

WORKING PAPER

April 1981

A Survey of Query Optimization Techniques

Farhad Arbab

IBM CORPORATION
LOS ANGELES SCIENTIFIC CENTER
9045 Lincoln Boulevard
Los Angeles, California 90045

CONTENTS

1.0	INTRODUCTION	1
1.1	Programming Languages vs. Query Languages	2
1.2	Domains of Optimization	3
2.0	SYNTACTIC APPROACHES	5
3.0	SEMANTIC APPROACHES	7
4.0	ACCESS PATH APPROACHES	10
5.0	MODELING OF PHYSICAL ORGANIZATION	12
6.0	SURVEY OF LITERATURE ON QUERY OPTIMIZATION	15
6.1	Aho, 1978	15
6.2	Chandra, 1977	16
6.3	Clausen, 1980	17
6.4	Hall, 1976	18
6.5	Hanani, 1977	20
6.6	Palermo, 1972	21
6.7	Pecherer, 1976	22
6.8	Rothnie, 1974	23
6.9	Sagiv, 1978	23
6.10	Smith, 1975	24
6.11	Stroet, 1979	24
6.12	Yao, 1979	25

1.0 INTRODUCTION

Language optimization is at least as old a problem as programming languages. However, with the advent of query languages it seems like interest is mounting on this classic subject. This paper presents a survey of the literature on the topic of query optimization, more specifically, query optimization in local data base systems. Discussion of query processing in distributed data base systems is eliminated, because in these systems the nature of the optimization problem is substantially different than in case of a local data base. In distributed systems, query optimization involves system parameters such as extent, availability, and costs of information and resources at each site, network topology, communication costs, and shipping strategies; issues which fall in an entirely different class than the ones we intend to cover in this paper. As a pointer to a recent work on this subject, the reader is referred to [KER80], where the authors study the problem of query optimization under a specific topology (star) for distributed data base systems, first for a central-local two computer network, and then generalize their results to the full star network topology.

In order to be able to put different methods and view points of a rather diverging group of authors and researchers into perspective, it is essential to develop a broad common framework. We will begin by a brief comparison of query and programming languages, in order to highlight the differences which we believe are significant enough to change the nature of a classic problem in the light of a new application. It should be understood that the intention of this comparison is not to run judgement on any particular programming or query language, nor to debate suitability or superiority of any class of languages for the general task of programming and information processing. The fact that languages of such a broad and varied spectrum as covering Fortran and CLU are lumped together into the same class, should not mean that we intend to imply that the attributes of the class apply equally to its members; and the same, of course, holds with respect to the class of query languages.

We will see that the differences between programming and query languages are significant enough to fuel the variety (and the novelty) of the methods one finds in the literature on query optimization and set them apart from classic methods for optimization of programming languages. To put these varied approaches in perspective, we propose a general classification of three orthogonal categories for optimization techniques.

Finally, a brief discussion of the papers on the subject which I think are significant, representative, or comprehensive is included, together with references to other related papers.

1.1 PROGRAMMING LANGUAGES VS. QUERY LANGUAGES

Consider a conventional programming language such as Fortran, Algol and associates, APL, Lisp, Simula, or CLU. Also, consider a typical query language such as relational algebra, relational calculus, QBE, Sequel, or Quel. Notice that we are not including the navigational directives of IMS or DBTG in the category of query languages, because by themselves they are not complete and must be imbedded within another language to become capable of expressing a query. This, of course, should not mean that it is not possible to design a query language based on the hierarchical or the network model of data.

A programming language has a data model which defines its basic types, and a set of operators including those which operate on the data model to produce new types (e.g. the structuring operators), operate on types to produce new objects of a given type, or operate on the objects of given types to produce new objects. An important class of this operator repertoire is that of flow of control operators and functional abstraction features of the language.

A query language too, has a data model and an operator repertoire. Its data model is generally simpler than that of an average programming language (exceptions: Lisp and APL) in that it is more homogeneous, and at the same time, higher level, because generally, such data structures are not immediately available in a programming language. This is an important point to notice, since higher level data structures capture and represent a lot more "information" than lower level ones; more on this issue later. The operator repertoire of query languages is, likewise, both simpler and more homogeneous than that of programming languages. With few fixed types, and no provisions for definition of new data types, there is no need for corresponding operators. Significantly, also missing are the flow of control and functional abstraction features. This leaves a more or less uniform set of operators that manipulate objects of relatively high level of abstraction.

Both query and programming languages are designed to serve the purpose of information processing. When information increases in volume and/or complexity, it becomes unmanageable. Information also has the property that when butchered into smaller pieces randomly, its overall complexity increases drastically. Hence the only right way to go about processing complex and/or large volumes of information is to break it down into manageable pieces by abstraction, i.e. to cut the mass of information into smaller pieces in such a way as to decrease the interaction of each piece with others.

Two important methods of information abstraction are operation abstraction and data abstraction. Operation abstraction is the effort of step-wise refinement of the process which data is to go through, and is achieved by modular system design and structured programming. Data abstraction is the effort of step-wise refinement of the view of data

which is to be processed, and is achieved by utilization of abstract data types (to the limit of the capabilities of the language.) In fact, operation abstraction and data abstraction go hand in hand and very much affect each other throughout the activity of refinement. This interaction can be understood better by noting that with operation abstraction in mind, one sees the world the way one wants to operate on it, whereas, data abstraction puts one in the position to operate on the world the way one sees it. The former leads one in building tools to operate, whereas the latter consists of manipulation and extraction of new views of the world such that they can be operated on with those tools.

Programming languages are generally well equipped to support operation abstraction through their flow of control operators, functional abstraction mechanisms, and/or subprogram and macro capabilities. In one way or another, these features make the operator repertoire of programming languages augmentable. Relative richness and atomicity of data types available in a programming language, in contrast with a query language, combined with extensibility of its data types and operators that manipulate them, provide for some degree of data abstraction.

Query languages, on the other hand, have a rigid and unaugmentable set of operators and do not provide operation abstraction features. Their data model is also non-extensible and this limits data abstraction to what can be represented by the underlying model; nothing more and, even more significantly, nothing less is directly representable.

1.2 DOMAINS OF OPTIMIZATION

In following sections, we will further elaborate on the simple model of a language discussed above, i.e. that a language consists of a data model and an operator repertoire, and will consider the issue of optimization. Our aim is to investigate the implications of the differences between the view points taken by query and programming languages on the way they are utilized to process information, and consequently, on how each should be optimized. We recognize three major categories of optimization techniques:

- Syntactic methods
- Semantic methods
- Access path methods

Traditionally, syntactic methods are the most important and the most commonly used optimization techniques in the realm of programming languages. As for database management systems, many use techniques which are combinations of different methods in order to optimize a query, and quite often, boundaries between these methods are not made very clear.

The above classification, thus, is mostly intended to serve as a crude analytical tool in our study of (query) optimization techniques rather than a categorical classification of actual systems or optimizers.

2.0 SYNTACTIC APPROACHES

A first cut approach to optimization is syntactic manipulation of the source, i.e. the kind of optimization that one can ideally expect from a user of a language to perform, without drastic modification to the algorithm involved. Users of a language are presented with a certain integral view of a machine and may (in fact, should) not be aware of whether the abstract machine they know as the language actually exists, or its behavior is simulated through (possibly a tandem of) compilation and/or interpretation processes. Invisibility of implementation details below the language surface is an important factor which defines one of the boarder-lines of syntactic approaches to optimization.

By syntactic optimization we mean achieving optimal utilization of the (extended) operator repertoire of the language, subject to the constraints that no drastic changes are made to the algorithm or the data types used in a program (or query.) Even though it is certainly within the capabilities of a user of a language to utilize its operators in an extremely optimal manner, it is often impractical to expect any user to do so. Users of a language miss and avoid optimal utilization of its operator repertoire for several well justified reasons, including convenience, clarity, locality, and self documentability; therefore, this task is commonly delegated to a specific processor, usually the language processor itself.

It is very uncommon for a programming or query language to be directly supported on a machine. Generally, programs and queries are "compiled" through a sequence of translation phases into a low level machine language before they can be executed. Strictly speaking, each phase of compilation expresses the original query or program in a different (sometimes somewhat implicit) notation or language, e.g. some representation of a parse tree, an intermediate language such as that of a hypothetical stack machine, or assembly and machine language. Every one of these "languages" is susceptible to its own set of correctness-preserving syntactic transformation rules in order to enhance utilization of its own operator repertoire.

There are basically two reasons that make syntactic optimization of intermediate languages even more important than transformations on the original source. First, the "users" of such intermediate languages are programs, rather than human beings, that lack creativity, to say the least. Second, it is extremely unreasonable to expect the real user to express his program or query in such a way as to compensate or avoid the "inoptimalities" that will be introduced by the mechanical substitution nature of the translation phases, even if the user is aware of the details involved (which very often he is not and should not be) and it is theoretically possible (which, again, quite often it is not.)

As a program (or query) gets translated into a lower level language in each phase of the compilation process (at least conceptually,) its higher level (data and operation) abstractions get expanded into more elaborate detail. This successive translation can obscure some of the suitable cases for application of optimizing transformations to the point where either they can no longer be detected, or it would require a sophisticated pattern matching algorithm and some degree of semantic analysis (that in essence would "uncompile" pieces of the lower level code) to detect them. This fact shows the importance of distributing the task of optimization among the phases of compilation, rather than trying to optimize at the lowest level once, and of course, applies not only to syntactic, but to any type of optimization.

We classify as syntactic optimization methods, those that involve transformations whose applicability can be determined by a superficial understanding of a program or query. Deciding which transformation rule is the most appropriate one at a given stage in a syntactic optimization process, can in fact be very much involved and complicated. Nevertheless, the information upon which the decision will be based are such that can be made available by a simple understanding of the notation or language in which the query or program is expressed, i.e. it merely involves syntactic properties of language constructs. Algebraic transformations, parse tree reordering techniques, and logical inference rules are all examples of syntactic methods of optimization.

Much work has been done on optimization of programming languages and the literature on this subject is rich. Most of these classic methods also, are syntactic and apply equally to query languages as well. It is beyond the scope of this paper to consider these optimization techniques and those who are interested are referred to [ALL71] for a systematic review of many such methods. A more recent paper on this subject, [MOR79], presents a technique based on a purely boolean approach for global elimination of partial redundancies and performs in a single algorithm what is normally done by successive application of several algorithms. The particular redundancies dealt with in [MOR79] are redundant computations and loop invariants. The authors observe that for well structured programs, the cost of their algorithm is very nearly linear with respect to the length of the program, and is very slightly dependent on its graphical structure. Because this approach is basically boolean and does not take into account the shape of the graph on which it is applied, the method of [MOR79] is language independent and works on both implicit and explicit loops.

3.0 SEMANTIC APPROACHES

Tracking our model of a language into a specific program or query, we recognize that correspondingly, it consists of a specific view and a specific sequence of operators. (where sometimes, e.g. in case of non-procedural languages, no temporal notion is associated with the "sequence".) In case of a program, the view is the set of abstract data types and their instances used in that program and in case of a query, for instance one expressed in relational terms, the view would be the set of relations used in the query.

Between the two components, it seems that the view is the principal constituent in the sense that it has a somewhat sounder effect on the algorithm of a program or query; it is possible to make, sometimes rather significant, changes to the sequence of operators in a query or program, without any significant changes to its view, whereas a change in view almost always mandates a change in the sequence of operators. This suggests that somehow, there is a tighter link between the semantics of a program or query and its view, than there is between the former and its sequence of operators.

The above observation is the grounds for the distinction that we make between syntactic and semantic optimization techniques. Whereas syntactic optimization is limited by the constraint of making no significant changes to the view (and therefore the semantics) of a program or query, by semantic optimization we mean reformations that result in the re-expression of the intention of a program or query in such a way as to best utilize the data model of the given language. Notice the distinction that we make between intention and (formal) semantics; as changes are made to a program or query, preserving the latter is often a much stronger constraint (consequently, more limiting, but an easier goal to achieve) than preserving the former.

The principal problem in preserving the intention is to discover it in the first place; intention and semantics are not only different concepts, but also there is often a wide gap separating them. Intention is easier to trace in case of non-procedural specification-like languages where a program or query is bound to represent the ultimate result more closely and more explicitly, than in case of procedural languages where even at the semantic level, each step of a program or query is still blind to its own role and contribution towards the ultimate result. Extensibility and augmentability (of the data model and the operator repertoire) in a language help to make users' intention tractable through the semantics of their program or query, by providing them with the means to construct and use whatever (operation or view) abstractions they feel (is closest to what) they want. Availability of low level abstractions, particularly in the form of atomic types and lower level data structures in the data model

of a language, also makes intention more tractable through semantics, by giving users a choice to use abstractions which directly represent their desired results, instead of including them.

Recalling our discussion on query vs. programming languages regarding the issues of extensibility, augmentability, and data model, should make it clear why we consider semantic optimization as a separate and important category in case of query languages. For most programming languages the distinction between syntactic and semantic optimization becomes blurred and somewhat arbitrary because of the closeness between the semantics and intention of their constructs. Besides, pragmatically, it makes sense to do very little semantic modification to a program and generally what needs to be done can be handled by a set of transformations that are only slightly more involved with the semantics than syntactic transformations are.

To reiterate, it is not so vital to understand the intention of a program in order to do a good overall optimization job on it. One can always trust that the composer of a program is an intelligent being who is given a suitable set of tools and even though he may use them in a less than optimal order and context for convenience, he has a rather good sense of tool selection and generally will not use wildly improper tools; in particular, he usually will not generate more information than what he is going to need. At least, one can argue that the effort is better utilized if it is invested towards optimization of individual "tools" (i.e., abstractions) rather than in questioning and second guessing the wisdom of their use.

The case of most query languages is different. Here the user has an unaugmentable set of operators and an unextensible data model which offers a rather high level data structure as a view. Without operation abstraction features and with limited data abstraction capabilities, the user does not have the suitable tools to succinctly express (or extract) the result that he wants. Instead, he often deals with information that contains the result which he seeks. This, in fact, shows a tradeoff in the design concept of query languages in order to gain generality, homogeneity, convenience, and ease of use, all at once. To do a good optimization job on a query, it is important to trace its user's intention through its semantics in order to find a better way to re-express it. In particular, it is important to detect and prevent generation of unnecessary and excessive information which is the result of application of non-precision all-purpose tools of the language, i.e. "too general" operators and a "too general" data model. In its simplest form, this involves cases like where a set is formed just to take a count of its members or to check if it is vacuous, or where it is not necessary to eliminate duplicate members in a set, etc. More involved semantic optimization would re-express the intention of a query in such a way as to make it susceptible to other optimization transformations, possibly in other "languages", in order to take advantage of its inherent properties, e.g. existence of more efficient data structures, parallelism, and concurrency.

The recent trend of general-purpose very-high-level languages changes the above picture to some extent. There are now language processors that do in fact try to perform semantic transformations on their intermediate representation in order to utilize more efficient internal data structures, among which SETL, a set oriented programming language [SCH75] can serve as an example [SCH79].

Semantic methods of query optimization involve transformations whose applicability can be determined by the properties of the data model as they relate to the query. Dependencies and constraints implied by the axioms of the data model, as well as those implied by the user's and universal views of the data base can be used in conjunction with the semantics of the query to make inferences about the intention of the user. after such analysis, the intention of a query can be rephrased as to conform better with the above constraints.

4.0 ACCESS PATH APPROACHES

The single most important distinguishing factor between query and programming languages is in their basic view of the storage hierarchy. The difference is in fact in the way that the secondary storage is treated by the language.

Programming languages clearly differentiate between the main memory and the secondary storage. The main memory is the center of activity and any meaningful processing can be done only on information that resides there. The secondary storage is merely a repository of raw and inactive information upon which no processing can be performed directly. All programming languages then, have to provide a number of special purpose operators, called the I/O operators, for transfer of information between the two segregated levels of storage. Such transfers are thus very explicit and are regarded as "side effects" of the processing that is taking place on the information which is already in the main memory, rather than being based on the merits of the information on the secondary storage. (The second class treatment of secondary storage in programming languages can also be observed in the concept and structure behind the I/O operators themselves, which traditionally have shared and inherited a number of attributes such as rudimentary, low level, unorderly, ad-hoc, and/or unbecoming of the language.)

The clear separation of the two storage levels in programming languages, together with the explicitness of information transfer between them, make it extremely difficult to attempt to optimize such transfers (consequently, the program itself) in conjunction with the overall intention of a program. As a result, language processors do not generally go any further than applying few syntactic transformations in that regard.

Query languages, on the other hand, see the secondary storage as a logical extension of the main memory and also share the built-in assumption that data, for the most part, resides on the secondary storage. Consequently, it is irrelevant for the user of a query language where a piece of information resides, and there is no need for explicit transfer of information by the so called I/O operators. The unified view of storage presented on the surface by query languages, is in sharp contrast with the view taken by programming languages, and makes it the responsibility of the language processor to both optimize and judiciously perform transfers of information between the two levels.

By access path optimization we mean transformations (usually on the intermediate representations of the query) or utilization of any other technique that will result in more efficient retrieval of information which is directly relevant to the result of the query. Access path optimization involves extensive knowledge about the data base instance in

question and its implementation details. Implementation details such as file and page organizations, availability of keys, indices, and links, and accessibility of each in terms of existence of local copies, should all be the concern of this class of techniques. Transient properties such as domain volumes or distribution of the values of a particular field, and other coincidental properties of the data base instance which are not inherent in its implementation, can also be taken advantage of by access path optimization methods. A great deal of optimization efforts of systems such as System R fall under this category.

An important subclass of access path optimization is optimal implementation of language operators in order to make retrieval more efficient. Examples of such techniques are optimal implementation of relational algebraic operators, most significantly Joins as in [GOT75], and the dynamic feedback method of Rothnie in [ROT74].

Access path optimization also involves issues common to information retrieval. Some of the techniques in information retrieval, e.g. the one in [HAN77], can be readily used to solve a problem common with query processing, whereas some others, e.g. the one in [AHO79] or in [YU78], are not so directly applicable.

Modeling the organization of the data structures which together comprise a so called physical database, independently being a subject of considerable interest by itself, is also of significance from the point of view of access path optimization. We will consider this issue in the next section.

5.0 MODELING OF PHYSICAL ORGANIZATION

Efforts towards modeling of physical organization of data bases date back to long before evolution of today's powerful, high-level, implementation independent query languages, which but intensify the significance of optimization. Generally, these efforts are aimed at capturing the characteristics of a (class of) data organization schemes into a model in which the cost of storage and/or the cost of access under the access method(s) supported by the organization can be formulated. Such a model, then, can be used to determine how well a particular data base organization fits in with the objectives of a specific system, or how does a data base organization rank among others covered by the same model in satisfying a given set of requirements; a valuable insight when one is to implement or reorganize a data base.

A cursory look at the literature on analysis and modeling of data base organization shows a variety of papers emphasizing different aspects of the subject. Some models are based on very "qualitative parameters" and are meant to provide practical guidelines for selecting a data base organization by imposing a type of ordering on the commonly used organizations against a set of more or less subjective and case dependent parameter values, e.g. [SEV77], where the authors present a chart in which dominant data base retrieval activity is plotted in the three dimensional space of quantity of records retrieved, volume of online update, and speed of response. Others present detailed analysis of a particular data base organization [CAR75 and KEE74]. Few others propose simple yet relatively general analytical models with a number of "quantitative parameters" providing degrees of freedom which make it possible for the same model to meaningfully cover a range of common data base organizations, e.g. [HSI70, SEV75, and YAO77].

Even though the choice of a physical data base organization and its corresponding access methods are important topics in long term planning for overall optimal performance of a database system, we are not here interested in this issue. Our interest is in models that are capable of expressing intrinsic attributes of a rather wide range of existing physical data base organizations in a manner which underlines the tradeoffs and compromises and contrasts explicitly the implicit features and preassumptions involved in the selection of a particular organization. For our purpose, using such a model is more preferable over utilizing a number of models each of which covers a specific data base organization or a few special cases in more elaborate detail, because a unified model can provide a framework in which generalized access algorithms and cost equations can be developed and used as the basis for generalized access path optimization techniques.

In [YAO77] such a generalized model for physical data base organization is presented. Yao's model can be better understood when put in perspective with that of Hsiao and Harary [HSI70] and Severance [SEV75].

Hsiao and Harary present a formal model in which a data base is divided into a file of records and a directory. Each directory entry corresponds to a keyword and contains the value of the keyword being indexed, the total number of records which contain that keyword value, and the number and head pointers to the sublists of records which contain that keyword value. As one varies the number of sublists of records, the model will represent organizations ranging between a multilist (one single sublist per directory entry) and an inverted file (as many sublists per directory entry as there are records with a given keyword value.) This model, thus, is basically a one dimensional model, i.e. the only parameter in the model is the number of sublists per directory entry.

Severance observes that the model of Hsiao and Harary cannot describe organizations with other than strictly list oriented record sequencing and proposes a two dimensional model in [SEV75]. In this model the structure of each record may be data direct or data indirect, meaning the record contains the actual data or the record contains a pointer to the actual data, respectively. The inter-record connection can similarly be address sequential or pointer sequential, meaning that the successor record is physically contiguous or the record contains a pointer to its successor, respectively. The proportion of data indirect records and the proportion of pointer sequential records in a data base are the two parameters of Severance's model which covers organizations like sequential, list, (a kin of) indexed sequential, and various inverted list organizations.

Yao notices that Severance's model can only represent a maximum of one level of indexing using its notion of data indirection. He also mentions other shortcomings of the model, including the fact that it cannot represent various cellular list organizations and the lack of provisions to express the concept of partitioning a list into sublists [YAO77].

Like Hsiao and Harary's, Yao's model also divides a data base into a directory and a file; however, each is broken into further levels. The directory consist of at least two levels of attributes and keywords, with zero or more index levels in between, depending on the particular organization being modeled. The function of the directory is to decode a given keyword (an "attribute-name,value pair"); the attribute-name is decoded (identified) at the attributes level and the value, at the keywords level. Index levels help this decoding by providing means of faster identification of the lower level items.

The set of records that contain a keyword is called a k-set. The file contains the k-sets corresponding to the keywords in the directory; this is the record level. Because it is sometimes desirable to partition a k-set into several subsets, each contained in a storage bucket, another level, called the accession level, is introduced above the record level,

when required. Corresponding to each k-set, there is an accession list pointing to the storage buckets which comprise the k-set. A tree representation of the access paths in the data base makes an access tree. The access tree has its root at level 0, all attribute names appear at level 1, followed by index levels, if any, the keywords level, the accession level, and finally, the records level as leaves of the tree at level n. Connectivity between the nodes of the access tree may be address sequential or pointer sequential, as in Severance's model. The set of nodes with the same immediate parent is called the filial set. The access tree levels, the average number of nodes per filial set in each level, and the proportion of pointer sequential connections at each level are the parameters of the model which together describe the static structure of a data base organization. The ratio of the free space distributed in the sequential blocks (buckets) of the filial sets at each level in order to delay an overflow, and a threshold ratio for each level after which an overflow would be handled by splitting of the filial set, as opposed to chaining, are the parameters that define the dynamic structure of the data base for updates and insertions.

Examples of how several of the most commonly used data base organizations can be represented are included in [YAO77] to show the generality of this model; they include inverted, multilist, cellular, and indexed sequential organizations. Generalized access algorithms for search and retrieval in the directory and the file are presented and generalized equations for the costs of storage and access time are developed based on the model. It is shown that Cardenas's cost equations for the inverted data base organization [CAR75] can be derived from this generalized equations as a special case.

6.0 SURVEY OF LITERATURE ON QUERY OPTIMIZATION

6.1 AHO, 1978

A subset of relational algebra is considered as a query language in [AHO78], which presents a formal study of the issues related to the optimization of queries expressed in this language. This subset is restricted to the three relational operators select, project, and natural join, and expressions in this language are called SPJ-expressions. Even though select, project, and join are not "relationally complete," SPJ expressions are powerful enough to cover most of the common queries. In passing, it should be mentioned that relational completeness as an indicator of the power of a language is itself the subject of some debate and Codd's definition of relational completeness, which is based on his version of relational calculus [COD72], has been questioned by many, among them [BAN78] and [CHA79]. In [CHA79], a class of queries called computable queries, is identified. Focusing on computable queries, the concept of boundedness, expressiveness, and completeness are defined and these definitions are compared to those of [COD72] and BAN78].

At the heart of the problem of optimization, is the issue of equivalence of two queries. In [AHO78], the authors define two notions of equivalence, weak equivalence, which is motivated by and relies on the universal relation assumption, and strong equivalence, which implies weak equivalence but is a stronger condition than the latter.

Next, the concept of a tableau, a two dimensional representation of SPJ-expressions, is defined. Tableaux are very much like a subset of QBE and are a subclass of the "conjunctive queries" of [CHAN77]. Since tableaux are another representation of SPJ-expressions, they are not relationally complete. In fact, even conjunctive queries of [CHAN77] are not relationally complete [SAG78]. It is shown in [AHO78] that the equivalence and optimization problem of SPJ-expressions can be reduced to analogous problems for tableaux. The advantage of the tableaux approach, however, is that it makes it possible to deal with functional dependencies in a mechanical manner. [AHO78] gives algorithms for converting an SPJ-expression to a tableau, based on either definition of equivalence. Even though the authors state that they do not know of an efficient algorithm to construct an SPJ-expression, given a tableau in general, they cite a reference where an efficient algorithm is given that works on a special subset of tableaux, called simple tableaux. [AHO78] gives a formal definition for simple tableaux and observes that "natural" queries expressed as SPJ-expressions often have simple tableaux.

The key idea in the equivalence test of two tableaux is the "containment mapping", which is a mapping from the rows of one tableau to another. Containment mapping is also at the heart of tableaux minimization, a process for elimination of the rows of a tableau whose closure is covered by other rows. Since the number of rows in a tableau is related to the number of joins in its corresponding SPJ-expression, minimization corresponds to elimination of joins, where possible.

In general, minimization and test for equivalence are NP-complete problems, even in case of tableaux. However, for simple tableaux, [AHO78] shows that there exist solutions for these problems that run in polynomial time, no longer than the fourth power of the size of the tableaux involved.

6.2 CHANDRA, 1977

First order queries are formally defined as what roughly corresponds to first order logic. Conjunctive queries are then defined as a subset of first order queries where only conjunction (AND) and existential quantifiers are permitted. Conjunctive queries include a large number of "natural" queries and many more general first order queries are indeed in part conjunctive, e.g. a first order query expressed in disjunctive normal form can be expressed as the union of a number of conjunctive queries. Conjunctive queries are at the heart of QBE, which then builds on this core to support more general queries, and they are also a superclass of the tableaux of [AHO78]. Nevertheless, conjunctive queries are not relationally complete, unless two additional operators, union and difference, are also incorporated [SAG78].

In [CHA77], the authors show that every conjunctive query has a unique (up to isomorphism, i.e. renaming) minimal equivalent query which can be obtained from the original query by "combining variables." This result is similar to the existence of minimal Finite State Automata, which are obtained by "combining states." Combining variables, like combining states in FSA, has the Church-Rosser property, which means that regardless of the order in which variables are combined, one will eventually get to the same unique minimal result. However, it is proved in [CHA77] that unlike the case for Deterministic Finite State Automata, minimization of conjunctive queries (as well as the problem of their equivalence) is an NP-complete problem.

6.3 CLAUSEN, 1980

In [CLA80] a relational data base system, called TGR, is described which uses microprogrammed data base primitives for searching. Data base queries are presented to the system in the form of relational calculus expressions. The optimization algorithm works on relational calculus expressions in conjunctive normal form. The major optimization concept in [CLA80] is the idea of dynamic feedback optimization which can be classified as a semantic approach to optimization. The method is adapted from [ROT74] and can best be explained through an example. The following example is taken from [CLA80] directly. Consider the two relations R1 and R2 shown below:

Relation R1	A1	A2
	3	6
	4	5
	10	2
	15	0

Relation R2	A1	A2
	2	8
	5	7
	9	6
	8	3

Now, consider a query equivalent to:

```
get R1.A1:for-all R2(R1.A1>R2.A1&R1.A2<R2.A2)
```

The above query is executed as follows:

For the tuple $\langle 3,6 \rangle$ from R1, relation R2 is scanned to see if the qualification of the query is satisfied, i.e. for-all $R2(3 > R2.A1 \& 6 < R2.A2)$. The qualification is not satisfied because of the tuple $\langle 5,7 \rangle$ of R2. From this we gain the information that if condition $F1 = (t1.A1 \leq 5 \mid t1.A2 \geq 7)$ is true for a tuple t1 in R1, then the query's qualification is bound to fail for t1. This eliminates the need to scan tuples of R2 for such tuples of R1, entirely. F1 is called an elimination filter.

Similarly, the fact that the tuple $\langle 10,2 \rangle$ of R1 satisfies the query's qualification, leads to another filter, $F2 = (t1.A1 \geq 10 \& t1.A2 \leq 2)$, called a true filter. A true filter such as F2, when satisfied for a tuple t1 in R1, eliminates the need to scan through R2 and designates t1 as satisfying the qualification of the query, automatically.

A third possibility for optimization which is only mentioned in [CLA80], but is implemented in TGR is prevention of duplicate tuple formation. This method too is based on that of Rothnie's as implemented in DAMAS [ROT74]. Suppose that while scanning the tuples of relation R we encountered a tuple such as <1,2,3>. Let us assume that the true and elimination filters are currently such that this tuple will pass to be incorporated in the result, and let us further assume that the result is (or uses) the last two fields of each tuple of R only, e.g. <2,3> in case of the tuple in question. This means that any other tuple of R whose last two fields are <2,3> is superfluous and its incorporation in the result will merely yield duplicate tuples. Thus a third filter, say F3, is constructed for relation R which eliminates all such tuples, preventing their unnecessary participation in further operations and saving the corresponding scans of other relations involved.

The above examples were in fact simple. In practice, there are more complex queries and queries which deal with multi-variable expressions (as opposed to the above example where only two variables, i.e., two relation scans, were involved.) Because this method does not always guarantee an optimal, or even more efficient execution, guidelines should be developed as to where and when to use each of the filters. In TGR (and DAMAS), they use three options to enable/disable each filter mechanism. However, the problem of when to set/reset an option is not satisfactorily solved.

In terms of relational algebra, this approach is similar to combining a selection with other relational algebraic operations, in particular, with joins, and being able to modify the selection filter dynamically. However, the actual building of the filters, or in fact the filter templates as in [CLA80], requires a global understanding of the whole query. This kind of information is less explicit when a query is expressed in terms of a procedural language such as relational algebra, as opposed to when it is written in a descriptive language such as relational calculus.

6.4 HALL, 1976

In [HAL76] a single query defined in relational algebra is considered. The paper advocates that the overhead of optimization should be removed in case of experienced users who express themselves concisely. However, it is not at all clear how this objective is met or even addressed in the proposed method. The method presented in this paper is purely syntactic. The author considers two types of transformations, namely, reordering of operators, and recognition of common subexpressions. The author's criteria for transformations considered in his paper is that "because it is extremely difficult to estimate cost, it is best to avoid transformations that depend critically on cost." This criteria can be better understood considering the limit to which cost can be defined on a

purely syntactic basis. The four types of transformations presented in this paper are:

- Combining sequences of selections into a single selection
This is achieved by combining filters of each of the selections in the sequence into a conjunction. Combining filters has two advantages. First, it can be cheaper to evaluate the conjunction than to evaluate the filters one by one; and second, the boolean proposition obtained after conjunction can be simplified.
- Combining sequences of projections
A sequence of projections can be reduced to a single projection. In some systems, such as the author's PRTV system, where each projection implies a sort, this transformation is very important.
- Idempotency and null relations
Any relational operation with null relation as one of its operands is redundant and can be removed. Further, idempotency laws of relational algebra permit transformations like replacement of the union of a relation with itself by the same relation. Similarly, null expression and idempotency laws of boolean algebra can be utilized as transformation rules.

In general, idempotency rules, null relation and expression removal, and common subexpression removal should be done concurrently. A general common subexpression identification algorithm is presented in [HA74b]. This algorithm transforms the parse tree into a lattice. In short, it starts at the bottom of the tree and climbing one level at a time, removes all common subexpressions, while applying idempotency laws of relational and boolean algebra, and removing expressions involving null relations. The method is theoretically sound and is proved to be correct. The same algorithm is used in [HAL76] and it is also the basis for similar algorithms used by other authors.

- Moving Selections towards leaves of the tree
For each relational algebra operation there is a distributive law that can be used to distribute selection over that operation. The paper gives these distributive laws for all operations and presents an algorithm for pushing selections down a tree.

In case of join the factor & residue method of [HA74a] is used. The same technique is used by several others.

Distributing a selection filter over a union does not necessarily improve performance. If the two relations are disjoint, then we cannot lose by distribution of selection. If the two relations overlap, then applying the selection filter twice for the common tuples may well offset the saving by reduction in cardinality. If there is no significant reduction in cardinality, then we may lose by distributing the selection. This is a case where mere syntactic

information is not sufficient for optimization. Disjointness of the two relations can be determined through the data base definition. Determination of cardinalities may involve deeper implementation and instance dependent information.

In case of distributing a selection over an intersection or a difference, the the above mentioned worsening due to insignificant changes in cardinality will be accentuated.

For joins, particularly as they approach cartesian product, distribution of selection is always an improvement.

6.5 HANANI, 1977

The method of [HAN77] aims at optimizing evaluation of a boolean expression represented as an AND/OR tree, and even though it is assumed that the expression is to be evaluated for the records of a file, it can in fact be extended to work in case of several files as well. It is very desirable to have this sort of intelligence either in the data management system underlying the data base, or at a level as close to the data management level as possible, so that higher levels will not be bothered with the implementation and instance dependent details involved.

In Hanani's approach, two numbers are associated with every terminal node I of the tree, a $t(I)$, the time it takes to check whether a given record has the attribute I , and a $p(I)$, the probability that a given record has the attribute I . The paper suggests using the frequency of the records having attribute I as $p(I)$, and the count of machine instructions required to decide whether a record has the attribute I as $t(I)$.

Two theorems are then presented which make it possible to associate two similar numbers, t and p , with every non-terminal (AND or OR) node in the tree. For a non-terminal node, t and p are the minimum time required to check whether the subexpression represented by the node evaluates to true for a given record, and the probability that it evaluates to true, respectively.

Finally, an evaluation algorithm is presented which based on an ordering imposed by the t and p values, evaluates the expression in optimal time.

In [LAI79] an extension of Hanani's method is presented which makes it possible to handle "records" whose fields are not all available at the same time, i.e. some cost may be involved in retrieval of certain secondary fields. This clearly also handles the case where a query is dealing with several records of different files at the same time. Laird's method is essentially the same as that of Hanani, except that three valued

logic is used instead of the standard boolean logic. The third value of undefined is assigned as the value of any relational expression involving unavailable attributes. The theorems and the algorithm are then extended to handle this three valued logic.

In [GUD79] a basic implicit assumption of Hanani's algorithm is discussed which in practice will lead to non-optimal solutions. The main assumption under which Hanani's approach will in fact yield optimal results is that each keyword (attribute) can appear in the query only once. This means that the query $I1 \& I2 | I1 \& I3$ is not a valid query in Hanani's method. Suggestions are made which even though do not remove this restriction entirely, extend Hanani's algorithm to include most of the practically interesting cases. The proposed extension breaks the one-to-one correspondance between attributes and keywords and while it still expects each keyword in the query to be unique, it permits different keywords in a query to have the same attribute. Consequently, a query like $DEGREE=BSC \& (AGE < 30) | DEGREE=MSC \& (AGE < 40)$ will be an acceptable query under the extension of [GUD79], whereas in Hanani's original method, it is not.

6.6 PALERMO, 1972

In [COD72] an algorithm is presented for the reduction of relational calculus expressions to relational algebra formulae. This reduction algorithm is in no sense meant to be an efficient or a practical method, and merely serves to prove the possibility of such conversion and existence of such an algorithm. Palermo, in [PAL72], takes Codd's algorithm as a basis and through a number of improvements, yields a more efficient reduction algorithm. The improvements are simple and intuitively clear in nature but are introduced and discussed in a mathematical framework. The end result is an algorithm for reduction of relational calculus expressions to relational algebra, which unlike that of Codd's, is practically viable.

One major improvement discussed in [PAL72] is substitution of the construction of the cartesian product of relations defined for each variable of the query, by formation of joins and unions. This leads to a gradual growth of the result, as opposed to gradual trimming of Codd's cartesian product.

Semi-join, an intermediate object introduced in [PAL72], is a formalization of the concept of constructing a secondary index on the join field of a relation. Semi-joins can be utilized to prevent multiple retrieval of tuples and make it possible to construct an algorithm in which the order of exploring relations can be determined dynamically.

6.7 PECHERER, 1976

Searching the product space of large sets is a common problem in all languages supporting a set like view of data, and often requires the movement of enormous quantities of data. Of course, explicit search of the product space is not always required; directories and secondary indexes can be utilized to reduce references to and the volume of the product space. However, these techniques do not eliminate the problem.

[PECH76] first considers the product of n relations (sets of tuples) and presents a method for finding an optimal order to obtain the product through nested iterations. Since the result (output) is independent of the order in which the product is formed (the order of the nested loops), the objective is to find an ordering that reduces the amount of input. Pecherer proves that the ordering is optimal if and only if the ratio $V_i/(N_i-1)$ is in non-increasing order for $i=1$ to n representing the order of relations, where N_i is the number of tuples in relation i and V_i is the total volume of relation i in bits.

Next, [PEC76] considers restricted products (products followed by a selection) and observes that the input data volume is minimized if the input tuples are filtered as soon as possible. When for all i , the probability of tuples of relation R_i appearing in the final restricted product (i.e. the probability of the tuples of relation R_i satisfying the filtering predicate) are known and are independent of each other, [PEC76] gives a more general version of the $V_i/(N_i-1)$ ratio whose non-increasing order yields a guaranteed optimal ordering for the relations.

Last, [PEC76] discusses a slightly improved iteration technique and, once more, modifies the $V_i/(N_i-1)$ ratio such that its non-increasing order would reflect the optimal order in which the product can be formed.

The analysis of [PEC76] is based on the assumption of a single processor, limited memory, and single channel to access the relations.

In his thesis [PEC75], Pecherer deals with efficient retrieval in relational databases and gives efficient algorithms for four relational operators, restrict (select), product, project, and division. He also shows that these operators are sufficient for expressing almost all relational queries.

A related work is that of Kambayashi who in [KAM79] discusses utilization of functional and multivalued dependencies in order to find an efficient ordering to perform joins. The method of [KAM79] yields the optimal ordering for joins in a special case.

6.8 ROTHNIE, 1974

In [ROT74] a relational data base system called DAMAS, developed at MIT, is discussed. The paper presents a promising approach to query optimization but only in an implicit form and wrapped in the architectural details and implementational idiosyncrasies of DAMAS. The paper is not intended to be an "optimization paper" and talks about DAMAS in general. The novel dynamic feedback optimization method of Rothnie is adapted by Clausen [CLA80] in another relational data base system called TGR, and is discussed elsewhere in this paper.

6.9 SAGIV, 1978

A generalization of tableaux of [AHO78], called sets of tableaux, is proposed. It is noted that neither tableaux, which correspond to expressions involving the three operations select, project, and join, nor conjunctive queries of [CHA77] which are a superclass of tableaux, are relationally complete, unless two additional operations, union and difference, are also permitted.

The generalization of tableaux in [SAG78] incorporates into sets of tableaux the union operator and shows that every relational expression involving the operators select, project, join, and union, can be represented by a set of tableaux. The theory of tableaux can easily be extended to sets of tableaux.

Sets of tableaux are further generalized to sets of elementary differences in order to also include the difference operator to a limited extent; thus every relational expression involving the five operators select, project, join, union, and difference, in which the project operator is not applied to subexpressions that include the difference operator, can be represented as a set of elementary differences.

Next, the problem of containment for single tableaux is considered [AHO78], and it is shown to be NP-complete. Because containment is at the heart of equivalence test and minimization, it follows that these problems are also NP-complete for single tableaux, sets of tableaux, and sets of elementary differences. Yet, polynomial-time algorithms for three special cases which correspond to some class of practically useful queries are presented.

Three transformations that can be applied to sets of elementary differences to obtain equivalent forms are introduced in [SAG78]. These transformations have the finite Church-Rosser property and employ the

containment test for single tableaux. Finally, a necessary and sufficient condition for the equivalence of two sets of elementary differences that are irreducible under these transformations is proved.

6.10 SMITH, 1975

In [SMI75] the architecture of a smart relational database interface is presented. A relational algebra interface, called SQUIRAL, which follows this approach is discussed. The type of optimizations performed is mostly syntactic tree-manipulation like in Hall's, with some semantic and simple access path methods such as coordinating sort orders, using indexes, and maintaining locality in page referencing, also being exploited.

The paper presents a good systematic description of query optimization process regarded as a special purpose programmer who aids the user in efficient query implementation. The significant concept in this automatic programming approach is the exploitation of concurrency. Each operator is viewed as an independent task with a well defined interface. The concurrency inherent in the query can be taken advantage of if each task is allowed to begin execution as soon as it has a sufficient number of tuples on which to operate. Some tasks, such as selection, will be able to work on single tuples; these operator tasks can be pipelined. Others would require entire relations before they can operate.

Pipelining and paralleling of operations are important optimization issues even when multiple (real or virtual) processors are not available. In [BUR76] these issues are addressed in the context of set oriented programming languages. In [ARB80] the same issues are discussed utilizing the approach of [BUR76], with the target of discussion being a very high level database manipulation language.

6.11 STROET, 1979

In [STR79], Stroet and Engmann present a formal syntax for expressions based on relational algebra. As a query (a relational algebra expression) is analyzed, a tree is constructed and the optimization phase consists of applying tree transformations which will improve the evaluation of the query. Transformations are very similar to those used by Hall, except that here, they explicitly deal with trees.

The formal syntax for relational algebraic expressions is included in the appendix and the paper gives a good formal coverage of tree manipulation techniques for syntactic optimization of queries.

6.12 YAO, 1979

In [YA079] the problem of access path optimization is considered. A model is presented for systematic synthesis (or analysis) of a large collection of "two-variable" query access strategies which otherwise would have to be analyzed separately and individually. It, thus, serves as a general framework within which different access strategies can be represented, analyzed, and compared, using a unified approach.

Because Yao's model deals with simple queries which only refer to one or two relations, the approach requires an external heuristic procedure for decomposition of complex queries into simple ones (i.e. semantic optimization). Once this is done, for each simple query it is possible to find the optimal access path using the model. Basically, it is also possible for the heuristic decomposer to use the model and its cost factors to reach a better decomposition resulting in a "more optimal" overall performance for the complex query.

The model recognizes seven basic classes of single relation access strategies based on the order in which relational algebraic operations Restrict (select), Project, and Join are applied, whether or not inter-relation links (one-to-many relationships between fields of two relations) are used to access tuples, and, finally, when records (tuples) are actually retrieved. Some classes have variations reflecting minor reorderings of operator sequences that do not really change the class properties. For example, RAJ represents class 1 of single relation access strategies and includes all such strategies which begin with a restriction (selection), R, followed by access (actual retrieval of a record), A, and, finally, a join, J. Depending on where a projection operator, P, is inserted, RAPJ and RAJP represent two other variations of the same class. Another example is class 7 which includes the two variants LRA and RLA, where L represents the use of an inter-relation link traversal method.

Two-relation access strategies are classified by identifying which basic single-relation access strategy is used for each of the two relations involved, e.g. RAJ/RAJ. The possible combinations of the seven basic single query access strategies lead to 31 basic types of two-relation access algorithms, or 339 if one counts all individual variations of each class.

In a given data base implementation it may quite well be that not all seven basic access strategies can be supported by the storage structure, and this may eliminate many of the theoretically feasible two-relation access strategies. A summary of storage structure requirements to support each of the seven basic access classes is presented in [YA079], and based on the assumption that "cost" is measured by the number of page accesses, cost equations for each access operator is quoted from [YA78b] which

contains their detailed derivations. Costs for single and two-relation access strategies are then computed using these access operator costs.

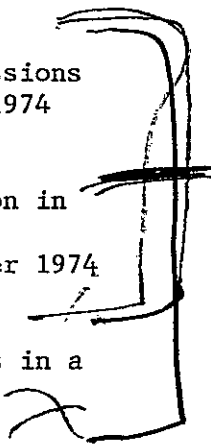
An obvious application of this cost model is to use it when several access algorithms are available, in order to choose the most appropriate one for a query. This approach will yield the optimum result only if the optimal access algorithm for the specific query under consideration is available on that particular data base implementation. Given the rather large number of possibilities, very often a compromise is made and only few different access algorithms are implemented, e.g. in System R. This can severely limit the number of choices, thus, defeating the purpose of a detailed cost model with fine resolution when the end result is selection of one of a few coarse and bulky options. For example, even though Yao's model can be used to evaluate the performance of the System R's optimizer by checking how close it gets to the optimum given by the model, one may not automatically conclude that incorporation of this model in System R for the purpose of access algorithm selection will be an improvement, unless many more access strategies are also implemented.

Another application of the model, as briefly discussed in [YA079], is to use it in conjunction with a synthesizer program which forms an access algorithm for each simple query out of the well defined access operators, based on the type of the query and using operator costs as the measure for optimality. This method will result in the best access algorithm for every simple query, to the limit of the capabilities of a given storage structure. Notice, however, that this dynamic access algorithm synthesizer will not replace the heuristics for decomposition of complex queries.

REFERENCES

- AHO79 Aho A. V. and Ullman J. D.
Optimal Partial-Match Retrieval When Fields
Are Independently Specified
ACM TODS, Vol. 2 No. 4, June 1979
- AHO78 Aho A. V., Sagiv Y., and Ullman J. D.
Efficient Optimization of a Class of Relational Expressions
SIGMOD 1978, also ACM TODS, Vol. 4 No. 4, December 1979
- ALL71 Allen F. E. and Cocke J.
A Catalogue of Optimizing Techniques,
In "Design and Optimization of Compilers"
R. Rustine, Editor
Prentice Hall, Englewood Cliffs, NJ, 1971, pp 31-50
- AND77 Anderson H. D. and Berra P. B.
Minimum Cost Selection of Secondary Indexes
for Formatted Files
ACM TODS Vol. 2 No. 1, March 1977
- ARB80 Arbab F.
Notes on The Semantics and Optimization of a VHLL
IBM LASC Report, G320-2706, October 1980
- BAN78 Bancilhon F.
On the Completeness of Query Languages
for Relational Databases
Proc. 7th Symp. on Math. Found. of Comp. Sci.
Springer-Verlag Lect. Notes in Comp. Sci., September 1978
- BAN80 Banerjee J., Hsiao D. K., and Ng F. K.
Database Transformation, Query Translation, and
Performance Analysis of a New Database Computer
in Supporting Hierarchical Database Management
IEEE Trans. on Soft. Eng., Vol. 6 No. 1, January 1980
- BEC80 Beck L. L.
A Generalized Implementation Method for
Relational Data Sublanguages
IEEE Trans. on Soft. Eng., Vol. 6 No. 2, March 1980
- BLA77 Blasgen M. W. and Eswaran K. P.
Storage and Access in Relational Data Bases
IBM System Journal 1977 No. 4, pp 363-377
- BUR76 Burge, W. H.
An Optimizing Technique for High Level Programming Languages
IBM T. J. Watson Research Center, RC 5834, February 1976
- CAR73 Cardenas A. F.
Evaluation and Selection of File Organization -

- A Model and System
CACM, Vol. 16 No. 9, September 1973, pp 540-548
- CAR75 Cardenas A. F.
Analysis and Performance of Inverted
Data Base Structures
CACM, Vol. 18 No. 5, May 1975, pp 253-263
- CHA77 Chandra A. K. and Merlin P. M.
Optimal Implementation of Conjunctive Queries in
Relational Data Banks
Proc. 9th ACM Symp. on Theory of Computing, May 1977, pp 77-90
- CHA78 Chang C. L.
An Optimization Problem in Relational Databases
IBM San Jose Research Report No. RJ2287, July 1978
- CHA79 Chandra A. K. and Harel D.
Computable Queries for Relational Data Bases
Proc. ACM Symp. on Th. of Comp., April-May 1979
- CHO78 Choy D. M. and Schkolnick M.
Implementation of Unclustered Links in
Relational Data Base Systems
IBM San Jose Research Report No. RJ2199, March 1978
- CLA80 Clausen S. E.
Optimizing the Evaluation of Calculus
Expressions in a Relational Database System
Info. Systems, Vol. 5 pp 41-54, 1980
- COD72 Codd E. F.
Relational Completeness of Database Sublanguages
In "Database Systems", Rustin R. Editor
Prentice Hall, 1972
- DEW79 DeWitt D. J.
Query Execution in DIRECT
Proc. of SIGMOD 1979 Int. Conf. on Manag. of Data
- GIL75 Gilles Farley J. H. and Schuster S. A.
Query Execution and Index Selection for
Relational Data Bases
Tech. Rep. CSRG-53, Univ. of Toronto, March 1975
- GOT75 Gotlieb L. R.
Computing Joins of Relations
Proc. of SIGMOD 1975
- GRI79 Griffiths Selinger P., Astrahan M. M.,
Chamberlin D. D., Lorie R. A., and Price T. G.
Access Path Selection in a Relational Database
Management System
Proc. of ACM-SIGMOD 1979 Int. Conf. on Manag. of Data
(also, IBM San Jose Res. Rep. No. RJ2429, January 1979)

- GUD79 Gudes E. and Hoffman A.
A note on "An Optimal Evaluation of Boolean
Expressions in an Online Query System"
CACM Vol. 22 No. 10, October 1979
- HA74a Hall P. A. V. and Todd S. J. P.
Factorisations of Algebraic Expressions
IBM UKSC Report, UKSC0055, April 1974
- HA74b Hall P. A. V.
Common Subexpression Identification in
General Algebraic Systems
IBM UKSC Report, UKSC0060, November 1974
- HAL76 Hall P. A. V.
Optimization of Single Expressions in a
Relational Data Base System
IBM J. Res. & Dev., May 1976
- HAN77 Hanani M. Z.
An Optimal Evaluation of Boolean
Expressions in an Online Query System
CACM Vol. 20 No. 5, May 1977
(see also [LAI79] and [GUD79])
- HSI70 Hsiao D. and Harary F.
A Formal System for Information Retrieval from Files
CACM, Vol. 13 No. 2, February 1970, pp 67-73
(Corrigendum, CACM, Vol. 13 No. 4, April 1970, pp 266)
- KAM79 Kambayashi Y.
Efficient Procedures for Query
Processing in Relational Databases
Dept. of Info. Science Rep. No. ER79-02
Kyoto University, Japan, January 1979
- KAT79 Katz R. H. and Wong E.
An Access Path Model for
Physical Database Design
Memo No. UCB/ERL M80/1, UC Berkeley, December 1979
- KEE74 Keehn D. G. and Lacy J. O.
VSAM Data Set Design Parameters
IBM System J., Vol. 13 No. 3, 1974, pp 186-212
- KER80 Kerschberg L., Ting P. D., and Yao S. B.
Query Optimization in Star Computer Networks
Bell Labs. Holmdel Database Res. Rep. #2, March 1980
- LAI79 Laird P. D.
Comments on "An Optimal Evaluation of Boolean
Expressions in an Online Query System"
CACM Vol. 22 No. 10, October 1979
- 

- MCS78 McSkimin J. R.
READS - A Relational Data Access System for
Real-Time Applications
Proc. IEEE Compsac 78, pp 295-300
- MOH78 Mohan C.
An Overview of Recent Data Base Research
Tech. Rep. SDBEG-5, April 1978
Dept. of CS, Univ. of Texas at Austin
(also, ACM-SIGBDP's "DATABASE", Fall 1978)
- MOR79 Morel E. and Renvoise C.
Global Optimization by Suppression of
Partial Redundancies
CACM Vol. 22 No. 2, February 1979
- PAL72 Palermo F. P.
A Data Base Search Problem
IBM San Jose Research Report RJ1072, July 1972
- PEC75 Pecherer R. M.
Efficient Evaluation of Expressions in a
Relational Algebra
Proc. ACM Pacific Conf., April 1975, pp 44-49
- PEC76 Pecherer R. M.
Efficient Exploration of Product Spaces
Proc. ACM-SIGMOD 1976, pp 169-177
- ROT74 Rothnie J. B.
An Approach to Implementing a Relational
Data Management System
Proc. ACM-SIGMOD 1974, pp 277-294
- SAG78 Sagiv Y. and Yannakakis M.
Equivalence Among Relational Expressions with
the Union and Difference Operations
JACM Vol. 27 No. 4, October 1980
(also: TR-241, Dept. of EECS, Princeton University, 1978)
- SCH75 Schwartz J. T.
On Programming
An Interim Report on the SETL Project
Courant Institute of Mathematical Sciences, 1975
- SCH78 Schmidt J. W.
On the Implementation of Relations:
A Key to Efficiency
Tech. Rep. CSRG-89, Univ. of Toronto, January 1978
- SCH79 Schonberg E., Schwartz J. T., and Sharir M.
Automatic Data Structure Selection in SETL
6th ACM Symp. on Prin. of Prog. Langs., January 1979

- SET74 Sethi R.
Testing for the Church-Rosser Property
JACM Vol. 21 No. 4, October 1974
- SEV75 Severance D. G.
A Parametric Model of Alternative File Structures
Info. Systems, Vol 1 No 2, 1975, pp 51-55
- SEV77 Severance D. G. and Carlis J. V.
A Practical Approach to Selecting Record Access Paths
Computing Surveys, Vol. 9 No. 4, December 1977
- SMI75 Smith J. M. and Chang P. Y. T.
Optimizing the Performance of a Relational
Algebra Database Interface
CACM, October 1975
- STR79 Stroett J. W. M. and Engmann R.
Manipulation of Expressions in a Relational Algebra
Info. Systems, Vol. 4 pp 195-203, 1979
- ULL80 Ullman J. D.
Principles of Database Systems
Computer Science Press, 1980
- WON76 Wong E. and Youssefi K.
Decomposition- a Strategy for query processing
ACM TODS Vol. 1 No. 3, September 1976
- YA78a Yao S. B. and DeJong D.
Evaluation of Database Access Paths
Proc. ACM SIGMOD Int. Conf. Manage. of Data
Austin, Texas, May 1978, pp. 66-77
- YA78b Yao S. B.
Optimization of Query Evaluation Algorithms
Tech. Rep. TR283, Comp. Sci. Dept.
Purdue Univ. W. Lafayette, Ind., August 1978
(A more detailed version of [YA079])
- YA077 Yao S. B.
An Attribute Based Model for Database
Access Cost Analysis
ACM TODS Vol. 2 No. 1, March 1977
- YA079 Yao S. B.
Optimization of Query Evaluation Algorithms
ACM TODS, Vol. 4 No. 2, June 1979
- YU78 Yu C. T., Luk W. S., and Siu M. K.
On the Estimation of the Number of Desired
Records with Respect to a Given Query
ACM TODS, Vol. 3 No. 1, March 1978