DATABASE PORTALS:

A NEW APPLICATION PROGRAM INTERFACE

by

M. Stonebraker and L. A. Rowe

Memorandum No. UCB/ERL M82/80

2 November 1982

DATABASE PORTALS: A NEW APPLICATION PROGRAM INTERFACE

by

Michael Stonebraker and Lawrence A. Rowe

Memorandum No. UCB/ERL M82/80

2 November 1982

ELECTRONICS RESEARCH LABORATORY

This research was supported by the Navy Electronics Systems Command through grant number N00039-81-C-0569 and the Defense Advanced REsearch Projects Agency through grant number N00039-C-0235.

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

ABSTRACT

This paper describes the design and one proposed implementation of 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 ***portal***, allows a program to request a collection of tuples at once and supports novel concurrency control schemes.

November 2, 1982

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

1. INTRODUCTION

There have been several recent proposals for user interfaces whereby a person can "browse" through a database [CATE80, HER080, 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 is edited rather than a text file. 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 *employee* relation is displayed. 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

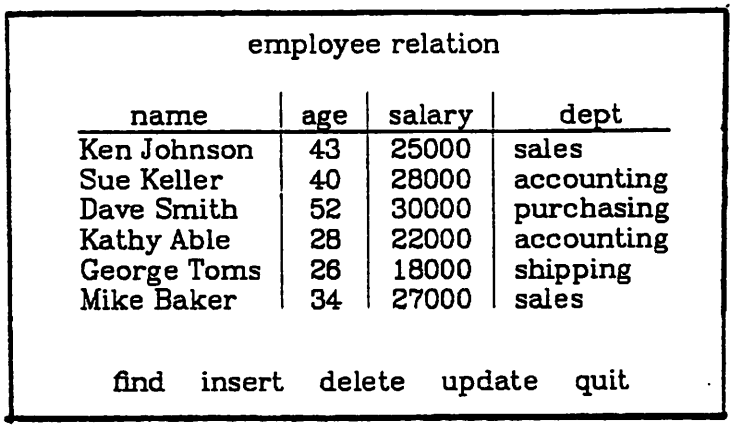

Figure 1. Relation editor interface.

30,000 and entering a new value.

Other operations are listed at the bottom of figure 1. The *find* operation scans forward or backward through the data from the row the CRT cursor is on until the first row is found that satisfies a user specified predicate. The *insert* and *delete* operations allow the user to enter or remove rows from the table. The *update* operation commits changes to the database so they become visible to other users. Lastly, the *quit* 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,[1] 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 to such an interface when used to implement a browser such as the one discussed above. First, the relation editor can scroll

[1] A database cursor is an embedded query language concept not the cursor displayed on a CRT.

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). Second, 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. The program issues one request and receives several records. This protocol simplifies the browsing program code.

Next, the browser must scan forward or backward to the first tuple that satisfies a predicate. This function is needed to implement the *find* operation described above. Of course, the predicate could be tested in the application program but 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 *update* 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 an application program interface that supports 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 *portal*, 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. The DBMS manages portals and 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 *view* that is *ordered*. 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

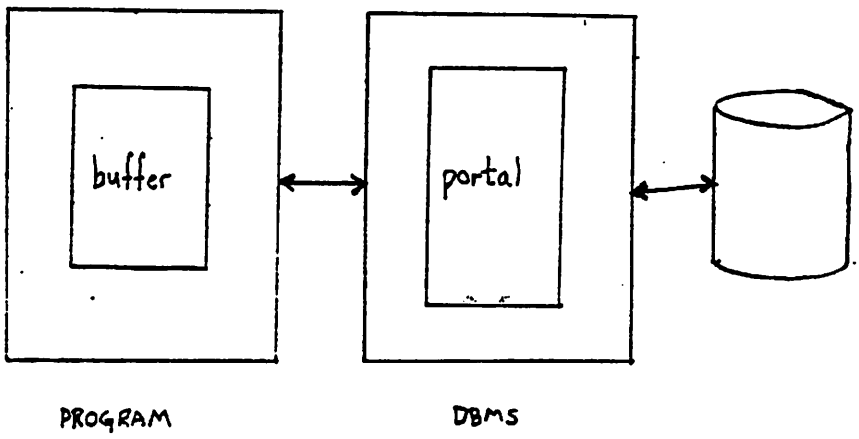

Figure 2. General Model for Portals.

field that contains a unique sequence number, called a *line identifier* (*LID*) [STON82a] that represents the 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"),
- scroll from the current position of the cursor (e.g., the tuples whose LID exceeds the LID of the tuple pointed at by the cursor by less than 24), or
- surround a particular tuple in the portal (e.g., the tuples with an LID within 12 of the LID of 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 one proposed implementation of this new application program interface. Section 2 presents the design of the portal abstraction. Section 3 describes a new collection of tactics that a database system can use to implement portals. Section 4 discusses some issues in designing versions of the language constructs for different programming languages and contains some other comments on their implementation and use.

2. APPLICATION PROGRAM INTERFACE

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[2]

let portal **be** (target-list) [**where** qualification]

where *portal* is the name of the portal, *target-list* is a comma separated list of expressions that define the columns or attributes in the portal, and *qualification* 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

[2] [x] indicates that x is optional.

defines a portal, named *p*, that contains the name, salary, and birthyear of employees whose salary is greater than $25,000.

The query that defines a portal 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 query can also include programming language variables so that it can be defined at run-time. 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. Depending on the implementation tactic chosen by the DBMS, the query might be executed and a temporary relation created to store the portal data. Other implementation tactics are described in the next section. For now, a portal can be thought of as a view. The open command also specifies the program variable into which data will be fetched and an optional lock mode that selects a concurrency control mechanism for the portal. The general format of the open command is

open portal **into** variable [**with** lock-mode = n]

where *portal* is the name of the portal, *variable* is a program buffer, and *n* 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 below.

A portal remains open until it is explicitly closed by a close command. The format of a close command is

close portal

Figure 2 shows a PASCAL program fragment that declares a buffer, defines a portal, and opens it. The buffer, named *buf*, has a field with the same name as each attribute in the portal. Notice that even though the line identifier was not explicitly defined in the target-list of the portal definition, it is included in the buffer record. A column, named *LID*, is implicitly defined for each portal.

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 *p* and stores it into *buf*. When the program run-time environ-

```
        { 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 2. PASCAL program fragment that declares a portal.

ment passes this command to the DBMS, it also passes the number of records that can be stored in the buffer The DBMS returns the number of tuples requested to the program. The attribute 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. The programmer can use this function to determine if any data was returned or if the buffer is only partially filled. For example, if the portal in figure 2 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 **where**-clause 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 *p*, 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:[3]

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

[3] {x|y} indicates that x or y must appear.

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 in figure 2 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 *buf*[*1*]. Subsequent returned tuples follow the selected one in *LID* order and do not necessarily satisfy the qualification. 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 and the tuples immediately before and after it will be fetched.

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. This facility can be used to implement a search operation which scans for the first record after the current one that satisfies a user-specified predicate. The following command fetches the appropriate data

fetch buf **around** p.LID > n **and** q

where *n* is the *LID* of the current record and *q* is a string variable that contains the user-specified predicate. 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 com-

mand

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 portal abstraction also includes update commands to insert, delete, and replace tuples in the buffer. Appropriate commands are also passed to the DBMS which change the portal so that subsequent fetches will see the updated data. When a transaction is committed, portal changes become visible to other DBMS users.

Because portals are defined by queries, some updates cannot be unambiguously mapped onto the underlying relations. This problem is identical to the problem of updating relational views [DAYA78, STON75]. However, since portal

updates affect single tuples only, several special purpose view update algorithms appear possible for this restricted case.

The general format of the replace command is

replace buffer-reference (target-list)

where *buffer-reference* is a program reference to a record in the buffer (e.g., *buf*[*i*]). For example, the following command changes the age of the tuple stored at *buf*[*4*]:

replace buf[4] (age = 25)

This command does not change tuple number 4; it changes which ever tuple was last fetched into *buf*[*4*].

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 is removed from the buffer. The new record is assigned the *LID* of the element it is being inserted before. The *LID*'s of all records following the new element are incremented. The new tuple and its *LID* 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 *LID* of the buffer element referenced is set to zero to indicate that it has been deleted. The *LID*'s of all records that follow it in the buffer are decremented. Then, the *LID* and the deleted record value 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. The following lock modes are provided.

1. The tuples returned by a fetch command are locked, and tuples locked by the previous fetch command are unlocked. Updates are committed when the next fetch command does not span the updated tuples.

2. This option is the same as number 1 except that each update is committed immediately upon a replace, delete, or append command.

3. This option is a variant on optimistic concurrency control [BARG80, 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. If the portal tuple to be modified is the same as the recreated tuple, the update is committed. Otherwise, an error is returned to the program. Append commands are committed immediately. This locking mode allows a browsing application to set no long-term read locks during a session.

4. 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. The 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.

5. Transactions are defined explicitly by the program. A begin and end transaction command are executed to delimit the beginning and end of the transaction. A transaction can be an arbitrary collection of fetch, insert, delete, and replace commands.

6. All commands between opening and closing a portal are considered one transaction.

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 next section describes several tactics for implementing portals.

3. IMPLEMENTATION STRATEGIES

This section describes four strategies for implementing the portal abstraction. It is expected that a data manager would implement most (or all) of them. For each portal the DBMS would select one based on the estimated size of the portal and hints from the user program. Selecting an implementation for a portal is analogous to optimizing a query in a conventional relational system. This section also describes the transaction management facilities needed to implement the six lock modes for portals.

3.1. Portal Implementation

The first strategy for implementing portals 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 disk pointer[4] to each tuple used to construct it. For example, given the portal

```
let p be (EMP.name, EMP.age, EMP.dept, DEPT.mgr)
        where EMP.dept = DEPT.dname
```

defined on the *EMP* and *DEPT* relations described in section 2, a temporary relation is created for this portal 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
```

If *TEMP* is organized as an ordered relation [STON82a], the DBMS will automatically create and maintain the *LID* attribute using an auxiliary storage structure called an *ordered B–tree* (*OB–tree*). 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 3). The leaf pages in the tree contain pointers to the tuples in the relation (i.e., *TID*'s). The LID ordering of the tuples is represented by the order of the *TID*'s in the leaf pages. Hence, traversing the leaf pages from left to right scans the tuples in *LID* order (i.e., the first *TID* in the leftmost page is the tuple with *LID* 1). Non-leaf pages contain a pointer to the next level of the index or a leaf page and a count of the number of tuples in that subtree.

The tree structure and the tuple counts can be effectively used by the DBMS to retrieve or update tuples based on their *LID*. For example, to find the i-th 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 i subtrees, i.e.,

$$s_i = \sum_{j=1}^{i} count_j$$

[4] In a relational DBMS, a pointer to a tuple in a relation is called a *tuple identifier* (*TID*).

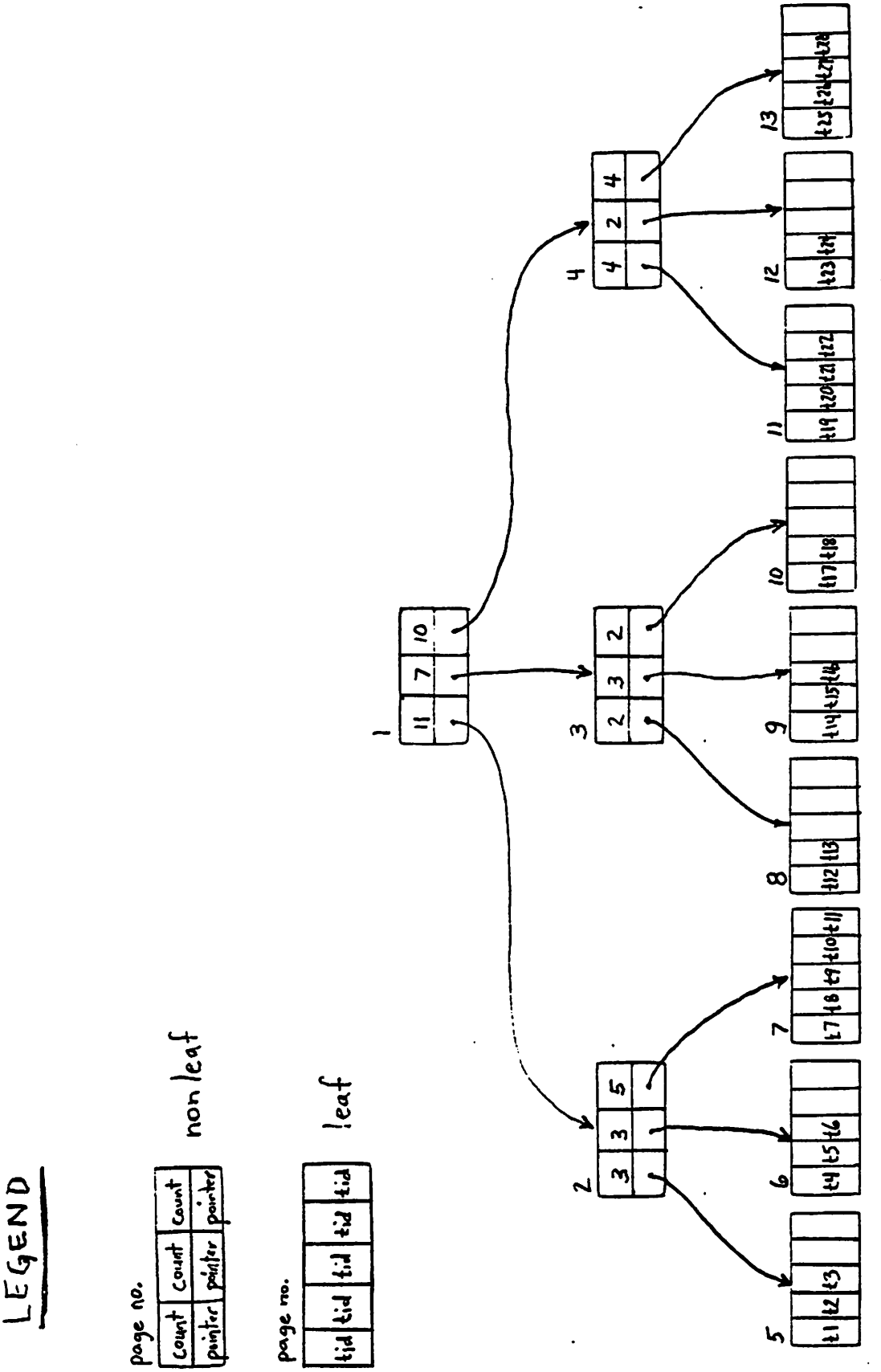

Figure 3. An OB–tree.

the subtree that contains the l-th tuple is pointed to by the entry at

$$\min_i \{ s_{i-1} < l \leq s_i \}$$

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 x, the calculation to select the correct subtree for intermediate levels is

$$\min_i \{ x+s_{i-1} < l \leq x+s_i \}$$

The value for x is s_{i-1} at the next outer level. The *TID* for the l-th tuple is stored in the leaf page at entry $l - x$.

For example, in figure 3 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 x 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 *TID* 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].

In the first implementation strategy, 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 program buffer can hold n records. First, the following query is executed to find the LID of the first qualifying tuple

retrieve (l = min(TEMP.LID)) **where** TEMP.LID > 10 **and** TEMP.age < 25

Then the DBMS can execute a query to return n tuples beginning with the l-th tuple. The query to retrieve these tuples is

retrieve (TEMP.LID, TEMP.all) **where** l ≤ TEMP.LID **and** TEMP.LID ≤ l+n-1

After and **around** position fetches can be implemented using a similar technique.

Fetch previous commands can be implemented by scanning the OB–tree backwards. Fetch commands that include joins with other relations are easy to implement 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 primary relation(s). Finally, restriction commands are implemented by creating a new temporary.

The advantages of this implementation are that large portals can be browsed and that forward and backward searching can be implemented efficiently. The disadvantages are the time and space it takes to create the temporary relation.

A possible improvement to 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 and the retrieval is executed. An update command can only modify data that has already been fetched so the data to be changed must be in the temporary.

Incrementally constructing the temporary reduces the time needed to open the portal because the retrieval to create the temporary is deferred. However, this implementation introduces more variability in the time to execute a fetch command because the portal query may have to be resumed. The space required for the temporary will be reduced if the user specifies a query that generates a large portal, but does not examine all of the data in it.

Another improvement is possible when the relation on which the portal is defined is already maintained by the DBMS as an ordered relation. If the portal definition selects all fields from this relation with no restriction, then the DBMS can directly utilize the underlying primary relation structure and no copy is required.

The second strategy for implementing portals is to store the temporary relation in primary memory. The representation in memory can use an OB—tree or a conventional data structure, such as an AVL-tree, hash table, or array. The implementation of portal commands is identical to that described above. The advantage of this implementation is that portal commands will be faster because primary memory is faster than secondary storage. Update commands will also be faster because only the intentions list has to be written to disk. The disadvantage of this implementation is that only small portals can be stored in pri-

mary memory. Of course, a main memory implementation can also be incrementally materialized to reduce space requirements.

The third strategy for implementing portals is to store pointers to the tuples in the primary relations in the temporary relation (i.e., the temporary is 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 *EMP* relation. The ordered temporary relation could be defined by

retrieve into TEMP(EMP.TID) **where** EMP.salary > 20000

Fetch commands that involve only the *LID* attribute can be implemented by restricting *TEMP* to the qualifying entries and using the *TID*'s to access the *EMP* tuples. The advantage of this implementation is that it reduces the space required to store the temporary relation. The disadvantage is that it requires an extra disk read to fetch the data so portal commands will be slower.

The fourth strategy for implementing portals is to materialize the portal dynamically and to buffer only the amount of data needed by the current fetch command. For example, suppose the browsing program issued a sequence of fetch commands that scrolled forwards through the portal. The DBMS would execute the portal query to generate tuples to be returned by the current command and would keep them in main memory buffers. The next fetch command would be implemented by continuing the portal query and discarding the tuples buffered for the previous fetch. If the browsing program issues a fetch command that requires data that has already been discarded, the portal query must be restarted at the beginning.

The advantage of this implementation is that very large portals can be browsed without having to make a copy of the data. The disadvantages are that some commands will be slow and that fetch previous commands cannot be

implemented efficiently. An obvious improvement to this strategy is to buffer more data than was returned by the last command which would allow some fetch previous commands to be implemented.

3.2. Concurrency Control

The implementation of the six lock modes 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 either committed immediately (lock mode 2) or at a later time (lock modes 1 or 3-6).

Lock modes 1 and 2 can be used only if the portal is implemented by dynamic materialization (i.e., strategy four discussed above). An update is committed when the tuple is not included in the next fetch command (i.e., it is removed from the buffers). The DBMS locks tuples which are buffered in main memory. Locks can be released immediately if the portal is defined on a single primary relation. If a portal is defined by a join, the lock is 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.

Locks do not have to be released on every fetch. For example, it may be advantageous to perform lock releases periodically. Releasing locks is analogous to garbage collection of free space by a programming language run-time system. However, in contrast to garbage collection which is performed when free space is exhausted, a DBMS wants to release locks as soon as possible to increase parallelism.

Lock mode 2 differs from lock mode 1 only in the time at which updates are committed back to the underlying primary relation(s). Locking is implemented the same way it is for lock mode 1.

Lock mode 3 which requires refetching the tuple being changed can be implemented as follows. The primary relation(s) are not locked. When a replace or delete command is executed, the *TID*'s in the temporary relation are used to lock and refetch the values from the primary relation(s). The update is aborted if the value in the primary relation is different than the value in the temporary relation. Otherwise, the primary relations are updated and the locks are released. Lock mode 4 can be implemented in the same way.

Lock mode 5 and lock mode 6 can be implemented in an obvious way. In lock mode 5, the program indicates when the begin and commit operations should be executed. In lock mode 6, the DBMS begins the transaction when the portal is opened and commits updates when the portal is closed.

4. DISCUSSION

This section discusses several issues concerning the design and implementation of the portal abstraction. 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 pro-

gramming 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. 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 exceeded the static buffer size.

Ordered relations can also be generalized to n dimensions [STON82a]. 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.

The second design issue concerns how the portal commands are integrated into existing query language embeddings that do not have an explicit open command (e.g., EQUEL [ALLM76]). The basic idea is to generalize the notion of a range variable to include portal constructs. For example, the command

```
range of buf is p(EMP.all)
   where EMP.age < 40
   with lock-mode=3
```

would be equivalent to

```
let p be (EMP.all) where EMP.age < 40
open p into buf with lock-mode=3
```

Lastly, a database system that implements portals must be able to save and restore the currently executing query because programs can open multiple portals and because several implementation strategies discussed in Section 3 are based on restarting the portal query.

5. CONCLUSIONS

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.

Acknowledgements

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.

References

[ALLM76] 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, Utah, March 1976.

[ASTR76] Astrahan, M. M., et. al., "System R: A Relational Approach to Data," ACM TODS, June 1976.

[BARG80] Bhargava, B., "An Optimistic Concurrency Control Algorithm and Its Performance Evaluation Against Locking," Proc. International Computer Symposium, Taipai, Taiwan, Dec. 1980.

[CATE80] Catell, R., "An Entity-based Database User Interface," Proc. 1980 ACM-SIGMOD Conference on Management of Data, Santa Monica, CA, May 1980.

[DAYA78] Dayal, U., and Bernstein, P., "On the Updatability of Relational Views," Proc. 4th Very Large Data Base Conference Montreal, Canada, October 1978.

[ESWA76] Eswaren, K., et. al., "On the Notion of Consistency and Predicate Locks in a Relational Database System," CACM, November 1976.

[GRAY78] Gray, J., "Notes on Data Base Operating Systems," IBM Research, San Jose, CA, Report RJ 2188, February 1978.

[HERO80] Herot, C., "SDMS: A Spatial Data Base System," TODS, December 1980.

[JOY79] Joy, W., "The vi Text Editor," unpublished working paper.

[KNUT73] Knuth, D., "The Art of Computer Programming, Vol 3: Sorting and Searching," Addison Wesley, Reading, Mass., 1973.

[KUNG81] Kung, H. and Robinson, J., "On Optimistic Methods for Concurrency Control," ACM TODS, June 1981.

[LYNN82] Lynn, N., "Implementation of Ordered Relations in a Data Base System," University of California, Berkeley, CA, Masters Report, Sept. 1982.

[MARY80] Maryanski, F., "Query By Forms," (unpublished presentation).

[ROWE79] Rowe, L. and Shoens, K., "Data Abstraction, Views and Updates in RIGEL," Proc. 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass. May 1979.

[ROWE82] Rowe, L. and Shoens, K., "FADS - A Forms Application Development System," Proc. 1982 ACM-SIGMOD Conference on Management of Data, Orlando, FL, June 1982.

[SCHM77] Schmidt, J., "Some High level Language Constructs for Data of Type Relation," ACM TODS, Sept. 1977.

[STAL81] 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.

[STON75] Stonebraker, M., "Integrity Constraints and Views by Query Modification," Proc. 1975 ACM-SIGMOD Workshop on Management of Data, San Jose, CA, May 1975.

[STON82] Stonebraker, M. and Kalash, J., "TIMBER: A Sophisticated Relation Browser," Proc. 8th International Conference on Very Large Data Bases, Mexico City, Mexico, September 1982.

[STON82a] Stonebraker, M., et. al., "Support for Document Processing in a Relational Database System," Electronics Research Laboratory, Memo M82/15., March 1982.

[WASS79] Wasserman, A., "The Data Management Facilities of PLAIN," Proc. 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass., May 1979.

[ZLOO82] Zloof, M., "Office-by-Example: A Business Language That Unifies Data and Word Processing and Electronic Mail," IBM Systems Journal, Fall 1982.