Office-by-Example: An Integrated Office System and Database Manager

KYU-YOUNG WHANG, ART AMMANN, ANTHONY BOLMARCICH, MARIA HANRAHAN, GUY HOCHGESANG, KUAN-TSAE HUANG, AL KHORASANI, RAVI KRISHNAMURTHY, GARY SOCKUT, PAULA SWEENEY, VANCE WADDLE, and MOSHÉ ZLOOF

IBM Thomas J. Watson Research Center

Office-by-Example (OBE) is an integrated office information system that has been under development at IBM Research. OBE, an extension of Query-by-Example, supports various office features such as database tables, word processing, electronic mail, graphics, images, and so forth. These seemingly heterogeneous features are integrated through a language feature called *example elements*. Applications involving example elements are processed by the database manager, an integrated part of the OBE system. In this paper we describe the facilities and architecture of the OBE system and discuss the techniques for integrating heterogeneous objects.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*parsing*; D.4.1 [**Operating Systems**]: Process Management—*concurrency*; E.2 [**Data Storage Representations**]: H.1.2 [**Models and Principles**]: User/Machine Systems—*human factors*; H.2.0 [**Database Management**]: General—*security, integrity, and protection*; H.2.1 [**Database Management**]: Logical Design; H.2.2 [**Database Management**]: Physical Design—*access methods; recovery and restart*; H.2.3 [**Database Management**]: Languages—*query languages*; H.2.4 [**Database Management**]: Systems—*query processing*; H.4.1 [**Information Systems Applications**]: Office Automation—*word processing*; H.4.3 [**Information Systems Applications**]: Communications Applications—*electronic mail*; I.3.6 [**Computer Graphics**]: Methodology and Techniques—*interaction techniques*; *languages*; I.4.0 [**Image Processing**]: General—*image processing software*; I.7.2 [**Text Processing**]: Document Preparation—*languages*

General Terms: Algorithms, Languages, Management, Performance

Additional Key Words and Phrases: Integration, memory-resident database, query optimization, screen management, two-dimensional parsing

1. INTRODUCTION

Office automation is an application of computer and communication technology that helps people working with various types of information. Its prime purpose is to help people work more efficiently and effectively. Recently, owing to rapid advances in computer technology and a sharp decline of hardware costs, the

Authors' current addresses: K.-Y. Whang, A. Ammann, A. Bolmarcich, M. Hanrahan, G. Hochgesang, K.-T. Huang, G. Sockut, P. Sweeney, and V. Waddle, IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, N.Y. 10598; A. Khorasani, IBM, P.O. Box 790, Poughkeepsie, NY 12602; R. Krishnamurthy, M.C.C., 9430 Research Blvd., Austin, TX 78759; M. Zloof, M. M. Zloof, Inc., 186 Clinton Ave., Dobbs Ferry, NY 10522.

trend in office automation has been converging toward *integration*, the combining of different types of information, functions, and hardware devices. The term *integrated office system* has been used quite commonly during the last few years. Although its concept has not yet been clearly defined, it is certain that an integrated office system should promise improvements in communication among people, among different systems, and among different applications.

We classify integrated office systems into four categories on the basis of their levels of integration. The first category includes an office system, called a window system, that allows users to run different application programs together in the same environment, but in separate windows. Such a system runs on top of an operating system with its own interface to each application program. An example of a system in this category is IBM's TopView [26]. In some sense, this type of system is not truly integrated because it is merely a collection of application programs.

The second category involves a system that loosely integrates different applications by providing the means of exchanging information. Such an exchange, however, requires translation between different data formats and possibly interactions with the file system. IBM's DISOSS [21] and PROFS [22] fall into this category.

The third category is characterized by a tightly integrated system that supports common data structures for information exchange, as well as a common operating environment. Information can be exchanged between different applications by simply copying it, but without the need for changing contexts. (We call this type of integration *surface integration*.) Xerox's Star [37], Apple's Macintosh [2], Lotus's Symphony [33], and Lotus's 1-2-3 [32] belong to this category.

The last category encompasses tightly integrated systems that achieve integration through a database management system rather than through simple common data structures that enable surface integration. In such systems, different applications can exchange information by using the full expressive power of the database language supported by the database management system. Thus, a system in this category is no longer a collection of individual applications. Instead, it provides a set of primitives, upon which many applications can be built. In this sense, the language supported by the system characterizes general office applications.

Office-by-Example (OBE), an integrated office system that has been under development at IBM Research, belongs to the last category. OBE extends the concept of Query-by-Example (QBE) [43], a relationally complete database language [8, 9]. It supports various features needed in a typical office environment: database tables, text processing, electronic mail, menus, forms,[1] graphics, and images. More features may be added without much difficulty. Moreover, OBE integrates these seemingly heterogeneous features through a language feature called *example elements*. In applications involving example elements, different features are mapped and processed by the database manager, which is an integrated part of the OBE system.

[1] Currently, OBE provides only hard-wired forms. Interactive form specification facilities such as in [36] and [42] are not provided.

The database manager constitutes the backbone of the entire OBE system, providing all processing needs for complex applications. Since the database manager provides most of the computational power, its performance directly determines the entire system's performance. Our design of the database manager is based on the concept of *memory-residency* of data. We discuss how the assumption of memory-residency can be reasonably approximated in a practical environment. A practical system does not allow a large amount of real memory for each user all the time in a time-sharing environment. Therefore, we emphasize the importance of proper coupling of the memory-residency idea to the operating system's scheduling algorithm. We shall also present the results of tests on OBE's performance. OBE uses the VM/CMS operating system [20].

The idea of memory-residency of data has also been investigated in [10], [13], [30], [31], and [38]. DeWitt et al. [10] compare performance of conventional data structures when a large amount of real memory is available. Garcia-Molina et al. [13] present a new logging/recovery scheme with specialized hardware in a (real) memory-resident database system. Lehman and Carey [30, 31] introduce a new indexing structure called *T-tree* and present join algorithms based on the T-trees. Shapiro [38] introduces a hash-based join algorithm that can be applied efficiently when there is a main-memory buffer whose size is equivalent to the square root of the size of the table processed. In [10] and [38] the large main memory is considered as the buffer for the disk-resident databases. On the other hand, in [13], [30], and [31] main memory is regarded as the main depository of data. Here, disks are used only for permanent storage of data and backup. Our approach belongs to the latter group, in which different sets of data structures and algorithms have to be devised to take full advantage of the memory-residency of data.

The purpose of this paper is to introduce the facilities of OBE and the architecture of the current implementation. In doing so, we describe the design decisions made to enhance performance and usability. In particular, we emphasize the following three aspects: (1) a fast, powerful database system for performance, (2) an implementation technique for integrating heterogeneous objects, and (3) user-friendly features with a rich set of color-graphics presentation facilities. The major contributions of this paper are summarized as follows:

—Integration of heterogeneous office objects through a database system using a powerful relational database language.

—A complete implementation of a memory-resident database system. To the authors' knowledge, there is no other systematic implementation of such a system except for some Prolog systems. However, Prolog systems mainly emphasize language issues rather than database management system (DBMS) issues.

—Development of many implementation techniques that make the memory-resident DBMS a practical system. We note that a memory-resident system has an environment vastly different from those of conventional disk-based systems. Therefore, we need a cost model, an indexing technique, algorithms, and data structures that differ from those of disk-based systems.

The paper is organized as follows: In Section 2 we present the features and facilities that are available to the OBE users. In particular, we discuss the OBE language, including database manipulation, screen manipulation, business graphics, text processing, electronic mail, authorization, and on-line help with an emphasis on integration of heterogeneous objects through example elements. In Section 3 we present details on system implementation issues. In particular, we cover screen management, two-dimensional parsing, object storage, query processing, query optimization, performance, integration of heterogeneous objects, concurrency control, recovery, and authorization. Finally, Section 4 briefly reviews the status of the system and concludes the paper.

2. FEATURES AND FACILITIES FOR THE OBE USER

In this section we present the OBE language and other features that are available to OBE users. An earlier version of the OBE language and features is described in detail in [46].

2.1 Objects

OBE supports manipulation of types of *objects* such as

Database table:	A set of *tuples* of data in which each tuple contains a data item for each column.
Text box:	A block of text (e.g., a memo), which OBE formats according to the user's specifications.
Window:	A unit of execution. It contains OBE objects, operators, and commands. It can be stored in a database.
Menu:	A set of alternatives that the user can select for OBE's next action.
Image object:	A picture.
Graphic object:	An object for graphic presentation on the terminal. For example, a bar chart might present the result of a database query.

Each object (except a window) is associated with a window. Each object has a header (an area for the object's name).

2.2 Operators

OBE supports activities such as word processing, data retrieval and writing, electronic mail, and graphics. OBE treats different types of objects and activities as consistently as possible. For example, an OBE user can type certain *operators* (e.g., "P." for Present and "U." for Update) in different *contexts* (parts of objects) without using different syntaxes. An operator's meaning in one context resembles the operator's meaning in a different context, but details of the meanings can differ. For example, a "P." in the header of a text box means "present the names of all stored text boxes," whereas a "P." in the row of a table's skeleton means "present all tuples that satisfy a specified condition." Such context-dependent semantics let OBE implement a variety of functions with a limited set of syntactic symbols.

OBE provides these operators:

P. Present (display) an object, data from a table, or the names of all objects of a specified type.
I. Insert an object or data into a table.
D. Delete an object or data from a table.
U. Update an object or data in a table.
G. Group data according to the values of the column or example element for which G. is specified.
S. Send a text box to a user via electronic mail.
X. Execute a window.

Operators that write need not take effect immediately in the database on disk.[2] A *save* command writes to the database the results of all write operators that have taken place since the previous save. Additional operators that are specific to database manipulation appear in Section 2.6.

An *example element* is a construct that lets the user match data values (e.g., an "item" column in a "sales" table and an "item" column in a "supply" table). An example element can also integrate different activities (e.g., text processing and data retrieval), as we explain in Section 2.7. Usually the presence of the same example element in two or more positions means that the data instances that correspond to those positions should have the same value. An example element's syntax is an underline followed by any alphanumeric string.

2.3 Screen Manipulation

OBE provides these commands (and a few others) for manipulation of objects on the screen:

Add box, add table, add window, etc.:	Add the specified type of object (e.g., a text box) at the position that the cursor selects.
Erase:	Erase or shorten the object or part of an object (e.g., row of a text box) that the cursor selects.
Expand:	Expand the part of an object (e.g., width of a table column) that the cursor selects.
Move:	Move the object that the cursor selects to another position.
Copy:	Copy the object that the cursor selects to another position.
Locate:	Locate the next object after the one that the cursor selects.
Push down:	Push the object that the cursor selects to the bottom of the list of objects to display.
Zoom:	Zoom out to give the user a view of all the objects in a window.

The user usually invokes these commands by pressing function keys. The user can change the assignment of function keys to commands.

[2] We use *write* as a generic term for insert, delete, and update.

OBE can display one or more windows to the user. They are positioned within a coordinate system called the *field of windows*. The field of windows is three dimensional, in the sense that one window may overlay all or part of another window. The display screen is a viewport showing a portion of the field of windows. Similarly, a window opening is a viewport over the *field of objects* associated with a window. An object associated with a window is not necessarily visible within the window opening, and one object may overlay another object within the same window. The user can move the display screen over the field of windows to show different portions of the field and can move each window individually with respect to the other windows. Similarly, the user can move a window opening over the window's field of objects and can move each object with respect to the other objects in that window.

2.4 Text Processing

The user can type text into a text box. When the user has been modifying a text box, OBE formats it after every press of the ENTER key (i.e., what you see is what you get). The user can also print a hard copy of any object.

The user positions the cursor and presses function keys to invoke the following functions for manipulating text in a text box:

—Scroll the text within the text box.
—Add a line of text by pushing down all text starting at the cursor, even if it is in the middle of a line.
—Erase a block of text, starting at the cursor's current position and ending at its position when the user presses the function key a second time.
—Move a block of text. The block starts at the cursor's current position and ends at the position it is in when then user presses the function key a second time. The destination is its position when the user presses the function key a third time.
—Copy a block of text. The interactions resemble those for moving text.
—Locate and optionally change a specified string of text.
—Move the cursor to a tab stop.

A function that involves multiple interactions (e.g., move) provides prompting and allows cancellation of the function between interactions.

The user can select any number of the following formatting options any number of times within a text box:

—Change the left and right margins and the tab stops.
—Suspend or resume justification (alignment of text to the right margin).
—Suspend or resume concatenation of each line to the next line.
—Suppress such concatenation for a particular pair of lines.
—Center a particular line.

2.5 Electronic Mail

The "S." command invokes electronic mail to send a text box to one or more users (receivers) at the same site or at different sites. The sender specifies the box by naming it or by selecting it with the cursor. The sender specifies the

receivers by naming them or by using an example element that matches an example element in a table that names the receivers.

When mail arrives at a receiver, OBE stores the mail as a text box in a window named *MAIL* and notifies the receiver. The receiver views mail simply by displaying the MAIL window, which contains the mail items in reverse chronological order of reception.

2.6 Database Manipulation

A user manipulates tables by typing operations in a skeleton of a table (at the beginning of a table's row and/or inside a table's row). The operations resemble those of Query-by-Example [43].

The following operations can appear at the beginning of a table's row:

P. Present all columns of any number of the table's tuples. The entries inside this row can contain expressions that specify which tuples to present; the default is all tuples.

I. Insert any number of tuples into the table. The entries inside this row contain expressions that represent the data to be inserted.

D. Delete any number of tuples from the table. The entries inside this row can contain expressions that specify which tuples to delete; the default is all tuples.

¬ Apply a specified query or write only where the table does *not* contain a tuple that matches the expressions in the entries inside this row. We call the ¬ operator *row negation.*

The following operations (and a few others) can appear in entries inside a table's row:

An expression	Apply a specified query or write only to tuples whose value in this column matches the expression. An expression can include example elements, constants, null, and arithmetic expressions.
P.	Present this column of any number of the table's tuples. Entries inside this row can contain expressions that specify which tuples to present; the default is all tuples.
U.expression	Update this column (replace the old value by the expression's value) in any number of the table's tuples. Entries inside this row can contain other expressions that specify which tuples to update; the default is all tuples.
An aggregate operator	Apply the aggregate operator to the selected tuples. Possible operators include CNT. (count), SUM. (sum), AVG. (average), MAX. (maximum), and MIN. (minimum).
ALL. or UNQ.	Include duplicate data values in performing the operation (if ALL.) or remove duplicates before performing it (if UNQ.). Inclusion is the default when using an aggregate operator, whereas removal is the default when not using an aggregate operator.

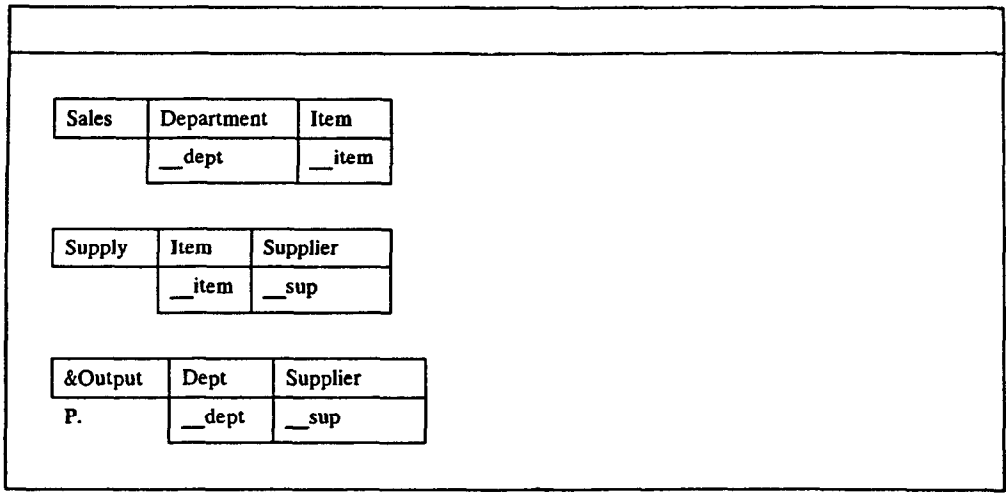

Fig. 1. An OBE program with a user-created output table.

A user can also create a *condition box* to specify any number of conditions (constraints on operations). Each condition contains at least one example element. The query or write applies only to tuples that satisfy all applicable conditions in the condition box and all applicable expressions in the table rows. A condition in a condition box is a Boolean combination of comparisons (e.g., ">" or "=") of expressions.

2.7 Integration of Objects

QBE uses example elements to map data among database tables. OBE extends this concept of mapping to integrate different types of objects. We illustrate integration in the following examples.

Mapping data into a user-created output table. Consider two database tables: Sales and Supply. Suppose we want a table of departments and suppliers who supply items sold by each department. A QBE (also OBE) program to do this is shown in Figure 1[23].

In Figure 1 the symbol "&" in the table name indicates that the table is a user-created output table. A *user-created output table* does not represent a table stored in the database; instead, it only presents output results.

Mapping data into text. This example illustrates the use of example elements to map data from a database table to text. Suppose manager Lee wishes to send a letter to each of her employees to inform them of her impending vacation. The EMP table contains the names, locations, managers, and electronic mail user ids of the employees. An OBE program for sending the letters is shown in Figure 2 [45].

In this example the S. command sends the object named by the command (Note) to the persons listed after the TO keyword of the command. OBE sends to each of Lee's employees a separate copy of Note, personally addressed with that employee's name and location.

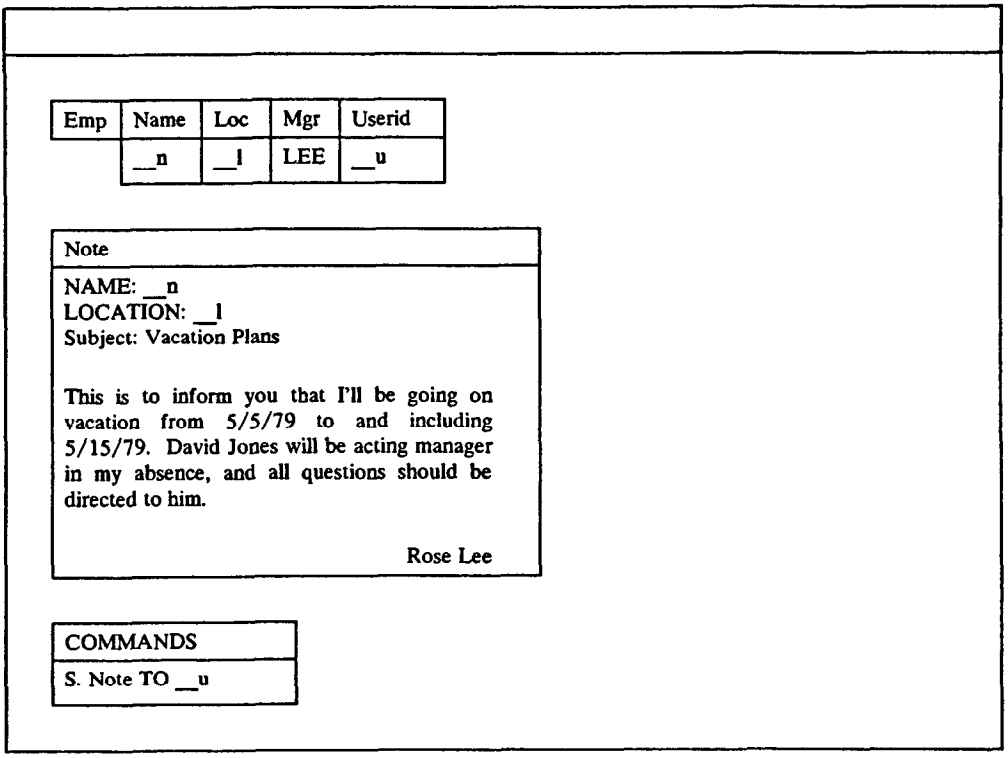

Fig. 2. An OBE program that maps data into text.

Mapping data into a graph. This example illustrates the use of example elements to map data from a database table to a business graph. Suppose we want a horizontal bar graph showing hardware and software sales for the year 1983. Further, suppose that we have a database table containing hardware and software sales for each state and calendar quarter. An OBE program for generating the result in a graphical form is illustrated in Figure 3.

In Figure 3 the query specifies that the sums of the hardware and software sales in the year 1983 be plotted in a horizontal bar graph grouped by states. The G._S example element expression in the horizontal bar graph specifies the variable for the vertical axis. In this example, two horizontal bars (one for hardware sales and one for software sales) are drawn for each state spread along the vertical axis. The expressions SUM._HW and SUM._SW specify the variables corresponding to the bar lengths.

2.8 Business Graphics and Images

2.8.1 *Direct Manipulation.* Two important concepts of the QBE language that have been carried over to the design of the graphic query interface are direct manipulation of objects and the use of example elements to map data from database tables, which we just described in Section 2.7. In QBE the database tables are the objects for direct manipulation. Similarly, for queries involving

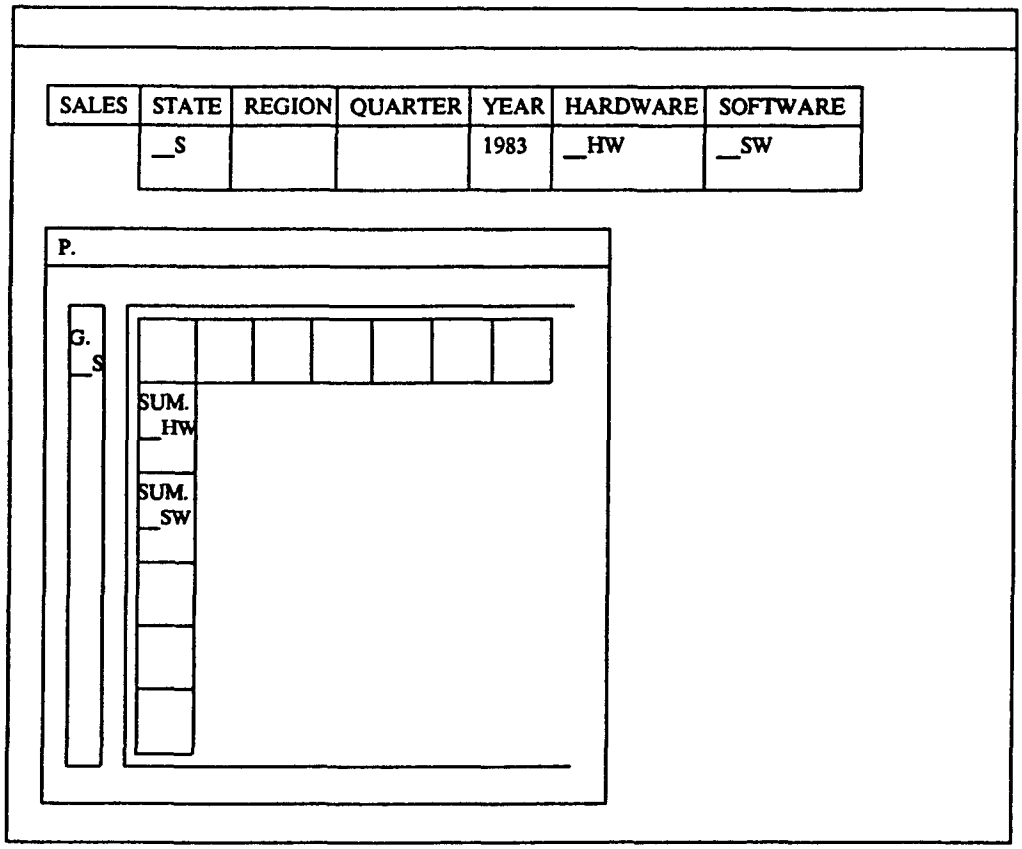

Fig. 3. A query for generating a horizontal bar graph.

business graphs, different types of business graphs are the objects for direct manipulation. Also, in creating a query, example elements can be used to link database tables and graphic objects.

In this section we first describe the visual interface of graphic objects and the concept of direct manipulation in composing a query. Then, we show the flexibility and ease of creating different graphic queries.

2.8.2 *Business Graphic Objects.* A graphic programming environment allows a person to converse with a computer rapidly through the power of a graphic interface. The idea of depicting abstractions graphically has been applied to many areas, such as general programming environments, computer-aided design, computer-aided instruction, and so forth.

Similarly, in an OBE program, business graphic objects are used for creating queries. Within each graphic object, a specific type of business graph is shown along with colored fields in which the user places example elements to specify data linkage. The colored fields of the graphic object allow the user to specify directly the color of the output data associated with an example element and the spatial relationship among output data associated with different example

elements. An input graphic object looks similar to its corresponding output object. The use of the similar objects as both input and output entities is an important feature for satisfying the user's intuition.

Figure 3 shows an example horizontal bar graph as the input and output object. All the colored fields of both vertical and horizontal bar areas (shown as sequences of squares) in the input object are used as the input programming area. If the user specifies example elements side by side in the horizontal bar area, the result will be a stacked bar graph with corresponding colors. On the other hand, if the user puts example elements in the vertical bar area, the result will be a parallel bar graph.

The system supports nine types of business graphs [18]: line graph; surface graph; scatter graph with optional linear, logarithmic, or exponential regression; pie graph; vertical bar graph; horizontal bar graph; curve graph; histogram with normal curve fitting; and statistical graph. In addition, the system has a graph menu object that allows the user to select a specific type of graph.

2.8.3 *Creating and Modifying Queries by Using Business Graphs.* Figure 3 shows how easily a user can create a query using business graphs. Modifying a query is also easy because the OBE user can compose a query related to an existing one by revising the latter only slightly. The example in Figure 3 is a query for generating the horizontal bar graph of both hardware and software sales in 1983 grouped by states. Now, suppose the user also wants to generate the horizontal bar graph of both hardware and software sales in 1983 grouped by regions. The two graphs use the same data and differ only in a variable of a base axis. However, many existing integrated software packages require the user to go through a sequence of menus to generate the new query. In OBE, to generate such a new graph, the user simply moves the example element_S in the column STATE to the column REGION and reprocesses the query. We have such a tremendous saving of effort owing mainly to our mechanism of integrating graphic objects with the database. Graphic queries can include all other OBE query language capabilities, such as the condition box to specify constraints, arithmetic operators to create formulas for derived data, and so forth.

2.8.4 *Interactive Graphic Editing.* Having generated the output graph on the screen, the user can edit it interactively in a very flexible manner. Examples of editing functions for graphic objects are (1) modifying the width or spacing of all bars in a bar graph by pressing a single function key; (2) expanding or shrinking the vertical scale by pressing a function key; (3) modifying the bar colors by changing the color areas in the legend; (4) adding the headings, footings, and comments by typing text strings into the object directly; (5) highlighting a segment of a pie graph by moving it slightly apart from the whole pie, and so forth. The edited graphic objects can be stored in the database for future use, or they can be superimposed into text or image objects.

All the operations are normally assigned to function keys. Their behaviors depend on the cursor position and the content of the selected object. Readers are referred to [18] and [19] for the details of the graphics facility in the OBE system.

2.8.5 *Facilities for Image Objects. Image* is one of the objects that OBE supports. The system provides image viewing and editing facilities through the image

object. Currently, image objects are not used in conjunction with database tables for creating queries. The system only supports the functions for viewing and editing stored images. The functions include shrink, expand, copy, move, scroll, zoom in, zoom out, annotate, and so forth. Images can be created using a scanner and are stored as CMS files or OBE objects in a compressed form. In the current implementation, only monochrome images are supported. The CMS file type for image objects is restricted to MONOIMG for a black and white image.

To view an image on the screen, the user has to get an image object into the window by issuing the ADD IMAGE command. To display an image stored in a CMS file, the user moves the cursor to the object header and types

P. CMS ⟨name⟩ MONOIMG ⟨mode⟩,

where ⟨name⟩ is the name of an image already stored, and ⟨mode⟩ identifies the disk containing the image. On the other hand, to display an image stored in the OBE's object storage system, the user types "P. object_name" in the header of the image object.

2.9 Authorization

OBE is primarily intended to provide a single-user environment, in which individual users keep their private data in their own databases. Nonetheless, since information must often be shared, OBE also supports sharing. We define these two modes as *private mode* and *shared mode*. OBE accommodates both modes in a unified framework by delegating the decision on the mode of operation to the users of the system. Thus, in a private mode, users own their own databases in which they define their own objects. On the other hand, in a shared mode, a group of users establishes a shared database in which different users can create their own objects. These objects can be shared by other users. OBE also allows users to share their private objects in private databases.

The flexibility in sharing is accomplished by an *authorization* scheme having the following characteristics:

(1) The scheme allows users to share objects (or parts of objects) selectively.
(2) The scheme minimizes the traditional role of the database administrator (DBA) to allow maximum independence to individual users.

In this scheme the DBA, who is defined as the database creator, receives no authoritarian power to override individual users' rights to their own objects. The DBA has only two types of authorities. The first is the authority to browse (i.e., only read) the database's directory. It provides an essential service to the user community when certain difficulties arise. For instance, suppose that an employee leaves after creating objects in the shared database. These objects may never be detected without the DBA's ability to browse the entire directory. The second is the authority to choose members of a user community who can share the database. The DBA chooses members by having the *creation* authority allow them to create objects in the shared database. In OBE, users are the sole owners of their own objects—even in a shared database. A database is owned by the users, not by the DBA.

OBE uses the following syntax for specifying and manipulating the authority statements. This syntax is an extension of the one defined in QBE [44].

$$\left\{\begin{matrix} I. \\ D. \\ U. \\ P. \end{matrix}\right\} AUTH(\langle list\ of\ authorities \rangle)\ Grantee\ Object$$

Here, ⟨*list of authorities*⟩ is a list of individual authorities, *Grantee* is a user or a group of users who receives the authorities, and *Object* is the object being authorized. The authorization information can be inserted (I., *grant*), deleted (D., *revoke*), updated (U.), or presented (P.). An authority statement is specified in the header of an object, except for a table, where it is specified in the data area under the table header.

OBE supports the *P.*, *I.*, *D.*, *U.*, *X.*, _* authorities for most OBE objects and the *creation* authority for the database. The symbol _* implies all the authorities applicable to the object being authorized. In general, applicable authorities vary, depending on the type of an object. For example, the I. authority applies only to insertion of tuples in database tables. Insertion (via I.) of objects (including database tables) requires the *creation* authority.

The syntax as described above allows only *data-independent* authorization, which can be enforced without examining the data. In general, *data-dependent* authorization, which can be enforced only by examining the data, can be specified by replacing Object in the syntax by an *associated* query. Implementing data-dependent authorization this way, however, poses some difficulties. First, in the presence of multiple, different authorizations on the same database table, the union of different sets of authorized tuples may not convey meaningful semantics. This problem is aggravated if the union of different sets of tuples cannot be formed because of different projection requirements of the associated queries. A simple union of sets of columns in associated queries would not be a solution because it will reveal some data that would not be shown when individual authorizations are considered. Second, managing (i.e., storing and retrieving) associated queries and their relationships with the objects they authorize could be troublesome.

To solve these problems, we decided to use *views* to implement data-dependent authorization [15]. This approach has several advantages. First, a view may be regarded as a named query. By using the name of a view in place of Object in the authorization statement, the same syntax can be used. Second, even in the presence of multiple authorizations on the same table, a user deals with only one set of authorized tuples at one time by designating a view by its name. Third, by using names, we can easily manage views. Further, we can explicitly establish the relationship between an authorization statement and its associated query (i.e., view). The view facility has not been implemented in OBE, although it has been planned. We believe, once the view facility is available, that data-dependent authorization can be easily incorporated in a way similar to the one discussed above.

2.10 On-Line Help

OBE's on-line help facility lets a user obtain information on using OBE without always having to consult a manual. The help information describes actions that

the user can perform in the context that the cursor selects. For example, for the header of a text box, the actions include inserting or deleting the text box, moving the text box, and so forth. Pressing the function key for HELP displays a scrollable help box that contains the context's name and the help information. OBE displays the box in one half of the screen so that it does not overlap the selected context. Thus the user can simultaneously see the help information and manipulate the context. This ability is particularly useful for tasks that involve multiple interactions (such as moving an object). The help box remains on the screen until the user erases it by pressing HELP again.

3. SYSTEM ARCHITECTURE

In this section we present issues in implementing our integrated office system and database manager.

3.1 Implementation of Integration

The OBE system maps example elements among heterogeneous objects by using preprocessing and postprocessing steps. The preprocessor creates a user-created output table and associates it with each nontable object containing example elements. This user-created output table is added to the query. Then, the query processor (described in Section 3.4) evaluates this modified query and creates result tables containing the output data for the user-created output tables in the query. Finally, the postprocessor scans the result tables to map the data to the associated objects.

For example, when the program in Figure 2 is processed, the preprocessor modifies the query by adding the user-created output table in Figure 4.

The query processor executes the query and creates an output table consisting of (name, location, userid) tuples. For each tuple in the output table, the postprocessor replaces the example elements in the text box with the name and location values and sends the text box with these substitutions to the electronic mail recipient specified by the userid value.

We choose this technique of integrating heterogeneous objects because the only information in the object to be related with the database is that associated with example elements. The purpose of using a user-created output as an intermediary is exactly to extract this essential information. In addition to simplicity, this technique provides modularity. Thus it is easy to integrate new object types to be added to the system. Currently, this extendability is the system programmer's responsibility in our system. However, providing the extendability (i.e., adding new object types) as a user task should not be very difficult to achieve.

3.2 Parsing and Translation

The parsing and translation component of OBE translates between the screen manager's data structures and those recognized by the query processor. During this process, the query being translated is checked for syntactic correctness. Parsing in OBE is significantly more complex than in a conventional compiler. This results from both OBE's two-dimensional nature and the tailoring of its behavior to the needs of nonprogrammers.

&Q			
P.	_n	_l	_u

Fig. 4. User-created output table added to query in Figure 2.

The parsing phase produces a parse tree containing a parsed version of the screen objects handled by the query processor, that is, tables and condition boxes. A translation system translates the parse tree to an internal form for the query processor.

3.2.1 *Lexical Analysis.* The lexical analysis component of OBE recognizes the basic tokens of the language: constant elements, example elements, and operators. Since OBE is designed to minimize the syntax its users must know, complications sometimes arise in recognizing the language. For example, in conventional programming languages, and in most database query languages, literal strings must be explicitly quoted. In OBE, the user can type in the (unquoted) value. Thus, a user simply types JONES rather than "JONES". (Of course, constants that have the same structure as keywords, like "P.", must be quoted. However, this happens infrequently.) This requires the lexical analyzer to have an indefinite look ahead to resolve tokens (up to the length of the field being scanned). In practice, the lexical analyzer must occasionally backtrack to the last previous blank within the token currently being formed.

A further complication is that keywords are context dependent. So although "AND" *is* a keyword in the condition box, it is *not* a keyword in other contexts. This leaves the user free to use it as a constant or the name of a table column in other contexts. This problem is resolved by keeping a table of keywords and the contexts in which they actually are keywords. A token can be a keyword only if the table associates it with the current context.

3.2.2 *Parsing Two-Dimensional Objects.* Whereas a program in a linear language like PASCAL is (formally) a single long string, OBE objects are data structures with an internal structure. Unlike a conventional programming language, this structure must be recorded in the parse tree for the subsequent translation step. For example, a "P." as a row operator must be translated differently from a "P." inside a row. Whereas a linear language's parser simply ignores empty lines, OBE's parser must preserve this structure to preserve the relative positioning of subsequent fields. Thus, there are two problems: (1) the complications of maintaining a position within a structured object and (2) "wiring" this structure into the grammar so that an expression's context within an object is represented within the parse tree.

Like a conventional linear language, the context-free part of OBE is handled by a parser generator [29]. However, as in lexical analysis, OBE is structurally more complex than a linear language. Since the input to a programming language's compiler is a sequence of lines, it can maintain its state as a 1-tuple: ⟨*buffer_pos*⟩, its position in the current line buffer. An OBE program consists of the data structures for a window and its objects. Each object consists of an array of rows, with each row consisting of one or more fields. Thus, a screen object is OBE's version of a line buffer. The parser's state is then a 3-tuple: ⟨*row_number*,

column_number, field_pos⟩, where *row_number* and *column_number* are the row and column of the current field, and *field_pos* is the position within the field being examined. These are incremented in reverse order: After the contents of the current field have been examined (*field_pos*), the columns within the current row are parsed (*column_number*), and then successive rows are parsed (*row_number*).

Each type of object is represented by a nonterminal symbol in the grammar, with the symbol deriving the information parsed for the object. An object's rows and fields are represented by nonterminals in the same fashion. So the symbol for an object derives the information parsed for its rows, and the symbol for each of its rows derives the contents parsed for the fields in that row, etc. For example, condition boxes are represented by the nonterminal "⟨CND_BOX⟩", and a row in the box is represented by "⟨CND_ROW⟩". Syntactically, each row can be a boolean OR expression ("⟨OR_EXP⟩"), so in the grammar we have (in BNF):

⟨CND_BOX⟩ ::= ⟨CONDITIONS⟩.
⟨CONDITIONS⟩ ::= ⟨CONDITIONS⟩ ⟨CND_ROW⟩.
⟨CONDITIONS⟩ ::= ⟨CND_ROW⟩.
⟨CND_ROW⟩ ::= ⟨CND_ROW_EDGE⟩ ⟨OR_EXP⟩.

The beginning of an object, a row in an object, or a field in an object is represented in the grammar by *edge symbols*, terminal symbols ending in "_EDGE⟩". (The edge symbol for a condition box row is "⟨CND_ROW_EDGE⟩".) *Edge symbols* are emitted when the lexical analyzer begins scanning a new object, row, or field. They signal the beginning of a new syntactic unit in OBE the same way in which semicolons end statements in PL/I.

3.2.3 *Translation to Internal Form.* The parse tree is translated to an internal form for query processing by code generated by a program transformation system. Our transformation system differs from other systems in that it provides for both negative and recursive patterns, a capability that existing systems seem to lack. (Cameron and Ito [7] critique the limitations of existing program transformation systems in the course of describing a manual methodology for programming such translations. Partsch and Steinbrueggen [34] survey the literature on program transformation systems. Freytag and Goodman [12] apply program transformation technology to translating relational query languages.) In addition, the system generates its translation into a conventional programming language for compilation (with a highly optimizing compiler), whereas most such systems are interpretive, often implemented on top of LISP.

The program transformation system is given an abstract statement of the problem as pattern matching and replacement against parse trees in units called *patterns*. A pattern is a procedure with a *match condition* that must be satisfied for the pattern to produce a result (a section of the result tree). If the match condition is satisfied, further tests are performed to select the pattern's result. Although quite different in syntax and semantics, the concepts of the pattern matching apparatus are loosely modeled after those of SNOBOL4 [16].

Once the translation has been stated, the transformation system compiles the patterns into PL.8 procedures as an off-line operation. (PL.8 is a PL/I-like

system programming language [3].) Thus, the translation system treats PL.8 as its machine language. These procedures are, in turn, compiled using the PL.8 compiler and incorporated into OBE. The translation system also provides debugging facilities to monitor the progress of the translation as it is performed. These and the translation system's other capabilities will be described in more detail in a future paper.

3.3 Object Storage

Since OBE is intended for use in an office environment, its storage system has design goals different from those of conventional database systems. We list below some of the important design goals:

—A variety of objects, not just tables, must be stored.
—The system is streamlined either for private usage for single users or for sharing with low contention.
—Good response time for small ad-hoc queries is essential, even if it requires a sacrifice of database capacity. Roughly speaking, the response time must be competitive with text editors for similar functions, such as searching for a specific data value.
—There must be no complex installation requirements like those typical for a large database system. For this reason, we rely on the host operating system (VM/CMS) for file management.
—The system must be small in size.

There are two components in the object storage system. The first is the lower level storage manager, which is independent of object types and deals primarily with the transfer and control of data between disk and main memory. It is described under the heading of memory-resident database management. The second, higher level component is a *hierarchical relational memory system* (*HRMS*), which implements additional storage management for relational database objects on top of the primitive functions provided by the first component. We do not describe the hierarchical portion of the system, since the current implementation does not support hierarchies among tables. See [1] for more details on the object storage system.

3.3.1 *Memory-Resident Database Management.* All objects in OBE are collected into databases. A database consists of a directory and a set of objects. Each object or directory is stored in one operating system (CMS) file. Multiple databases can be stored in a given *minidisk* (a logical disk unit in the CMS operating system), but a particular database must reside entirely on one minidisk. When a minidisk is accessed, all the database directories are read into main memory. Individual objects are read into main memory as they are referenced. Directories and objects remain in main memory throughout an OBE session. Changes to objects are accumulated in main memory until a save operation forces a committal to disk.

Typically, in OBE, we assume information is organized into small independent databases. Accordingly, we maintain separate directories for different databases. This assumption matches closely the office organization, in which individual

users keep their private data in their own databases and occasionally share them with others. It also matches our implementation strategy that keeps the directory in main memory. Clearly, it would be undesirable to keep in main memory the entire directory of a large centralized database, most of which would not be referenced in a given session.

A directory consists of (directory) entries, along with an indexing structure for efficient search. One directory entry is associated with each object in the database. It contains, among other things, the name and size of the CMS file that stores the object. An object is read into main memory in its entirety upon its first reference. It is read into its own *area* allocated according to the size information in the directory entry. We henceforth call an area containing an object a *segment*. Area is a primitive data type of the PL.8 language. All the memory references inside an area are based on offsets from the starting address of the area and therefore are relocatable. The property of *relocatability* is essential in performance because it obviates address transformations when the segment is loaded from disk, written to disk, or copied between different locations in main memory. Storage within a segment is managed in a stacklike fashion from either end. When storage within a segment is exceeded, a larger area is allocated, and the contents of the segment are transferred.

Once a change has been made to a segment, it is marked as modified. When changes are to be saved, an attempt is made to write all modified segments back to disk. For a save operation to succeed, all modified segments must be committed. Thus, the user has the view that the entire database is saved as a unit. Should committal of changes to disk fail, all changes made to segments since the last successful save are lost. Detailed discussion on concurrency control and recovery appears in Sections 3.7 and 3.8.

3.3.2 *HRMS.* HRMS is a higher level storage management system for relational database objects, built on top of memory-resident database management. Specifically, it provides data structure and access functions for tables and indexes. Each table is stored in a segment of the memory-resident database, along with associated indexes and table definition (schema) information.

A table is represented as a doubly linked list of tuples. A tuple is an array of pointers to column values. Thus, the size of the array is the number of columns of the table. The special *null* value is represented as a null pointer. This representation has the advantage of not wasting extra space in column data to distinguish the null value from ordinary values. In a given column, data values are shared whenever possible; that is, tuples having the same value for a particular column will have equal pointers. The sharing of column values can be maintained easily when an index is defined for that column.

All data are stored as variable-length strings. HRMS supports four data types: *character*, *numeric*, *date*, and *time*. Data of type character are interpreted as character strings. Numeric data are interpreted as a form of floating decimal, which consists of a 1-byte binary exponent followed by as many packed decimal digits as necessary. Date is retained as the number of days since some arbitrarily chosen origin. Last, time is kept as the number of seconds since midnight. These representations permit simple numeric comparison of dates or times with no further conversion.

To support associative accesses, HRMS provides two types of indexes: the *single-column* and the *multiple-column index*. A single-column index is an index defined for a column of a table, and it can be permanently stored in the database. A multiple-column index is an index defined for a list of columns, but it cannot be permanently stored in the database. In Section 3.5 we discuss how a multiple-column index is constructed. We note here, however, that a multiple-column index is always constructed to be specific to a query, using the information given by the optimizer. Hence, it is not relevant to store the index permanently in the database.

An index (of either type) is implemented as an array of *tuple identifiers* (*TIDs*), which are pointers to tuples. Accordingly, accessing an index requires a binary search. This scheme contrasts with the conventional method of implementing an index for main-memory data access as a binary search tree such as the AVL tree [27]. Since our scheme does not store the tree structure explicitly, it requires less memory space for storing indexes. Specifically, our implementation requires one-sixth to one-third the memory space of our earlier implementation, which used the AVL tree. The scheme achieves further reduction in memory space by storing only pointers to tuples (TIDs) in the index; the key values can be found easily (by two pointer references) from the tuples to which they point. In comparison, in some conventional disk-based database systems, index storage cost is a major problem because the index must store original key values.

Updating an index is straightforward. For example, when an index entry is inserted, the upper part of the index is block-copied to a new memory location, and a new entry is inserted; then, the lower part of the index is block-copied next to the new entry. For block-copying we use the instruction MOVE LONG (meaning *move a long character string*) in the 370 architecture [25]. This instruction is a very efficient operation in IBM mainframes. Experiments show that copying 1 megabyte of memory using this instruction takes a small fraction of a second in a 3081 processor. For this reason, updating indexes is hardly a problem in our system.

In addition to tables permanently stored in the database, HRMS also supports temporary tables for composing output results of queries. Since OBE does not allow duplicate tuples in the output (unless the ALL. operator is specified), the system automatically eliminates those duplicates. We use an efficient method based on hashing for this purpose.

3.4 Query Processing

The query processor evaluates a query in three steps. First, it translates the query into an internal data structure. Second, it calls the query optimizer to obtain the access plan. Finally, it executes the query according to the directives specified in the access plan. In the following subsections we discuss important design considerations in the first and the last steps. The query optimizer is discussed in detail in Section 3.5.

3.4.1 *Query Translation.* In the translation step the query processor receives from the parser a syntactically correct query in an internal form. From this representation the query processor composes an efficient global data structure capturing all the semantics of the query. Typically, this process carries out the

following two tasks:

(1) various types of binding that can be determined statically (i.e., independently of data),
(2) semantic error checking.

The purpose of static binding is to avoid the excessive run-time overhead of repeating costly computation. Static binding lets this computation be done once and for all by storing the result in the data structure that the run-time modules subsequently use. The simplest type of binding is the resolution of names of example elements, tables, and columns. These names are recognized and translated into an internal representation. For example, tables are given unique integer identifiers, and the column names are converted to their corresponding positions as defined in the table schema. This conversion requires database accesses to obtain such information.

Another type of binding is what we call *operator binding*. The OBE language allows a symbol to play different roles according to the context. For example, when an expression_$X = 5/9/85$ occurs in a condition box, the operator "/" is a division operator. When $5/\_X/85$ occurs in a column of a table, the meaning of the operator "/" is determined by the context; that is, depending on the type of the column, it may either be a division operator or a date operator. The query processor keeps track of the context and determines the correct meaning whenever there is potential ambiguity. Let us note that, although the parser itself keeps track of certain context information during parsing, this type of ambiguity cannot be resolved solely by syntax.

There are other types of bindings like *cross references* and *query graphs*. Cross references give all the occurrences of an example element in expressions. They are extensively used in semantic error checking and query optimization. The query graph is a representation of the query showing the tables, columns, and their relationships defined through the example elements. The query graph is vital information for the query optimizer.

Semantic error checking is interleaved with various steps of the translation. Numerous cases of semantic errors are detected. Owing to the two-dimensional nature of OBE, these errors are highly dependent on the context and therefore very difficult to detect during the parsing phase. For example, a query consisting of one row with example elements and row negation is erroneous because there is no way of binding these example elements. Capturing this error syntactically would significantly and unnecessarily complicate the syntax.

3.4.2 *Query Execution.* Executing a query consists of retrieving the data, checking for satisfaction of the conditions, computing aggregation if necessary, and constructing the output result. In conventional database systems, retrieval of data (especially disk I/Os) was given the main emphasis when considering the performance. Since the data are memory resident in OBE, however, trimming any other operations can notably improve the performance. In this subsection we discuss the central query execution algorithm, with an emphasis on performance aspects.

In executing a query, the query processor takes several steps. First, it computes the subset of the tuples from each table that satisfies all the selection and join

conditions. It then performs aggregation operations. Finally, it carries out projection operations. If an aggregation operator appears in the condition box, often subunits of the query must be processed in a partial order. We call these subunits *subqueries*. If subqueries are involved, the next higher level subquery follows the aggregation operations, delaying projection operations until after the highest level subquery is evaluated. This general structure of execution can be modified through peephole optimization—merging and commuting some steps—whenever feasible.

Computing the subset of the tuples satisfying all the conditions involves a join algorithm. There can be numerous join algorithms but we choose here the *nested-loop join strategy*, which is also called the *inner/outer loop* join strategy. An abstract version of the algorithm is given in Figure 5.

An important feature of this algorithm is that it need not create temporary tables. Creating temporary tables could be harmful in our system because it takes not only CPU time but also a significant amount of main memory space. In memory-resident databases, main memory is a scarce resource. Thus, the space taken up by temporary tables could sometimes be a threat to our memory-residency assumption.

Yet a more important feature of the nested-loop join strategy is its performance. We believe that the nested-loop join (with proper usage of indexes)[3] alone is adequate in memory-resident databases. Typically, two join algorithms are used in many database systems: the nested-loop and sort–merge join strategies [41].

In disk-based database systems the sort–merge join strategy has an advantage over the nested-loop join strategy when a majority of tuples in the tables are accessed. Since it processes all the tuples in a block sequentially, it avoids a pathological situation in which accessing one tuple requires one block access. On the other hand, the nested-loop join strategy is advantageous when only a small fraction of tuples needs to be processed. Since it accesses the tuples in a random fashion, it is very likely to incur one block access per tuple. However, sorting is not needed in this strategy. Since only a small fraction of tuples is considered, the saving of the sorting cost can easily compensate for the disadvantage of random access. Clearly, the sort–merge join strategy should be used only when the benefit of sequential access exceeds the cost of sorting the tuples in the tables.

In OBE we choose to implement only the nested-loop join strategy. Basically, we argue that an index (a single-column or multiple-column index) can be built in time equivalent to that for sorting a table, both on the order of $n \log n$, where n is the number of tuples satisfying restriction predicates. Once the table is sorted or an index created, since there is no concept of blocking in a memory-resident database, most of the benefit of using the sort–merge join strategy can no longer be achieved. In fact, a table can be viewed as a set of blocks each containing only one tuple. Thus, we conclude that, in a memory-resident database, the nested-loop join strategy is as good as or better than the sort–merge join strategy in most cases.

[3] Readers are warned that some authors [30] *define* the nested-loop join as a method that does not use (or create and use) any indexes. Clearly, the nested-loop join by this definition would have an unacceptable performance.

```
/* using indexes as designated by the optimizer */
for each tuple t1 ∈ R1 do;
  if (t1) satisfies the conditions then
  for each tuple t2 ∈ R2 do;
    if (t1, t2) satisfy the conditions then
    for each tuple t3 ∈ R3 do;
        .
        .
        .
      If (t1, t2, t3, . . . , tk) satisfy the conditions then
      project the answer.
```

Fig. 5. Nested-loop join strategy for tables $R_1, R_2, R_3, \ldots R_k$.

Notice that many variations of hash join methods [10, 30] are, by our definition, nested-loop join methods, but they use hash indexes. Such indexes are created on the fly and are not permanently stored. One of our access structures defined in Section 3.5 considers creating an index on the fly for a specific query. However, we did not implement hash indexes because we decided to use the same index structure for both permanent and temporary indexes.

We also have incorporated some modifications to the nested-loop join strategy to avoid wasted work. For instance, let table R_1 join with tables R_2 and R_3 in a query. Further, let the strategy choose tuples from R_1 first, R_2 next, and R_3 last. If no tuples of R_3 satisfy the conditions for particular tuples in R_1 and R_2, the algorithm in Figure 5 will attempt to choose the next tuple from R_2. From the query, however, we know that table R_3 joins with R_1 but not with R_2. Thus, any new choice of a tuple from R_2 is futile. Instead, the algorithm should choose a new tuple from R_1 to make any meaningful changes in the outcome of the conditions. In general, if no tuple from a table is found to satisfy the conditions, the algorithm should choose a new tuple from the *parent* of the table in the query. The parent of a table is defined to be the one connected to the table by a join condition (i.e., sharing an example element) that is the closest predecessor of the table in the join sequence (i.e., ordering of tables that are joined).

In summary, we have implemented the query processor using the modified nested-loop join strategy. We have argued that using this strategy alone is adequate for performance in memory-resident database systems. Let us note that this argument views the data as being in main memory, whereas the data are really in virtual memory. In Section 3.6 we discuss how well virtual memory approximates the real memory when used in conjunction with a specific operating system scheduling algorithm.

3.5 Query Optimization

The query optimizer is a component of the database manager that finds an optimal sequence of access operations for processing a query. The optimizer's results are recorded in a data structure called the *access plan.* The access plan is subsequently interpreted by various components of the query processor to evaluate a query. In this section we discuss the query optimizer's algorithm, cost model, and data structures.

Before proceeding we define some terminology. A *node* is a table appearing in the query with associated predicates specified in the columns of the table. We use the term *node* to distinguish different instantiations of the same table. For

example, if a table appears twice in a query, we treat the two instances of the table as separate nodes. A *join loop* is the looping structure in the query processor implementing the nested-loop join strategy. A *restriction predicate* is a predicate that contains only literal values. A *join predicate* is a predicate that contains an example element. It is called a *bound join predicate* if all of its example elements have been assigned specific values (i.e., *bound*); otherwise, it is called an *unbound join predicate*. We define a *restriction column* as a column of a table associated with a restriction predicate, and a *join column* as a column associated with a join predicate. We call a node with row negation a *negated node*, and a node without row negation an *ordinary node*. Finally, we define *access structures* as the methods of accessing data that the optimizer can select in the process of optimization.

3.5.1 *Optimization Algorithm.* The optimizer determines the optimal order of processing nodes in the join loop and specific access structures to be used in processing individual nodes. It finds an optimal solution by exploring the search space using a branch-and-bound algorithm [17].

A branch-and-bound algorithm can be characterized by three rules: (1) a rule for finding a lower bound for a subtree of the search tree, (2) a branching rule, (3) a rule for resolving a tie among candidate subtrees to be explored. In the query optimizer we define the accumulated cost of processing nodes up to (but not including) the root of a subtree as the lower bound for the subtree. We employ the newest bound branching rule, which selects the most recently created subtree that has not been pruned, as the branching rule, and we use a fixed priority scheme to resolve a tie. We define the order of priority among nodes as follows:

(1) an ordinary node with a bound equality restriction or join predicate,
(2) a negated node with all its join predicates bound,
(3) an ordinary node with a bound inequality restriction or join predicate (i.e., a range or not-equal predicate),
(4) an ordinary node without any bound predicate.

A negated node having an unbound join predicate should never be given a priority because it can be evaluated only when all the predicates are bound.

3.5.2 *Access Structures.* On the basis of the features that HRMS and the query processor provide, the optimizer defines four access structures:

—Table Scan,
—Single-Column Index (existing),
—Single-Column Index (created),
—Multiple-Column Index.

For Table Scan, every tuple in the table is sequentially accessed. For Single-Column Index (existing), tuples are accessed associatively through an existing single-column index. The specific index to be used is designated by the optimizer. For Single-Column Index (created), tuples are accessed through a single-column index, but the index must be created for each query and dropped at the completion of the query. Finally, for Multiple-Column Index, tuples are accessed through a

multiple-column index, for which the optimizer designates a list of restriction columns and a list of bound join columns having equality predicates. The index is created for each query as follows: First, the tuples of the table that satisfy the restriction predicates are selected; then, an index is constructed for the list of columns having bound equality join predicates. The multiple-column index has an advantage of reducing the size of the table because it selects only those tuples that satisfy restriction predicates. Because of this reduction, the cost of both creating the index and accessing tuples using this index can be reduced.

3.5.3 *Cost Model.* We define minimum query processing cost as the criterion for optimality. Since the query processing cost must be determined before actual execution of the query, we need a proper model for estimating this cost. Since we assume that the data are resident in main memory, ideally the only cost incurred in processing a query will be CPU computation cost and there will be no I/O cost. For this reason, the cost model incorporates only CPU cost.

Modeling CPU computation cost is not as straightforward as counting the number of I/O accesses, as is done in many conventional disk-based systems. In particular, analyzing the entire program code for computation cost would be next to impossible. As a solution to this problem we propose an approach using both experimental and analytic methods: We first identify the system's bottlenecks and then build a cost model on the unit costs of these bottlenecks.

In OBE the bottlenecks have been identified with the aid of the PLEA8 execution analyzer [35]. We have found, by experiments, that most cost is incurred in evaluation of various predicates specified in the columns of tables or in condition boxes. Thus, roughly, the optimization criterion becomes the minimum total number of predicates to be evaluated. To provide a better resolution, however, we classify the costs into two categories:

(1) the number of evaluations of the expressions involved in predicates (unit cost = C_1),
(2) the number of comparison operations needed to finally determine the outcome of predicates (unit cost = C_4).

We also define three other basic costs:

(1) the cost of retrieving a tuple from a (memory-resident) table (unit cost = C_5),
(2) the cost of unit operation in creating an index (unit cost = C_2: note that there are $n \log_2 n$ unit operations in creating an index, where n is the number of tuples in the table),
(3) the cost of unit operation in the sorting used to prepare a multiple-column index (unit cost = C_3).

Thus, the five parameters C_1, C_2, C_3, C_4, and C_5 constitute a complete set of unit costs that defines the cost model. Further details in constructing the complete cost model on the basis of these unit costs appear in [40]. The values of the five parameters have been determined by experiments and proved to be stable for a long period of program development. Currently, in our system, expression evaluation (unit cost = C_1) proves to be the costliest operation.

3.6 Performance

The OBE database manager has been designed with a rather unconventional assumption that data reside in main memory. Naturally, one important question is to what extent this assumption would be satisfied in a real environment. Obviously, a steady tendency toward cheaper memory hardware is encouraging. Yet, it will not solve all the problems because the cheaper the memory becomes, the larger the data requirement will be. Thus, we need other means of satisfying the memory-residency assumption.

3.6.1 *Operating System Environment.* In OBE the memory-residency assumption is approximated by virtual memory and the scheduling algorithm of VM/370 [24], a working-set algorithm. If a pure demand paging algorithm were used, OBE might suffer from serious performance degradation due to potential thrashing when many users share the system. However, in conjunction with the working-set algorithm, VM/370's virtual memory provides an excellent approximation of real memory.

We describe below a very simplified version of the scheduling algorithm to investigate its effect on the memory-residency assumption. The scheduling algorithm uses two types of time units: *time slice* and *dispatch time slice.* For convenience, let us call them a *long time slice* and a *short time slice.* In addition, there are two types of queues for virtual machines: *dispatch list* and *eligible list.* The short time slice is the unit time for allocating CPU among members in the dispatch list. A long time slice is a fixed multiple of short time slices during which a virtual machine is entitled to stay in the dispatch list. If there are other virtual machines with higher priorities when the long time slice expires, a virtual machine may be relocated to the eligible list to wait for another long time slice to be assigned to it.

The scheduler puts virtual machines in the dispatch list only to the extent that the total memory requirement of the virtual machines does not exceed the size of real memory. The memory requirement of a virtual machine (*working set*) is first estimated as the average number of memory-resident pages for that virtual machine while in the dispatch list. The number thus obtained is adjusted according to some formula that provides a feedback mechanism to stabilize the performance toward the system's global goal for paging activities. Once the set of virtual machines on the dispatch list is determined, paging is controlled on a demand basis.

Ideally, if the access patterns of virtual machines are constant, the virtual machines must get sufficient memory with which to work. Of course, there is paging due to contention among virtual machines, but, since the amount of real memory always exceeds the total expected demand, the amount of paging should be minimal.

Let us summarize the ramifications of the scheduling algorithm:

—As long as an OBE query is evaluated within one long time slice, there will be no additional I/Os, except for the initial loading of data (one access for each page).

—Even when a query spans multiple long time slices, provided that a long time slice is long enough to dominate the cost of initial loading, the I/O time will be negligible compared with the CPU time.

Table I. Performance Results of Test Set 1

Degree of join	Number of tests	Virtual CPU time (ms)	Elapsed time (seconds)
1	4	176	3.8
2	7	247	4.6
3	5	328	4.2
4	4	223	4.3
5	1	222	4.0

—There will be no significant thrashing because the memory requirement of a virtual machine is satisfied while the machine is on the dispatch list.

—The system's feedback mechanism, imbedded in the estimation of the working set, tends to stabilize the overall paging activity.

We have informally argued the advantages of the working-set scheduling algorithm, the detailed analysis of which is beyond the scope of this paper. Nevertheless, we believe that virtual memory, in conjunction with the working-set algorithm, would serve as a reasonable approximation to real memory in practical environments.

3.6.2 *Test Results.* Extensive tests have been made on the performance of OBE; some of the results are presented here. Specifically, we present two sets of tests using IBM's 3081 and 4341 processors. Across all the queries tested, we found that 4341 figures are approximately 7 times as large as 3081 figures. Thus, in Table I we transformed 4341 figures to an equivalent 3081 figures by dividing them by 7.

Table I shows the results of the first set of tests, which were made under the following conditions. The system was shared by a number of users, but it was not CPU bound. The queries tested were arbitrarily chosen from a large pool of test queries used for debugging purposes. Each table in the query contained approximately 500 tuples, with each tuple occupying about 100 bytes. Each table had one index on the key column. The sizes of the output results of the queries ranged from 0 to 50 tuples.

In Table I, *degree of join* is the number of tables joined in a query, *number of tests* the number of queries tested having the same degree of join, *virtual CPU time* the total CPU time the virtual machine consumed in processing a query, and *elapsed time* the real time measured from the start of a query to its end. Virtual CPU time and elapsed time were averaged over the test queries of the same degree. Let us note that queries of degrees 4 and 5 took less time than those of smaller queries. This happened because these queries produced, on the average, a smaller number of output tuples owing to more selective predicates in the queries.

Table II shows the results of the second set of tests. The tests were made under two different load conditions: First, the machine was used by a single user; second, it was shared by a moderate number of users. The tables in the queries had different numbers of tuples; thus, we represent a table by the number of tuples it contained, for example, 1K, 2K, 10K. The size of a tuple was 186 bytes, and the size of the result of a query was controlled to be exactly 1000. Test

Table II. Performance Results of Test Set 2

Queries	Total data volume (Mbytes)	Elapsed time (seconds)	
		Single-user machine	Shared machine
1K ⋈ 2K	1.1	1.40	
1K ⋈ 5K	2.1	1.43	
1K ⋈ 10K	3.9	1.44	
1K ⋈ 2K ⋈ 5K	2.8	1.85	<10
1K ⋈ 2K ⋈ 10K	4.6	2.18	
1K ⋈ 2K ⋈ 2K ⋈ 5K	3.6	2.26	
1K ⋈ 2K ⋈ 5K ⋈ 10K ⋈ 10K	9.7	2.85	

queries were designed as joins among tables without any restriction predicates (i.e., *unconditional joins*). Each table had indexes on all columns. However, only indexes on the join columns were utilized because we tested only unconditional joins. The indexes occupied approximately 18 percent of the total space. The symbol "⋈" represents a join between tables. In Table II, since the elapsed time for the shared machine varied depending on the system load, we presented only an approximate upper bound.

Finally, we note that a formal benchmarking has been done, and the results can be found in [4] and [6]. The contents in Table II came partially from those results [5]. We briefly summarize the results here:

(1) OBE does very well with selections and joins, as indicated in Tables I and II.

(2) OBE is somewhat slow in update, insertion, and deletion. For update of nonkey columns and deletion, maintenance of indexes causes the major cost. However, note that this problem does not come from our index update algorithm; instead, it comes from our decision to provide indexes for all columns by default, although there is a provision for selective indexing. Experiments indicate that the cost of updating all indexes is comparable to (somewhat higher than) that of updating one index in conventional disk-based DBMSs. Insertion and update of key columns are quite slow; the reason is duplicate elimination by default. Note that many other DBMSs, such as SQL/DS or INGRES, do not provide this feature by default; instead, they allow duplicates in a table. Also note that duplicate elimination in a permanent table is different from that in a temporary table, for which we use an efficient hash-based method.

(3) OBE is somewhat slow for certain types of projections, in which our efficient, hash-based duplicate elimination is not used. Nevertheless, this problem is not inherently related to the memory-residency assumption. We believe that this problem can be fixed by more careful design of data structures for these operations.

(4) We also address the problem of the total size of the physical memory (of the entire system, not per user) being less than the size of the data. The experiments indicate that the performance is gradually degraded as the size of data crosses over the physical memory size. It shows a tendency to degrade approximately four times as the ratio of the data size to the physical memory

size doubles. Note that, as long as the system's total memory is greater than the data requirement of a single user, there is no significant degradation due to thrashing, as discussed in Section 3.6.1. Of course, the response time would be increased (approximately linearly) as the number of users increases. If a system uses pure demand paging, the degradation would be far more significant in a multiuser environment owing to thrashing.

(5) The tests are based on the fact that the tables are already in main memory. If they are not, they have to be brought in main memory at the first reference. Nevertheless, reading a table into main memory is very fast in our system because we store a table as *one* record in a file. The file system has a built-in mechanism to try to store a contiguous virtual file address space in contiguous physical disk pages as much as possible. Typically, the system takes 0.67 second to read in 1 megabyte of data.

For more details of the benchmarking results, readers are referred to [4].

3.7 Concurrency Control

Here we sketch OBE's *concurrency control*—the portion of the OBE system that assures correct operation when multiple users access shared data simultaneously. An application for reserving rooms, for example, could use such an ability. A recent paper [39] describes the concurrency control in more detail.

We define two terms:

—An OBE *subtransaction* is the set of actions that the user specifies before pressing the PROCESS key. This set might include nested executions of windows and selections from menus.

—An OBE *transaction* is a sequence of subtransactions that satisfies one of the following:
  (1) The sequence's length is 1, and the subtransaction only reads.
  (2) The sequence's first subtransaction and possibly others write, and only the last subtransaction saves on disk. The length might be just 1.

During an OBE session, the user can execute any number of transactions. A transaction is the unit of activity for which the concurrency control assures correct operation (consistency). The transaction's operations take effect atomically (as a group), as in a transfer from a checking account to a savings account at a banking machine. Transactions take effect as if they were *serial* [11], that is, as if their operations were not interleaved with other transactions' operations.

Most research in concurrency control for transaction-oriented systems has taken place in the context of DBMSs. Probably most of this research can apply to office systems, and readers who have studied concurrency control for database systems will recognize several of the techniques that OBE uses. However, some issues in concurrency control that apply to OBE, and some other office systems may differ from those that apply to a DBMS. These differences influenced our design decisions. The differences include less conflict among transactions in an office system, possible user interactions (e.g., selection from a menu) *during* transactions, and persistence of local copies of data (in virtual memory) across transactions.

We believe that most OBE transactions will use unshared databases. Therefore, in our design trade-offs, we gave a high priority to optimizing the concurrency control toward those transactions.

OBE applies a type of *optimistic* concurrency control [28] to each subtransaction. When evaluating a query or write, we read and write local copies of objects. Any user interactions during the subtransaction take place now. We assume that the local copies are *fresh*; that is, no other users have written the objects in the database on disk after this user most recently read or wrote them in the database. If a referenced object has no local copy yet, then we read the object from its file into virtual memory, that is, we create a local copy.

After the evaluation, OBE *validates* (checks freshness of the copies of all objects that the transaction has referenced). If all are fresh, then we complete the subtransaction. This includes saving, if the user specified saving. If any referenced copies are *stale* (not fresh), then we reject the subtransaction (if the transaction has written) or restart the subtransaction (if the transaction has not written). A rejection includes erasing local copies of referenced objects that OBE has written or that are stale. The erasing aborts the transaction. OBE behaves as if the aborted transaction never occurred. Rejections should occur rarely. A restart involves rereading stale referenced objects from the database into local copies and then reevaluating the query.

OBE currently uses *freshness indicators* as a mechanism to implement the determination of freshness. A freshness indicator is a piece of information that OBE maintains. A freshness indicator exists for a user for an object if and only if that user has a fresh copy of that object. We have explicitly separated the policy of freshness from the well-known mechanism that most implementations of such a policy use, that is, *timestamps*. The separation lets us use different mechanisms for the same policy.

The optimistic strategy assures correctness. The strategy includes some locks that are essential to ensure integrity of certain low-level actions. We supplement the strategy with other, inessential locks only to improve performance by reducing the frequency of restarts, as we explain below. OBE uses *mortal* locks. When OBE locks a mortal lock, it specifies a lifetime for the lock. Usually OBE explicitly unlocks the lock later. If, however, OBE does not explicitly unlock the lock, then the lock automatically becomes unlocked after its lifetime has expired. This finiteness of lifetime led to our term *mortal lock.*

OBE specifies a very long lifetime (24 hours) for all essential locks, that is, for checking freshness, reading from the database, and saving in the database. This provides the effect of a conventional lock. OBE specifies a brief lifetime (starting with 1 minute) only for inessential locks that it uses to make successful validation likely, as we describe below. We expect that even the brief locks will almost always be unlocked explicitly, as conventional locks are. Locks should time out only under unusual circumstances, such as very slow user interaction during a subtransaction. A user can interact slowly without blocking other users indefinitely because the locks will time out. OBE does not set long (24-hour) locks around user interactions.

If we restart a subtransaction (begin a second pass), then we set brief (1-minute) read locks for all objects that the subtransaction has referenced. If a

copy becomes stale during the second pass, we again restart, but with a longer lifetime of the locks (2 minutes). If we restart more times, we use even longer lifetimes (4 minutes, 8 minutes, etc.). This strategy should succeed eventually, usually on the first or second pass. OBE does not rely on the existence of the brief (minute) locks during a restarted subtransaction. OBE uses them only to increase the probability of success by decreasing the probability that another user will save a referenced object before this user's next validation.

3.8 Recovery

Recovery is a function that a database management system must provide to guarantee the integrity of data. In OBE, recovery is required when a failed save operation has left the database in an inconsistent state. A save operation may fail because of a system error or inadequate disk storage space.

Saving a database consists of several steps. First, each modified segment (object) is written to disk in files of type *commit*. Next, the new directory is written to disk with the same file type. The existence of a directory file of commit type signifies that the transaction has committed. The creation of this directory is guaranteed to be an atomic operation by the operating system. Finally, old versions of modified segments, including other deleted segments, are erased. The new files are subsequently renamed to the types of their original versions.

If the save operation fails before the new directory is successfully written to disk, the transaction is aborted, and recovery is accomplished by erasing all files of type commit. On the other hand, once the new directory exists, recovery is done by reexecuting the part of the save operation that follows the operation of creating the new directory.

OBE's recovery scheme has several advantages. First, individual transaction backup is easy: It is done by simply discarding the memory-resident copy. Accordingly, there is no need for expensive systemwide transaction logging for the recovery from soft failures. In this sense OBE's recovery scheme is similar to the deferred update policy [14]. Second, a systemwide checkpointing is not needed for recovering physical consistency. In OBE a transaction commit leaves the database in a globally consistent state, both logically and physically. Physical consistency is implied at a transaction commit because no two transactions can write in the same disk block without violating logical consistency, which is maintained by the concurrency control module. We obtain this property because the objects, which are the smallest granules of locking, are stored in disjoint disk blocks and thus cannot share the same disk block. System checkpointing is a cumbersome operation, especially in environments such as OBE, in which manipulating private data is the primary mode of operation.

This recovery scheme also has a disadvantage: It lacks the capability of recovering from hard failures because it does not store the update log permanently on disk. (Let us note that the files of type commit can serve as an update log, if stored permanently on disk.) However, the operating system's recovery and backup facilities partially compensate for this drawback.

3.9 Authorization

There are two phases in authorization: definition and enforcement. Authorization is defined by inserting an authorization statement according to the syntax described in Section 2.6. There are three system tables involved in defining authorization. When an authority statement is inserted, a tuple is placed in the *AUTHORITY* table, after verifying that the user is eligible for granting the specified authority. At the same time, if the authority statement involves a table object, associated column information must be stored in the *AUTHCOL* table. Lastly, the *AUTHSTRUCTURE* table is needed for authorizing the X. operation on a window.

A complicated situation occurs when the X. authority on a window is granted. Since a window is an executable object, its authorization implies authorization on objects it contains (we call them *subobjects*). Thus, the system must verify the user's eligibility for granting authorities on all subobjects, as well as on the window itself. If verification of eligibility fails on any of them, the X. authority on the window must not be granted. Let us note that this verification can be omitted, if the X. authority on the window has ever been granted to a different user. In the process of defining the authorization on the window, the subobject structure of the window is recorded in AUTHSTRUCTURE tables—in the database of the window and in those of subobjects. The entire procedure is repeated recursively if the original window specifies an X. operation for another window.

Authorization thus defined is enforced every time an access to the database is requested. In most cases verifying the authorization is as simple as looking up the AUTHORITY table. However, a complex action needs to be taken in certain situations. For example, we encounter such a situation when an object, which is a subobject of a window that has been authorized for the X. operation, is deleted. Since such a window becomes nonexecutable after deletion of the subobject, any X. authorization on this window must be revoked. (Let us note that a nonexecutable window is still a valid OBE object and thus should not be deleted.) Besides, if the window is called in another window, the revocation must be propagated recursively.

When executing a window, an interesting question arises on which window definition should be used. Specifically, we have to choose from a window that appears on the screen and the one stored in the database. Normally, we use the one on the screen since a user should be allowed to modify the window and execute it without having to store it in the database. When the user executes a window for which the user has the X. authorization, however, the definition must be fetched from the database. Otherwise, there is a danger of forging by modifying the window on the screen.

4. CONCLUSIONS

We have presented the facilities and architecture of OBE, a prototype office system and database manager that has been under development at IBM Research. In particular, we have emphasized three aspects of office systems: integration,

usability, and performance. OBE integrates heterogeneous objects by extending the concept of example elements that was originally defined in QBE. OBE ensures a high degree of usability by providing a flexible screen manager and a color-graphics presentation package. OBE implements a memory-resident database manager for high performance.

We have classified integrated office systems into four categories. We have shown that OBE achieves the highest level of integration of heterogeneous office objects through a database system using the power of a relationally complete database language. We have presented a simple and effective implementation technique for this integration.

We have presented a complete memory-resident database system. We have argued that a memory-resident system has an environment vastly different from those of conventional disk-based systems. We emphasize that simply keeping the data in main memory does not automatically guarantee good performance. Instead, all the data structures and algorithms must be designed to maximize the benefit of keeping the data in main memory. Indeed, OBE's new indexing structures, join algorithms, and optimization algorithms have been designed with this goal in view.

We have proposed a new approach to developing the cost model in a CPU-intensive system based on both experimental and analytic methods. In this approach, we identify the system's bottlenecks and construct the cost model on the basis of their unit costs. Through this process, we identified the predicate evaluation as the dominant CPU computation. This contrasts with the case of conventional database systems, in which the cost of I/O accesses to retrieve tuples dominates.

We have analyzed an operating system environment that provides a reasonable approximation for memory-residency of data. By employing a working-set algorithm, the system prevents potential thrashing due to heavy usage of virtual memory. We emphasize that coupling the virtual memory with the working-set algorithm is crucial in realizing a practical memory-resident database system. On the other hand, a pure demand paging scheme would not work in a practical time-sharing environment because of thrashing.

We have presented a concurrency control scheme that is suitable for office environments and memory-resident databases. This scheme uses freshness to validate transactions and mortal locks to improve performance.

An important feature that has been implemented, but not included in this paper, is the interface to a procedural language. We have implemented a facility for invoking OBE from PL.8 programs. This feature is necessary for applications requiring more than standard features of OBE (e.g., interfacing OBE to a different database system).

Currently, a single-user version of the OBE system is fully operational. This version includes all the system components discussed in Sections 2 and 3, except for the concurrency control module and the authorization module. The concurrency control module has been completely designed and implemented but has not been incorporated into the rest of the system. The authorization module has been designed and partially implemented.

We believe that the ideas contained in OBE and novel design decisions made in the prototype implementation are a major contribution toward the research on future integrated, high-performance, user-friendly office systems.

ACKNOWLEDGMENTS

Besides the authors, many other people contributed to the project at different stages. John Ahn implemented a procedural interface to PL.8. Stuart Haber implemented an encryption facility. Gil Barzilai installed the system in a shared segment. Carolyn Turbyfill and Dina Bitton, as consultants from Cornell University, benchmarked the system performance. Kevin Hoffman, Kerri Maner, and Richard Hansen helped with programming. Steve Morgan and Erdwin Chua participated in the initial stage of the project. Finally, Linda Rubin, Bruce Hepp, and Ann Werges helped the project with administrative support, marketing, and documentation.

We thank Susan Betz for the discussion on the scheduling algorithm of VM/370. Fred Lochovsky and the referees suggested several improvements.

REFERENCES

1. AMMANN, A. C., HANRAHAN, M. B., AND KRISHNAMURTHY, R. Design of a memory resident DBMS. *IEEE COMPCON Spring 85 Digest of Papers*. IEEE, New York, Feb. 1985, pp. 54–57.
2. APPLE COMPUTER CO. *Macintosh User's Guide*. Apple Computer, Inc., San Francisco, Calif., 1983.
3. AUSLANDER, M., AND HOPKINS, M. An overview of the PL.8 compiler. In *Proceedings of the SIGPLAN 1982 Symposium on Compiler Construction* (Boston, Mass., June 23–25). ACM, New York, 1982, pp. 22–31.
4. BITTON, D., AND TURBYFILL, C. Performance evaluation of main memory database systems. Tech. Rep. TR 86-731, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., Jan. 1986.
5. BITTON, D., AND TURBYFILL, C. Main memory database support for office systems: A performance study. *Proceedings of IFIP WG8.4 Working Conference on Methods and Tools for Office Systems* (Pisa, Italy, Oct. 19), 1986, pp. 113–134.
6. BITTON, D., HANRAHAN, M. B., AND TURBYFILL, C. Performance of complex queries in main memory database systems. In *Proceedings of the 3rd International Conference on Data Engineering* (Los Angeles, Calif., Feb.). IEEE Computer Society, New York, 1987, pp. 72–81.
7. CAMERON, R. D., AND ITO, M. R. Grammar-based definition of metaprogramming systems. *ACM Trans. Program. Lang. Syst. 6*, 1 (Jan. 1984), 20–54.
8. CODD, E. F. A relational model of data for large shared data banks. *Commun. ACM 13*, 6 (June 1970), 377–387.
9. CODD, E. F. Relational completeness of data base sublanguages. In *Data Base Systems*, R. Rustin, Ed. Courant Computer Science Symposium, vol. 6, May 1971, Prentice-Hall, Englewood Cliffs, N.J., 1972, pp. 65–98.
10. DEWITT, D., KATZ, R. H., OLKEN, F., SHAPIRO, L. D., STONEBRAKER, M. R., AND WOOD, D. Implementation techniques for main memory database systems. In *Proceedings of the International Conference on Management of Data* (Boston, Mass., June 18–21). ACM, New York, 1984, pp. 1–8.
11. ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624–633.
12. FREYTAG, J. C., AND GOODMAN, N. Rule-base transformation of relational queries into iterative programs. In *Proceedings of International Conference on Management of Data* (Washington, D.C., May 28–30). ACM, New York, 1986, pp. 206–214.

13. GARCIA-MOLINA, H. ET AL. A massive memory database system. Unpublished manuscript, Dept. of Electrical Engineering and Computer Science, Princeton Univ., Princeton, N.J., 1983.
14. GRAY, J., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLU, F., AND TRAIGER, I. The recovery manager of the system R database manager. *ACM Comput. Surv. 13*, 2 (June 1981), 223–242.
15. GRIFFITHS, P. P., AND WADE, B. W. An authorization mechanism for a relational database system. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 242–255.
16. GRISWOLD, R. E., POAGE, J. F., AND POLONSKY, I. P. *The SNOBOL4 Programming Language*, 2nd ed. Prentice-Hall, Englewood Cliffs, N.J., 1971.
17. HILLER, F. S., AND LIEBERMAN, G. J. *Introduction to Operations Research*, 3rd ed. Holden-Day, San Francisco, 1980.
18. HUANG, K. T. Visual business graphics query interface. In *Advanced Computer Graphics*, T. L. Kunii, Ed. Springer-Verlag, Berlin, 1986, pp. 233–243.
19. HUANG, K. T., AND ZLOOF, M. M. Business graphics interface to databases. In *Proceedings of Conference on Graphics Interface.* Canadian Information Processing Society (Ottawa, Ontario, Canada, May 22–June 1). 1984, pp. 29–33.
20. IBM. *IBM Virtual Machine/System Product: CMS User's Guide*, SC19-6210-0. 1st ed. IBM Marketing, Sept. 1980.
21. IBM. *IBM Distributed Office Support System/370VSE.* GH12-5137-0. 1st ed. IBM Marketing, May 1981.
22. IBM. *IBM Professional Office System: Programming RPQ P09033 User's Guide*, SH20-5503-1. 2nd ed. IBM Marketing, Nov. 1982.
23. IBM. *Query-by-Example Terminal User's Guide*, SH20-2078. 4th ed. IBM Marketing, Mar. 1983.
24. IBM. *VM/SP: System Logic and Problem Determination Guide (CP)*, LY20-0892-2. 3rd ed. IBM Marketing, Sept. 1983.
25. IBM. *IBM System/370 Principles of Operation*, GA22-7000-9. 10th ed. IBM Marketing, May 1983.
26. IBM. *Top View.* Personal Computer Software. IBM Marketing, 1984.
27. KNUTH, D. *The Art of Computer Programming.* Vol. 3, *Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.
28. KUNG, H. T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2 (June 1981), 213–226.
29. LALONDE, W. R., LEE, E. S., AND HORNING, J. J. An LALR(k) parser generator. In *Proceedings of the IFIP Congress 71* (Ljubljana, Yugoslavia, Aug. 23–28). Elsevier Science, New York, 1971, pp. 151–153.
30. LEHMAN, T., AND CAREY, M. Query processing in main memory database management systems. In *Proceedings of Conference on Management of Data* (Washington, D.C., May 28–30). ACM, New York, 1986, pp. 239–250.
31. LEHMAN, T., AND CAREY, M. A study of index structures for main memory database management systems. In *Proceedings of the 12th International Conference on Very Large Data Bases* (Kyoto, Japan, Sept.). Very Large Data Base Endowment, Saratoga, Calif., 1986, pp. 294–303.
32. LOTUS DEVELOPMENT CO. *Lotus 1-2-3 User Manual.* Cambridge, Mass., 1983.
33. LOTUS DEVELOPMENT CO. *Symphony User Manual.* Cambridge, Mass., 1984.
34. PARTSCH, H., AND STEINBRUEGGEN, R. Program transformation systems. *ACM Comput. Surv. 15*, 3 (Sept. 1983), 199–236.
35. POWER, L. R. EPLEA using execution profiles to analyze and optimize programs. IBM Res. Rep. RC9932. IBM T.J. Watson Research Center, Yorktown Heights, N.Y., Apr. 1983.
36. ROWE, L. A., AND SHOENS, K. A. A form application development system. In *Proceedings of International Conference on Management of Data* (Orlando, Fla. June 2–4). ACM, New York, 1982, pp. 28–38.
37. SEYBOLD, J. Xerox star. *Seybold Report 10*, 16 (1981), Seybold Publications, Media, Pa.
38. SHAPIRO, L. Join processing in database systems with large main memories. *ACM Trans. Database Syst. 11*, 3 (Sept. 1986), 239–264.
39. SOCKUT, G. H., AND KRISHNAMURTHY, R. Concurrency control in office-by-example (OBE). In *Proceedings of IEEE-CS Office Automation Symposium* (Gaithersburg, Md., Apr.). IEEE, New York, 1987, pp. 164–170.

40. WHANG, K.-Y. Query optimization in office-by-example. IBM Res. Rep. RC11571. IBM T.J. Watson Research Center, Yorktown Heights, N.Y., Dec. 1985.
41. WHANG, K.-Y., WIEDERHOLD, G., AND SAGALOWICZ, D. Separability—An approach to physical database design. *IEEE Trans. Comput. C-33*, 3 (Mar. 1984), pp. 209–222.
42. YAO, S. B., HEVNER, A. R., SHI, Z., AND LUO, D. FORMANAGER: An office forms management system. *ACM Trans. Off. Inf. Syst. 2*, 3 (July 1984), 235–262.
43. ZLOOF, M. M. Query-by-example: A data base language. *IBM Syst. J. 16*, 4 (1977), 324–343.
44. ZLOOF, M. M. Security and integrity within query-by-example database management language. IBM Res. Rep. RC6982, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., Aug. 1978.
45. ZLOOF, M. M. QBE/OBE: A language for office and business automation. *IEEE Computer 14*, 5 (May 1981), 13–22.
46. ZLOOF, M. M. Office-by-example: A business language that unifies data and word processing and electronic mail. *IBM Syst. J. 21*, 3 (1982), 272–304.

Received July 1986; revised January 1987; accepted August 1987