Supporting Maybe Algebra in the Associative Search Language Machine (ASLM)

L.L. Miller
Dept. of Computer Science
Iowa State University
Ames, Iowa 50011

and

A.R. Hurson*
Dept. of Electrical Engineering
Computer Engineering Program
Pennsylvania State University
University Park, PA 16802

Abstract

Changes in the user population continues the trend to more simplicity for the user and increased sophistication of the computer systems. In addition, advances in data storage technology have enforced the growth and creation of incomplete data files. A growing number of users require information from the database which cannot be obtained through the operators found in traditional query languages. Such needs have increased the expectation that a system should be able to assist the user in gathering and interpreting the appropriate data. The recent discussion of null values and maybe algebra is an attempt in this direction. However, the current literature has not addressed the practical issues of such theoretical concepts. This presentation examines the inclusion of the maybe algebra operators in a database machine. In addition, modifications designed to increase the time and space efficiency of the maybe algebra operators as well as the quality of the results are discussed.

* This work has been partially supported by Ben Franklin Challenge Grant and Locus Inc.

1. Introduction

Incomplete information continues to be a problem for systems involved in information utilization. The problem of how to handle missing information has been studied by a number of researchers [5,6,7,8,10,18,19,20,24,31,32]. The current discussion of the use of the **universal relation assumption** (URA) [9,15,16,22,26,29], has increased the interest of dealing with null values.

A number of different uses for **null values** have been given in [1,26], but the problem of unknown data value represents the most common usage of null values. Codd [6] introduced a comprehensive extension to relational algebra that allows the user to examine potentially interesting data relationships based in part on unknown (incomplete) data. The theoretical foundations of Codd's new maybe algebra have been examined by Biskup [2]. Recently, Codd [7] has examined the inclusion of the inapplicable data value into the maybe algebra.

Recent developments in technology have led researchers to seek a hardware solution to the design of database systems, namely, the database machines [25]. Such machines have been developed with the purpose of improving the performance of the database operations. In [24], we looked at the design considerations for incorporating the maybe algebra operators into the design of a relational database machine called Associative Search Language Machine (ASLM). In the present work, we examine the implications of implementing Codd's recent work in ASLM.

The architecture of ASLM is briefly overviewed in Section 2. The basic concepts of implementing relational algebra for unknown data values in ASLM are reviewed in Section 3. The impact of Codd's recent work on the ASLM design are presented in Section 4.

2. Database Machines

Several different classifications of database machines have been proposed. Rosenthal [27] has classified these machines as large backend, distributed network data node, and smart peripheral systems. Champine [4] has a similar classification using four classes: backend system, storage hierarchy, intelligent controllers and database computers. Su et. al. [28] have classified database machines as cellular logic systems, backend computers, integrated database machines and high speed associative memory systems. These classifications are similar in that for the most part they are based on how the database machine will be used. Bray and Freeman [3] have proposed a classification based on the concept of parallelism and the location where the data is searched. Based on this criteria, we have the following five groups: single processor indirect search, single processor direct search, multiple processor direct search, multiple processor indirect search and multiple processor combined search.

These classifications suffer from the fact that there is some overlap between the categories. For example, there is no clear way to determine when a smart peripheral system becomes a backend computer as functions are moved from the host to the peripheral system.

However, one common feature of such classifications is the existence of a special purpose hardware system designed to manipulate the database. This approach is based on the need to remove the database functions from the host machine. The hardware specification of the database operations in the backend computer, the ability to overlap operations between the host and the backend, and elimination of the problems associated with the 90-10% rule [12] have made the backend approach more promising than the other categories. In the remainder of this section, the design of a backend database machine, ASLM, is discussed.

2.1 ASLM - a backend database machine

ASLM (Associative Search Language Machine) is a backend database machine [13]. The system is composed of a general purpose frontend machine supported by ASLM (Figure 1). The frontend system acts as the interface between the user and the backend machine. Security validation of the user and the user's query, translation of the user's query into ASL primitives and transmission of the final results to the user are the major functions of the frontend system.

Reducing the semantic gap and alleviating the transportation problem have been the general motivations behind the design of ASLM. Semantic gap reduction has been achieved through: i) the one to one relationship between the set operations and associative operations. ii) hardware implementation of the basic relational operations and iii) elimination of the address accessability of the data. The transportation problem has been resolved by screening the data close to the secondary storage. Solving these problems has determined the organization of ASLM.

Figure 1. ASLM Architecture.

ASLM is composed of four modules: i) controller, ii) secondary storage interface, iii) an array of preprocessors, and iv) a database processor.

Controller: The controller is a microprogrammable control unit. The contents of the writable control memory will be determined by the user's program. More specifically the controller: i) stores the ASL microinstructions generated by the ASL compiler, ii) decodes the ASL microinstructions, and iii) propagates the control sequences to the appropriate modules in the backend. The simplicity of the controller is primarily due to the use of associative hardware for the execution of the ASL primitives, since for the execution of the ASL primitives, there is no need for address generation and the majority of the operations are strictly sequential.

Secondary Storage Interface: The secondary storage interface is a collection of random access memory modules, augmented by some hardware facilities. This module is an interface between secondary storage and ASLM. It accesses blocks of data from secondary storage and distributes them in a tuplewise fashion among the preprocessors.

Preprocessors: Because of the practical limitation on the size of the associative memory, it may not be possible to store all available data in the associative memory in order to perform an operation on the data items. Due to this restriction, all the systems proposed earlier based on fully associative memory, are not capable of handling large databases. In contrast, the design of ASLM overcomes this problem through consideration of two points.

First, queries almost always refer to a subset of attributes in the tuples. Second, in each query, a small subset of tuples will satisfy the search criteria [12].

The preprocessors act as a filter which screens the data, selecting the valid data. The valid data is then placed in an associative stack in the database processor. In addition, the preprocessors perform a projection over the relevant attributes. In a database operation attributes of a relation can be classified into three groups: i) members of the search argument (SA), ii) members of the output set (OS), and iii) the remaining attributes. It is the task of the preprocessors to validate tuples, on the basis of the SA attributes and to project them over the OS attributes.

The module is composed of an array of identical and independent processors which communicate with the database processor and the secondary storage interface. The preprocessors are initialized by the controller according to the query. After the preprocessors have been initialized, they perform operations on the tuples independent of any direct supervision from the controller.

Database Processor: The database processor is composed of a set of associative stacks enhanced by some hardware capabilities for direct implementation of the relational operators. The result of the operations of the array of preprocessors on a relation r with relation scheme R is a relation r_S with relation scheme R_S, which is stored in an associative stack.

$R_S \subseteq R$ and $\forall t_S \in r_S \; \exists \; t \in r$ such that

$t_S = t[OS]$ and $t[SA] \equiv$ search criteria.

The associative stacks act as intermediate storage which holds part of the relation in order to perform the required relational operations on it. The database processor receives the database operations from the controller and performs them on the data stored in the appropriate associative stacks. At the end of the operations defined by the query, the result is transferred to the general purpose computer.

The module is a set of identical and independent associative stacks of size w*d (where w is the width and d is the depth) augmented by some hardware circuits. The associative stacks can be linked together to make a memory size (K*w)*d ($1 \leq K \leq n$, where n is the number of associative modules) capable of holding d tuples of size K*w bits, or they could be linked to form a memory of size w*(K*d) capable of holding K*d tuples of size w bits. Because of the generality of the tuple sizes in a database, this facility enables the system to adjust the available memory modules based on the length of the tuples and the cardinality of the relation. In addition, this increases the space efficiency of the database processor. The independence of the associative modules enhances the modularity and as such the fault tolerance of the system.

Database processor is enhanced by a group of small special purpose modules to facilitate the direct implementation of the relational operators. One of these modules, the concatenation unit is used to reformat and realign tuples in the associative stacks. Such an action is necessary during some

operations such as join where the join attributes in the source tuples should be organized according to the position of join attribute(s) in the target tuples. The concatenation unit is capable of performing basic logic operations on a collection of shift registers (4 units).

ASLM is based on the concept of variable length tuples, where tuples and attribute fields are separated from each other by tuple separator markers and attribute separator markers, respectively. The null value is represented in this format as two adjacent attribute separator markers. The fields of a tuple t defined by t[OS] are expanded by the preprocessors to their predefined size in the normal manner. Fields containing null values are expanded to their appropriate size with don't care symbols. The ASLM architecture is used in Section 3 to examine the implementation issues of Codd's maybe algebra [6].

3. Implementing Maybe Algebra in ASLM

In this section, the ASLM implementation of maybe algebra [24] is briefly overviewed. We assume knowledge of the relational model to the level of [21, 30] and of maybe algebra to the level of [6]. We ignore the obvious changes required to the query language and concentrate on the compilation process and the architectural changes required to add the maybe algebra operators to ASLM.

i) The null value is simply taken as a data value by relational operators such as union, intersection, difference and projection. Such an interpretation of null values means there is no change in the implementation of the operators over traditional relational algebra. The details of these operators can be found in [13].

ii) The result of a maybe select operation is the set of tuples that yield the maybe truth value for the select condition under Codd's

null value substitution principle [6]. A query of the form:

```
maybe select from r where
A = "a" AND B = "b" AND C = "c"
```

can be written as

```
select from r where
(A = "a" OR A = ⊥) AND (B = "b" OR B = ⊥)
AND (C = "c" OR C = ⊥) AND (A = ⊥ OR B = ⊥ OR C = ⊥).
```

In the event the condition is defined over a single attribute, the result in ASLM can be determined by an equality search against $\perp$. For more general search conditions, the maybe select can be implemented in the database processor as a sequence of 2^d-1 select operations, where d is the number of definite attributes used in the search condition. The sequence of operations will be determined by the compiler and hence will be passed to the database processor prior to the execution. The results of the selects are placed together on the resultant associative stack. The operation

```
Maybe select from r where a<1 AND b>2
would require

select from r where a<1 AND b=⊥;
select from r where a=⊥ AND b>2;
select from r where a=⊥ AND b=⊥.
```

Such an approach means that the maybe select operator can be implemented directly in the current design of ASLM. No architectural changes are required to implement the operator. Since the search condition is introduced into the process by the user's query, the 2^d-1 select conditions required can be generated directly by the compiler. Naturally, an increase on the number of searches reduces the performance of the maybe select operator. However, since no changes have been introduced into the design of ASLM, no degradation of performance of the true algebra operators will be experienced.

iii) The maybe join operation places a tuple in the result relation when the test for a join between two tuples results in a maybe truth value. For each tuple in r_1 (source relation), the relation r_2 (target relation) is searched 2^d-1 times, where d is the number of definite attributes in the join set. As in the maybe select, the searches are looking for tuples in r_2 with different combinations of the definite attributes of the join set that contain one or more null values.

In the ASLM design, the join of relations r_1 and r_2 is initiated by loading the relevant data from each relation into two associative stacks. The tuples from the source relation are popped one at a time and passed to the

concatenation unit. The concatenation unit realigns the join set attributes to conform to the positions they have in the target relation. Then, for each reformatted attribute the target relation is searched in associative fashion. In our new environment, compiler distinguishes the true join and the maybe join during the compilation. For true join, two additional sequences of select operations will be initiated to exclude all the tuples in the source and target relations which have at least one null value in their join set attributes. For the maybe join, the concatenation unit is triggered to enumerate all possible permutations of the join set attributes (i.e. 2^d-1 cases as above). This enhancement to the concatenation unit is the only hardware overhead of the maybe algebra over the original design of the system.

Example: maybe join r_1 and r_2 ($A_1 = A_2$ and $B_1 = B_2$)

r_1 (A_1 B_1 C)	r_2 (A_2 B_2 D)	maybe join r_1 and r_2
a_1 $\perp$ c_1	a_1 b_1 d_1	A_1 A_2 B_1 B_2 C D
a_2 b_2 c_2	a_2 $\perp$ d_2	a_1 a_1 $\perp$ b_1 c_1 d_1
	a_2 b_2 d_3	a_1 $\perp$ $\perp$ b_2 c_1 d_4
	$\perp$ b_2 d_4	a_2 a_2 b_2 $\perp$ c_2 d_2
		a_2 $\perp$ b_2 b_2 c_2 d_4

Maybe join has the potential to have drastic effects on the resource utilization and the performance of a database system. For example, if we wish to use the maybe join to join r_1 and r_2 over A = B, where R_1 and R_2 are the respective schemes and A ϵ R_1 and B ϵ R_2, then we get one copy of r_2 for each

tuple t_1 in r_1 where $t_1(A)$ is null. Similarly, we get a copy of r_1 for each tuple t_2 in r_2 where $t_2(B)$ is null.

Interestingly, the loss in physical performance is paralleled by a loss in logical performance. The value of such an operator must be rated by its ability to supply a "useable" set of tuples which the user can examine to see the potential relationships between data values. The relation sizes suggested by the previous example fall beyond the useable category. Zaniolo's generalized join removes the join over two nulls, but still has the potential to produce extremely large relations [33].

Based on this discussion, we present a practical restriction on the maybe join operator. In its current form, the operator will likely cause stack overflow in ASLM when either the number of attributes in the join set is small or the two relations have a high percentage of null values for the join attributes. To deal with this problem, two solutions are being incorporated into the ASLM design.

a) Maybe join over a single attribute is detected by the ASL compiler and ignored. The user is notified that such an operation is expected to result in a low information result. In [24], we showed that the same information can be obtained by examining the two relations.

b) During the maybe join operation on two or more attributes, the system will be able to detect that a large percentage of nulls are resulting in a large number of maybe joined tuples. When the stack containing the maybe join result reaches a predefined load density, the operation is terminated and the user is notified of the low information quality of the data. By using the relative sizes of the two relations involved in the join, the system can set the predefined load density for the stack in question. In general, we feel that this approach is sufficient since it is unlikely that the user will want to look at such low information content data. In cases where the user wants to override the load density check, ASLM allows the

override but uses the predefined load density to periodically query the user if he/she wants to continue.

The use of these two restrictions will result in higher information value results. The first decision can be made during the compilation while the second decision is determined during the execution time. This can be done by the concatenation unit which controls the sequence of the maybe join operations.

Additional methods of improving the performance of the maybe join are given in [14]. For example, the order the operations are performed in can greatly influence the overall performance. For the query

$$r_1 \overset{\text{maybe}}{\bowtie} r_2 \overset{\text{maybe}}{\bowtie} r_3 \overset{\text{maybe}}{\bowtie} r_4 \overset{\text{maybe}}{\bowtie} r_5$$

a better level of performance is expected in general for

$$(r_1 \overset{\text{maybe}}{\bowtie} r_2) \overset{\text{maybe}}{\bowtie} (r_3 \overset{\text{maybe}}{\bowtie} r_4 \overset{\text{maybe}}{\bowtie} r_5)$$

Such optimization of the query will not reduce the number of tuples in the final result, but will reduce the number of intermediate tuples used in the join operations.

iv) The maybe division operator can be implemented using the operators previously defined. The following algorithm provides the sequences of operations for r(R) maybe ÷ s(S) in ASLM:

```
Algorithm maybe divide;
begin
(maybe join r(R) and {t1 ε s(S)}) U (join r(R) and {t1 ε s(S)});
  place result in stack S1;
  project S1 over R-S;
i=2;
while there is an unprocessed tuple in s do
begin
(maybe join r(R) and {ti ε s(S)}) U (join r(R) and {ti ε s(S)});
  place result in stack S2;
  project S2 over R-S;
  intersect S1 and S2 placing the result in S1;
  i:=i+1
end;
Place r÷s on stack S2;
Place S1-S2 on stack S1
end
```

Example r(A B C) s(B C)

a_1 b_1 ⊥	b_1 c_1
a_2 b_2 c_2	b_2 c_2
a_1 b_2 c_2	
a_2 b_1 c_1	
a_3 b_1 c_1	

Stack S_1 after projection over R-S	a_1	Stack S_2 after projection Over R-S	a_1	$S_1 \wedge S_2$	a_1
	a_2		a_2		a_2
	a_3				
Stack S_2 r÷s	a_2		S_1-S_2		a_1

The maybe divide operation is extremely slow, but does not require any additional hardware changes beyond the changes required to implement the maybe join operator. As such, it offers no further degradation of the performance of the true algebra operators.

In the next section, we examine the problem of incorporating Codd's recent extension [7] into the design of ASLM.

4. Incorporating Codd's extension into ASLM

In a recent work [7], Codd has incorporated another type of null value into his maybe algebra, namely, the inapplicable data value. Following Codd's notation we will use the marks A and I to mean unknown but applicable and inapplicable, respectively. Codd restricts substitution as follows

$$\text{db value} \rightleftharpoons A \overset{*}{\rightleftharpoons} I$$

where * means that special authorization is required. Such a substitution pattern allows Codd to retain three valued logic rather than extending it to four valued logic as suggested by Vassiliou [31]. Since the I-mark may eventually be replaced by a database value (e.g. db value), a condition such as COMMISSION > 10K evaluates to maybe when tested against a tuple containing an I-mark for the COMMISSION attribute.

The most significant change required in ASLM to incorporate the I-mark is in handling the storage of the null values (marks in Codds new terminology). As described in Section 2, the ASLM storage format used when only the A-mark is supported as an empty field. Incorporating the I-mark means that two unique symbols must be used at the storage level. For the sake of simplicity, we will assume in this write up that the A and I marks are the two unique symbols.

Currently, the preprocessors identify an empty field and expand it with don't care symbols to fit the field length. Inclusion of the I-mark means that a new policy must be adopted. As in our previous work [14,24], it is our goal to extend the maybe operators imposing as few architectural changes as possible on the ASLM design. It is our feeling that any changes that seriously impede the performance of the more commonly used true algebra operators must be avoided.

Interpretation of the marks is required at two levels. First, at the system level, it is clear that two occurrences of one of the marks uses the same symbol are viewed by the hardware as being equal (i.e. A=A & I=I for different occurrences of the marks). Second at the logical (or Codd's semantical) level, the comparison of two marks must result in the maybe truth value.

To provide the appropriate logical interpretation of the marks without major architectural changes means that we must continue our policy of multiple searches. However, to search for both the A and I marks would be prohibitively expensive. For d attributes in the join set, $\sum_{i=1}^{d} [\binom{n}{i} * \sum_{j=0}^{i} \binom{i}{j}]$ searches would be required.

To avoid such extreme loss of performance for the maybe algebra operators, we propose a compromise approach. For attribute fields consisting of more than one byte, a null value field (either A or I mark) is expanded by the preprocessors with don't care symbols. Therefore with a simple adjustment

in the mask used in the search, such fields can be searched for don't cares as in the current design. For example, the attributes

Attributes	Field Length
A	3 bytes
B	4 bytes
C	2 bytes

containing the values @I@A@A@ (@ is the attribute separator marker) would be expanded by the preprocessors to the form

By adjusting our search mark to block out the first byte of the field, ASLM can use the same search policy described in the previous section for the maybe algebra operators.

For single byte fields, ASLM is forced to duplicate a search, searching first for the A mark and then for the I mark. In the worst case (i.e. all the attributes in either the selection condition or join set are single byte attributes) the operation degenerates to the $\sum_{i=1}^{d} [\binom{d}{i} * \sum_{j=1}^{i} \binom{i}{j}]$ searches. It is a reasonable approach however, since it is extremely unlikely that there will be many single byte attributes involved in either selections or join sets. One alternative approach that we considered was to have the preprocessors replace the A & I marks with don't care symbols prior to expanding the fields. Such an approach maintains the old level of performance but the semantics of the mark is lost in later operations on the maybe result. It is our feeling that retaining this information is worth the occassional loss in performance of the maybe operations caused by single byte attributes.

The main change in the ASLM design is to upgrade the concatenation unit. The changes are modest and do not change the performance of the true algebra operators. In particular, the mask generation and search generation capabilities must be enhanced.

5. Conclusion

The question of inclusion of Codd's extension into the ASLM design has been examined. It is shown that the ASLM design can be upgraded to include the I-mark with only modest changes to the preprocessors and concatenation unit.

As in our previous work [14,24], the primary concern in including Codd's extension is that the performance of the true operators should not be degraded by the changes. As such, the performance of the maybe operators remains slow as compared to the true operators.

Reference List

1. ANSI/XS/SPARC Study Group on Database Management Systems, Interim Report, ANSI, Feb., 1975.

2. Biskup, J., "A Foundation of Codd's Relational Maybe-Operations," ACM TODS, Vol. 8, No. 4, 1983, pp. 608-636.

3. Bray, O.H. and H.A. Freeman, Database Computers, Lexington Books, 1979.

4. Champine, G.A., "Four Approaches to a Database Computer", Datamation, December 1978, pp. 101-106.

5. Codd, E.F., "Understanding Relations", FDT, 7:3-4, Dec. 1975, pp. 23-28.

6. Codd, E.F., "Extending the Database Relational Model to Capture More Meaning", ACM TODS, Vol. 4, No. 4, 1979, pp. 397-434.

7. Codd, E. F., "Missing Information (Applicable and Inapplicable) in Relational databases," SIGMOD Record, Vol.15, No. 4, 1986, pp. 53-78.

8. Date, C.J., "Null Values in Database Management," Proceedings of the 2nd British National Conference on Databases, Bristol, England, July 1982, Also Chapter 14 in Date, C.J., Relational Databases: Selected Writings, Addison-Wesley, 1986.

9. Fagin, R., A.O. Mendelzon, and J. Ullman, "A Simplified Universal Relation Assumption and its Properties", ACM TODS, Vol. 7, No. 3, 1982, pp. 343-360.

10. Grant, J., "Null Values in a Relational Database", Information Processing Letters, Oct. 1977, pp. 156-157.

11. Grant, J. "Partial Values in a Tabular Database Model", Information Processing Letters, August 1979, pp. 97-99.

12. Hsiao, D.K., "Database Computers," Advances in Computers, 1980, pp. 1-64.

13. Hurson, A.R., An Associative Backend Machine for Database Management, Ph.D. Dissertation, University of Central Florida, 1980.

14. Hurson, A.R., and Miller, L.L., "A Database Machine Architecture for Supporting Incomplete Information," Journal of Computer Systems and Science Engineering, Vol. 2, No. 3, 1987.

15 Kent, W., "Consequences of Assuming a Universal Relation," ACM TODS, Vol. 6, No. 4, 1981, pp. 539-556.

16. Korth, H., G.M. Kuper, J.U. Feigenbaum, A. Van Gelder and J. Ullman, "System/U:A Database System Based on the Universal Relation Assumption, ACM TODS, Vol. 9, No. 3, 1984, pp. 331-347.

17. LaCroix, M. and A. Pirotte, "Generalized Joins", ACM SIGMOD Record, Sept. 1976, pp. 14-15.

18. Lien, Y.E., "Multivalued Dependencies with Null Values in Relational Databases," VLDB, Rio the Janeiro, Brazil, 1979, pp. 61-66.

19. Lipski, W., "On Semantic Issues Connected with Incomplete Information Databases", ACM TODS, Vol. 4, No. 3, 1979, pp. 262-296.

20. Lipski, W., "On Databases with Incomplete Information", JACM, Vol. 28, No. 1, 1981, pp. 41-47.

21. Maier, D., The Theory of Relational Databases, Computer Science Press, Rockville, Maryland, 1983.

22. Maier, D., J. Ullman, and M. Vardi, "On the Foundations of the Universal Relation Model", ACM TODS, Vol, 9, No. 2, pp. 283-308.

23. Merrett, T.H., "Relations as Programming Language Elements", Information Processing Letters, Vol. 6, No. 1, 1977, pp. 29-33.

24. Miller, L.L. and A.R. Hurson, "MAYBE Algebra Operators in Database Machine Architecture," Proceedings of the 1986 Fall Joint ACM-IEEE Computer Conference, Dallas Texas, 1986, pp. 1210-1218.

25. Ozkarahan, E., Database Machine and Database Management, Prentice Hall, New Jersey, 1986.

26. Parker, D.S. and P. Atzeni, "Assumptions in Relational Database Theory," Proceedings of the ACM Symposium on Principles of Database Systems, Mar. 1982, Los Angeles, pp. 1-9.

27. Rosenthal, R.S., "The Data Management Machine, A Classification", Third Workshop on Computer Architecture for Non-numeric Processing, 1977.

28. Su, S.Y.W., H. Chang, G. Copeland, P. Fisher, E. Lowenthal, and S. Schuster, "Database Machines and Some Issues on DBMS Standards", National Computer Conference, 1980, pp. 191-208.

29. Ullman, J., "The U.R. Strikes Back", Proceedings of the ACM Symprosium on Principles of Database Systems, Mar. 1982, Los Angeles, pp. 10-22.

30. Ullman, J., Principles of Database Systems, (Second Edition), Computer Science Press, Rockville, Maryland, 1982.

31. Vassiliou, Y., "Null Values in Database Management Denotational Semantics Approach," ACM SIGMOD Conference, 1979, pp. 162-169.

32. Vassiliou, Y., "Functional Dependencies and Incomplete Information," VLDB, Montreal, 1980, pp. 260-269.

33. Zaniolo, C., "Relational Views in a Database System: Support for Queries", IEEE COMPSAC, 1977, pp. 267-275.