The Sequoia 2000 Object Browser

Jolly Chen, Ray Larson, and Michael Stonebraker

Sequoia 2000 Technical Report 91/4
University of California
Berkeley, Ca.

Abstract

In this paper we explain the paradigm that we are following for Sequoia 2000 object browsers. It is intended to be a keyboard-free interface, and is based on the "move and zoom" paradigm popularized for Navy ships by SDMS [HERO80].

1. Introduction

As noted in [STON92], the Sequoia 2000 project seeks to provide the concept of **bottomless storage** for Global Change researchers. These clients wish to store large numbers of disparate objects in a storage system, including satellite imagery, aerial photographs, polygonal maps including topographic maps, computer programs, documents, output of simulation runs, etc. In all, we expect data set size to approach 100 Tbytes on the server we are constructing, aptly termed **BIGFOOT.**

In many cases, researchers simply need a storage system into which they can place objects and then reliably retrieve them at a later time. Such **production** use of a storage system typifies the access provided by file systems and Data Base Management Systems (DBMS). Here, retrieval is assured by knowing exactly what is wanted and then obtaining it using file system or DBMS commands.

However, a large repository such as BIGFOOT will also be accessed by casual or naive users. Such users will typically want to **browse** the repository to locate items in which they might be interested. Location of repository objects has points in common with information retrieval (IR) techniques that have been developed over the last 30 years to access bibliographic databases and electronic card catalogs for libraries. In traditional IR applications, each document is indexed by an expert, who specifies a set of **index terms** from a **controlled vocabulary** (such as the **Library of Congress Subject Headings,** or a **thesaurus** of **authorized** index terms) which describe the subject material of the document. In online IR systems these manually assigned index terms are supplemented with **keywords** automatically extracted from the titles, abstracts, and (when available) the full text of the document. Commonly, the individual words of controlled index terms (which may be descriptive phrases or classification numbers) are also treated as keywords.

A user who wishes to find documents in a collection that might be relevant to his interests, specifies his request by indicating a Boolean combination of search criteria on specific fields, e.g:

author = "Knuth" and (subject = "algorithms" or subject
= "typesetting")

Systems with this sort of **Boolean query language** include commercial bibliographic search services, such as DIALOG and BRS, and online library catalogs like the MELVYL system at the University of California [DLA87].

More advanced retrieval methods have been developed that replace Boolean logic with more sophisticated matching techniques. These techniques, such as the *vector space* and *probabilistic* models of IR attempt to rank the documents in the database in order of their similarity, or probability of relevance, to a given natural language query [SALT91]. Systems using these methods include I3R [CROF87], CITE [DOSZ82] and CHESHIRE [LARS91].

In the Sequoia environment, there are two problems with the traditional IR paradigm:

1) Traditional IR depends on the the existence of a human (or a program) to **index** the document by specifying a collection of **keywords** It is not clear how to specify such keywords about a map, for example.

2) Not all Sequoia objects will be text documents. Many other kinds of objects must be stored. It is not clear how to apply the traditional IR paradigm to computer programs, for example.

For these reasons, we have taken a much more general approach to repository access, which we now describe. Since repository objects reside in a POSTGRES data base, we make use of several of the novel features provided by POSTGRES [STON90, KEMN91]. Hence, in Section 2, we briefly describe features of POSTGRES that are relevant to our browser, which is described in Sections 3 and 4.

2. Postgres constructs

Traditional relational DBMSs support a data model consisting of a collection of named relations, each attribute of which has a specific type. In current commercial systems possible types are floating point numbers, integers, character strings, money, and dates. It is commonly recognized that this data model is insufficient for future data processing applications, and we now describe a more general data model and query language.

The fundamental notion in POSTGRES is that of a **class,** which is a named collection of **instances** of objects. Each instance has the same collection of named **attributes** and each attribute is of a specific **type.** Moreover, each instance has a unique (never-changing) identifier (OID).

A user can create a new class by specifying the class name, along with all attribute names and their types, for example.

create EMP (name = c12, salary = float, age = int)

A class can optionally **inherit** data elements from other classes. For example, a SALESMAN class can be created as follows:

create SALESMAN (quota = float) inherits EMP

In this case, an instance of SALESMAN has a quota and inherits all data elements from EMP, namely name, salary and age.

POSTGRES contains an extensive type system and a powerful notion of functions. There are three kinds of types in POSTGRES, base types, arrays of base types, and composite types, which we discuss in turn.

Some researchers, e.g. [STON86, OSBO86], have argued that one should be able to construct new **base types** such as bits, bitstrings, encoded character strings, bitmaps, compressed integers, packed decimal numbers, radix 50 decimal numbers, money, etc. Unlike many next generation DBMSs which have a hard-wired collection of base types (typically integers, floats and character strings), POSTGRES contains an **abstract data type (ADT)** facility whereby any user can construct arbitrary new **base** types. Consequently, it is possible to construct a class, DEPT, as follows:

```
create DEPT (dname = c10, manager = c12, floorspace
    = polygon, mailstop = point)
```

Here, a DEPT instance contains four attributes, the first two have familiar types while the third is a polygon indicating the space allocated to the department and the fourth is the geographic location of the mailstop.

Arrays of base types are also supported as POSTGRES types. Therefore, if employees receive a different salary each month, we could redefine the EMP class as:

```
create EMP (name = c12, salary = float[12], age = int)
```

Arrays are supported in the POSTQUEL query language using the standard bracket notation, e.g:

```
retrieve (EMP.name) where EMP.salary[4] = 1000
```

Composite types allow an application designer to construct **complex objects,** i.e. attributes which contain other instances as part or all of their value. Hence, complex objects have a hierarchical internal structure, and POSTGRES supports two kinds of composite types. First, zero or more instances of any class is automatically a composite type. For example, the EMP class can be redefined to have attributes, manager and co-workers, each of which holds a collection of zero or more instances of the EMP class:

```
create EMP (name = c12, salary = float[12], age = int,
    manager = EMP, co-workers = EMP)
```

Consequently, each time a class is constructed, a type is automatically available to hold a collection of instances of the class.

In the above example manager and co-workers have the same structure for each instance of EMP. However, there are situations where the application designer requires a complex object which does not have this rigid structure. To accomodate this need, POSTGRES supports a final constructed type, **set,** whose value is a collection of instances from all classes.

POSTGRES also contains a powerful function facility, and details of its capabilities appear in [KEMN91]. Here, we merely illustrate one kind of POSTGRES function, namely C functions.

A user can define an arbitrary number of **C functions** whose arguments are base types or composite types. For example, he can define a function, area, which maps an instance of a polygon into an instance of a floating point number. Such functions are automatically available in the query language as illustrated in the following query which finds the names of departments for which area returns a result greater than 500:

```
retrieve (DEPT.dname) where area
    (DEPT.floorspace) > 500
```

C functions can be defined to POSTGRES while the system is running and are dynamically loaded when required during query execution. Because C functions are arbitrary C procedures. they can run arbitrary POSTQUEL commands during execution.

3. The Sequoia 2000 repository

As noted in the previous section, POSTGRES supports a wide variety of data types, which may be assembled into classes. In addition, nearly arbitrary functions may be defined on such types and classes. Using this data model, we have designed class definitions for popular repository objects. In this section we discuss the class definitions for documents and maps to illustrate the concept.

There are three levels or layers of POSTGRES classes in the repository schema, these are:

Object layer - contains the classes with actual objects, such as text documents, images, maps, video segments, etc. Any POSTGRES class in the repository database may also be treated as an object.

Description layer - contains the classes which **describe** objects on the object layer. It consists of a "catalog" of repository objects in addition to the normal POSTGRES **metadata** describing classes, attributes, rules, ADTs, functions, etc.

Access layer - contains the classes used to index or provide access to objects on the object and description layers. It includes structured sets of attributes (such as thesaurii) and keyword indexes.

The repository schema is intended to be user-extensible and capable of accommodating virtually any form of document. Moreover, POSTGRES inheritance can be freely used to allow subclasses to inherit the characteristics of more general repository classes. For example, a "troff document" might inherit the characteristics of a "markup language document", which in turn inherits the characteristics of a "text document".

In this section we will briefly discuss the general document **(DOC)** class, the text document **(TEXTDOC)** class, and the map **(MAP)** class. Figure 1 shows the attributes of the DOC class. This class is an abstraction of the characteristics expected to be shared by any object in the repository, and is therefore the "lowest common denominator" of object representation. The general DOC class includes information about the type of document **(doctype),** and links to more specific information about the particular internal format of the stored document **(format).** The DOC class also includes a user-defined procedure **(displayproc)** for rendering the document on a workstation screen. Any instance in the DOC class may be linked to descriptive information about the particular document in the main catalog **(catentry).** Lastly, the DOC class provides storage for the actual document as a POSTGRES **large object.**

```
create DOC (

doctype = int4,              Code for type of document (text, map,
                             image, video, etc.)
catentry = CATALOG,          Link to an object description in the
                             main catalog
format = oid,                Link to a class representing the internal
                             format of documents (TeX, troff, etc.)
displayproc = FUNCTION,      A user-defined POSTGRES function
                             that will render the document
relateddocs = set,           A function returning a set of related
                             documents for this document
                             (such as separate illustrations
                             accompanying a text document)
document = large_object,     The actual document contents
)
```

The Command to Create the DOC Class
Figure 1

At the next, more specific, level of abstraction the repository documents are grouped into subclasses of DOC that share certain general characteristics (based primarily on their content). The TEXTDOC class is one of these classes which is appropriate for textual material. Figure 2 shows the create statement for this class:

```
create TEXTDOC (

plaintext = FUNCTION,        A function to strip formatting and
                             convert to plain ASCII text
docvector = DOCVECTOR,       Link to the term-frequency weight
                             vector for this text document in
                             the DOCVECTOR class in the access
                             layer.
)
inherits DOC
```

The Command to Create the TEXTDOC Class
Figure 2

The primary feature of this class is a function that takes the raw document as its argument and returns a stream of plain ASCII text, with any markup language or format-specific data removed. The results of applying this function are used in building the keyword indexes and vectors that provide access to the contents of the text document. Whenever an insertion is made to the TEXTDOC class, a rule [STON90B] is defined which will cause this indexing to occur. The **docvector** attribute points to the frequency weighted vector representation of the text document. This is used in calculating similarity coefficients between documents and between queries and documents for use in retrieval and relevance feedback operations.

The **MAP** class is the parent for a set of specific classes that contain geographic information, as noted in Figure 3. This class contains the most general descriptive and functional information for all types of maps and spatial data files stored in the repository. Descriptive elements include the **scale** of the map, the **projection** (i.e., how the curved surface of the Earth is translated systematically onto a flat surface for depiction). The MAP class also records the coordinate system used in the map data (such as Latitude/ Longitude coordinates or Universal Transverse Mercator (UTM) coordinates) and the **area** covered by the particular map, which may be an arbitrary polygon in the UTM coordinate system.

The **maptext** function extracts textual elements in the map data (such as titles, place and region names) and returns ASCII text. This text is used in indexing, in conjunction with authoritative lists of geographic names, to provide additional access to the map.

The MAP class is inherited by classes for particular types of map representation (e.g. vector or raster), and specific map data formats. These include GRASS (Geographical Resources Analysis Support System) raster and vector maps, USGS Digital Elevation Maps (DEM), and other map data formats.

```
create MAP(

scale = float4,              scale of map
projection = text,                   type of map projection (Mercator, etc.)
coordsys = text,             The coordinate system used in the map.
                             (e.g., latitude/longitude)
area = geoarea,              coverage of map using the geoarea type
                             (e.g. a polygon of longitude and latitude)
maptext = FUNCTION, User-defined POSTGRES function to
                             strip words from map data)
)
inherits DOC
```

The Command to Create the MAP Class
Figure 3

4. The Sequoia 2000 repository browser

4.1. Introduction

As noted in Section 1, we wish a browsing paradigm that is appropriate for a wide range of repository objects. Moreover, we wish to avoid the traditional "type a query of keyterms" interface popular in information retrieval environments. As a result, we are focusing on a "move and zoom" interface that allows a user to "navigate" around in an N-dimensional space with little or no typing. In Section 4.2 we present our thoughts on the user interface. Then, in Section 4.3 we formalize our browser notions. Section 4.4 continues with our approach to eliminating "clutter" in Sequoia browsers. Lastly, in Section 4.5 we present example browsers that can be quickly built using our paradigm.

4.2. Browser user interface

The browser will run on a high resolution, graphical user interface that enables the user to perform most operations by direct manipulation. While the keyboard will still be available for input, we believe that a **keyboard-free** interface will be more intuitive and easier to use for casual or naive user who will be **browsing** the repository instead of **searching** for a specific object. Our interface assumes an underlying space of n dimensions and will display a two or three-dimensional subspace of the n-dimensional space at any one time. For example, browser dimensions might include latitude, longitude, and time, and a two dimensional presentation of longitude and latitude might be visible on the screen. In addition, all dimensions have a background **landscape** which will give the browsing user visual clues about the dimension. For example, the background landscape for latitude and longitude might be a topographic map of the United States.

All repository objects are located in this n-dimensional space using a mechanism described in Section 4.3. Intuitively, a map might be associated with a polygon of latitude and longitude indicating its area of coverage and a point in time indicating its date of publication. This polygon is then the location of the map in 3-space.

Repository objects are represented by icons and projected onto the subspace which is visible on the screen. For example, a map object might be represented by a suitable icon, and a given map would have its three dimensional icon projected into two dimensions for screen display. If the screen should become too cluttered by a large number of icons, the system automatically attempts to reduce clutter by aggregation, as indicated in Section 4.4.

The user can browse the repository by selecting relevant dimensions to form the subspace and then moving across the background landscape for that subspace with onscreen panning controls. The user may zoom into regions of interest and may **pick** particular repository objects to reveal greater detail.

The user is not limited to traversing the currently viewable dimensions. Other dimensions will also be displayed on the screen as linear slider bars. The user is then able to browse those dimensions by moving the controls on the slider bars. In addition, the user may **filter** out intervals in a dimension by manipulating the slider bars to select various intervals of interest. As an example, if the current viewable dimensions are longitude and latitude, then the time dimension would be indicated by a slider. By moving graphical controls on the slider bar, the user may elect to view only those data objects within a certain time range. In addition, by moving the slider through the time dimension, the user may **animate** data objects by **fast-forwarding** or **rewinding** them through time.

By using the slider bars as filters, by changing the dimensions currently visible on the screen, and by panning across the landscape the user will be able to easily browse through the repository and focus on areas or items of interest. Then, he can zoom into such items to reveal detail, ultimately resulting in a presentation of the complete object.

4.3. Browsing Paradigm

Our browsing paradigm is a mechanism by which a skilled user can construct a browser. Our paradigm requires a predefined data type, pixel, which is a displayable pixel. As noted in Section 2, arrays of pixels are automatically defined. Hence, we will denote an n dimensional array of pixels, by SCREEN-n.

Each browser can then be defined by specifying:

1) A collection of classes. Each browser is capable of accessing the instances in a collection of classes in the data base. This collection must be specified by the person constructing the browser. Such classes can be specializations of the repository classes discussed in the previous section or they can be arbitrary user-defined POSTGRES classes.

2) A collection of POSTGRES data types, T1, ..., Tn, which make up the **dimensions,** in which objects can be displayed. For example, a browser could be defined on the dimensions, longitude and latitude.

3) A data type, T, which is semantically a polygon in the n-space defined by the collection of dimensional data types.

4) For the data type T, the user must provide three functions which respectively compute the **size** of an instance of T, the **intersection** of two instances of T and the **union** of two instances of T. These functions are required to index objects in a manner to be presently described.

5) A value of type, T, which specifies the region of interest, for example the United States.

6) The identity of a map in the repository. This map forms a background **landscape** for the browser. For example, it might be a map of the United States. This map has a co-ordinate system identical to the one of the browser. Moreover, given any value of type T, the region of the map that intersects T can be efficiently found. Details of the multiple Sequoia map representations are beyond the scope of this paper.

7) A **rendering** function, R, which takes an argument of type T and returns a value of type SCREEN-n. This function is responsible for choosing the display characteristics for the various dimensions.

8) For each class in 1) above, the user must provide a function FF which takes an instance of the class as an argument and returns a value of type, array of T. This is the **locator** function which will determine the position of the class instance in the browser n-space. Our paradigm allows FF to return an array of locations to allow an object to be positioned multiple times in n-space. For example, a object corresponding to the University of California could be positioned at each of the 9 campuses.

9) For each class, an **icon** must be defined. This icon is a function, I, which takes three arguments, a class instance, a value of type T, and an integer i, and returns a value in SCREEN-i. The

reason for the class argument is to allow the icon to be customized for the particular class instance. For example, the icon could be a picture of an employee or it might display his age. Such data elements must be obtained from each class instance. The other two arguments allow the icon to be **scaled** to the correct location and number of dimensions.

10) For each class a **display** function must be provided. If the user **picks** an object, this function can display relevant information in a separate window. Hence, it is a function which takes a class instance as an argument and produces a result of type SCREEN-2.

Any given browser is now straightforward. Each object in each class is mapped into one or more polygons in the n-space of the browser by applying the locator function discussed above. POSTGRES supports user-defined access methods, and we have used this facility to construct an R-tree index [GUTM84], which efficiently supports multidimensional spatial search. Hence, an R-tree index is built on the result of the locator function. To find the objects that should appear on the screen, this index is searched to find the objects which intersect the region of n-space that corresponds to the screen. The landscape is projected and clipped to the screen dimensions. All visible objects are then turned into icons and superimposed on the landscape, and the result displayed,

The operations defined in Section 4.2 are all straightforward to implement by applying the above algorithm to define the current contents of the screen. If an object is picked, it is easy to identify the actual object and display additional information using the display functions mentioned above.

4.4. Dealing with clutter

A potential difficulty with using the browser is the large number of icons that will be presented to the user on the screen. As more objects are added to the repository, the problem of **clutter** will become more serious. We aim to reduce clutter through the automatic **aggregation.** Clutter arises from having too many icons in a given area of screen. We can alleviate this problem somewhat aggregrating icons into groups. Distinguishable **set icons** will represent a set of objects instead of a single object. Set icons will be labeled with some characteristics of the set such as the size of the set. The user may view the contents of a set by selecting the set icon. Aggregation will be done automatically to maintain an tolerable level of clutter. Different users may have different tolerances, e.g. expert users can often deal with more clutter than novice users. Various methods of aggregation are possible. One way is to group objects by class so that objects of the same class that are near to one another or represented by a single icon. Another method of aggregation is by proximity -- every object within a certain vicinity is lumped together into a group. Still another aggregation method is by logical viewgroups. For example, objects of different types may be of logical interest when they are near each other. If so, they should be aggregated. We will investigate the use of these and other aggregation methods to reduce clutter.

4.5. Example browsers

One browser we are currently constructing uses latitude and longitude as dimensions. The browser places all browsing objects into this geographic space The background landscape is a topographic map. As such, this browser has points in common with Image Query and with SDMS [HERO80].

We will also be developing a set of experimental browsers for text documents. Some interesting work on direct manipulation interfaces and visual abstractions for conventional

Boolean systems has been reported by Shneiderman [SHNE91], and for more advanced IR systems by Croft [CROF87], but we intend to take a different approach based on the browsing paradigm discussed above.

Obviously, the geographic browser just described can also be used to browse text documents that describe a particular place or region by having the locator function place an "icon" representing the document in the region described. However, text documents may lack such conventional "dimensions" as latitude and longitude. Instead, automatic clustering methods and document similarity measures developed in the vector space and probabilistic models of information retrieval [SALT91] might be used to present 2 or 3 dimensional "slices" of an n-dimensional document space. In such a representation each document, or surrogate cluster "centroids" for a set of sufficiently similar documents would be represented by icons. The similarity between documents would be indicated by how close together their icons are on the screen.

In this space, particular clusters of documents might be labelled by some small set of keywords that occurred frequently within that cluster, but rarely in the rest of the document. The user would be able to pan over and zoom in on individual clusters, which would eventually narrow the search space from the entire collection down to a few closely similar documents.

4.6. References

[CROF87] Croft, B. and Thompson, R. ‘‘I3R: A New Approach to the Design of Document Retrieval Systems.’’ *Journal of the American Society for Information Science* (Nov. 1987).

[DLA87] Division of Library Automation. *MELVYL Online Catalog Reference Manual.* (Berkeley, CA: Division of Library Automation, University of California, 1987).

[DOSZ82] Doszkocs, T. ‘‘From Research to Application: The CITE Natural Language Information Retrieval System.’’ *Research and Development in Information Retrieval: Proceedings, Berlin, 1982.* (New York, Berlin: Springer Verlag, 1983)

[GUTM84] Gutman, A., ‘‘R-trees: A Dynamic Index Structure for Spatial Searching,’’ Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass. June 1984.

[HERO80] Herot, C., ‘‘SDMS: A Spatial Data Base System,’’ *ACM TODS* (1980).

[KEMN91] Kemnitz, G. (ed.), ‘‘The POSTGRES Reference Manual, Version 3.0’’ Electronics Research Laboratory, University of California, Berkeley, CA, Report M91/10, November 1991.

[LARS91] Larson, R., ‘‘Classification Clustering, Probabilistic Information Retrieval and the Online Catalog,’’ *Library Quarterly*, vol. 61 (April 1991).

[OSBO86] Osborne, S. and Heaven, T., ‘‘The Design of a Relational System with Abstract Data Types as Domains,’’ ACM TODS, Sept. 1986.

[SALT91] Salton, G., ‘‘Developments in Automatic Text Retrieval,’’ *Science* Aug. 30, 1991.

[SHNE89] Shneiderman, B., et al. ‘‘Three Evaluations of Museum Installations of a Hypertext System,’’ *Journal of the American Society for Information Science* (May 1989). ,ip [SHNE91] Shneiderman, B. ‘‘Visual User Interfaces for Information Exploration,’’ *Proceedings of the 54th ASIS Annual Meeting, Washington, DC, Oct. 1991.*

[STON86] Stonebraker, M., ``Inclusion of New Types in Relational Data Base Systems,´´ Proc. Second International Conference on Data Engineering, Los Angeles, Ca., Feb. 1986.

[STON90] Stonebraker, M. et al., ‘‘The Implementation of POSTGRES,’’ *IEEE Transactions on Knowledge and Data Engineering* (March 1990).

[STON90B] Stonebraker, M., et. al., "On Rules, Procedures, Caching and Views," Proc. 1991 ACM-SIGMOD Conference on Management of Data, Atlantic City, N.J., June 1990.

[STON92] Stonebraker, M., "An Overview of the Sequoia 2000 Project," (elsewhere in this proceedings).