IBM Research

VIEWS, AUTHORIZATION, AND LOCKING IN A RELATIONAL DATA BASE SYSTEM

D. D. Chamberlin / J. N. Gray
I. L. Traiger

December 19, 1974

RJ 1486

Yorktown Heights, New York

San Jose, California

Zurich, Switzerland

Limited Distribution Notice

This report has been submitted for publication elsewhere and has been issued as a Research Report for early dissemination of its contents. As a courtesy to the intended publisher, it should not be widely distributed until after the date of outside publication.

Copies may be requested from:
IBM Thomas J. Watson Research Center
Post Office Box 218
Yorktown Heights, New York 10598

RJ 1486 (#22785)
December 19,1974
Computer Science

VIEWS, AUTHORIZATION, AND LOCKING IN A RELATIONAL DATA BASE SYSTEM

D.D. Chamberlin
J.N. Gray
I.L. Traiger

IBM Research Laboratory
San Jose, California 95193

ABSTRACT: It is the contention of this paper that a uniform mechanism can support the notions of view, authorization and locking. A particular mechanism is described where rectangular views are built through the data definition and manipulation facility of the high-level relational language SEQUEL. Both authorization and lock description are then handled by attaching access qualifiers to the columns of the views.

INTRODUCTION

In the interest of brevity we assume that the reader is familiar with the notion of a relational data base. In particular, we assume a familiarity with the work of Codd /1,2,3,4,5/ and Boyce and Chamberlin /6,7,8/. The examples in this paper will be drawn from a data base which describes a department store and consists of three relations:

```
EMP(NAME,SAL,MGR,DEPT)
SALES(DEPT,ITEM,VOL)
LOC(DEPT,FLOOR)
```

The EMP relation has a row for each employee, giving his name, salary, manager's name, and department. The SALES relation gives the dollor volume of each item sold by each department. The LOC relation gives the floor on which each department is located.

In /6/ and /7/, Boyce and Chamberlin introduced SEQUEL, a data sublanguage based on English keywords and intended for interactive problem-solving by users who are not computer specialists. SEQUEL is a unified data definition and data manipulation language, based on the concept of a mapping, which allows users to select certain attributes from those rows of a table which satisfy some criterion. For example, the user may request the names and salaries of all employees in the shoe department:

```
SELECT NAME , SAL
FROM   EMP
WHERE  DEPT = 'shoe';
```

SEQUEL also allows attibutes to be selected from two or more tables which have been joined together according to some

stated criterion. For example:

```
SELECT NAME , FLOOR
FROM   EMP , LOC
WHERE EMP.DEPT = LOC.DEPT;
```

produces a table of the names and floors of each employee by joining EMP and LOC on the DEPT column. (For a more complete treatment of joins, see /4/ and /6/.)

This paper may be viewed as an extension of the ideas in /7/, which developed the concept of view and showed its applicability to extensible data structures, authorization, and integrity constraints, and /14/, which discussed the problems of locking relations and concluded that one must lock logical subsets of relations.

RELATIONS AND VIEWS Defining relations

The SEQUEL system postulates a finite collection of base relations. The description of a relation includes a list of named columns. Each column (e.g. SAL) has the attributes scope (e.g. positive_integer), comparability (e.g. money/time), units (e.g. dollars/year), representation (e.g. DECIMAL(6)) and role description (e.g. "yearly compensation for services rendered").

The formal definition of EMP might be:

```
DEFINE EMP TABLE AS:
     NAME(SCOPE=ALPHA(*),DOMAIN=NAME,REPR=CHAR(*)),
     SAL(SCOPE=POS_INT,DOMAIN=MONEY,UNITS=DOLLARS,REPR=DEC(6)),
     MGR LIKE NAME,
     DEPT LIKE NAME EXCEPT (DOMAIN=DEPARTMENT),
     KEY = NAME,
     ORDER = ASCENDING NAME,
     INDEX NAME;
```

where the expressions to handle NAME, MONEY, POS_INT and

DOLLARS have been previously defined.

Defining views

Simple variations of base relations may be obtained by:

(a) Renaming or permuting columns;

(b) Converting units or representation of a column;

(c) Selecting that subset of the rows of a relation which satisfy some predicate;

(d) Projecting out some columns of a relation

(e) Linking existing relations together into joins, hierarchies or networks which can then be viewed as a single larger table.

Such variations can be obtained using the data definition facility. For example,

```
DEFINE ITALIAN_EMP TABLE AS:
    LIKE EMP EXCEPT (SAL.UNITS=LIRA,SAL.REP=DEC(9));
```

defines a view of Italian employees paid in lira and expands the representation field appropriately. Thereafter, ITALIAN_EMP may be used as a relation. It may be placed anywhere in a SEQUEL statement that one could place the base relation EMP. All fetches from ITALIAN_EMP will have the salary field converted from dollars to lira. All stores into ITALIAN_EMP will store tuples into EMP with the salary field converted from lira to dollars.

To give a more sophisticated example, the table of employees and their locations is defined by:

```
DEFINE EMP_LOC TABLE AS:
        SELECT EMP , LOC
        WHERE EMP.DEPT = LOC.DEPT;
```

This statement defines the view:

EMP_LOC (NAME, SAL,MGR,DEPT,FLOOR).

Any SEQUEL query evaluates to a virtual relation which may be displayed on the user's screen, fed to a further query, deleted from an existing relation, inserted into an existing relation, or copied to form a new base relation. More importantly for this discussion, the query definition may be stored as a named view. The principal difference between a copy and a view is that updates to the original relations which produced the virtual relation will be reflected in the view but will not affect the copy. The view is a dynamic picture of a query, whereas a copy is a static picture.

There is a need for both views and copies. Someone wanting to record the monthly sales volume might run the following transaction at the end of each month:

```
MONTHLY_VOLUME(DEPT,VOL) = SELECT DEPT , SUM(VOL)
                           FROM SALES GROUPED BY DEPT;
```

which computes the dollar volume of each department. The new base relation MONTHLY_VOLUME is defined to hold the answer, and its columns inherit the attributes of the SALES relation. On the other hand, the current volume can be gotten by the view:

```
DEFINE CURRENT_VOLUME (DEPT,VOL) TABLE AS:
                SELECT DEPT , SUM(VOL)
                FROM SALES GROUPED BY DEPT;
```

Thereafter, any updates to SALES will be reflected in the CURRENT_VOLUME view. Again, CURRENT_VOLUME may be used in

the same ways base relations can be used. For example one can compute the difference between the current and monthly volume.

The semantics of views are quite simple. Views in SEQUEL can be supported by a process of substitution in the abstract syntax (parse tree) of the statement. Each time a view is mentioned, it is replaced by its definition. This fits well with the notion of nested mappings. Thereafter, the SEQUEL compiler and interpreter can treat views and nested mappings in a uniform way.

To summarize then, any query evaluates to virtual relation. Naming this virtual relation makes it a view. Thereafter, this view can be used as a relation. This allows views to be defined as row and column subsets of relations, statistical summaries of relations, named joins, hierarchies and networks of relations. This mechanism contributes to:

(a) Data independence: giving programs a logical view of data, thereby isolating them from data reorganization.

(b) Data isolation: giving the program exactly that subset of the data it needs, thereby minimizing error propagation.

Views and update

Since only base relations are actually stored, only base relations can be physically updated. To make an update via a view, it must be possible to propagate the updates down to the the underlying base relations. Any view can support read operations.

If the view is very simple (e.g. ITALIAN_EMP above) then this is straightforward. If the view is a one-to-one mapping of tuples in some base relation but some columns of the base are missing from the view, then update and delete present no problem but insert requires that the unspecified ("invisible") fields of the new tuples in the base relation be filled in with the "undefined" value. This may or may not be allowed by the integrity constraints on the base relation.

Beyond these very simple rules, propagation of updates from views to base relations becomes complicated, dangerous, and sometimes is impossible /5/. Views derived from joins are not third normal form relations, /3/ and hence necessarily have unpleasant update properties. The types of updates which can be supported for various types of view will be discussed in a forthcoming paper. The following basic principles underlie our approach to the problem:

uniqueness rule: An insertion, deletion, or update to a view is permitted only if there is a unique operation which can be applied to the underlying base relations and which will result in exactly the specified changes to the user's view.

rectangle rule: An insertion, deletion, or update via a view must affect only information visible within the rectangle of the view.

These rules are illustrated by the following examples:

```
DEFINE MY_DEPT TABLE AS:
      SELECT EMP , LOC
```

```
WHERE  EMP.DEPT = LOC.DEPT
AND    EMP.MGR  = USER;
```

where USER is a variable selected from the profile of the user of this program. This view is built from the join of the two base relations EMP and LOC. It allows one to see the name, salary, manager, department, and floor of each employee who reports directly to the named user. If the manager's name is Smith, it defines the rectangle: NAMExPOS_INTx('Smith')xDEPARTMENTxFLOOR which is a subset of the cartesian product which underlies the EMP_LOC relation defined previously. No actions using the MY_DEPT view can affect a tuple outside this rectangle. The SEQUEL statement:

```
DELETE MY_DEPT
WHERE SAL > 15000;
```

would not delete all over-paid employees; it would only delete those overpaid employees who work for Smith. It really translates into the statement:

```
DELETE EMP
WHERE SAL > 15000

AND MGR = 'Smith';
```

Since NAME is a key for EMP and DEPT is a key for LOC, MY_DEPT is a simple view which supports update, delete and insert. Of course, any tuple Smith updates must have manager Smith before and after the update. Similarly, any tuple he inserts must have manager Smith, and he can only delete tuples with manager Smith. Each of these restrictions derive from the rectangle rule.

To give an example of the uniqueness rule, imagine that

there is an employee who works in a department not listed in the LOC relation. For example, suppose the tuple (SCOTT,14000,SMITH,BOOK) appears in the EMP relation but that there is no book depatrment in the LOC relation. Because of this, SCOTT will not appear in the join (in the virtual EMP_LOC relation defined previously) and so SCOTT will not appear in Smith's view MY_DEPT (which is a row subset of the EMP_LOC relation). Now if Smith inserts the tuple (FITZGERALD,1300,SMITH,BOOK,5) into his MY_DEPT view, this might be implemented by inserting (FITZGERALD,13000,SMITH,BOOK) into EMP and (BOOK,5) into LOC. These inserts would add both Fitzgerald and Scott to Smith's view since they would add both to the join. This "side effect" is in violation of the uniqueness rule. Because of the possibility of such side effects, the MY_DEPT view cannot support insertions.

Another application of the uniqueness rule disallows support of insert or update to the CURRENT_VOLUME view defined previously, because there is not a unique way of propagating an updated SUM(VOL) to updates on the individual VOL entries in the base SALES relation.

AUTHORIZATION

If only one user has access to a data base, there seems little point in having any authorization mechanism beyond authentication on entry, although one still wants views for

the reasons of conversion, isolation, etc., listed above. However, if several people expect to selectively share data then there must be some mechanism to protect and authorize access. Since one of the merits of a relational data base system is simplicity, we want a simple mechanism to dynamically create and share relations. This simplicity is important for a community of individuals who control their own data, as well as for a more centrally controlled system where authorization is handled by a (human) data base administrator. Following standard practice /7,9,10,11/ we use the view mechanism as the basis of the authorization mechanism. The user has a catalog of named (base and virtual) relations. These give his only access to the data base. Each time a user defines a new base relation, a fully authorized view of it is placed in his catalog. The kinds of authorization we recognize are:

GRANT: the ability to grant this view to someone else or define a view on top of this view.

REVOKE: the ability to selectively reduce or revoke authorizations to this view.

DESTROY: the ability to destroy this view.

INSERT: the ability to insert into this view.

DELETE: the ability to delete tuples in the view.

And for each column of the view:

UPDATE: the ability to update values in this column.

We do not distinguish read access because read restriction can be gotten by eliminating columns from a view. All

columns in a view are readable. Also, we do not distinguish "statistical" access or "manipulative" access. All known proposals for such access control are complicated to understand and easy to subvert. Owens /12/ and Stonebraker /13/ both present a convincing case against distinguishing statistical access. Our approach to statistical access is to use the view mechanism. For example, the CURRENT_VOLUME view described above gives only statistical access to the SALES relation in a very simple and understandable way.

Update authorization is attached to the column of a view rather than to the entire view. Since relational operators distribute over a view, touching each tuple, it makes sense to authorize each visible tuple uniformly. However, some fields (columns) within a tuple are more sensitive than others. For example, a manager may be authorized only to read the name and salary of an employee but to update the floor of an employee in the MY_DEPT view.

So each column of a relation has the attributes: scope, comparability, units, representation, role description, and update authorization. The view itself carries the additional attributes: grant, revoke, destroy, insert, and delete.

Base relations when created have all fields updatable and are fully authorized for all operations. The creator may immediately define a view with non-updatable keys by (for example):

```
DEFINE EMPLOYEE TABLE AS:
        LIKE EMP EXCEPT (NAME.UPDATE = 'NO');
```

A derived view never has greater authorization than its parent view. If the view is not simple, then it automatically loses insert, delete, and update authorization. This is a good example of the interplay between authorization and views.

Granting and revoking authorization

Granting the EMPLOYEE view to Jones conceptually places a copy of the EMPLOYEE definition into Jones' catalog of relations.

Any user having grant authority to EMPLOYEE can grant it to another user with the same or reduced authority. For example:

```
/BASE GRANT EMPLOYEE TO JONES:
     (GRANT = 'NO' , REVOKE = 'NO' , DESTROY = 'NO');
```

This allows Jones to use EMPLOYEE, inserting in it, deleting from it, updating it, but prevents him from destroying the view or revoking it from someone else. Also it prevents him from granting the view to another or defining a view on top of EMPLOYEE. (Otherwise Jones could define an identical view and grant that view.) If Jones already has a relation named EMPLOYEE, the grant will fail.

Since Jones probably does not have the relation EMP in his catalog and since EMPLOYEE is defined in terms of EMP, the view must be interpreted in the context of the definer. On the other hand, the variable USER in the definition of MY_DEPT is local to the user of the view. Standard

mechanisms are used to distinguish the definer's context from the user's context.

A second issue is revocation. When the definer destroys a view, it is deleted from the catalog of all users to whom it was granted. This also invalidates all views which derive from that view. When anyone with revoke authority modifies the authorization of a view, that modification is again propagated to all views derived from that view. Further, anyone with revoke authority for the view may selectively revoke access to the view. For example:

REVOKE EMPLOYEE FROM JONES;

revokes Jones' access to EMPLOYEE. One may imagine base relations and views organized into a hierarchy. If one view entry is granted from another or is defined in terms of another, then changes in the parent will affect the child and all its descendants.

Checking authorization

When a transaction is "compiled" one may tell by the syntax of the statement which views are used by the transaction and for each view one can establish whether it is being granted, revoked, read, inserted into, deleted from or updated. We believe that much authorization will be value dependent and therefore must be checked at the time the transaction is run. For example, if a view is qualified by a selection criterion then each tuple which is inserted, deleted or updated must satisfy this criterion. For example all tuples

entering and leaving MY_DEPT must be checked to see that the value of the MGR field is the name of the person running the transaction.

The entire SEQUEL system is carefully constructed so that mappings can be easily and uniformly composed. Once the update, insert, or delete is resolved to the underlying views, the translated tuples are tested against the selection criteria for the rectangles of those views. This process continues recursively until only base relations remain. If authorization, the uniqueness rule or the rectangle rule is compromised at any step, the operation faults. If a transaction tries to store outside its view, it is given a protection exception. If it tries to read outside its view, it is given the empty set as a response.

LOCKING

If several concurrent transactions access common data then there must be some protocol to synchronize their accesses. This protocol should be invisible to the user. The system is responsible for deciding what locks are required and whether they should be shared or exclusive locks (read or write access). Usually, a SEQUEL statement is the unit of consistency and locks are released at the end of a statement. To get consistency that spans multiple SEQUEL statements, the user may bracket the sequence of statements by the verbs: BEGIN_TRANSACTION and END_TRANSACTION. If two users each want to change the same data, one must wait for

the other to finish. Under certain circumstances, one user may be forced to backup to the beginning of his transaction. If the transaction has not done any terminal input-output this is invisible to the user (except that the transaction takes a long time). If the transaction has done some I/O then backup will be automatic but visible. The issues of deadlock detection, preemption, and backup are resolved by the SEQUEL system using a priority-seniority scheme.

In /14/ it is shown that if each transaction wants to see a consistent view of the data base, then locks must be held to the end of the transaction. It is further shown that the use of indices requires that transactions lock entire relations or that they lock logical subsets of relations.

To see that transactions must lock logical rather than physical subsets of a relation, imagine that Smith wanted to lock for read access all members of his MY_DEPT view. Scanning the EMP relation and locking all tuples with manager Smith would not prevent a new tuple with manager Smith from entering the EMP relation. For example, if Smith made a list of all his employees who make less than 15000 dollars and then made a list of all his employees who make less than 10000 dollars, the second set might not be a subset of the first! This problem of phantom tuples requires that Smith lock the logical set of tuples such that MGR='Smith'. This suggests the concept of predicate locks described as:

(RELATION, PREDICATE, ((F1,A1),... (Fn,An)))

where PREDICATE is a selection criterion giving a row subset of the RELATION and the lock requests access of type Ai to field Fi. The kinds of access are read (shared) and write (exclusive). So for example,

```
(MY_DEPT,(MGR='SMITH'&SAL<10000),
     ((NAME,read),(MGR,read),(SAL,write))),
```

is a lock appropriate to the transaction:

```
UPDATE MY_DEPT SET SAL = 1.10 * SAL
WHERE SAL < 10000;
```

which gives a 10% raise to each underpaid employee in Smith's view. The reason for specifying the kind of access to each column is to allow greater concurrency. Reference /14/ contains a deeper discussion of the resolution of such locks. However their similarity to views should be obvious. Each view describes a rectangle of a (virtual or base) relation. Similarly, a predicate lock describes a rectangle of a relation and the access attributes of that rectangle. The view, authorization and lock mechanisms must each translate operations on a view to operations on base relations. Also, the view, authorization and the lock mechnisms each need to check that each tuple falls within the rectangles prescribed by the views and locks.

In most cases, locks will be finer than views (will be subrectangles of views) but in some complex cases the locks may extend to the entire base relation because the software is not smart enough to find the minimal lock predicate. Figure 1 illustrates the relationship between predicate locks, authorized views, and the base relation.

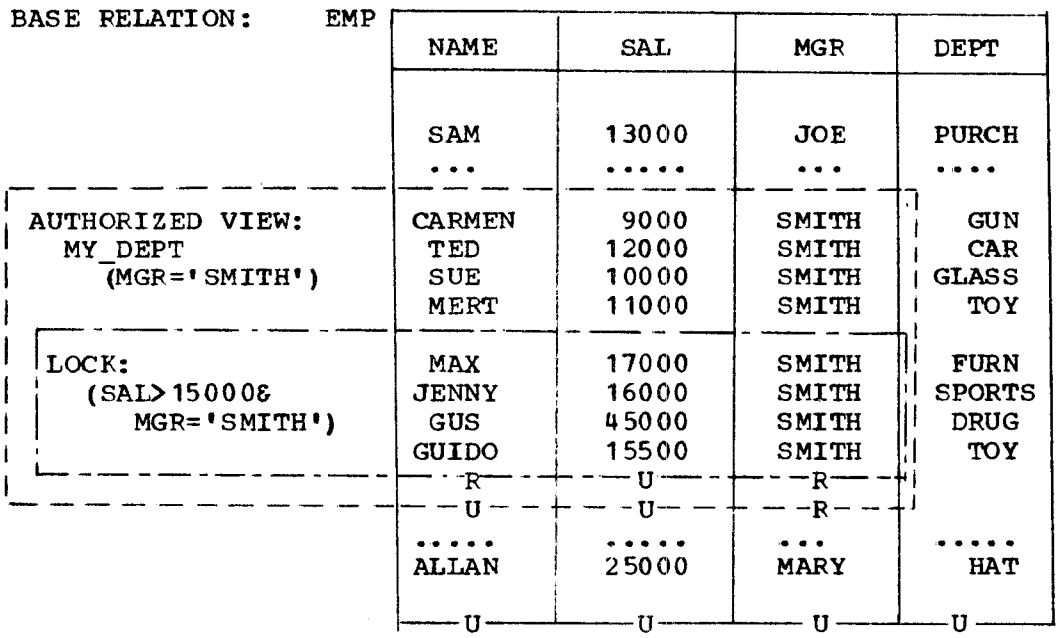

Figure 1. Base relation EMP, viewed from authorized view MY_DEPT and locked with respect to 'SAL>15000' in that view. The capital letters at the base of each rectangle give the authorization of that rectangle for that column.

SUMMARY

Views prescribe what can be seen.

Authorization prescribes what can be done to what is seen.

Locks are a dynamic kind of authorization which prescribe what can be done to what is seen at this instant.

Each of these concepts is an extension of its predecessor and all of them can be based on the concept of a defined relation with a qualifying predicate (a subrectangle of a virtual relation), where each column is tagged with read or write access and the whole view has authority qualifiers such as insert and delete.

STATUS OF IMPLEMENTATION

A single user SEQUEL system with SELECT, INSERT, DELETE, DEFINE, integrity constraints, and very sophisticated index selection was implemented by Morton Astrahan, Don Chamberlin and Paul Fehder and has been operational since June 1974. It is being experimented with at various locations within IBM. Current work is focused on a concurrent user system called MUSE (Multiple User SEQUEL Environment) which will incorporate support for multiple views, and an advanced operating system interface.

ACKNOWLEDGMENTS

We have benefitted from stimulating discussions with Ted Codd and Frank King on update propagation and with Kapali Eswaran and Raymond Lorie on the locking problem.

REFERENCES

(1) E. F. Codd, "A relational model for large shared data banks," CACM, Vol. 13, No. 6 (June 1970) pp. 377-387.

(2) E. F. Codd, "A data base sublanguage founded on the relational calculus," Proceedings of 1971 ACM SIGFIDET Workshop on data description, access, and control, San Diego, California, November 1971.

(3) E. F. Codd, "Further normalization of the data base relational model," Courant Computer Science Symposia,

Vol. 6, Data Base Systems, Prentice Hall, New York, May 1971.

(4) E. F. Codd, "Relational completeness of data base sublanguages," Courant Computer Science Symposia, Vol. 6, Data Base Systems, Prentice Hall, New York, May 1971.

(5) E.F. Codd, "Recent investigations in relational data base systems," Proceedings of IFIP Congress 74, Stokholm, Sweden, August 1974.

(6) R. F. Boyce, D.D. Chamberlin, "A structured English query language," Proceedings of ACM SIGFIDET Workshop, Ann Arbor, Michigan, May 1974.

(7) R. F. Boyce, D. D. Chamberlin, "Using a structured English query language as a data definition facility," IBM Research report: RJ 1318, San Jose, California, Dec. 1973. To appear in CACM.

(8) R. F. Boyce, D. D. Chamberlin, W. F. King III, M. M. Hammer, "Specifying queries as relational expressions," Proceedings of ACM SIGPLAN/SIGFIDET Interface Meeting on Programming Languages and Information Retrieval, Gatisburg, Md., November 1973.

(9) Anononomous, "Information Management System Virtual Storage System (IMS/VS) - System/Application Design Guide," IBM form No: SH20-9025, pp. 3.72-3.73, IBM Corporation, Palo Alto California. 1974..

(10) CODASYL, "Data Base task group report," ACM, New York, 1971.

(11) R.C. Summers, C.D. Coleman, E.B. Fernandez, "A programming language approach to secure data bases," IBM Los Angeles Scientific Center Technical Report: G320-2662, Los Angeles, California, May 1974.

(12) R. Owens, "Primary access control in large scale time-shared decision systems," Project MAC report TR-89, MIT, Cambridge , Mass., July 1971.

(13) M. Stonebraker, E. Wong, "Access control in a relational data base management system by query modification," Electronics Research Laboratory memorandum: ERL-M438, UC Berkeley, California, May 1974.

(14) K.E. Eswaran, J. N. Gray, R. A. Lorie, I. L. Traiger, "The notions of consistency and predicate locks," IBM Reserarch report: (in preparation), San Jose, California, Dec 1974.