TECHNIQUES FOR PROCESSING OF AGGREGATES IN RELATIONAL DATABASE SYSTEMS

by

R. Epstein

Memorandum No. UCB/ERL M79/8

21 February 1979

TECHNIQUES FOR PROCESSING OF AGGREGATES IN RELATIONAL DATABASE SYSTEMS

by

Robert Epstein

Memorandum No. UCB/ERL M79/8

21 February 1979

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TECHNIQUES FOR PROCESSING OF AGGREGATES IN RELATIONAL DATABASE SYSTEMS

by
Robert Epstein
(University of California at Berkeley)

ABSTRACT

Extremely rich and powerful semantics are possible when multiple aggregates are allowed within a single query, and when aggregates are allowed to be nested inside each other. Such queries, however, can be extremely difficult to process. This paper describes a system for processing of arbitrarily complex aggregates on data in relational database systems. The topics include processing scalar aggregates and aggregate functions, combining the processing of multiple aggregates, linking of aggregate functions, optimizing domain references, computing multivariable aggregates, and computing aggregates on unique values. The algorithms described have all been implemented as part of the INGRES relational database system.

Research sponsored by the U.S. Army Research Office Grant DAAG29-76-G-0245.

INTRODUCTION

This paper describes a system for processing arbitrarily complex aggregations of data within a relational database system. Examples will be given using QUEL [HELD75], the query language of the INGRES relational database system. For other discussions on the syntax and semantics of aggregation see [DATE77] and [ASTR76].

This paper begins by defining the two classes of aggregates: scalar aggregates and aggregate functions. It is not difficult to process these aggregates when only a single aggregate appears in a query. Languages such as QUEL allow arbitrarily complex aggregation because extremely rich and powerful semantics are possible when multiple aggregates are allowed within a single query, and when aggregates are allowed to be nested inside each other. Such queries, however, can be extremely difficult to process. In sections II, III, and IV this paper presents simple efficient algorithms for processing these aggregations. There are several optimization techniques particular to aggregates which can dramatically improve the processing cost of a query. These optimizations are discussed in sections V and VI. Lastly, section VII contains an overview of the processing strategy.

The examples presented in this paper are all based on the two relations:

```
employee(number, name, salary, manager, startdate)
dept(number, name, store, floor, manager)
```

I. TYPES OF AGGREGATION.

Aggregates can be divided into two catagories which we shall call scalar aggregates and aggregate functions. A scalar aggregate can be computed independently of the query it is contained in, and when computed will yield a single scalar value. For example, the query to find the average salary in the employee relation can be expressed as:

```
range of e is employee
retrieve (avgsal = avg(e.salary))
```

This query will compute a single tuple with one domain named "avgsal" whose value will be the average of the salary domain of the employee relation.

Aggregates can have an optional qualification. For example, what is the average salary of those people who work for manager 128?

```
range of e is employee
retrieve (avg128 = avg(e.salary
                  where e.manager = 128))
```

Aggregate functions differ from scalar aggregates in that they return a set of values. The data to be aggregated is logically partitioned by one or more attributes. For example one could ask how many people work for each manager?

```
retrieve (cnt = count(e.name by e.manager))
```

This query would yield a separate count value for each unique manager in the relation. It is convenient to name the value being aggregated as the aggregate expression and the value determining the set grouping as the by-list (in this case "e.manager"). The "by-list" is not limited to a single domain but rather it can be an arbitrarily complex comma separated list of expressions. If the by-list has more than one expression, then it is defined to be the concatenation of the expressions.

In summary, the generic form of a scalar aggregate is:

agg_operator(agg_expression where qualification)

The generic form of an aggregate function is:

agg_operator(agg_expression by by_expression1,

..., by_expressionN where qualification)

The aggregate operators which commonly occur in database systems are:

count	sum	avg
min	max	countu
sumu	avgu	

The first five aggregates (count, sum, avg, min, max) have the obvious arithmetic meaning. The "unique" aggregates (sumu, avgu, countu) guarantee that the set of values of the expression being aggregated contains only unique values (i.e. duplicates have been removed).

Scalar aggregates are "local" in the sense that they are independent of the query in which they are nested. Aggregate functions are not "local". The attributes in the "by-list" are logically linked to the corresponding relations in the remaining query. As a simple example consider the query:

```
retrieve (e.manager, cnt=count(e.name by e.manager))
```

Both occurrences of the manager attribute refer to the same entity. The linking between the two references to "manager" is implicitly defined. If this linking did not occur then the query would be the same as:

```
range of e is employee
range of f is employee
retrieve (e.manager, cnt=count(f.name by f.manager))
```

This query yields the cross product of all "e.managers" with the set of counts. Thus defining the "by-list" of an aggregate function to be global to the query which contains it yields an intuitive and useful definition.

The linking of the "by-list" will occur as a function of the variables used -- not the attributes used. (A variable is a reference to a relation. For example, in the previous query, both "e" and "f" were declared to be variables which reference the "employee" relation.) As an example of linking the by-list, consider the query: "List each employee number together with the count of other employees who work for the same manager."

```
range of e is employee
retrieve (e.number, cnt=count(e.name by e.manager))
```

Since the variable "e" occurs in both the by-list (e.manager) and elsewhere in the query (e.number), the aggregate function is linked to the "e.number" field. The algorithms for determining how to link aggregate function will be discussed in section III. Note that some database management systems avoid the problem of linking aggregate functions by disallowing queries such as the one above.

II. PROCESSING AGGREGATES.

A scalar aggregate consists of an aggregate expression and an optional qualification. The steps for processing are straight forward and will surprise no one.

(1) Allocate an initially zero tuple to hold the aggregate result. Allocate a counter, initially zero, which can be used to count the number of tuples which qualify.

(2) For each tuple which satisfies the qualification, update the aggregate result and increment the counter.

The count of the number of tuples which satisfies is used for computing the avg function and is used for "min" and "max" for recognizing the first tuple. The aggregate in the query can then be replaced by the computed scalar value.

Aggregate functions require the maintenance of an aggregate value, count field, and the actual by-list value for each unique occurrence of the "by-list". The number of possible values in the "by-list" is potentially as large as the cardinality of the relation being aggregated. In the general case, the set of values can be maintained by creating a temporary relation which has domains for the count field, aggregate value, and "by-list". For example, the aggregate:

avg(e.salary by e.manager)

can accumulate its results in a relation of the form:

temp(count, manager, average)

The algorithm consists of:

(1) create a temporary relation with the necessary attributes.

(2) for each tuple which satisfies the qualification:

(2a) if a tuple with the identical by-list already exists in the "temp" relation, then update that tuple.

(2b) else append to the "temp" relation, a new tuple with the initial correct values.

At the end of processing a relation will exist with exactly one tuple for each manager. That tuple will hold the correct aggregate value for that manager.

This algorithm has the property that values of the by-list will be excluded if no corresponding tuple satisfies the qualification. For example, the query: "For each manager, determine how many employees have worked for the company more than five years."

```
range of e is employee
retrieve (e.manager, oldpeople =
        count(e.name by e.manager
                where e.startdate < 1975))
```

If a manager does not manage someone who started before 1975 then that manager will not appear in the "temp" relation. It is unreasonable to allow the count to be undefined; it should be zero.

This problem can be avoided if the "temp" relation is initialized with the by-list values. Thus before actually processing the aggregate function, if it is qualified, project the by-list into the "temp" relation. This guarantees that every value of the by-list will be present and have the default value of zero. The query above can be processed in two steps:

```
range of e is employee
retrieve into temp(count = 0, manager = e.manager)
```

followed by the actual aggregation into "temp". There is a general problem of what value to give an aggregate operating on an empty set; that is, one for which no tuples satisfy? It is generally reasonable to define sum and count on the empty set as equal to zero. Avg, min, and max could also be defined as zero. If the "null" value is supported, it may be reasonable to define avg on the empty set as equal to "null". Min and max could be initialized to the largest and smallest (respectively) possible values of the domain, or +- infinity if the computer hardware supports such values. Regardless of what is chosen for the default value, if the aggregate is qualified, it must be initialized to the default value.[1]

Notice that in step (2a), the "temp" relation is always accessed by equality on the manager domain. It will be accessed once for each tuple in employee which satisfies the

---

[1] For the INGRES implementation, we have chosen to default to zero in all cases. One reason for this is that the host hardware (PDP-11) does not support undefined or infinite values. We feel that the cost to simulate such values in software is prohibitive.

qualification. The optimal storage structure for "temp" is to key on manager.

An alternative processing strategy for computing an aggregate function is to first project the needed domains and then sort on the by-list being careful not to remove duplicates. Results will still be accumulated into a temporary relation but the reference pattern will be different. Since the tuples are sorted in order of the by-list, each tuple read will have either the same by-list as the previous tuple, or it will be an entirely new by-list and there will be no more references to any previous by-lists. If the tuples were unsorted, then it is possible that each tuple read would have a different by-list than the previous tuple and the references to the temporary relation would be somewhat random. The trade-off is the cost to project and sort verses the cost to randomly search the temporary relation.

We shall now briefly analyse when each method wins. Suppose we know that:

N = number of tuples to be aggregated.
U = number of unique values in the by-list.
P = number of pages needed in the aggregate temp relation. (U / number of tuples per page)
B = number of main memory page buffers available.

We will assume that "temp" relation is either in a hash structure or some order preserving structure such as B-tree or ISAM. The following table summarizes the logical page accesses for each case.

	hash	b-tree or isam
random distribution	N	N + index pages
sorted distribution	U	P

The minimum access is P accesses. Let's now conjecture that

B >= P. In this case, all page accesses will be in core and there will be only P physical page accesses in any of the above cases. Since most aggregate function result tuples are small, there are many per page. In addition, the value of U tends to be small in many typical cases. Thus the assumption that B >= P is commonly true in practice. To the extent that this holds, the best structure to use is hash, and sorting does not help. If B < P and U is large, then sorting clearly wins.

III. NESTED AGGREGATION AND LINKING OF AGGREGATE FUNCTIONS

The order in which several different aggregates are performed may be critical. Here are two examples.

```
range of e is employee
retrieve (tiny = min(e.salary
        where e.salary != min(e.salary)))

retrieve (cheap = min(e.salary),
                  costly = max(e.salary))
```

The first query asks for the second smallest salary. One aggregate is nested inside the other. Since the outer aggregate is dependent on the result of the inner aggregate, they must be done in the order of innermost first. In the second query, it is immaterial which aggregate is processed first since they are not nested inside each other.

All aggregates are nested either in another aggregate or in the main query. While aggregates may be deeply nested, for processing purposes, one need only keep track of one level of nesting. For simplicity we shall refer to the

query in which an aggregate is immediately nested as the "outer query". The aggregate itself we shall refer to as the "inner query" or "inner aggregate".

Once a scalar aggregate is processed, it can be replaced in the query by its scalar value. When an aggregate function is processed, it can be replaced by a reference to the domain in the "temp" relation which was used to process the aggregate. Aggregate functions have an additional complexity -- the by-list must be linked to the query in which the aggregate was nested. For example the following query:

```
range of e is employee
retrieve (e.manager,
                manavg = avg(e.salary by e.manager))
```

is transformed into

```
range of t is temp
retrieve (e.manager, t.avg)
        where e.manager = t.manager
```

where "temp" contains the manager and average salary by the processing algorithm in section II. The qualification (e.manager = t.manager) was added to correctly link the relation holding the aggregate function (temp) to the remaining query. This is a direct result of the definition that the variables in the by-list are global to the query. A more complicated example is

```
retrieve (e.manager,
        manmin = min(e.salary by e.manager),
        manmax = max(e.salary by e.manager))
```

after processing the aggregates, this becomes

```
range of t1 is temp1
range of t2 is temp2
retrieve (e.manager, t1.min, t2.max)
        where e.manager = t1.manager
        and     e.manager = t2.manager
```

The algorithm for linking a single aggregate function to the query it is nested in, is as follows:

(1) Generate a list of all variables which occur in the outer query or in the by-list of any aggregate functions contained in the outer query. Do not include variables which appear in scalar aggregates or in the non by-list portion of an aggregate function.

(2) Generate a list of all variables in the by-list of this aggregate.

(3) If the variables in the by-list intersect the variables in the outer query, then link the aggregate "temp" relation to the outer query on all the by-list domains.

It may happen that there are no variables in common between the by-list and the outer query. In that case no linking is required.

Suppose that aggregate2 is nested inside aggregate1 which is nested inside the main query. Aggregate2 is local to aggregate1 only. If any linking is done it is to link the "temp" relation of aggregate2 to the variables in aggregate1. For example, for each manager find the second highest paid employee:

```
retrieve (e.name, e.manager)
where
        e.salary = max(e.salary by e.manager
                where max(e.salary by e.manager)
                        != e.salary)
```

This will be processed in three steps.

(1) Compute max(e.salary by e.manager) into "temp1".

(2) Replace the aggregate with its value set. The query is now:

```
range of t1 is temp1
retrieve (e.name, e.manager)
where
```

```
e.salary = max(e.salary by e.manager
        where t1.max != e.salary
        and     t1.manager = e.manager)
```

(3) Compute the next aggregate into "temp2". The remaining query is now:

```
range of t2 is temp2
retrieve (e.name, e.manager)
where
        e.salary = t2.max
and
        e.manager = t2.manager
```

IV. MULTIVARIABLE AND UNIQUE AGGREGATION.

Multivariable and unique aggregates can be processed by first projecting those tuples which satisfy the qualification of the aggregate into a temporary relation. The aggregate can then be expressed in terms of the temporary relation and processed as previously described.

For example, suppose we want to know how many people who earn more than $10,000 work on each floor:

```
range of e is employee
range of d is dept
retrieve (d.floor, people =
        count(e.number by d.floor
                where e.salary > 10000
                and e.manager = d.manager))
```

1) First reduce the aggregate to one variable:

```
retrieve into temp1(e.number, d.floor)
where
        e.salary > 10000
and
        e.manager = d.manager
```

2) Next process the aggregate into "temp2"

```
range of t1 is temp1
count(t1.number by t1.floor)
```

3) The remaining query is now:

```
range of t2 is temp2
retrieve (d.floor, t2.count)
        where d.manager = t2.manager
```

If the aggregate calls for unique values, then the temporary relation (in this case "temp1") will have to be sorted to remove duplicates. The optimal sort order is to sort on the by-list first. Thus when the results are aggregated, the tuples will already be sorted into the correct sets.

Suppose we want to know how many managers there are. Consider the queries:

```
    range of e is employee
(1) retrieve (mancnt = count(e.manager))
(2) retrieve (mancnt = countu(e.manager))
```

Query 1 will give the number of occurrences of e.manager; this is of course only the cardinality of the employee relation. Query 2 gives the count of unique occurrences; which is the desired result. This query can be processed in the identical manner as the query above:

1a) First reduce the aggregate

```
retrieve into temp(e.manager)
```

1b) Next eliminate duplicates.

```
remove duplicates from temp
```

2) Process the one variable aggregate

```
range of t is temp
retrieve (mancnt = count(t.manager))
```

3) The remaining query is now:

```
retrieve (mancnt = CONSTANT)
```

The issue of unique values is clearly defined when there is only one relation involved. For multi-variable queries, the definition of non-unique aggregates is vague. To solve a multivariable query, it is necessary to reduce the query to one variable and then aggregate. Presumably the aggregate processor can call the normal query processor to solve an aggregate-free, multi-variable query. The presence or absence of duplicates in the resulting relation depends on how the query is actually processed [YOUS78]. Furthermore, in a system which makes dynamic processing decisions, the manner in which a query is processed can change as the information about the relations change. The definition is vague primarily because allowing duplicates does not fit in well with relational theory [CODD70].

As examples, consider the query: "What is the average salary for employees on each floor?"

```
range of e is employee
range of d is dept
retrieve (d.floor, flooravg =
        avg(e.salary by d.floor
                where e.manager = d.manager))
```

If there are two people with the same salary on the same floor, then "avg" will yield a different answer than "avgu". When reducing the query to one variable, those duplicates must be preserved. Now consider the query:

```
range of e is employee
range of d is dept
```

```
retrieve (e.manager, mancnt =
        count(e.number by e.manager
                where e.manager = d.manager
                and d.number > 10))
```

Suppose there is a manager who manages two different departments. In that case, the employees of that manager may be counted twice, depending on how the query is processed. The solution in this case is to use a "countu" instead of a "count".

V. COMBINING AGGREGATES.

There are many cases when more than one aggregate can be processed in one scan through the data. Doing so is purely an optimization step and is not in any way essential to the semantics of the query. The advantages of recognizing such cases are enormous. For only a slight increase in algorithmic complexity, multiple aggregates can be processed for nearly the same cost as doing just one. We will describe a sufficient set of conditions for determining whether two or more aggregates can be processed at the same time. Most of the rules are intuitively obvious.

Rule 1 - Not nested.

Two aggregates cannot be run together if one is nested inside the other since one aggregate depends on the final result of the other.

Rule 2 - Same dependence on uniqueness.

Aggregates can be divided into three catagories: (1) unique aggregates (sumu, countu, avgu), (2) non-unique aggregates (sum, count avg), and (3) don't care aggregates (min, max). Unique and non-unique aggregates can

never be run on the same data since by definition, one requires that any duplicate tuples be removed, and the other requires that duplicates remain. The following table describes the conditions under which two aggregates can be combined.

	unique	non-unique	don't care
unique	?	never	?
non-unique	never	always	always
don't care	?	always	always

The "?" in the above table means that the answer depends on the expression being aggregated. If both aggregates are on the same expression then the two aggregates can be safely combined otherwise they cannot.

Rule 3 - Same by-lists

If two aggregate functions are being combined, they must have the same by-lists otherwise the "temp" relation cannot be guaranteed to have exactly one tuple for each occurrence of the by-list.

Rule 4 - Same qualifications.

The reason for combining two aggregates is to save scanning the relation being aggregated. If two aggregates had different qualifications, they might require different access paths. The simpliest rule to adopt is that if the qualifications are not identical, do not combine the aggregates.

Rule 5 - Must range over the same variables

If the aggregates expressions are not over the same variables, then the inclusion of one with the other might involve the generation of duplicates which otherwise might not be present. This rule is essential if rule 4 is relaxed, otherwise, the queries which fall in this catagory are usually not meaningful queries.

VI. ELIMINATING SOURCE VARIABLES BY USING THE BY-LIST

It frequently happens that references to the original source relation can be replaced by references to the by-list. The goal is to reduce the number of variables in the query. An example should serve to illustrate the technique. We already have seen that the query:

```
range of e is employee
retrieve (e.manager,cnt=count(e.name by e.manager))
```

will be transformed into

```
range of t is temp
retrieve (e.manager,t.count)
        where e.manager = t.manager
```

Since the domain "t.manager" is itself a complete projection (with duplicates removed) of "e.manager", the references to "e.manager" can be replaced yielding:

```
retrieve (t.manager, t.count)
        where t.manager = t.manager
```

We are detecting that the two occurrences of "e.manager" are the same. This greatly simplifies the query as it reduced the number of relations involved from two to one. Whenever this type of by-list replacement is done, the linking term (e.manager = t.manager) becomes unnecessary and can easily be detected and removed.

There are, of course, many cases when the replacement would serve no purpose, for example in the query:

```
retrieve (e.number, e.manager,
                mancnt = count(e.name by e.manager))
```

After processing the aggregate, this would be transformed into:

```
retrieve (e.number, e.manager, t.count)
        where e.manager = t.manager
```

Replacing only "e.manager" in the target list would serve no particular purpose since "e.number" would still be referenced and the query would still involve two relations. Note also that a "blind" application of by-list replacement can yield semantically incorrect transformations:

```
retrieve (e.number, t.manager, t.count)
        where t.manager = t.manager
```

This would lose the crucial linking information between "e" and "t".

The general algorithm for by-list replacement must be able to replace all occurrences of a variable or else not do any replacements. The algorithm is:

(1) For each aggregate function nested in the query, do steps (2) through (5).

(2) If the variables remaining in the query, intersects the variables in the by-list of this aggregate function, proceed; else go to (1)

(3) Examine the query looking for expressions which are identical to one of the by-list expressions.

(3a) If an expression is found, then record its address and which part of the by-list it matches. Also record the variables which the by-list replaces.

(3b) If an expression is found which does not match any of the by-list, record the variables it contains if any.

(4) Compare the variables remaining in the query (the list from 3b) with the list of those which could be replaced by the by-list (the list from 3a).

(5) If these lists intersect, then do not modify the query; else actually perform the by-list replacement.

(6) If any replacement was actually done, then scan the

qualification of the query and remove unnecessary clauses (e.g. clauses of the form expression equals itself).

Here is an example of the by-list replacement algorithm:

```
range of e is employee
retrieve (e.manager, e.startdate,
        manmax = max(e.salary by e.manager),
        mancnt = count(e.number by
                        e.manager,e.startdate),
        datmax = max(e.salary by e.startdate))
```

The three aggregate functions would be processed first. The "max by manager" aggregate into "temp1", the "count by manager and startdate" aggregate into "temp2", and the "max by startdate" aggregate into "temp3". The query would then be:

```
range of t1 is temp1
range of t2 is temp2
range of t3 is temp3
retrieve (e.manager, e.startdate,
                t1.max, t2.count, t3.max)
where
        e.manager = t1.manager
and
        e.manager = t2.manager
and
        e.startdate = t2.startdate
and
        e.startdate = t3.startdate
```

Now perform the by-list optimization:

(1) Consider the aggregate max(e.salary by e.manager)

(2) The query contains variables e, t1, t2, and t3. The aggregate by-list contains variable e. Therefore, we proceed to step (3).

(3) Step 3a will find that e.manager occurs three times

in the query and can be replaced by t1.manager. Step 3b will find the e.startdate, t1.manager, t2.manager, t2.startdate, and t3.manager will still be remaining if the replacements found in 3a are done.

(4) The list from 3a uses variable e. The list from 3b uses variables e, t1, t2, and t3.

(5) Since the two list intersect, no replacement is done and we return to step (1).

(1) Consider the aggregate count(e.number by e.manager,e.startdate)

(2) The query contains variables e, t1, t2, and t3. The aggregate by-list contains variable e. Therefore, we proceed to step (3).

(3) Step 3a will find that e.manager occurs three times and can be replaced by t2.manager. It also finds that e.startdate occurs three times and can be replaced by t2.startdate. Step 3b will find t1.manager, t2.manager, t2.startdate, and t3.manager will still be remaining if the replacements found in 3a are done.

(4) The list from 3a uses variable e. The list from 3b contains variables t1, t2, and t3.

(5) Since the two lists do not intersect, the replacements are performed. The query now becomes:

```
retrieve (t2.manager, t2.startdate,
                t1.max, t2.count, t3.max)
where
        t2.startdate > 1975
and
        t2.manager = t1.manager
and
        t2.manager = t2.manager
and
        t2.startdate = t2.startdate
and
```

t2.startdate = t3.startdate

(1) Consider the aggregate max(e.salary by e.startdate).

(2) The query contains variables t1, t2, and t3. The aggregate by-list contains variable e. Since they do not intersect, we return to step (1).

(1) There are no more aggregates to be considered so go to step (6).

(6) Since at least one replacement was done, the qualification is scanned to remove unnecessary equality clauses. The final query is:

```
retrieve (t2.manager, t2.startdate,
                t1.max, t2.count, t3.max)
where
        t2.startdate > 1975
and
        t2.manager = t1.manager
and
        t2.startdate = t3.startdate
```

VII. CONCLUSIONS.

The algorithms presented in this paper provide a very powerful yet relatively simple methods for performing aggregation. In summary here is an overview of the entire aggregate processing procedure:

(1) Generate a list of all aggregates to be processed. The list must be in innermost to outermost order to

guarantee that nested aggregates are done in the correct order.

(2) For each aggregate in the list, perform steps 3 through 6.

(3) Scan the list of remaining aggregate to see which aggregates can be processed at the same time as the current one; according to the rules of section V.

(4) Process the aggregate.

(4a) If it is an aggregate function, create a temporary relation to hold the results (call it temp1).

(4b) If the aggregate function has a qualification, project the by-list into "temp1".

(4c) If the aggregate is multivariable or unique project the qualifing tuples into "temp2".

(4d) If the aggregate is unique, remove duplicates from "temp2".

(4e) Compute the aggregate.

(4f) If it is an aggregate function, link "temp1" to the outer query as necessary according to the algorithm of section III.

(5) Replace the aggregate(s) in the query with its/their scalar value(s) or with a reference to the temporary relation.

(6) If it is an aggregate function, try to replace domains referenced in the by-list according to the algorithm of section VI.

ACKNOWLEDGEMENTS

Michael Ubell was responsible for writing nearly every query which unveiled a bug in the earlier algorithms. Both Dan Ries and Michael Ubell were involved in analysing most of the algorithms presented here.

REFERENCES

[ASTR76] Astrahan, M. M. et. al., "System:R A Relational Approach to Data Base Management", ACM Transactions on Database Systems 1, No. 2, (June 1976).

[CODD70] Codd, E.F.; "A Relational Model of Data for Large Shared Data Banks," CACM vol. 13, no 6, June 1970.

[DATE77] Date, C. J., "An Introduction to Database Systems", Second Edition, Addison-Wesley Publishing, 1977.

[HELD75] Held, G.D.; Stonebraker, M.R.; Wong, E.; "INGRES - A Relational Data Base System", Proc. NCC vol. 44, 1975.

[YOUS78] Youssefi, K.; Wong, E; "Query Processing in a Relational Database Management System", Electronics Research Laboratory, Memorandum UCB/ERL M78/17, March 1, 1978.