Dynamic Information Visualization

Yannis E. Ioannidis[*†]
Department of Computer Sciences, University of Wisconsin, Madison, WI 53706
yannis@cs.wisc.edu

Abstract

Dynamic queries constitute a very powerful mechanism for information visualization; some universe of data is visualized, and this visualization is modified on-the-fly as users modify the range of interest within the domains of the various attributes of the visualized information. In this paper, we analyze dynamic queries and offer some natural generalizations of the original concept by establishing a connection to SQL. We also discuss some implementation ideas that should make these generalizations efficient as well.

1 Introduction

Dynamic queries [1, 3, 6] have been introduced as a mechanism that achieves three goals:

- compared to using textual languages (e.g., SQL), users can query data in a directed-manipulation manner that shortens learning time, speeds performance, lowers error rates, and increases human retention over time;
- the query result is immediately visualized; and
- the queries are parameterized and the parameters can be modified dynamically, with the corresponding query results being modified accordingly as well.

The following example is used throughout this paper to illustrate the concepts introduced. Although there are several details or variations of dynamic queries that are not captured in this example, we believe that it is representative of their most important features. (For example, although some existing dynamic-query systems do capture disjunctive queries, we will concentrate on conjunctive queries only.) Consider the set of employees of a company, and assume that the attributes of interest are name, salary, age, and department. A typical display of this information and the corresponding control panel for dynamic queries may be as shown in Figure 1. The left side of the figure is occupied by a display of points in the salary-age space, each point representing an employee with the corresponding salary and age. The shape of each point represents the corresponding employee's department. The right side of the figure is occupied by double-dragbox sliders, one for each of the data attributes (name, salary, age, and department). By moving the left dragbox of a slider, users specify the minimum value of interest in the corresponding attribute. Similarly, its right dragbox indicates the maximum value of interest. As users adjust the dragboxes of some slider, points on the data display disappear or reappear, based on whether or not their corresponding attribute values fall within the specified range.

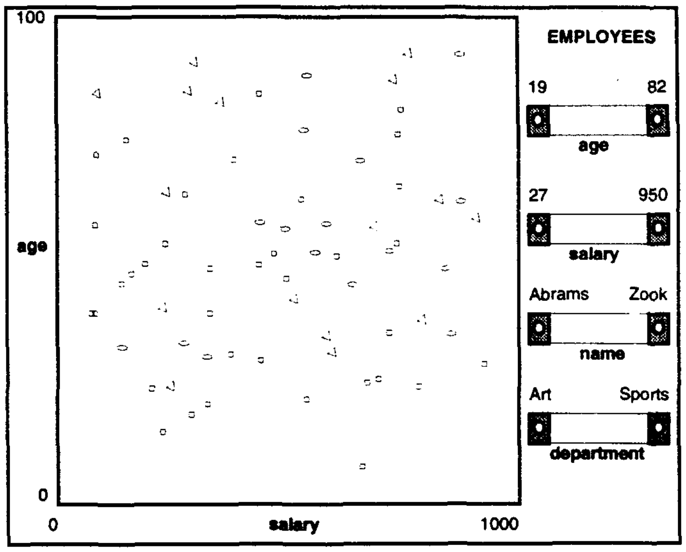

Figure 1: Typical display for dynamic queries

Using SQL as a vehicle to express the textual equivalents of the visual dynamic queries, one may view the query of Figure 1 as equivalent to the following family of queries:

```
select *
from EMP
where name between C1 and C2
  and age between C3 and C4
  and salary between C5 and C6
  and department between C7 and C8.
```

(Without loss of generality, we assume in the above that all data comes from a single relation EMP.) By moving the sliders, users instantiate the constant parameters C1-C8, and the entire set of attributes of the qualifying EMP tuples are visualized in the chosen fashion.

In this short paper, we introduce some slightly different conceptual interpretations of dynamic queries and, based on

[*] Partially supported by the National Science Foundation under Grants IRI-9224741 and IRI-9157368 (PYI Award) and by grants from DEC, IBM, HP, AT&T, Oracle, and Informix.

[†] We want to thank Richard Biegle and Ben Shneiderman for many useful comments on an earlier draft.

those, propose some generalizations. We also provide some thoughts on how these generalizations may be implemented by modifying the data structures that are typically used in existing dynamic-query systems.

2 Multiple Interacting Dynamic Queries

One can easily imagine situations where users need to explore via dynamic queries multiple universes of data, e.g., to identify correlations between their attributes. This presents no technical or conceptual difficulty for unrelated universes; in fact, in this case, there is no reason for having more than one of them on the screen at the same time. The interesting case is when between these universes there are conceptual connections that need to be explored. Then, as users manipulate the sliders associated with one such set, they would expect to see the visualizations of the other sets being affected as well. This requires that the result of a dynamic query be considered a proper, named data set itself that can, in turn, be used as input in another dynamic query as well. In technical terms, a dynamic query should be seen as essentially a *dynamic view* that is *visually materialized*.

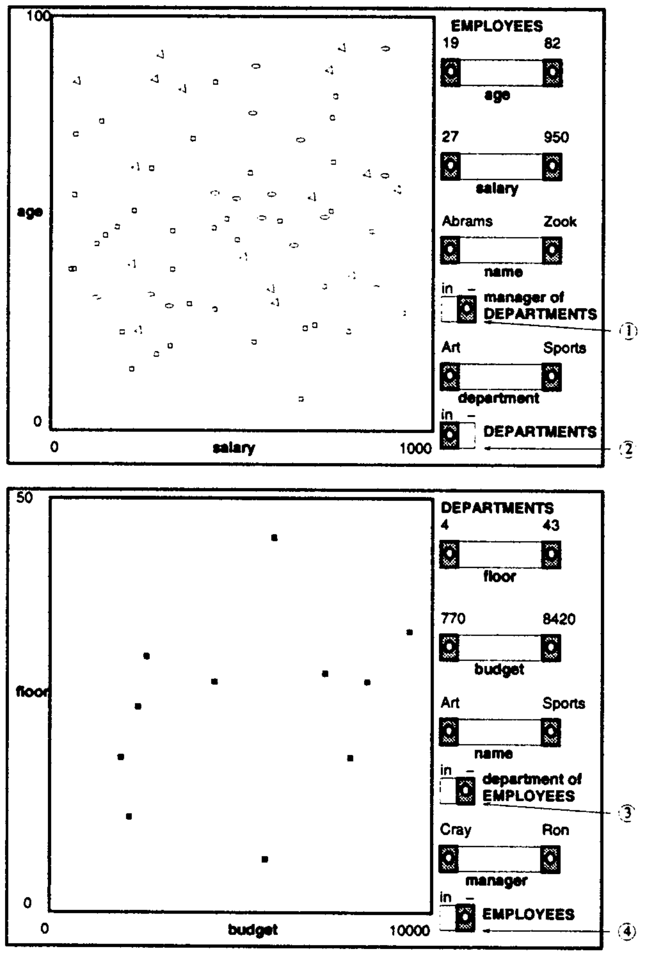

Figure 2: Possible display for multiple interacting dynamic queries

Continuing with our example, assume that in addition to a company's employees (relation EMP), our interests include the set of its departments (relation DEPT), with attributes the department's name, floor, budget, and manager. The two sets are naturally connected through the department of an employee and/or the manager of a department. Exploring these connections is achieved by defining two dynamic views, say EMPLOYEES and DEPARTMENTS, one of which may refer to the other through (semi-)joins. For instance, the definition of EMPLOYEES could be as follows:

```
create view EMPLOYEES as
  select *
  from EMP
  where name between C1 and C2
    and age between C3 and C4
    and salary between C5 and C6
    and department between C7 and C8
    and (C9 or (name in
          (select manager from DEPARTMENTS)))
    and (C10 or (department in
             (select name from DEPARTMENTS))).
```

In the above, C9 and C10 are boolean constants. Users can set each one to 'False' or 'True' to indicate whether or not EMPLOYEES and DEPARTMENTS are tightly coupled, respectively. C9 corresponds to the potential coupling on a department's name, while C10 corresponds to the potential coupling on a manager's name. Figure 2 shows one possible example of the corresponding dynamic visual setup (ignore the switches indicated by 3 and 4). Switches 1 and 2 correspond to the variables C9 and C10, respectively. Switch 1 is set to -, which corresponds to C9='True', indicating that the DEPARTMENTS dynamic view does not affect the one of EMPLOYEES through the manager's name connection (i.e., the displayed EMPLOYEES do not have to be managers of one of the displayed DEPARTMENTS). Switch 2 is set to `in`, which corresponds to C10='False', indicating that there is indeed coupling through the departments's name. Note that by presenting everything as a semijoin, these switches remain local to the area dedicated to employees like the attribute sliders, so their use should be rather intuitive.

The department name and manager name attributes can also be used in the opposite direction, i.e., to impose restrictions in the DEPARTMENTS view. In this case, switches 3 and 4 of Figure 2 should be present and switches 1 and 2 should be ignored. Setting all switches to `in` at the same time gives rise to mutually recursive queries, which are likely to be computationally expensive and have semantics that are confusing to most users. Therefore, we believe that interactions between dynamic views should always be in one direction. The only exception is when there is only one connecting attribute between the two sets. In that case, mutual coupling between two dynamic views presents no problem (in fact, what arises is *bounded recursion* [2]).

3 General Dynamic Views

As we have seen above, by looking at dynamic queries as dynamic views, one may define other views in terms of them. In that case, we say that the *dependent* view (e.g., EMPLOYEES) is defined in terms of the *determining* view (e.g., DEPARTMENTS). This can be applied at arbitrary levels, with dynamic views being defined in terms of dynamic views that are also defined in terms of dynamic views, etc, as long as no recursion is formed. As the parameters of some dynamic-view definition are modified (through slider actions), the visualizations of the results of all dependent views will potentially change as well. These visualizations may appear in different areas or superimposed, if that makes sense.

Dynamic-view definitions may in principle be arbitrary queries, each having its own possibly parameterized qualifications and even arbitrary target lists. For example, the sliders of dynamic queries are used as output devices as well: whenever the visualization changes, the dragboxes of all the sliders are adjusted to indicate the minimum and maximum values of the corresponding attributes among the data points that are currently displayed. Thus, the slider for, say, the age attribute can be seen as a visualization of the result of the following family of views built on top of EMPLOYEES:

> **select** min(age), max(age)
> **from** EMPLOYEES.

In addition to simple aggregates, other typical target lists may be functions of individual attribute values (e.g., arithmetic functions on numeric fields) or aggregate functions on attributes, all involving data related to a single universe. For example, on top of Figure 1 one may superimpose the average salary and age per department as solid-black points of the corresponding shape placed at the appropriate coordinates (Figure 3). These would correspond to the dynamic query

> **select** department, avg(age), avg(salary)
> **from** EMPLOYEES
> **group by** department.

4 Parameterized Functions

As mentioned above, one of the possibilities for dynamic views calls for target lists that involve functions of existing values assigning new values to one or more of the attributes of some elements. Depending on the application, there may be functions that are important and commonly occurring. Just as the qualifications of dynamic views/queries can be parameterized, in exactly the same way their target lists can be parameterized as well to generate a separate family of functions, each parameter being associated with a slider. Users can then explore the possible values of these parameter and observe the effect that the corresponding queries have on the relevant visualizations.

For example, the EMPLOYEES dynamic view may have the following target list:

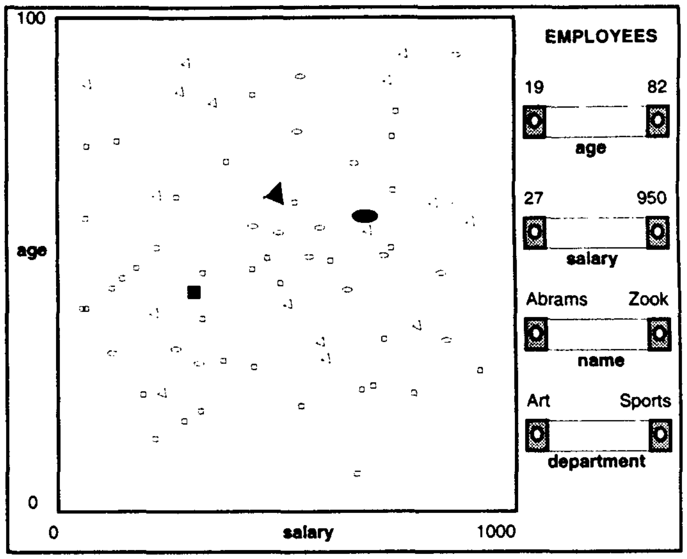

Figure 3: Salary-age space superimposed with averages per department

> **select** name,
> age = age + C11,
> salary = (1 + C12)*salary,
> department.

Associated with the age and salary attributes are increment and decrement buttons that correspond to the C11 and C12 parameters (Figure 4). Users can manipulate these buttons to

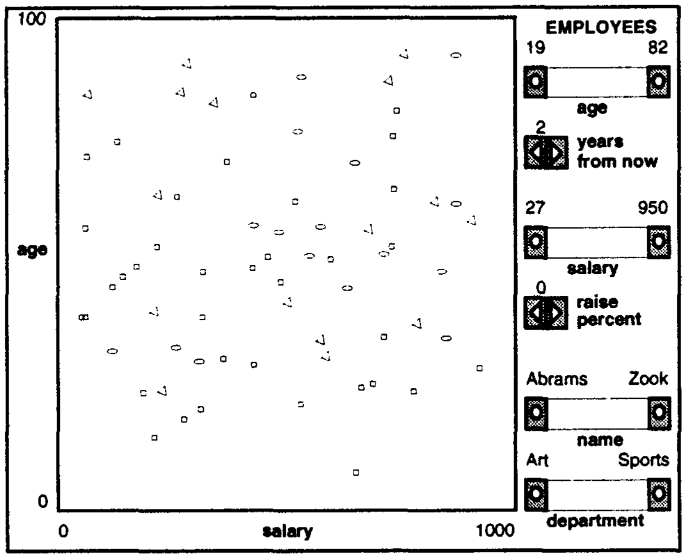

Figure 4: Salary-age space controllable through parameterized salary and age functions

specify what values should be visualized for these attributes of the qualifying tuples. (The buttons could have been replaced by sliders that can take only specific values if the application required such a restriction.) When users push on the increment (resp. decrement) button of age, all visualized points move up (resp. down) by the indicated amount. When users push on the increment (resp. decrement) button of salary, all visualized points move to the right (resp. left) by the indicated percentage. Moreover, all other relevant visualizations on the screen are affected accordingly.

5 Hypothetical (What-If) Updates

Motivated by the above example, we now discuss one more alternative conceptualization of some dynamic queries. In particular, we concentrate on dynamic queries that visualize existing objects (i.e., in SQL terms, queries whose target lists involve attributes of objects in the same relation) and not defined in terms of other dynamic views. In the traditional conceptualization of dynamic queries, the query result is continuously modified as the sliders of the relevant attributes are manipulated for specifying what is to be visualized. In an equivalent view of the process, the database contents are continuously *hypothetically* modified as the sliders of the relevant attributes are manipulated for specifying what is not to be *hypothetically* deleted (or reinserted).

For example, in this approach, instead of the query given in Section 1 above, Figure 1 corresponds to the following SQL statement (we take the liberty to use the keyword **hypothetical** in front of updates to emphasize that no actual database modification takes place and also the keyword **retain in** to denote the complement of **delete from**[1]):

> **hypothetical retain in** EMPLOYEES
> **where** name **between** C1 **and** C2
>   **and** age **between** C3 **and** C4
>   **and** salary **between** C5 **and** C6
>   **and** department **between** C7 **and** C8.

This now allows us to introduce all other forms of hypothetical database modification. In particular, we may take dynamic queries whose target lists include functions on attribute values (Figure 4) and treat those as *hypothetical updates*. While in the traditional interpretation, nonqualifying objects are *hypothetically* deleted, in this interpretation, they are still visualized, but retain their original values in all their attributes (while those qualifying have their corresponding attribute values changed based on the update functions). Thus, users can choose one of two different modes (delete or update) and, with essentially very similar actions, perform two rather different visual explorations of the data.

6 Implementation Thoughts

One of the primary reasons for the success of current dynamic-query systems is their efficiency. User actions on the sliders are immediately reflected in the visualization of the corresponding set (response times in the order of 100ms [6, 7]). A crucial question that arises is how efficiently can the generalizations mentioned above be implemented. We will only discuss the issue of joins, i.e., visualizations of multiple tightly coupled universes (Figure 2). In particular, we will show how to adapt one of the currently used data structures [4] (the Grid File for disk-based data or Grid Array for memory-resident data) for visualizing individual universes so that it can be used for multi-universe visualizations. For the most part, all current systems assume that all data resides in memory, so we also make that assumption and concentrate on Grid Arrays. The essence of our discussion, however, is not affected by this assumption, and our adaptations are equally valid for Grid Files as well.

Grid arrays/files are quite well known [5], so we do not describe them in any detail. For dynamic queries on an individual universe (Figure 1), the grid has as many dimensions as there are attributes in the universe, e.g., that for employees will have four dimensions. Every bucket of the grid captures a hyper-rectangle of the four-dimensional space and includes a list of pointers (or range of pointers, if clustered) to the records whose values fall within the ranges corresponding to the bucket on all dimensions. A simple example from a two-dimensional space (e.g., salary and department) is shown in Figure 5 (essentially copied from [4]).

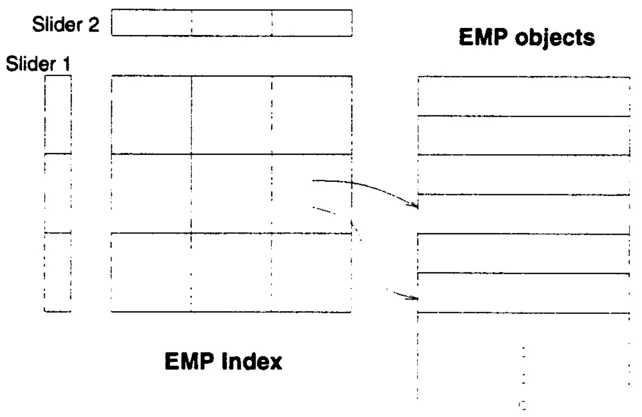

Figure 5: Grid array for two-dimensional space of employees

To manage two universes that are tightly coupled through one attribute (generalizing to multiple attributes is straightforward, as long as no recursion arises), their grid arrays remain intact, each based on the attributes that can be manipulated by their corresponding sliders. The actual object arrays are enhanced with the following information. First, every object of the determining universe points to all the objects of the dependent universe that it is associated with. Returning to our example, consider the EMPLOYEES dynamic view and assume that the only potential connection between EMPLOYEES and DEPARTMENTS is through the department's name. Then, the proposed data structure will have every DEPT object pointing to all the EMP objects with the same department name (Figure 6). For elements that are dangling (e.g., departments with no employees), there will be no

[1]That is '**retain in** R **where** Q' is equivalent to '**delete from** R **where not** Q'.

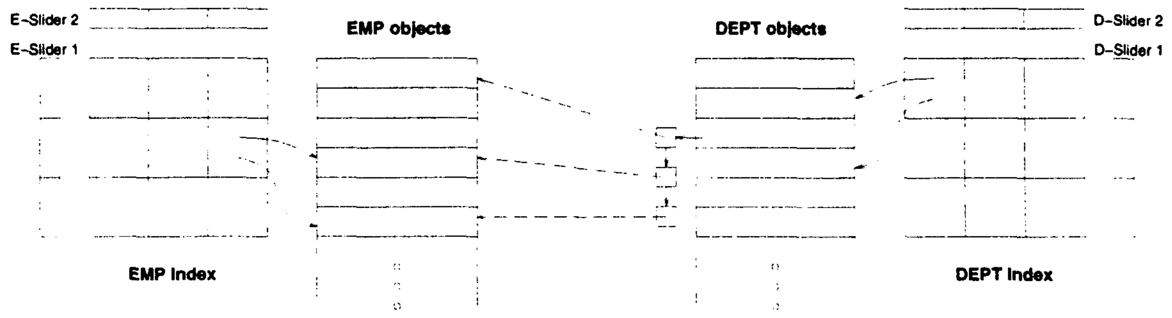

Figure 6: Connection of grid array for spaces of employees and departments

such pointer.

Second, every object of the dependent universe stores the number of currently visible objects in the determining universe that are associated with it. (This is not shown in the figure; it would be one additional field in the EMP objects, whose value could be 1 or 0, since every employee is associated with at most one department.)

The above data structures should make the interaction between the two dynamic views (actually any two dynamic views that are tightly coupled) rather efficient. For example, any modification on the DEPARTMENTS sliders already visits all the DEPT objects that are to disappear or reappear in the DEPARTMENTS visualization. Now, their pointers to EMP objects will have to be dereferenced as well so that the corresponding EMP objects also disappear or reappear (the latter if the corresponding employees satisfy all the other conditions on the employee sliders). In the process, the additional field on EMP objects mentioned above will be updated, so that subsequent manipulations of the EMP sliders can immediately check to see if the appropriate EMP objects are visible or not with respect to their connections with visible DEPT objects.

Finally, note that in the above example, there is a functional dependency from the dependent universe to the determining universe (more precisely, from employees to the attributes of departments). In that case, an alternative implementation is to build an index on the dependent universe (EMP) that involves the attributes of both universes. (Unless there is also a referential integrity constraint, the domain of the determining attributes should include the null value.) Then, any manipulation of the DEPARTMENTS sliders would trigger the same processing actions on two indices, the one on EMP and the other on DEPT. In this approach, there is no need for the direct object-to-object pointers mentioned earlier. Which of the two approaches is more efficient is an interesting open question that we plan to investigate in the future.

In the absence of the appropriate functional dependencies, the second approach cannot be directly applied. It would require that duplicates of the dependent objects be maintained, or at least multiple pointers from the index to the same object. For example, if DEPARTMENTS depended on EMPLOYEES, every department would have to be replicated once for each employee in it. This duplication is likely to hamper performance, something that we also intend to investigate in an upcoming study.

7 Conclusions

We have presented some new conceptualizations of dynamic queries, and based on these, we have described some generalizations. We have argued that the resulting interfaces should be rather intuitive to users and have also discussed informally how they can be implemented so that response time remains within acceptable limits. We intend to implement the above ideas and test their effectiveness and efficiency.

References

[1] C. Ahlberg and B. Shneiderman. Visual information seeking: Tight coupling of dynamic query filters with starfield displays. In *Proc. CHI94 Conference on Human Factors in Computing Systems*, pages 313–317, Boston, MA, April 1994.

[2] Y. Ioannidis. A time bound on the materialization of some recursively defined views. *Algorithmica*, 1(4):361–385, October 1986.

[3] IVEE Development AB, Goteborg, Sweden. http://www.ivee.com, June 1996.

[4] V. Jain and B. Shneiderman. Data structures for dynamic queries: An analytical and experimental evaluation. In *Proc. 2nd International Workshop on Advanced Visual Interfaces*, Bari, Italy, May 1994.

[5] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.

[6] B. Shneiderman. Dynamic queries for visual information seeking. *IEEE Software*, 11(6):70–77, November 1994.

[7] E. Tanin, R. Beigel, and B. Shneiderman. Incremental data structures and algorithms for dynamic query interfaces. *ACM-SIGMOD Record*, 24(4), December 1996.