# DATABASE PORTALS: A NEW APPLICATION

# PROGRAM INTERFACE†

Michael Stonebraker Lawrence A. Rowe

Department of Electrical Engineering and Computer Science University of California Berkeley, CA 94720

This paper describes the design and proposes
an implementation for a new application program interface to a database management system. Programs which browse through a database making ad-hoc updates are not well served by conventional embeddings of DBMS commands in programming languages. A new embedding is suggested which overcomes all deficiencies. This construct, called a allows a program to request a collection of tuples at once and supports novel concurrency control schemes.

† This research was supported by the Navy Electronics Systems Command under Contract N00039-78-G-0013. There have been several recent proposals for user interfaces that allow a user to ‘‘browse" through a database [CATE80, HERO80, MARY80, ROWE82, STON82, ZLOO82]. Such interfaces allow one to select data of interest (e.g., ‘‘all employees over 40") and then navigate through this data making ad-hoc changes. A simple illustration of a browsing program is described with the aid of figure 1. This program allows a user to ‘‘edit" a relation. It is similar to a full screen, visual text editor (e.g., *vi* [JOY79] or Emacs [STAL81]) except that a relation rather than a text file is edited. This example browser will be used to motivate the need for a new programming language interface to a database management system. In figure 1 data from an relation is displayed.

employee relation

center; c | c | c | c l | n | n | l. name age salary dept _ Ken Johnson 43 25000 sales Sue Keller 40 28000 accounting Dave Smith 52 30000 purchasing Kathy Able 28 22000 accounting George Toms 26 18000 shipping Mike Baker 34 27000 sales find insert delete update quit

Figure 1. Relation editor interface.

Since only a few rows of the relation can fit on the screen at one time, cursor commands are provided to scroll forward and backward. In other words, the screen provides a ‘‘portal" onto the employee relation which the user can reposition. Commands are also provided so a user can edit the data on the screen. For example, Dave Smith’s salary can be changed by repositioning the cursor to the field containing 30,000 and entering a new value. Other operations are listed at the bottom of figure 1. The operation scans forward or backward through the data from the current row until the first row satisfying a user specified predicate is found. The and operations allow the user to enter or remove rows from the table. The operation commits changes to the database so they become visible to other users. Lastly, the operation exits the editor. The data manipulation facilities supported by conventional programming language interfaces [ALLM76, ASTR76, SCHM77, ROWE79, WASS79] allow a program to bind a query to a database cursor,

A database cursor is an embedded query language concept and not the cursor displayed on a CRT. open it, and fetch the qualifying tuples sequentially. Moreover, one can specify that a query or collection of queries is to be a transaction [ESWA76, GRAY78]. The DBMS provides serializability and an atomic commit for such transactions. There are several drawbacks when such an interface is used to implement a browser such as the one discussed above. First, the relation editor can scroll backwards, thereby requiring that the cursor be repositioned to a previously fetched tuple. This feature is not supported by a conventional programming language interface (PLI). Secondly, current PLI’s return one record at a time. When the user scrolls forward or backward, a browsing program would prefer that the DBMS return as many records as will fit on the screen. This protocol would simplify the browsing program code. Next, to implement the operation the browser must scan forward or backward to the first tuple that satisfies a given predicate. Of course, the predicate could be tested in the application program. However, this would duplicate function already present in the DBMS. A cleaner and more efficient solution would be to use the DBMS search logic through a new programming language interface. Lastly, to implement the operation the relation editor must be able to commit updates incrementally during the execution of a single query. Conventional transaction management facilities do not support this kind of update. This paper describes programming language constructs that provide the data manipulation and transaction management facilities required to implement database browsers. The basic idea is to have the database management system support an object, called a We chose this term rather than window to avoid confusion with window managers through which a browser might display its output. that corresponds to the data returned by a single query and allow a program to retrieve data from it. Figure 2 shows a general model for the proposed system.

Figure 2. General Model for Portals.

The DBMS allows a program to selectively retrieve or update data from the portal with a new collection of DBMS commands. A portal can be thought of as a relational that is The query that defines the portal retrieves the data in some particular sequence which establishes the ordering of tuples in the portal. Each tuple will have an extra field that contains a unique sequence number, called a (*LID*) [STON83] which represents the current position of the tuple in the portal. Line identifiers are automatically updated when tuples are inserted into or deleted from the portal so the position of each tuple is always represented by the line identifier. Commands are provided which return collections of portal tuples to the application program. For example, a program can request tuples which:

- match a predicate (e.g., ‘‘all employees over 40"),
- match a given range of LID values (e.g., have an LID between 509 and 522)
- are within a given distance from an indicated tuple (e.g., the tuples less than 12 away from the tuple corresponding to Jones)

Changes made to the data in a portal are propagated to the relations that define it when the update is committed. Six commit modes are supported so that different forms of concurrency control can be implemented by an application program. In addition to modes that allow one or more queries to be treated as an atomic transaction, a mode is provided that allows a transaction to be committed incrementally. This paper describes the design and a proposed implementation of this new application program interface. Section 2 presents the design of the portal abstraction. Section 3 describes a proposed implementation of portals and two performance oriented variations. Then, Section 4 indicates the suggested concurrency control alternatives. Lastly, Section 5 discusses some issues in designing versions of the language constructs for different programming languages and compares portals to other application program interfaces. The application program interface includes language constructs to define a portal, to open and close a portal, to fetch tuples from a portal, to update tuples in a portal, and to further restrict a portal. A portal is defined by specifying a query that selects the tuples that are in it. The general format of a portal definition is similar to the definition of a cursor [ASTR76] and is **let** portal **be** (target-list) [**where** qualification]

[x] indicates that x is optional. where is the name of the portal, is a comma separated list of expressions that define the columns or attributes in the portal, and is a predicate that determines which tuples are in the portal. For example, given an employee relation with the following attributes EMP (name, address, age, salary, years-service, dept) the command **let** p **be** (EMP.name, EMP.salary, birthyear = 1982 − EMP.age) **where** EMP.salary > 25000 defines a portal, *p* that contains the name, salary, and birthyear of employees whose salary is greater than $25,000. The portal definition can be a multiple variable query. For example, given a department relation DEPT (dname, mgr, floor, budget) a portal that contains employee and department information can be defined by **let** p1 **be** (EMP.name, EMP.dept, DEPT.floor) **where** EMP.dept = DEPT.dname This portal contains the name, department, and department floor for all employees. The portal definition can also include programming language variables. For example, the following declaration **let** p2 **be** (EMP.name) **where** EMP.salary > x **and** q includes two program variables, *x* and *q*, that allow the employee's salary and some other predicate (e.g., ‘‘EMP.age < 20") to be substituted at run-time. The definition of a portal causes the query to be parsed and stored by the DBMS. Then, opening a portal causes the values of run-time variables in the portal query to be passed to the DBMS. The open command also specifies the program buffer into which data will be fetched and an optional lock mode for the portal. The general format of the open command is **open** portal **into** variable [**with** lock-mode = n] where is the name of the portal, is a program buffer, and is an integer that identifies a lock-mode. The program buffer is an array of records declared in the application program which determines the maximum number of tuples that can be retrieved from the portal by one command. Lock modes and transaction management are discussed in Section 4. A portal remains open until it is explicitly closed by a close command. The format of a close command is **close** portal Figure 3 shows a PASCAL program fragment that declares a buffer, defines a portal named and opens it. { declare buffer }

**var** buf: **array** [1..10] **of** **record** LID: integer;
name: **array** [1..20] **of** char; salary: real;
age: integer **end**

**begin**

```
        . . .
let p be (EMP.name, EMP.salary, EMP.age)          where EMP.salary > 25000
open p into buf
        . . .  end
```

Figure 3. PASCAL program fragment that declares a portal.

The buffer, named has a field for each user-defined attribute in the portal. A portal also has an implicitly defined field which must be included in the buffer record. Data can be retrieved from the portal and stored in the program buffer by the fetch command. For example, the command **fetch** buf fetches data from and stores it into When the program run-time environment passes this command to the DBMS, it also passes the number of records that can be stored in the buffer. The DBMS returns to the program the number of requested tuples. The data values returned from the portal are automatically converted to the appropriate data types and stored in the buffer. A built-in function is provided that indicates how many records were actually stored in the buffer by the last fetch command. For example, if the portal in figure 3 contained only 5 records, the fetch command above would not fill the buffer. On the other hand, if the portal contained 50 tuples, the command would fetch only the first 10 tuples because only that number can fit in the buffer. The program can retrieve the next 10 tuples by executing a fetch command with a as follows: **fetch** buf **where** p.LID > 10 This command fetches 10 tuples beginning with tuple number 11. Notice that the portal name, in this case is used to reference tuples in the portal. A fetch command can have an arbitrary qualification that will restrict the tuples retrieved to those that satisfy a predicate. For example, the program might want to retrieve employees under 20 who make more than $40,000. The command to retrieve these records is **fetch** buf **where** p.age < 20 **and** p.salary > 40000 The fetch command can also be used to retrieve data by position and to search forwards or backwards. The general format of the fetch command is:

{x|y} indicates that x or y must appear. **fetch** [**previous**] buffer [ {**where | after | before | around**} qualification ] A *position* fetch uses the keyword **after**, **before**, or **around** rather than **where**. A fetch with an **after**-clause indicates that the first tuple that satisfies the qualification and the tuples immediately after it in the portal ordering are to be retrieved. For example, if the following command was executed on the portal defined in figure 3 it would retrieve 10 tuples beginning with tuple number 40: **fetch** buf **after** p.LID = 40 Tuples 40 to 49, if they exist, would be stored in *buf*. The tuple that satisfies the qualification (i.e., tuple number 40) is stored in Subsequent returned tuples follow the selected one in order and do not necessarily satisfy the qualification in the fetch command (e.g., ‘‘p.LID = 40"). In contrast, all tuples returned by a restriction fetch (i.e., one that includes a **where**-clause) must satisfy the qualification. The keyword **before** indicates that the first tuple that satisfies the qualification should be stored at the end of the buffer. Consequently, the buffer will contain the qualifying tuple and the tuples that immediately precede it. The keyword **around** indicates that the qualifying tuple should be stored in the middle of the buffer. The qualification in a position fetch can be an arbitrary predicate such as ... **after** p.LID > 10 **and** p.age < 25 which retrieves tuples beginning with the first one found after tuple number 10 that satisfies the qualification on age. Most browsers also allow users to search backwards. The **fetch previous** command can be used to implement this function. It scans backward through the portal rather than forward. For example, the command **fetch previous** buf **before** p.LID < n **and** q searches for the first record before the current one that satisfies a search predicate. The qualification in a fetch command can be any boolean combination of terms involving portal variables (e.g., ‘‘p.age = 40") and application program variables (e.g., ‘‘q" from the example above). It is also possible to support qualifications involving join terms to other data base relations. A command is provided which allows a programmer to restrict the portal to a smaller subset of the data that it currently contains. The format of the restrict command is: **restrict** portal **where** qualification This command removes from the portal all tuples which do not satisfy the qualification. For example, **restrict** p **where** p.age > 25 removes all employees 25 and under from the portal. A restrict command is equivalent to defining a new portal with a qualification obtained by AND’ing the new qualification to the one that defined the portal. The restrict command functions in much the same way as a marking [RIET81] of a relation, although our other commands and suggested implementation are quite different. The portal abstraction also includes commands to insert, delete, and replace tuples in the buffer. The general format of the replace command is **replace** buffer-reference (target-list) where is a program reference to a record in the buffer (e.g., For example, the following command changes the age of the tuple stored at **replace** buf[4] (age = 25) The insert command appends a tuple to the portal. The general format of this command is: **insert** (target-list) **before** buffer-reference This command inserts the tuple before the buffer array element referenced. The elements in the buffer are moved down to make room for the new data. Since the buffer is fixed size, the last record must be removed from the buffer. The new record is assigned the of the element it is being inserted before, and the of all records following the new element are incremented. The new tuple and its are passed to the DBMS which updates the portal. The last update command allows tuples to be deleted. The format of this command is: **delete** buffer-reference The of the buffer element referenced is set to zero to indicate that it has been deleted. The of all records that follow it in the buffer are decremented. Then, the deleted record and its original are passed to the DBMS which updates the portal. Update commands are passed to the DBMS which records the changes so that subsequent fetches will return the new data. The lock mode selected when the portal is opened will determine when the update is committed to the database. Lock modes will be discussed in Section 4. gsize 12 delim $$ This section describes a basic strategy for implementing the portal abstraction and two improvements on this theme for augmented performance. The basic strategy is to create an ordered temporary relation that contains the portal data. Portal commands would then be translated into conventional queries on this temporary relation. A tuple in the temporary relation must contain a column for each attribute in the portal and a to each tuple used to construct it.

In a relational DBMS, a pointer to a tuple in a relation is called a (For example, given the portal defined in Section 2, **let** p **be** (EMP.name, EMP.age, EMP.dept, DEPT.floor) **where** EMP.dept = DEPT.dname a temporary relation is created by executing the following query: **retrieve into** TEMP(EMP.name, EMP.age, EMP.dept, DEPT.mgr, EMP_TID=EMP.TID, DEPT_TID=DEPT.TID) **where** EMP.dept = DEPT.dname is organized as an ordered relation [STON83], and the DBMS will automatically create and maintain the attribute using an auxiliary storage structure called an (An OB–tree is similar to a B+–tree (i.e., data is stored in the leaves of the tree and a multi-level index is provided to access the data as indicated in figure 4).

Figure 4. An OB–tree.

The leaf pages in the tree contain for tuples in the relation. The LID ordering of the tuples is represented by the order of the in the leaf pages. Hence, traversing the leaf pages from left to right scans the tuples in order (i.e., the first in the left-most page is the tuple with 1). Non-leaf pages contain pointers to the next level of the index or a leaf page and counts of the number of tuples in that subtree. The tree structure and the tuple counts can be used by the DBMS to retrieve or update tuples based on their For example, to find the tuple, the DBMS begins at the root page and selects the subtree that contains the tuple by performing a simple calculation. Assuming that $s_i$ is the number of tuples in the first subtrees, which is defined by the following formula `$s sub i ~=~ sum from {j=1} to {i} ~ count sub j$` the subtree that contains the tuple is pointed to by the entry at `$min from i ~"{" ~ s sub i-1 ~<~l~<=~s sub i ~ "}"$` This process is performed iteratively until the algorithm reaches a leaf page which is guaranteed to contain the tuple. The calculation at intermediate levels of the tree to select a subtree must take into account the number of tuples that precede the first tuple in the subtree. Assuming that this number is the calculation to select the correct subtree for intermediate levels is `$min from i ~"{" ~ x+s sub i-1 ~<~l~<=~x+s sub i ~ "}"$` The value for is $s_{i-1}$ at the next outer level. The for the tuple is stored in the leaf page at entry For example, in figure 4 to find the tuple with LID 17, the algorithm will examine page 1 and select the second subtree because 17 is between 11 ($s_1$) and 18 ($s_2$). Examining page 3 with equal to 11, the algorithm selects page 10 because 17 is between 16 ($x + s_2$) and 18 ($x + s_3$). Page 10 is a leaf and the for tuple 17 is stored in the first entry (*l* - *x*). Insertions into an OB–tree are implemented by inserting a TID for a new tuple into the appropriate leaf page and updating the counts. A standard B–tree split algorithm is used if the leaf page is full [KNUT73]. Deletions and replaces are implemented in a similar way. A complete description of these operations and a prototype implementation of OB–trees are described in [LYNN82]. The DBMS executes portal commands by transforming them into queries on the temporary relation. For example, the fetch command **fetch** buf **where** p.age < 25 is implemented by executing the query **retrieve** (TEMP.LID, TEMP.all) **where** TEMP.age < 25 Recall that the number of records that can fit in the program buffer is passed to the DBMS along with the command so that only the requested number of tuples are returned. A position fetch is implemented by executing two retrievals. Suppose the position fetch was **fetch** buf **after** p.LID > 10 **and** p.age < 25 and that the buffer named *buf* in the program can hold *n* records. First, the LID of the first qualifying tuple is found **retrieve** (l = min(TEMP.LID)) **where** TEMP.LID > 10 **and** TEMP.age < 25 Then the query **retrieve** (TEMP.LID, TEMP.all) **where** l ≤ TEMP.LID **and** TEMP.LID ≤ l+n–1 returns tuples beginning with the tuple. **After** and **around** position fetches can be implemented using a similar technique. Fetch previous commands can be implemented by scanning the OB–tree backwards. Moreover, the execution of fetch commands that include joins with other relations is easy because the portal is stored as a relation. Update commands on the portal are implemented by executing queries to update the temporary relation and writing an intentions list that will be used by the transaction manager to update the affected relation(s). Finally, restriction commands are implemented by creating a new temporary relation. The first improvement on this strategy is to create the temporary relation incrementally. At any time the temporary relation contains all tuples with *LID*'s less than the maximum *LID* that has been fetched thus far. If the data required by a fetch command is in the temporary relation, a retrieval is executed to fetch it. Otherwise, the portal query is resumed to retrieve more data into the temporary relation before the retrieval can be executed. An update command can only modify data that has already been fetched so the data to be changed must be in the temporary. A second improvement is to materialize the portal dynamically and to buffer only a fixed amount of data, say B tuples. For example, one might buffer the tuple with the highest LID requested by the last fetch command and the previous B-1 tuples. However, it is not unreasonable for the DBMS to fetch tuples ahead of the current fetch command. Whenever a fetch command cannot be satisfied by data in the buffer, the portal query is resumed to retrieve tuples with higher LIDs. On the other hand, if the fetch requires data with lower LIDs than any tuples in the buffer, then the portal query must be restarted at the beginning. An OB tree can still be used to support this implementation of a portal. The LID of the first tuple in the buffer must be maintained by the DBMS as tuples are scrolled out of the buffer. This number must be subtracted from the LID used in all portal commands to yield correct responses from the OB tree. It is expected that B can be optimized to provide good response time for most portal users. A user who browses many records without locality of reference could obtain good response time with a large B. On the other hand, a user performing sequential processing would be satisfied with a small value. Lastly, note that a sufficiently large value of B approximates the first improvement described above. The techniques noted above involve creating a temporary relation corresponding to a portal. An alternate implementation would store pointers to the tuples in the primary relations in the temporary relation, using the temporary relation as a kind of secondary index. For example, given the portal definition **let** p **be** (EMP.all) **where** EMP.salary > 20000 the DBMS does not have to make a copy of the data in the relation. The ordered temporary relation could be defined by **retrieve into** TEMP(EMP.TID) **where** EMP.salary > 20000 Fetch commands that involve only the attribute can be implemented by restricting to the qualifying entries and using the to access the tuples. This is the suggested implementation of markings [RIET81]; however it requires an extra disk read to fetch the data so portal commands would perform more slowly. In the next sections we assume that portals are implemented by a dynamic buffering scheme with B tuples in the buffer supported by an OB tree. This section proposes concurrency control facilities for portals. Several lock modes are presented so that a portal user can select an option with appropriate consistency and performance characteristics. These alternatives are now enumerated. The tuples which currently reside in the buffer have a write lock. When a tuple scrolls out of the buffer, its lock is released. Updates are committed as they are received. This is expected to be the normal lock mode for portals. This option is the same as number 1 except that an update is not committed until it scrolls out of the buffer. This mode is appropriate when a user makes several changes that will be scrolled out of the buffer at the same time. Consequently, they would be made visible to other users together. This option is a variant on optimistic concurrency control [BHAR80, KUNG81]. The browsing program does not lock a tuple until it is deleted or replaced. When a tuple in a portal is modified, the tuple(s) from the relation(s) that define the portal are locked and the portal tuple is recreated from the real relations. If the portal tuple to be modified is the same as the recreated tuple, the update is allowed. Otherwise, the modification must be rejected. Updates are committed immediately; hence a browsing application holds locks only for the time required to read, validate and then write desired data. Like other optimistic concurrency control algorithms, a user must restart when an update is rejected. Unfortunately, this requires restarting the portal query and repositioning to desired data. The expense of this restart will make option 3 unattractive except in situations where the probability of conflict is very low. This option is the same as number 3 except that all tuples returned by the last fetch command are locked, refetched, and compared with the recreated values when an update is attempted. This update is committed only if they all are the same. This mode is appropriate if an update is determined by data elsewhere in the scope of the current fetch command. Transactions are defined explicitly by the program. A begin and end transaction command are executed to delimit the beginning and end of the transaction. Consequently, a transaction can be an arbitrary collection of fetch, insert, delete, and replace commands. All commands between opening and closing a portal are considered one transaction. The motivation for these lock modes is as follows. Modes 1 and 2 lock data that is being browsed only while the user can see it (i.e., when it has been fetched). Otherwise, the data can be changed by others. Modes 3 and 4 are similar to modes 1 and 2 except for the use of optimistic concurrency control which may be more efficient. Mode 5 gives control to the application programmer and mode 6 makes the entire browsing session a transaction. Modes 5 and 6 provide the most and least flexibility, respectively. The conventional definition of a transaction is that it is a collection of reads and writes which are atomically committed and serializable

[GRAY78, ESWA76]. Lock modes 3-6 obey this model. For example, lock mode 4 can be implemented as follows: **begin transaction** recreate the most recently fetched tuples **if** tuples changed **then** abort the replace or delete **else** update relation(s) **end transaction** Lock modes 1 and 2, on the other hand, do not correspond to any atomically committed and serializable collection of reads and writes. They both require that locks be held after the end of an atomically committed action. The properties of such locking schemes are an interesting topic for investigation. The implementation of the lock modes 3 through 6 for portals can use a conventional transaction manager that locks physical entities and supports operations to begin, commit, and abort transactions. The general strategy is to update the temporary relation when the update command is executed. In addition, updates for the primary relation(s) are generated and written to a log. These updates are committed at the appropriate time and all locks are released. Lock modes 1 and 2 require slight changes to a transaction manager in that locks can not be released at the time that a transaction is committed. Rather locks are candidates for release when the tuple scrolls out of the buffer. If a portal is defined on a single primary relation, they can be released at this time. However, if a portal is defined by a join, a lock can be released only if the tuple is not used to construct another portal tuple which is currently locked. For example, suppose the portal definition was **let** p **be** (EMP.name, EMP.dept, DEPT.floor, DEPT.mgr) **where** EMP.dept = DEPT.dname and two employees, say Smith and Jones from the toy department, are in the DBMS buffer. Consequently, the two *EMP* relation tuples and the *DEPT* relation tuple would be locked. If Smith's tuple was removed from the portal, the lock on his tuple in the *EMP* relation can be released. However, the lock on the toy department tuple could not be released because it is used to construct Jones' tuple in the portal. In other words, the buffer must be searched to see if the department tuple is used elsewhere before that lock can be released. Hence, deciding if a lock is releasable may be an expensive operation. However, lock reclamation does not have to be performed each time a tuple is removed from the buffer, and it may be advantageous to perform lock releases periodically. Such a mechanism is analogous to garbage collection of free space by a programming language run-time system. This section discusses several issues concerning the design and implementation of the portal abstraction. Then, it discusses the advantages of portals compared to conventional programming language interfaces. First, the language constructs presented in section 2 map a portal into a buffer which is a static 1-dimensional array. The constructs can be generalized to dynamic and n-dimensional arrays. If the programming language into which the constructs are embedded has dynamic arrays, the size of the program buffer can be redefined at run-time. The DBMS can pass a count of the number of records that will be returned by a fetch command before the records are returned. Using this information the run-time support routines in the user program can dynamically allocate an array to hold the returned records. This would relieve the program of executing multiple fetch commands when the number of returned tuples exceeds a static buffer size. Ordered relations can also be generalized to n dimensions [STON83]. In this case a relation can have several LIDs, one for each dimension. The language constructs discussed in section 2 can be easily generalized to support a portal with multiple LIDs which is mapped to an n-dimensional buffer. This feature would be especially valuable to browsers such as SDMS [HERO80] which implement 2 dimensional scrolling. Lastly, a database system that implements portals must be able to save and restore the currently executing query. This is necessary because programs can open multiple portals and because the implementation strategy discussed in Section 3 sometimes requires restarting the portal query. There are several advantages of portals compared to normal programming language interfaces. We enumerate several. All buffering is performed in the portal.

The application program is freed from the responsibility of this task. More flexible concurrency control is possible.

It is certainly possible to support traditional transaction processing with a portal. However, novel locking policies are also possible which can lead to more parallelism in some situations. Code duplication is not required.

A portal can easily implement the ‘‘restrict" command and the ‘‘fetch where qualification" command by translating them into appropriate DBMS commands. A conventional application program interface does not support this function, and providing it in application level code is redundant. Traditional transaction management can be efficiently supported.

A small value for the size of the buffer, B may be chosen for such applications. It is even possible to choose B=1 and effectively obtain a programming language interface similar to that of a conventional cursor-oriented one. In this case portals should be optimizable to provide efficiency nearly equal to a traditional application program interface. As such, portals can be considered to be a generalization of a traditional application program interface. Greater efficiency may be provided in some situations.

The application program must pass control to the DBMS once per fetch command for a portal implementation. On the other hand, control may change hands as often as once per tuple in a cursor oriented application program interface. A browsing application which sequentially scans a relation calls the DBMS once per screen (say each 24 tuples) using a portal whereas it might make a call once per tuple otherwise. Consequently, a portal might outperform a conventional interface for this situation. View management is easily accomplished.

Because portals are defined by queries which may span multiple relations, updating a portal is semantically identical to updating relational views [DAYA78, STON75]. The general problem of updating views is impossible; however, portal updates affect only a single tuple at one time. In this case, the affected tuple has a TID for every tuple in every relation which was used to compose it. One can simply make the obvious update to the specified underlying tuple(s). Although this algorithm is not free from semantic difficulties, it is the only practical candidate in this environment. A new application program interface to a relational database system has been described which makes it easier to implement database browsers. The interface is based on the concept of a *portal* that supports querying and updating an ordered view. Several lock modes were suggested that can be used to implement browsing transactions with varying consistency and parallelism requirements. At the current time OB–trees are operational [LYNN82]. Moreover, performance experiments [STON83] suggest that they perform comparably to normal secondary indexes. Space requirements are also comparable to a normal secondary index. Work is proceeding on implementing the support code for portals using OB trees so that their performance can be compared to traditional interfaces.

Several people have contributed ideas that have been incorporated into this proposal. We want to thank Paul Butterworth, Joe Kalash, Richard Probst, Beth Rabb, and Kurt Shoens for their contributions.

Allman, E. at. al., ‘‘Embedding a Relational Data Sublanguage in a General Purpose Programming Language," *Proc. ACM-SIGPLAN-SIGMOD Conference on Data Abstraction, Definition and Structure*, Salt Lake City, UT, March 1976. Astrahan, M. M., et. al., ‘‘System R: A Relational Approach to Data," *ACM TODS*, June 1976. Bhargava, B., ‘‘An Optimistic Concurrency Control Algorithm and Its Performance Evaluation Against Locking," *Proc. International Computer Symposium*, Taipai, Taiwan, Dec. 1980. Cattell, R., ‘‘An Entity-based Database User Interface," *Proc. 1980 ACM-SIGMOD Conference on Management of Data*, Santa Monica, CA, May 1980. Dayal, U., and Bernstein, P., ‘‘On the Updatability of Relational Views," *Proc. 4th Very Large Data Base Conference*, Montreal, Canada, Oct. 1978. Eswaren, K., et. al., ‘‘On the Notion of Consistency and Predicate Locks in a Relational Database System," *CACM*, Nov. 1976. Gray, J., ‘‘Notes on Data Base Operating Systems," Report RJ–2188, IBM Research, San Jose, CA, Feb. 1978. Herot, C., ‘‘SDMS: A Spatial Data Base System," *ACM TODS*, Dec. 1980. Joy, W., ‘‘The *vi* Text Editor," unpublished working paper, 1979. Knuth, D., Addison Wesley, Reading, MA, 1973. Kung, H. and Robinson, J., ‘‘On Optimistic Methods for Concurrency Control," *ACM TODS*, June 1981. Lynn, N., ‘‘Implementation of Ordered Relations in a Data Base System," Masters Report, EECS Dept., U. C. Berkeley, CA, Sept. 1982. Maryanski, F., ‘‘Query By Forms," unpublished presentation, 1980. van de Riet, R. et. al., ‘‘High Level Programming Features for Improving the Efficiency of Relational Database System," *ACM TODS*, Mar. 1981. Rowe, L. and Shoens, K., ‘‘Data Abstraction, Views and Updates in RIGEL," *Proc. 1979 ACM-SIGMOD Conference on Management of Data*, Boston, MA, May 1979. Rowe, L. and Shoens, K., ‘‘FADS - A Forms Application Development System," *Proc. 1982 ACM-SIGMOD Conference on Management of Data*, Orlando, FL, June 1982. Schmidt, J., ‘‘Some High level Language Constructs for Data of Type Relation," *ACM TODS*, Sept. 1977. Stallman, R.M., ‘‘EMACS The Extensible, Customizable Self-Documenting Display Editor," *Proc. 1981 ACM-SIGPLAN/SIGOA Symp. on Text Manipulation, SIGPLAN Notices*, 16, 6, June 1981. Stonebraker, M., ‘‘Integrity Constraints and Views by Query Modification," *Proc. 1975 ACM-SIGMOD Workshop on Management of Data*, San Jose, CA, May 1975. Stonebraker, M. and Kalash, J., ‘‘TIMBER: A Sophisticated Relation Browser," *Proc. 8th International Conference on Very Large Data Bases*, Mexico City, Mexico, Sept. 1982. Stonebraker, M., et. al., ‘‘Support for Document Processing in a Relational Database System," *ACM-TOOIS*, Apr. 1983. Wasserman, A., ‘‘The Data Management Facilities of PLAIN," *Proc. 1979 ACM-SIGMOD Conference on Management of Data*, Boston, MA, May 1979. Zloof, M., ‘‘Office-by-Example: A Business Language That Unifies Data and Word Processing and Electronic Mail," *IBM Systems Journal*, Fall 1982.