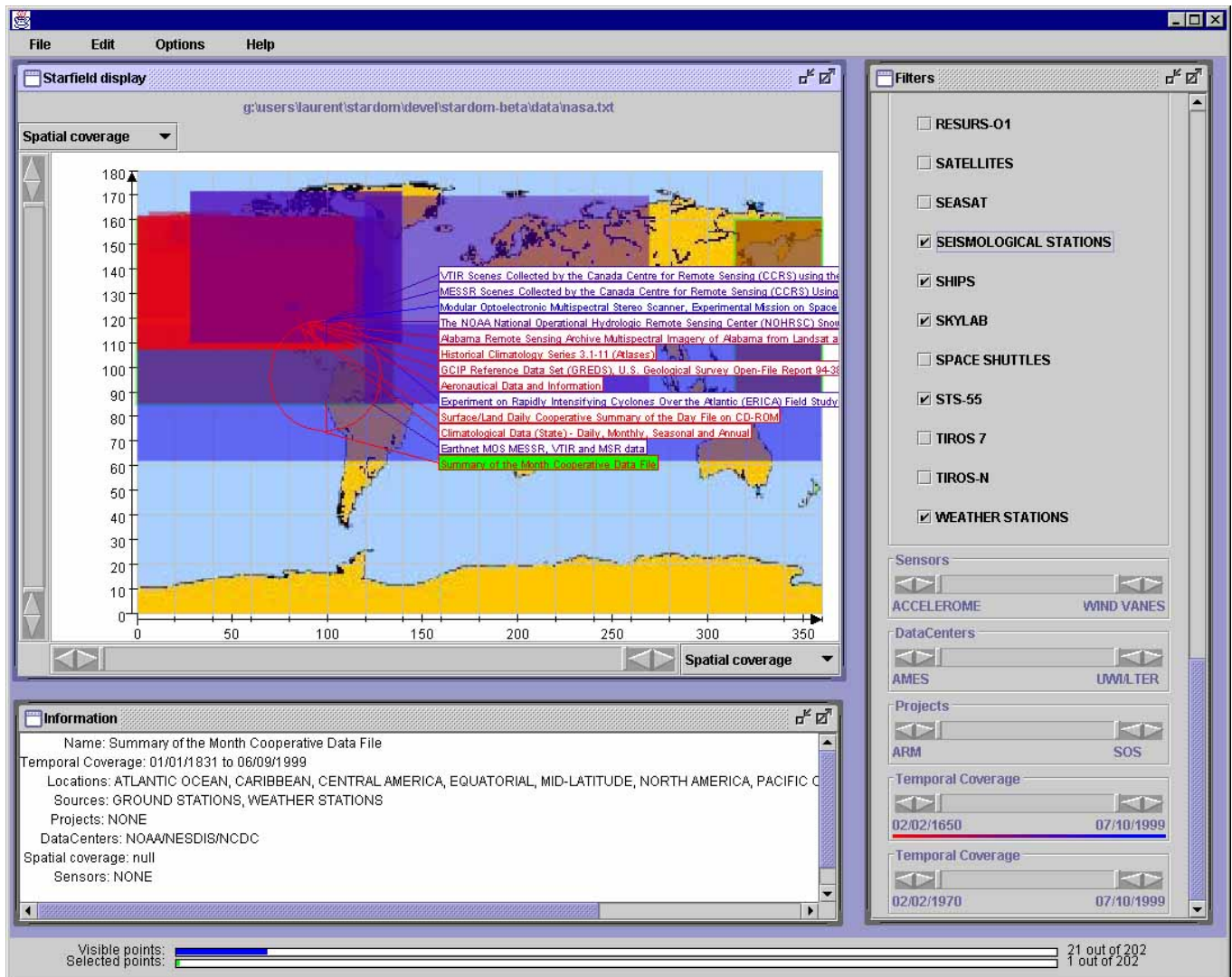


Figure 4-9: The transparency feature of Java 2 used in StarDOM. Though very attractive, this characteristic remains a burden for the update time of the starfield display.

4.2.5. Creation of an applet

The current version of StarDOM consists of an application that has to be downloaded in order to be run. Of course, this download occurs just once, and the user just needs to get the correct datasets file from the GCMD site. Though, it could be a better idea to use StarDOM as an applet in a future version. Two issues remain, first, the java virtual machine of the current Internet browsers do not support Java Swing components. A lot of StarDOM features are based upon these Swing components (e.g. the multi-window system). Sun provides a Java plug-in that can be downloaded from their web site, in order to make the current browsers "Swing-compliant". Besides, Sun provides converters to create Swing applets without difficulty from the developed application. The second problem concerns the security manager, like for every applet. There may be some

changes to perform on the StarDOM version before being allowed to transform it to an applet by the Java compiler.

4.2.6. Generalization of StarDOM to all kind of databases

StarDOM opens the way to a more powerful graphical database visualization tool. Even if StarDOM allows the visualization of the multi-valued attributes of a database, there is yet no possibility to navigate into a real database scheme, like the powerful VQE – Visage do (see [19] and [20]). The input data of StarDOM is only a simple text file, and the system is developed in a way to fulfill the requirements of NASA data, which can be expressed through a simple relational database diagram, on Figure 4-10.

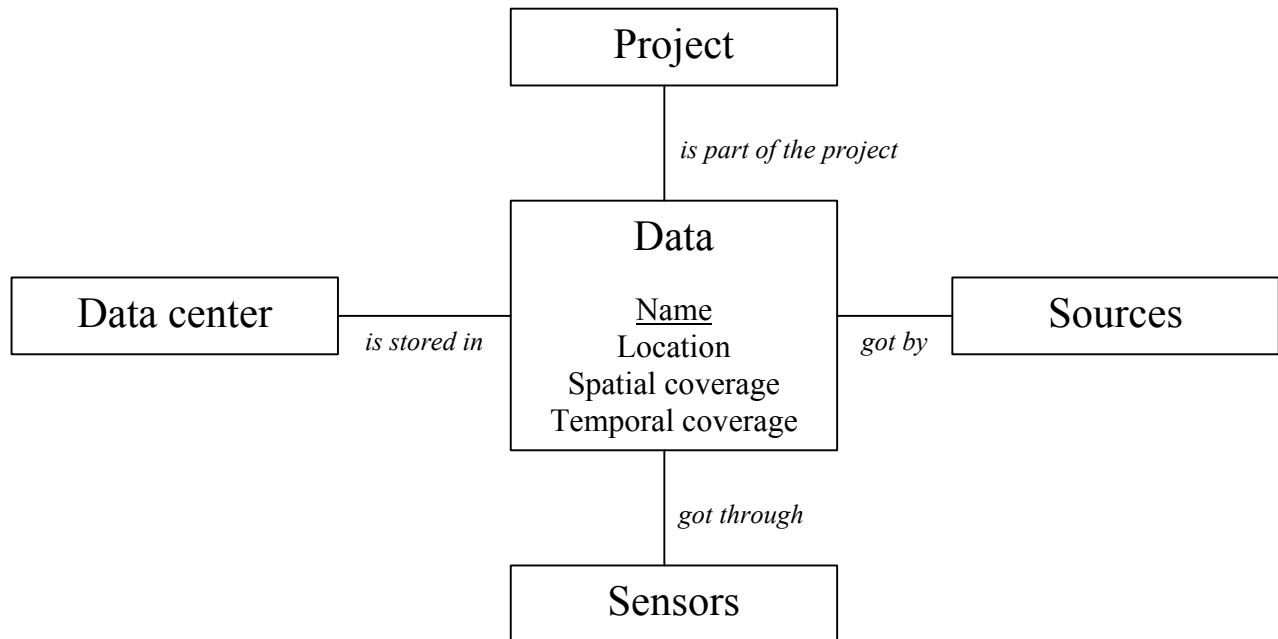


Figure 4-10: Possible database schema in the case of NASA EOSDIS databases. Every table is linked to the central object “Data”. As a consequence, all the data can be gathered in a single table, at the condition to allow multi-values for some attributes. The current implementation of StarDOM is based upon this feature.

If different kinds of databases, with large real object tables, are to be considered in StarDOM, a further implementation may take this into account. In the scheme of Figure 4-10, all the tables are linked to Data, and it is thus possible to regroup all the information into a single object, at the condition of allowing multi-valued attributes for some attributes. This is what StarDOM does basically. The relatively simple example given in [19] or [20] is difficult to execute with StarDOM. If a more complex database schema has

to be used by StarDOM, a specific text file has to be created, with many repeated fields. Already, the text file used by StarDOM contained many similar fields (e.g. two data with same data centers, or part of the same project). This remark is not important as long as the fields are not very large (e.g. the data center table consists of a single string value).

5. Conclusion

During this six-month project, StarDOM was substantially improved from a rudimentary visualization prototype to a more powerful tool including zooming, better labeling, color and size coding, multiple displays and faster querying. A new aspect of information visualization – range-valued and multi-valued attribute data – was explored. The corresponding data structures and display algorithms were designed and implemented. Representative data from NASA was selected and used for testing.

The power of Java will be an asset to integrate StarDOM with other technologies (e.g. databases with JDBC). Already, the use of “Swing” components helped to rapidly develop user-friendly interface components (e.g. multi-windows system, large number of widgets, nice interface display).

StarDOM seems to be a viable approach to the NASA EOSDIS data. It allows the visualization of all the types of attribute data from our NASA samples and could be used as a refinement phase for the Global Change Master Directory search interface.

On the other hand, this is still a prototype. Much more can be done to improve the user interface, and to refine the data structures to allow the visualization of larger number of records. By dealing with the particular format of EOSDIS data, StarDOM also opens the way to a more general, advanced visualization tool adaptable to all kinds of databases.

References

- [1] Williamson, Christopher, and Shneiderman, Ben, 1992. The Dynamic HomeFinder: Evaluating dynamic queries in a real-estate information exploration system, *Proc. ACM SIGIR'92 Conference*, Copenhagen, 338-346.
- [2] Ahlberg, Christopher, and Shneiderman, Ben, Visual Information Seeking: Tight Coupling of Dynamic Query Filters with Starfield Displays, *Proc. ACM CHI 94 Conference*, 313-317.
- [3] Ahlberg, C., and Shneiderman, B., 1993. The Alphaslider: A Compact and Rapid Selector, *Proc. ACM CHI 94 Conference*, 365-371.
- [4] Ahlberg, C., Williamson, C., and Shneiderman B., 1992. Dynamic queries for information exploration: An implementation and evaluation, *Proc ACM CHI'92 Conference*, 619-626.
- [5] Eick, Steven, 1993. Data visualization sliders, AT&T Bell Laboratories Report, Naperville, IL.
- [6] Williamson, C, and Shneiderman, B., 1992. The Dynamic HomeFinder: Evaluating dynamic queries in a real-estate information exploration system, *Proc. ACM SIGIR 92 Conference*, 339-346.
- [7] Osada, M., Liao, H, Shneiderman, B., Alphaslider: searching textual lists with sliders, *Proc. of the Ninth Annual Japanese Conf. On Human Interface*, Oct 1993.
- [8] Plaisant, C., Venkatraman, M., Ngamkajornwiwat, K., Barth, R., Harberts, B., Feng, W., Refining Query Previews Techniques for Data with Multivalued Attributes: The case of NASA EOSDIS, *University of Maryland Computer Science Technical Report CS-TR-4010*, also in *Proc. Of Advanced Digital Libraries 99 (ADL'99)*, IEEE Computer Society Press (May 1999).
- [9] Fekete, J.D., Plaisant, C., Excentric Labeling: Dynamic Neighborhood Labeling for Data Visualization, *CHI'99 Conference Proceedings*.
- [10] Plaisant, C., Shneiderman, B., Doan, K., Bruns, T., Interface and Data Architecture for Query Preview in Networked Information Systems, *ACM TOIS Practice and Experience Paper*, September 98.

- [11] North, C., Shneiderman, B., Snap-Together Visualization: Coordinating Multiple Views to Explore Information, *University of Maryland Computer Science Technical Report CS-TR-4020, HCIL 16th Annual Symposium & Open House Technical Reports*.
- [12] Beigel, R., Tanin, E., The Geometry of Browsing, *the 3rd Latin American Symposium on Theoretical Informatics, 1998*.
- [13] Tanin, E., Beigel, R., Shneiderman, B., Design and Evaluation of Incremental Data Structures and Algorithms for Dynamic Query Interfaces, *University of Maryland Computer Science Technical Report CS-TR-3796*.
- [14] Tanin, E., Beigel, R., and Shneiderman, B., Incremental Data Structures and Algorithms for Dynamic Query Interfaces, *ACM SIGMOD Record, Vol. 25, No 4, 1996, pp. 21-24*.
- [15] Bertin, J., Graphics and Graphic Information Processing, *Walter de Gruyter & Co., Berlin, 1981*.
- [16] Tufte, E.R., The visual Display of Quantitative Information, *Graphics Press, Cheshire, Connecticut, 1983*.
- [17] Mackinlay, J.D., Automating the Design of Graphical Presentations of Relational Information, *ACM Transactions on Graphics, 1986, 111-141*.
- [18] Ware, C., and Beatty, J.C., Using color as a tool in discrete data analysis, *technical report CS-85-21, Computer Science Dept., University of Waterloo, Waterloo, Ont., Canada, Aug. 1985*.
- [19] Derthick., M., Kolojejchick, J., Roth, S.F., An Interactive Visual Query Environment for Exploring Data, *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'97), ACM Press, October 1997*.
- [20] Derthick., M., Kolojejchick, J., Roth, S.F., An Interactive Visualization Environment for Data Exploration, *Proceedings of the Knowledge Discovery in Databases, AAAI Press, August 1997, pp. 2-9*.
- [21] Livny, M., Ramakrishnan, R., Beyer, K., Chen, G., Donjerkovic, D., Lawande, S., Myllymaki, J., Wenger, K., DEVise: Integrated Querying and Visual Exploration of Large Datasets, *Department of Computer Sciences, University of Wisconsin-Madison, <http://www.cs.wisc.edu/~devise>*.
- [22] Plaisant, C., Shneiderman, B., Doan, K., Bruns, T., Interface and Data Architecture for Query Preview in Networked Information Systems, *Practice and Experience Paper of ACM TOIS, September 1998. A short early version also appeared in SIGMOD Record, Vol.26, No.1, pp. 75-81 March 1997, as Previews for Networked Information Systems: A Case Study with NASA Environmental Data*.

- [23] Jain, V., and Shneiderman, B., Data Structures for Dynamic Queries: An Analytical and Experimental Evaluation, *Institute for System Research (ISR) technical report T.R. 94-47. Other technical reports: CS-TR-3287, CAR-TR-715. Proc. of the workshop in advanced visual interfaces, AVI 94 (Bari, Italy, June 1-4, 1994).*