DOCUMENT PROCESSING IN A RELATIONAL DATA BASE SYSTEM

by

M. Stonebraker, H. Stettner, J. Kalash, A. Guttman and N. Lynn

Memorandum No. UCB/ERL M82/32

6 May 1982

DOCUMENT PROCESSING IN A RELATIONAL DATA BASE SYSTEM

By

Michael Stonebraker, Heidi Stettner, Joseph Kalash

Antonin Guttman, and Nadene Lynn

Memorandum No. UCB/ERL M82/32

6 May 1982

This research was supported by the National Science Foundation through grant number 8007683, the Navy Electronics Systems Command through grant number N00039-81-C-0569, and the Air Force Office of Scientific Research through grant number 78-3596.

DOCUMENT PROCESSING IN A RELATIONAL DATA BASE SYSTEM

by

Michael Stonebraker, Heidi Stettner, Joseph Kalash,
Antonin Guttman, and Nadene Lynn

Dept of Electrical Engineering and Computer Science

University of California

Berkeley, Ca.

Abstract

This paper contains a proposal to enhance a relational database manager to support document processing. Basically, it suggests inclusion of data items which are variable length strings, the notion of ordered relations, and new operators which concatenate and break apart string fields.

I INTRODUCTION

Document processing is currently done by text editors with their own facilities for storage and manipulation of data. It would be desirable to support document processing in a database manager. In this way, database services such as concurrency control and crash recovery could be provided automatically for documents. Also, the access methods and indexing facilities of a data manager would not need to be duplicated in a text editor. Lastly, the capabilities of the database query language can be used to advantage in some kinds of document manipulation. For

example, if a document is stored in a relation:

```
CREATE WORD-SENT-DOC (sentence# = i2,
                      word# = i2,
                      word = c40)
```

then the following QUEL [HELD75] command counts the number of words in the document:

```
RANGE OF W IS WORD-SENT-DOC
RETRIEVE (word-count = COUNT (W.word))
```

More generally, the following code produces a histogram of sentence length.

```
RANGE OF W IS WORD-SENT-DOC
RETRIEVE INTO TEMP (W.sentence#,
              length = COUNT (W.word by W.sentence#))
RANGE OF T IS TEMP
RETRIEVE (T.length,
          freq = COUNT (T.sentence# BY T.length))
```

Hence, many complex document analyses can be performed with simple relational aggregate operators.

Text editors, such as VI [JOY79], manipulate documents which effectively have the format:

```
CREATE LINE-DOC (line# = i4, text = c255)
```

Basically, each line of text is a variable length string with an associated line number. In the above relation this string is limited to 255 bytes. The line# field supports two features of text editors. First, lines of a document are inherently ordered and this field supports such an ordering. In addition, many text editors allow one to move the cursor to a specific line in a document, neccessitating the use of a line# field.

Consequently one can conclude that LINE-DOC is a useful representation for text manipulation while WORD-SENT-DOC is preferred for text analysis. In order to effectively support both formats and the ability to transfer a document from one to the other and back, new database facilities are required. This paper proposes new mechanisms, including:

1) variable length strings
2) ordered relations
3) new substring operators
4) a new break operator
5) a generalized concatenate operator

In the next several sections we discuss these in turn and give examples of their utility. The context in which we present our constructs is the INGRES database system [STON76]. However, these features could easily be applied in most relational environments.

II VARIABLE LENGTH STRINGS

Our first text oriented facility is support for a field which is a variable length string. The syntax we propose is an extension of the CREATE statement as follows:

```
CREATE DOCUMENT (DOC# = i4, TEXT = c0)
```

Here a document relation would have a document number as the first field and the text of the document as a second field; c0 indicates a variable length character string field.

There are two possible mechanisms which could be used to store such variable length fields. The first is to use a database expert [STON80]. This paradigm supports entering a code for the string in the database and storing the actual string externally. Any of the popular tactics to

store and retrieve variable length stings could be used (e.g. first fit, best fit, cyclic best fit, etc.). Using this technique, the database manager need never know about variable length data.

The second method is to extend the access methods to allow variable length strings to be stored and manipulated directly. This could be accomplished easily by placing a length descriptor at the front of each variable length field and another such descriptor at the beginning of the entire record. This approach has been successfully implemented in System R [ASTR76].

III ORDERED RELATIONS

We first discuss simple one-dimensional ordered relations. Then, in Section 3.2 we generalize the notion to multidimensional ordered relations.

3.1 Simple Ordered Relations

In order to support documents which are stored in the format of one line per record, we need a structure as described above:

```
CREATE LINE-DOC (LID = i4, text = c0)
```

Here, we have a line number as the first field followed by the variable length text for that line. However, one has the problem that adding a new line in the middle of the document is a tedious task; all the subsequent lines must be remembered. This problem is also encountered any time a line or lines are moved within the document or a line is deleted.

A more desirable approach would be to have the database manager assist with line numbers. The notion of an ORDERED relation, supported

by a special storage structure is required. An unordered relation R can be ordered as follows:

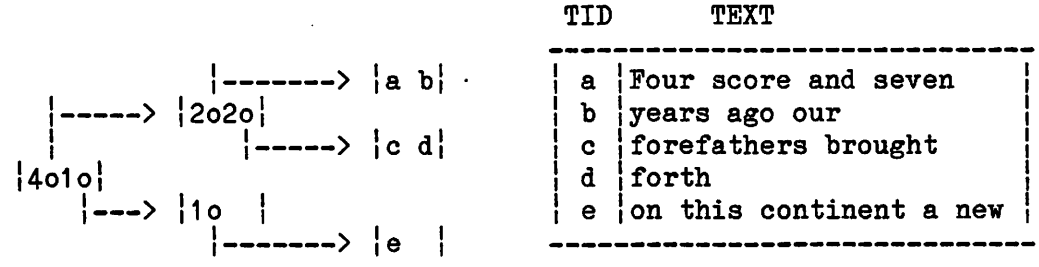
ORDER R WITH LID

This command would have the effect of discarding any keyed primary structure which might exist for R (e.g. B-Tree [COME79] or ISAM index) and building in its place the structure shown in Figure 1.

Here we indicate TEXT, an unordered relation with one field, and its ordered counterpart. Tuples in INGRES have an associated tuple identifier (TID) which is not normally visible to users. The auxiliary structure for an ordered relation is a B-Tree containing TIDs as its leaf

TID	TEXT
a	Four score and seven
b	years ago our
c	forefathers brought
d	forth
e	on this continent a new

Unordered relation



Ordered Relation With Access Path

Figure 1

node data. These TIDs indicate the the ordering of the tuples within the relation when traversed in left-to-right order. At higher levels of the tree pointers to lower tree levels are stored with a count field N indicating how many TIDS are present in the corresponding subtree. Figure 1 shows higher level blocks as containing either one or two pointers and associated count fields. Momentarily updating and search mechanisms for this structure will be discussed.

The user view of an ordered relation R(A, B, C) will have an additional LID field:

R(LID, A, B, C)

This field behaves like an ordinary field most of the time. Hence, one can use the LID field in a query e.g.:

RETRIEVE (R.ALL) WHERE R.LID = 100

This command will retrieve line 100 of the relation R. By examining the root node of the B-Tree one can discover in which subtree S line 100 exists by performing the computation:

$$\min_{S} \left| \text{sum}_{T<S} N(T) \right| >= 100$$

The computation is repeated at each level, saving along the way the lowest LID present in the current subtree. At the leaf level, TIDs are searched sequentially until a count equal to 100 minus the lowest LID on the block is reached. This is the TID for line 100.

A new line can be inserted as follows:

APPEND TO R (LID = 100, A = value, B = value, C = value)

This will add a new tuple to R which can be physically located anywhere in the file containing this relation. The new TID, t, must then be inserted in the B-tree. One first finds the 100th line as above, incrementing by 1 the affected subtree count at each level of the tree. Then t is inserted before the 100th TID. If this insert causes a block split, the conventional B-tree algorithm is used and the subtree counts are adjusted as appropriate.

A line can be deleted as follows:

```
DELETE R WHERE R.LID = 100
```

First, the 100th line is found and removed from the B-tree. The subtree counts must be adjusted as the tree is descended. Subsequently, the tuple with the indicated TID is physically removed from the relation.

Lastly, a line can be moved as follows:

```
REPLACE R (LID = 50) WHERE R.LID = 100
```

This is implemented by a line delete followed by an insert if the new LID is smaller than the old one. If a higher LID is assigned, then the insert must be done first.

It should be noted that secondary indices can be created for ordered relations. Moreover, any QUEL commands which do not affect the LID have the identical semantics as in current QUEL.

One last point is that the LID must be considered the primary key of an ordered relation. Suppose one allowed an ISAM primary key for an ordered relation: for example, employees indexed on empname. Then, assume one renamed Mr. Aardvark to Mr. Zoom. This will necessitate physically changing the TID of the tuple in question, forcing an auxiliary

update to the B-tree. Without back-pointers from the relation to the B-tree, there is no way to do this second update efficiently.

One generalization of simple ordered relations is appropriate. It would be helpful be able to change an ordinary relation with a user supported sequence field into an ordered relation. As such, the sequence field should define the ordering and thus should be suppressed as the same function is supported by an LID. The following syntax extension supports this notion:

```
ORDER R WITH field-1 [ = ASCENDING field-2 ]
                     [   DESCENDING       ]
```

For example, one could execute:

```
ORDER R WITH my-LID = ASCENDING A
```

This would have the effect of sorting R(A,B,C) by ascending values of A before building the structure of Figure 1. A would then be discarded as an attribute of R and would be replaced by an ordering field called my-LID. Consequently, the structure of R would become:

```
R(my-LID, B, C)
```

If a user wished to certain field A in addition to the LID field, he could execute the following commands:

```
RETRIEVE INTO R' (R.all)
ORDER R' WITH my-LID
```

By convention, INGRES creates R' sorted by ascending value of the first field; hence, R' is sorted on A. Consequently, the ordering command will add an LID field for the sorted relation. It should be noted that a "sort by" clause as in SQL [CHAM74] would expedite this process.

The inclusion of ordered relations and variable length strings as fields allows a relational database system to effectively support documents in the format of LINE-DOC from Section I. We now turn to multidimensional ordered relations.

3.2 Ordered Relations in Two Dimensions

Although simple ordered relations are helpful in processing documents, there are cases where a two-dimensional ordering is required. For example, WORD-SENT-DOC from Section I needs to be ordered by both sentence# and by word# within each sentence. Simple ordered relations cannot provide such an ordering. A second example is the application package TIMBER [STON82] which supports placement of icons representing data from a database on a graphics terminal in a manner similar to that of SDMS [HER080]. These icons have a two dimensional position on the screen, and panning in both the vertical and horizontal direction is supported. Again, a two dimensional ordered relation, allowing ordering by column within each ordered row, is required.

To support such two dimensional structures the ordered relation concept must be expanded in two different ways. First, one would want the possibility of having a relation such as WORD-SENT-DOC ordered by word# for each value of sentence#. The proposed syntax is as follows:

```
                          ┌                                    ┐
ORDER R BY field-1         │WITH field-2 = ASCENDING field-3     │
                          └                  DESCENDING        ┘
```

For example, we could perform the desired ordering on WORD-SENT-DOC as follows:

```
ORDER WORD-SENT-DOC BY sentence# WITH LID = ASCENDING word#.
```

The BY clause indicates that the ordering is produced for each value of sentence#. The WITH clause specifies sorting by ascending word# and replacing that field with an LID field. Hence, the user view of WORD-SENT-DOC becomes:

WORD-SENT-DOC(sentence#, LID, word)

The auxiliary structure for this ordered relation is shown in Figure 2.

Here a form of secondary index is created with one row of an index relation for each possible sentence number. The example in Figure 2 shows sentences 1 to 3. For each sentence, there is a pointer field to the root of a B-tree structure of the form of Figure 1. For each sentence#, an LID of I is assigned to the first word and increases for subsequent words.

Obvious extensions to the algorithms of Section 3.1 will support this structure. Hence, on insertion or deletion of new words the LID field will be dynamically adjusted. However the problem still exists that sentence# must be a user supported field. Consequently, whenever a new sentence is added or dropped, the user must adjust his own sentence numbers.

sentence#	pointer
1	a
2	b
3	c

The Auxiliary Structure

Figure 2

The second syntax extension corrects this deficiency:

```
                        |-                    -|
ORDER R WITH field-1    | = ASCENDING field-2 |
                        |_  DESCENDING       _|

                        |-                          -|
                        |,field-3 = ASCENDING field-4 |
                        |_          DESCENDING      _|
```

This syntax allows two ordering fields as follows:

```
ORDER WORD-SENT-DOC WITH
                LID-x = ASCENDING sentence#,
                LID-y = ASCENDING word#
```

This version of the command allows a two dimensional ordering first by sentence# and then by word#. A user now sees the relation:

WORD-SENT-DOC (LID-x, LID-y, word)

The support structure needed is a generalization of Figure 1. The leaf nodes of the B-tree for LID-x do not contain TIDs of tuples in the relation WORD-SENT-DOC. Instead, each contains a pointer to a second B-tree as in Figure 2 which supports a LID-y for each value of LID-x. Again obvious extensions of the algorithms of Section 3.1 will support such a structure. Lastly, note that this structure can be further generalized to N-dimensional orderings.

In the next three sections we turn to QUEL extensions useful in supporting document processing.

IV SUBSTRING OPERATORS

4.1 Extended Wild Cards

A common operation for text editors is the substitution of a second string for each occurrence of a first string. For example, the text editor "ex" allows substitution of lower case letters for the first occurrence of "THE" on each line with the following command:

```
1,$s/THE/the/
```

The "1,$" specifies the command is to affect all lines between the first and last (noted by $) in the document. The command "s/THE/the/" specifies substitution of "the" for each occurence of "THE".

It is currently possible to find each tuple of LINE-DOC which has the string "THE" with the following QUEL command:

```
RANGE OF L IS LINE-DOC
RETRIEVE (L.all) WHERE L.text = "*THE*"
```

Here, * is a wild card matching any variable length string. In addition to *, INGRES currently supports special characters which match any single character, one of a set of characters, a range of characters, or a regular expression of the above. Currently, these wild cards can appear only in the qualification of a QUEL command.

The proposed extension is the inclusion of *i as a "wild card" which matches any string. Values of i must be between 0 and 9. Moreover, *i can also appear in the target list and has the same value for a tuple as it does in the qualification. Consequently the substitution of lower case letters for "THE" can be accomplished as follows:

```
REPLACE L(text = "*1the*2") WHERE L.text = "*1THE*2"
```

With extended wild cards, we can do other text operations. For example, to delete a line up to the first instance of "the" we would perform:

```
REPLACE L(text = "the*2") WHERE L.text = "*1the*2"
```

To flip the portion of the line after "the" with the portion before "the", we would perform:

```
REPLACE L(text = "*2the*1") WHERE L.text = "*1the*2"
```

However, there are a number of operations which are not possible with extended wild cards. These include deleting the first 10 characters on each line and changing the 7th word of a line to a specific pattern. To perform such functions, we require the substring operators as described in the next subsection.

4.2 Substring Operators

At the present time QUEL provides support for referencing a field in a relation as:

```
tuple-variable.field
```

However there is no way to extract a portion of a field. This section proposes facilities to alleviate this weakness. We propose that a field can have one of five additional formats:

```
tuple-variable.field [X]
tuple-variable.field [X, Y]
tuple-variable.field [X, Y)
tuple-variable.field (X, Y]
tuple-variable.field (X, Y)
```

In all cases a substring of the field will be retained and it will be delimited by the strings which match the patterns to be specified by X and Y. The open bracket indicates exluding the string which marks the boundary of the field to be retained, while a closed bracket means including it. In all cases X matches the first substring starting from

the left side of the indicated field while Y matches the first substring starting from the end of the substring matched by X. For example, if the text field of a tuple in LINE-DOC has the value:

the fox jumped over the log

then:

```
L.text ["the"]          has value  "the"
L.text ["the", "the"]   has value  "the fox jumped over the"
L.text ("the", "over")  has value  "fox jumped"
L.text ["over", "fox"]  does not have a value
```

Both X and Y are of the form AB where

```
A = | i |
    | $ |

B = | "QUEL string" |
    |       s       |
    |       w       |
    |       c       |
```

Both A and B are optional. B can be any valid string in QUEL including the extended wild cards from Section 4.1. Moreover, s stands for any sentence, w for any word and c for any character. It may be desirable to add other options, such as digits, in the future. If B is not specified, the default is "?", the wild card that matches any single character.

The first portion A can be any integer i indicating the i-th occurence of a string matching B, or it can be $. The $ deals with changing the left-to-right search order for matching X and Y and is illustrated by example. Suppose LINE-DOC has a text field of

the boy and the girl like the man

The following constructs have the values indicated:

```
L.text ("the", "the")      has value  "boy and"
L.text (1"the",1"the")     has value  "boy and"
L.text (1"the",2"the")     has value  "boy and the girl like"
```

If we did not know how many "the" patterns existed on the line and wanted the pattern between the last two of them, we would use:

```
L.text ($"the", $"the")
```

Here, $ specifies beginning at the end of the line and searching backwards. If Y has a $ specified, it is matched first and then X is matched. If neither i nor $ is specified for X and Y, then the default is i=1.

With these substring operators, we can perform the examples not possible with extended wild cards. For example, to delete the first 10 characters in every line of LINE-DOC we would:

```
REPLACE L( text = L.text(10c,])
```

In order to change the 7th word on each line to be "tuple" we would:

```
REPLACE L( text = concat(L.text[,6w]," tuple ", L.text[8w,]))
```

Hopefully, an improved syntax for the last operation will be found in the future.

The next section discusses an operator to break a text field into all of its component words.

V THE BREAK OPERATOR

The syntax we propose to break apart a text field into its component parts is the following:

```
        |-                                                    -|
BREAK   |tuple-variable.field-1 BY X ASCENDING  field-2|
        |_                           DESCENDING       _|
```

This operator takes the field specified by field-1 and fragments it. The value of X as described in the previous section determines the endpoints of these fragments. In order not to lose semantic information, it is imperative to retain the ordering of the fragments. Consequently, field-2 represents either an ascending or descending sequence number specifying this ordering.

An example should clarify this operator. Suppose we wished to store LINE-DOC in a relation with one word per row. This would be accomplished as follows:

```
RETRIEVE INTO WORD-DOC
              line# = L.LID,
              word = BREAK[L.TEXT BY w ASCENDING word#])
```

This will create a new relation WORD-DOC with three fields as follows:

```
WORD-DOC (line#, word, word#)
```

The field word# is added to the relation and orders the words on a given line. The field word stores the actual words.

The last operator allows one to concatenate groups of records back together. In a sense it is the inverse of BREAK.

VI THE CONCAT OPERATOR

The generalized concatenate operator required is the following:

```
CONCAT [ tuple-variable.field-1
         BY tuple-variable.field-2
         WHERE qualification
         WITH ASCENDING tuple-variable.field-3]
```

Basically, CONCAT is an aggregate operator and is similar to the other QUEL aggregate operators such as SUM or AVERAGE. It groups together all fields which have a constant value for field-2, keeping only those which satisfy the qualification. However, instead of performing a numerical computation, the values obtained are sorted on field-3 and concatenated together.

Both the BY clause and the WHERE clause are optional, as is the case for aggregates. A few examples will further explain the CONCAT operator.

Suppose we wanted to restore WORD-DOC back to the original LINE-DOC. This is accomplished as follows:

```
RANGE OF W IS WORD-DOC
RETRIEVE INTO LINE-DOC(
          W.line#,
          text = CONCAT [W.word WITH ASCENDING W.word#])
```

Now consider the relation WORD-SENT-DOC from Section I. We can form a relation SENT-DOC with one tuple for each sentence as follows:

```
RANGE OF W IS WORD-SENT-DOC
RETRIEVE INTO SENT-DOC(
      W.sentence#,
      text = CONCAT[W.word
                BY W.sentence#
                WITH ASCENDING word#])
```

Application of another CONCAT operator would allow generation of a relation containing the entire document as a single tuple.

```
RANGE OF S IS SENT-DOC
RETRIEVE INTO DOC(
        text = CONCAT[S.text WITH ASCENDING sentence#])
```

It should be clear that two successive BREAK operators could restore SENT-DOC and WORD-SENT-DOC respectively.

VII VIEWS

Some of the representations which we have created for documents are space inefficient; for example, WORD-SENT-DOC contains both a word number and a sentence number for each word in a document. However, as noted in Section I these representations are often easy for an end user or application program to process.

Consequently, one might like to actually store SENT-DOC (which is space efficient) and allow the user to manipulate WORD-SENT-DOC. This requires views involving the extended operators proposed earlier. In the next several subsections we consider the mapping of each operator in turn.

7.1 Ordered Views

Since ordered relations are supported by a physical access path, there is some difficulty with the implementation of ordered views. Clearly, ordered views for unordered relations cannot be supported. Moreover, given an ordered relation R(LID, A, B, C) we can define a view as follows:

DEFINE VIEW V(R.LID, R.B) WHERE QUAL

where QUAL is an arbitrary qualification. If one applies standard query modification techniques [STON75], one can obtain an LID field for the view V. A problem arises because V.LID is not orderd sequentially, but will have gaps when a tuple of R does not satisfy the qualification QUAL. Defensive programming on the part of the user is required in order for a program on an ordered relation to work on an ordered view. In particular, one must not use

```
current LID + 1
```

to obtain the next row of an ordered relation. Rather, one must express this as:

```
min (E.LID where E.LID > current LID)
```

With this proviso, ordered views present no difficulty.

7.2 Substrings

Consider a view which contains a substring from a field in an existing relation. For example, suppose V is defined as

```
DEFINE VIEW V ( R.A, C = R.C[X,Y])
```

for some legal values of X and Y. We can map a REPLACE command such as

```
REPLACE V( C = "new") WHERE QUAL
```

into the following command:

```
REPLACE R(C = concat(R.C[,X), "new",R.C(Y,]) WHERE QUAL
```

Deletes and appends can be analogously transformed.

7.3 The BREAK Operator

Consider the view

```
DEFINE VIEW V (R.A,
               C = BREAK[R.C BY U ASCENDING j])
```

Here j is the name of the field added to V to contain the ordering of the fragments and U is any legal parameter of the BY clause. The update

```
REPLACE V( C = "new") WHERE QUAL and V.j = k
```

This would be converted to:

```
REPLACE R ( C = concat( R.C[,kU),
                        "new",
                        R.C((k+1)U,])
        WHERE QUAL
```

As long as the value k is explicitly stated in the qualification, the above semantics work correctly. If multiple values are specified, e.g.

```
V.j < k
```

or if a comparison is made to another field, e.g.

```
V.j = F.H
```

for some field H in another relation, then there is considerable difficulty.

Basically, we have a non-functional update [STON76] in which we are attempting to update the same tuple several times in a single command. One would have to set up a collection of QUEL updates and loop through them to handle this case.

7.4 The CONCAT Operator

Lastly, consider the view

```
DEFINE VIEW V (R. B
               C = CONCAT[R.C BY R.B ASCENDING R.A])
```

and the update

```
REPLACE V ( C = "new") WHERE QUAL and V.B = k
```

To support such a command we require the following algorithm. Let n be the number of bytes in the fixed length field C. Moreover, let !x! be the smallest integer greater than x.

```
For i = 0,..., !width ("new" / n )!
DO
```

```
RETRIEVE (R.all) WHERE R.A = i
if a tuple is returned then
   REPLACE R( C = "new"[(i-1)*n,i*n]) WHERE R.A =i

else
   APPEND TO R( C = "new"[(i-1)*n, i*n], A = i)

ENDO
```

Unfortunately, there is no single command which can perform this update, so a collection of commands is required.

It is evident that this view is more difficult to map than previous ones. However, one would expect users to want views with data broken apart into component pieces more frequently than they would want concatenated views. Hence, it is possible that one would choose not to support views involving the CONCAT command.

The conclusion of this section is that under many circumstances support for views containing our extended operators is possible. All that is required is defensive programming on the part of the user to avoid the commands which cannot be successfully transformed.

IX CONCLUSIONS

This paper has proposed a small collection of facilities including ordered relations, substring operators, CONCAT and BREAK. These allow a relational database system to be useful for text processing by supporting ordered documents and the ability to compose and decompose text fields.

It is expected that these facilities are straight-forward to add to a system like INGRES and will result in little, if any, performance degradation.

REFERENCES

[ASTR76] Astrahan, M. M. et. al., "System R: A Relational Approach to Database Management," TODS 2, 2, June 1976.

[CHAM74] Chamberlin, D. and Boyce, R., "SEQUEL: A Structured English Query Language," Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 174.

[COME79] Comer, D., "The Ubiquitous B-Tree," Computing Surveys, June 1979.

[HELD75] Held, G. et. al., "INGRES: A Relational Database System," Proc. 1975 National Computer Conference, Anaheim, Ca., May 1975.

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

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

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

[STON76] Stonebraker, M. et. al., "The Design and Implementation of INGRES," TODS 2, 3, September 1976.

[STON80] Stonebraker, M. and Keller, K., "Embedding Experts and Hypothetical Data Bases in A Relational Data Base System," Proc. 1980 ACM-SIGMOD Conference on Management of Data, Santa Monica, Ca., May 1980.

[STON82] Stonebraker, M. and Kalash, J., "TIMBER: A Sophisticated Relation Browser," Electronics Research Laboratory, University of California, Berkeley, Ca., Memo 82/17, January 1982.