The Stability of Query Evaluation Plans and Dynamic Query Evaluation Plans

Goetz Graefe

Oregon Graduate Center
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 88-003

The Stability of Query Evaluation Plans and Dynamic Query Evaluation Plans

Goetz Graefe
Oregon Graduate Center

Abstract

A database query embedded in a program written in a conventional programming language is optimized when the program is compiled. The query optimizer must make assumptions about the values of the program variables that appear as constants in the query and about the data stored in the database. These assumptions include that the query can be optimized realistically using guessed "typical" values for the program variables and that the database will not change significantly between query optimization and query evaluation. The optimality of the resulting query evaluation plan depends on the validity of these assumptions. If a query evaluation plan is used repeatedly over an extended period of time, it is important to determine when reoptimization is necessary. We aim at developing criteria when reoptimization is required, how these criteria can be implemented efficiently, and how reoptimization can be avoided by using a new technique called dynamic query evaluation plans for embedded queries. Note that the same problem occurs in database support for logic programs that rely on backtracking. A specific database query is executed repeatedly with different variable instantiations, i.e. different constants in the query predicate. The concepts presented in this paper will be included in the next version of the EXODUS optimizer generator.

1. Introduction

In many database applications, queries are embedded in application programs written in a conventional programming language like Cobol or C. These application programs are compiled once and then executed repeatedly over an extended period of time. In most database management systems, the embedded queries are optimized when the programs are compiled. The resulting query evaluation plan is stored in an 'access module' associated with the program's object code, and is activated by procedure calls. This organization avoids the optimization overhead when the program and the query are executed, thus allowing for fast execution and high transaction rates.

The disadvantage of query optimization at compile time is that varying constants in the query predicate or changes in the database cannot be reflected in the query evaluation plan.

Thus, queries might be executed according to query evaluation plans that are far from optimal.

An embedded database query is used to retrieve information pertinent to the program. The program data and the database query are connected by using one or more program variables as constants in the query predicate. The value of the program variables is not known at compile-time, i.e. when the query is optimized. Database query optimization does not genuinely depend on the values of constants in the database query. However, in order to optimize a complex query correctly, the optimizer needs to estimate intermediate result sizes, which in turn may depend on the constants in the query predicate. For example, the optimal join strategy depends on the number of tuples to be joined from each input. If the inputs are the results of selections, it is imperative that the optimizer estimate the select output size.

When optimizing an embedded query with program variables in the query predicates, database query optimizers guess a "typical" value for each variable or a selectivity for each predicate clause. Selectivities and costs of alternative query evaluation plans are estimated using the guessed values. In the case of complex queries with inequality constraints involving program variables, the resulting query evaluation plan can be far from optimal.

For example, consider a database query to find all employees with a salary greater than $30,000 and their departments. Assume that this query requires to join the employee relation with the department relation on the department number, and that the only two indices in the database are for the employee relation on salary and for the department relation on department number. If there are very few employees in this salary range, it probably is best to find qualifying employees using the salary index, to use the result as outer relation in the join and to perform repeated index lookups on the department relation. If there are very many such employees, however, it might be best to read the entire department relation (the smaller of the two relations) into a memory-resident hash table and then probe the hash table using the employee tuples, using a file scan on the employee file. In this example, the scanning strategies, the join order, and the join algorithm depend on the cardinality of the two relations and the

selectivity of the selection predicate. Imagine that the above query is embedded in an application program, and that the cutoff value, the constant $30,000, is replaced by a program variable. In this case, a conventional query optimizer cannot satisfactorily optimize the query.

Independent of values of program variables, another problem with stored query evaluation plans is that they are optimized for the database at the time when the query was optimized, not at the time when query is executed. In the mean time, the database may change such that the query evaluation plan is no longer optimal. We have to distinguish between changes that make a query evaluation plan infeasible and changes that make a query evaluation plan non-optimal. The former category includes removing relations and indices that are referenced in the query evaluation plan and has been addressed by earlier work [Chamberlin1981a]. The latter category includes creating new indices, inserting or deleting a large number of tuples, and modifying a large number of attribute values. We are concerned with the latter category.

Consider the above example again with a constant of $30,000. If most of the newly hired employees are highly paid specialists, the number of qualifying employees will change, as well as the portion of qualifying employees (i.e. the selectivity). If the query is optimized only once and the resulting access plan reused over a long period of time, the query evaluation plan will eventually become suboptimal.

So far, we have looked at the problem as it appears in conventional database management systems. In a database system used to support logic programs, the same problem occurs very frequently. Consider a model of execution that relies on backtracking, particularly Prolog [Warren1977a, Clocksin1981a]. The same clause is activated repeatedly with different variable instantiations. If the clause contains a database query, the same database query is performed with different constants in the query predicate, possibly requiring different query evaluation plans. The techniques described here are aimed at providing more flexibility and better performance for both conventional and non-conventional database management systems and applications.

The next section describes a new concept which we call the **stability of query evaluation plans**. In Section 3, we propose techniques to test efficiently whether a query evaluation plan is optimal, i.e. whether the query with the actual constants is optimal for the current state of the database. Section 4 describes how some of the problems addressed in this paper can be a solved, namely by choosing the scanning strategy dynamically. Section 5 extends these techniques to join processing. In Section 6, we develop a general technique, called **dynamic query evaluation plans**, with the goal of providing minimal overhead, maximal flexibility, or both. Section 7 briefly overviews the work in progress to assess the practical feasability of dynamic query evaluation plans. Section 8 contains a summary and our conclusions.

2. The Stability of Query Evaluation Plans

As pointed out in the introduction, the optimality of a query evaluation plan depends on whether or not the guesses and assumptions made during optimization are correct during query execution, or, to be more exact, on how significantly the guesses are wrong and the assumptions are violated. To date, there is no exact measure how much is significantly. It certainly depends on the database and the query; it also depends on the query evaluation plan itself. A query evaluation plan might be better than another one, but only slight changes in the database would favor the other one. For example, if the smaller of two join inputs fits into the available main memory, hash join is preferred over merge join [Shapiro1986a]. However, if the relations grow only by a small number of tuples such that the main memory hash table would overflow and the overflow must be resolved using temporary files [Gerber1986a, DeWitt1986a], merge join might be the preferred strategy. If the query in this example is an embedded query and changes in the database are quite likely between query optimization and query execution, it might be the safer choice to ignore the advantage of hash join and choose merge join.

We define the **stability of query evaluation plans** to be the robustness of query evaluation plans against changes in the database and in the query constants. Robustness expresses the amount of change that will not change the fact that the query evaluation plan is the optimal

one or its cost is very close to the optimum. In the example above, the query evaluation plan using hash join is not very robust[1].

The major problem with this definition is that it is not a concrete measure for robustness. In the next sections, we describe a way to think about the problem and how we plan to capture it in more concrete mathematical terms.

2.1. Range of Optimality

The range over which a query evaluation plan is optimal for a given query depends on the query constants, the database, and their relationship.

Consider the query given in the example above. We indicated two query evaluation plans for it, one of them using an index scan for the selection and an index lookup for the join, the other one using two file scans and a hash based join method. We encountered the problem that either one of the plans could be the optimal one, depending on the concrete situation. Both query evaluation plans have costs associated with them that depend on the cutoff value for the attribute salary. The cost functions could be parameterized, and plotted over the range of possible salary cutoff values. If we had a plot that shows the respective curves for each of the two query evaluation plans, it would be trivial to select the better plan given a particular cutoff value for the salary attribute.

When the database changes (e.g. by hiring many highly paid specialists), the curves change in their exact positions, but the general method would still work. We observe that the shape and position of the curves depend on the cardinality of the relations involved and the selectivity of the selection predicate. Consequently, the curves should be parameterized, too.

---

[1] In general, query evaluation plans involving hash join are less likely to be robust due to the fact that small changes can lead to hash table overflow and an ungraceful performance degradation. Merge join, on the other hand, tends to lead to more robust query evaluation plans because the cost of both sorting and merging grows smoothly with the size of the relations to be joined. We ignore here non-smooth growth due to duplicate attribute values as this problems typically is not significant.

So far, we have considered only cases with one program variable in the query predicate of an embedded query. Now imagine a query with two program variables. In order to plot a cost function that depends on two variables, we use a 3-dimensional plot. The result looks like a mountain range over a plane spanned by the two variables in the query predicate. If there are two access plans to choose from, two surfaces interpenetrate. For any given pair of values for the program variables, the cheaper query evaluation plan can readily be determined.

This model of thinking about access plans as functions and curves can be generalized for any number of dimensions and program variables in the query predicate. We restricted ourselves to the cases of one and two dimensions only for the sake of explanation.

In order to enable a database management system to decide which one of a set of query evaluation plans promises the best performance, the system must be knowledgeable about the relation cardinalities, the selectivities, and the cost functions. The cardinalities of the stored files are readily available in most systems. The cost functions for each of the algorithms involved is specified by the database implementor, and it is possible to design the database system in such a way that it is able to combine several cost formulas into one calculation. The remaining problem is to estimate selectivities and cardinalities of intermediate results depending on both the database state and the constants in the query predicate. What is needed is a method practical for database systems to concisely describe data distributions.

2.2. Distribution Description Tools

Several methods to estimate selectivities have been suggested and implemented in database systems, including histograms and inverted histograms [Kooi1980a, Kooi1982a, Piatetsky-Shapiro1984a]. Recently, we have suggested to use density functions [Graefe1987a]. If the density functions are assumed to be polynomials, it is possible to derive the coefficients from the moments. The moments can be collected very efficiently and possibly maintained by update transactions. One of the main advantages of density functions is that they can capture multi-dimensional distributions, whereas previous methods relied on the assumption that attributes

are statistically independent. Multi-dimensional distributions are described using multi-dimensional density functions, the coefficients of which can be derived using co-moments. Since multi-dimensional density functions are more costly to derive and to use than one-dimensional ones, it is important to note that it can be determined very easily from the co-moments whether two attributes are correlated and whether a multi-dimensional density function allows significantly better estimates.

In order to estimate the selectivity of a simple predicate, e.g. for what portion of our employees the salary is between $20,000 and $30,000, we need to know the density function for the salary attribute in the employee relation, say f_{salary}, and solve the integral

$$\int_{20,000}^{30,000} f_{salary}(x)\, dx$$

This integral is easy to solve numerically if the function f_{salary} is a polynomial. Integral of multi-dimensional functions are used to estimate the selectivity of predicates involving two or more correlated attributed.

Density functions also allow to estimate the cardinality of join results. If the join predicate is $x.a=y.a$ and the appropriate density functions $f_{x.a}$ and $f_{y.a}$ are known, the result cardinality can be calculated using the integral

$$\int_{A} f_{x.a}(a)\, f_{y.a}(a)\, da$$

over the join domain A. Again, if the density functions are polynomials, this integral can easily be solved numerically. More details can be found in [Graefe1987a].

2.3. Development of the Concept 'Query Evaluation Plan Stability'

The difficulty of capturing the concept of query evaluation plan stability stems from the fact that multiple dimensions need to be considered. In fact, expressing the concept in a formula and assigning a "stability coefficient" like

$$\frac{\textit{range of optimality in values and distributions}}{\textit{range of possible values and distribution}}$$

make only limited sense. We hope to develop a more concrete mathematical definition. Research on the decision trees to be introduced in the following sections is expected to provide more insight into this problem.

Fortunately, finding a definition is not crucial to operationalize the concept, and to implement significant improvements in database systems. For the time being, we think about the stability of query evaluation plans as a concept only, and concentrate on ramifications of the lack of stability in query evaluation plans.

3. Test of Optimality

In order to test whether a given query evaluation plan is optimal for a set of query constants given in the program variables, a special predicate is associated with each compiled query evaluation plan. When a plan is activated with a record of actual values for the query constants, the associated predicate is evaluated on this record and returns one of the values *TRUE* or *FALSE*. In the case of *TRUE*, the access plan is considered appropriate, and query processing proceeds as in existing database systems. In the case of *FALSE*, the access plan is considered suboptimal, and the database query optimizer is invoked.

In order to include the data distributions currently found in the database, the coefficients for the density polynomials (see Section 2.2) are kept in the catalogs, together with information needed to verify the feasability the query evaluation plan (e.g. existence if indices). If all information pertinent to one relation is kept physically clustered, no additional I/O is incurred for testing the optimality of a query evaluation plan, and only insignifant computational expense is needed for evaluating a polynomial.

It can be argued that the overhead for evaluation of the predicate is unacceptable for high-performance database systems. Consider, for example, a banking teller transaction. Probably, the optimizer selects to access the appropriate account records using a index on account numbers (assuming this index exists), and testing this access plan's optimality will be wasted

effort. While this is true, we would like to alert the reader to two facts. First, if the predicate used to test the optimality is compiled into machine code (just as predicates on data records should be), evaluation requires probably in the range of 20 to 100 instructions. Second, in extreme cases like the banking teller example, the predicate can be designed such that it always returns *TRUE* without inspecting the record of actual values, which requires only one instruction.

When the query optimizer has been reinvoked for a certain query after the original query evaluation plan was rejected, it is probably a good idea to keep both plans and choose dynamically among them in future activations of the query. In general, there might be a number of access plans to be choosen from dynamically. It is not even necessary to wait until the original query evaluation plan is rejected; rather, it should be possible to prepare more than one plan when the query is optimized originally. We will discuss this concept in more detail in the following sections.

4. Dynamic Decisions on Scanning Strategies

Dynamic decisions on the best scanning strategy are the first step towards dynamic query evaluation plans. The alternative scanning strategies are file scan and index scan, if a suitable index exists. Let us first review the rationale by means of an example.

Consider a moderately large file, e.g. with 10,000 employee records stored in 1,000 pages. If we need to retrieve all records, we should use a file scan, as this method ensures that we inspect each data page only once and allows high-performance techniques like read-ahead. If we retrieve only 10 of the 10,000 records, we do better by using an index (assuming one exists). We would have to read at most 10 data pages, and probably less than 30 index pages. If we retrieve 2,000 records, however, it is quite likely that we eventually have to read all 1,000 data pages and the index adds only to the overhead for three reasons. First, we have to read the index pages, second, if the index is an unclustered index, we inspect many pages more than once, and third, we cannot make use of read-ahead. Yao [Yao1979a] gives an estimation formula to

determine the number of page accesses from the number of qualifying records; other research refined these formulas for the case that clustering attribute and selection attribute are correlated, e.g. [Christodoulakis1983a, Zanden1986a]. Besides the considerations concerning pages access, there are also considerations concerning locking and concurrency (assuming locks are used for concurrency control). If only relevant records are accessed using the index, only those records need to be locked. For a file scan, all records must be locked. But then again, this can be done with a single call to the lock manager if a hierarchical locking scheme is used. Depending on the selectivity and the current system load, either one of the two strategies can be optimal.

The lesson from this example is that there are many considerations that favor index scan over file scan or vice versa, depending on the actual situation when the query evaluation plan is activated. The choice depends mainly on the selectivity of the predicate and to a small degree on the current system load, and thus can be performed by estimating the selectivity at run-time. If the interfaces to the file scan procedure and the index scan procedure are equal (or very similar), a dynamic choice of the scan method can be implemented with reasonable effort and run-time overhead. The task of the query optimizer is to determine the break-even point. More exactly, the query optimizer must determine the formulas to find the break-even point and to compare it with the actual selectivity, and include these formulas in the query evaluation plan.

5. Dynamic Decisions on Join Strategies

As we have seen in the example in the introduction, in some cases it is recommendable to select the join strategy at run-time. If more than one join operators are cascaded in a query evaluation plan, it may even be advantageous to delay the decision about the join order. Unfortunately, the decisions on the join strategies are interdependent. Most importantly, the physical (sort) order of intermediate results may affect the cost of alternative algorithms for the next processing step.

The query optimizer's task for this kind of access plans is more complex. Instead of following a fixed set of assumption and guesses about distributions, selectivities, etc., it must design an efficient decision procedure which can be executed when the query evaluation plan is activated. This decision procedure must have resolved the interdependence of partial decisions into a straight-forward decision tree, and include formulas for the break-even points between alternative plans.

Since the number of possible join strategies is very large, even for only moderately complex queries, it is not possible to include all query execution plans in the access module. There are two solutions to this problem. First, the optimizer can select a subset of query execution plans. The plans are selected such that they allow reasonably efficient query evaluation for any set of parameters. In order to keep this set small, plans with great stability must be selected. Second, instead of storing complete plans, only elements of the plans are stored, and linked together when the access module is activated.

6. Dynamic Query Evaluation Plans

Instead of a set of query evaluation plans, as suggested in the previous sections, we propose to avoid redundancy in the access module by designing the data structure for the query evaluation plans to be more flexible. The access modules of existing database management systems consist of a number of components, e.g. an indicator for file scan with a file name and a search predicate, an indicator for hash join with a hash function and a comparison function, etc. These components are bound together in a static query evaluation plan by the query optimizer, thus hiding the fact there are several components. **Dynamic access modules** consist of the same components, only the binding between components is more flexible. The only new component is the **decision tree** used to analyze the actual query constants and the data distributions. When an access module is activated, the first step is to evaluate the decision tree. Concurrently with the evaluation of the decision tree, this step sets up the bindings between the components of the access module.

Besides the decision tree designed by the optimizer, the access module must also contain the support functions for all possible query evaluation plans. These support functions include predicate functions for scans, comparison functions for sort and join, hash functions for hash join, etc. The physical organization of the access module must allow equally efficient execution of any of the query evaluation plans.

Introducing the dynamic choices outlined in the sections above are an important step towards more flexible and efficient database systems. However, instead of considering scanning strategies and join strategies separately, we intend to investigate how far the concept of **dynamic query evaluation plans** can be generalized. Once we have investigated the concept by a trial implementation and have developed implementation guidelines for database processing algorithms, selection of scanning and join strategies are special cases of a general problem. Other special cases that come immediately to mind are join orders and the placement and execution of aggregate functions. The principal goal of the implementation guidelines is to develop modular query evaluation plans that can be activated in a number of ways.

It can be argued that setting up the bindings dynamically inflicts too much overhead on query processing. Consider the example of a banking teller transaction introduced in Section 3. If there is no gain in using a dynamic access module, the decision tree can be an empty function. In this case, all bindings must be set statically, and "evaluating" the decision tree costs only one instruction. The techniques proposed here do not require that as many choices as possible must be delayed until run-time. Their advantage is that they allow to delay exactly as many choices as advisable.

7. State of the Implementation

In order to assess whether dynamic query evaluation plans are a viable alternative to existing static plans, we are undertaking a trial implementation of a run-time system using the new data structures. The methods currently implemented are file scan, select, project, duplicate elimination, hash join, division, aggregate functions, and print. Each of the algorithm is imple-

mented as an iterator, i.e. there are an *open*, *next*, and *close* procedure for each algorithm. Associated with each algorithm is a *state record*. Part of a state record are the addresses of the appropriate procedures. The arguments for the algorithms, e.g. predicates, are kept in the state records.

In queries involving more than one operator (i.e. almost all queries), state records are linked together by means of *input* pointers, also kept in the state records. Calling *open* for the top-most operator results in instantiations for the associated state record, e.g. allocation of a hash table, and in *open* calls for all inputs. In this way, all iterators in a query are initiated recursively. In order to process the query, *next* for the top-most operator is called repeatedly until it fails. Finally, a *close* call for the top-most operator recursively "shuts down" all iterators in the query. This model of query execution matches very closely the one being implemented in the E database implementation language [Richardson1987a].

The state records are the components of dynamic query evaluation plans. Instead of being linked together statically as they are in the trial implementation, however, they will be linked dynamically. For example, for the queries used in the "Wisconsin Benchmarks" [Bitton1983a, Boral1984a], i.e. queries of moderate complexity, as few as 3 to 8 pointers need to be assigned to make a dynamic query execution plan executable, truly a very low overhead. Very preliminary performance comparisons strongly suggest that the resulting query execution times are acceptable, and that calling the input procedures via a pointer in a record incurs only minimal overhead. Thus, we are confident that a database system using dynamic query evaluation plans will perform as well as one using conventional query evaluation plans. We are currently designing the required extensions for the EXODUS optimizer generator [Graefe1987b, Graefe1987c, Graefe1987d] to produce dynamic query evaluation plans.

8. Summary and Conclusions

In this paper, we have introduced a new concept called the *stability of query evaluation plans*. For cases in which the stability of a query evaluation plan does not cover the entire

range of possible queries as defined by the range of query constants and data distributions, we suggest a very efficient scheme to decide dynamically whether to reoptimize the query or to choose one of several existing query evaluation plans. Finally, we proposed *dynamic query evaluation plans* which build the query evaluation plan very efficiently at run-time from fragments prepared by the query optimizer using a decision and linking procedure also included in the access module by the query optimizer. Implementation work in progress indicates that the run-time overhead for dynamic query evaluation plans is negligibly low.

The proposed concepts can be expected to enhance significantly database systems for both conventional and non-conventional application domains. For conventional domains, the emphasis is on better and more flexible support for queries embedded in application programs. Within non-conventional domains, we envision the new techniques to be particularly advantageous for database systems supporting logic programming relying on backtracking, thus executing the same query repeatedly with different variable instantiations.

References

Bitton1983a.
D. Bitton, D.J. DeWitt, and C. Turbyfill, "Benchmarking Database Systems: A Systematic Approach," *Proceeding of the Conference on Very Large Data Bases*, pp. 8-19 (October-November 1983).

Boral1984a.
H. Boral and D.J. DeWitt, "A Methodology for Database System Performance Evaluation," *Proceedings of the ACM SIGMOD Conference*, pp. 176-185 (June 1984).

Chamberlin1981a.
D.D. Chamberlin, M.M. Astrahan, W.F. King, R.A. Lorie, J.W. Mehl, T.G. Price, M. Schkolnik, P. Griffiths Selinger, D.R. Slutz, B.W. Wade, and R.A. Yost, "Support for Repetitive Transactions and Ad Hoc Queries in System R," *ACM Transactions on Database Systems* **6**(1) pp. 70-94 (March 1981).

Christodoulakis1983a.
S. Christodoulakis, "Estimating Block Transfers and Join Sizes," *Proceedings of the ACM SIGMOD Conference*, pp. 40-54 (May 1983).

Clocksin1981a.
W. Clocksin and C. Mellish, *Programming in Prolog*, Springer, New York (1981).

DeWitt1986a.
D.J. DeWitt, R.H. Gerber, G. Graefe, M.L. Heytens, K.B. Kumar, and M. Muralikrishna, "GAMMA - A High Performance Dataflow Database Machine," *Proceedings of the Conference on Very Large Data Bases*, pp. 228-237 (August 1986).

Gerber1986a.
R. Gerber, "Dataflow Query Processing using Multiprocessor Hash-Partitioned

Algorithms," *Ph.D. Thesis*, University of Wisconsin, (October 1986).

Graefe1987a.
G. Graefe, "Selectivity Estimation Using Moments and Density Functions," *submitted for presentation at the ACM SIGMOD Conference 1988*, (November 1987).

Graefe1987b.
G. Graefe, "Rule-Based Query Optimization in Extensible Database Systems," *Ph.D. Thesis*, University of Wisconsin, (August 1987).

Graefe1987c.
G. Graefe and D.J. DeWitt, "The EXODUS Optimizer Generator," *Proceedings of the ACM SIGMOD Conference*, pp. 160-171 (May 1987).

Graefe1987d.
G. Graefe, "Software Modularization with the EXODUS Optimizer Generator," *IEEE Database Engineering*, (November 1987).

Kooi1980a.
R.P. Kooi, "The Optimization of Queries in Relational Databases," *Ph.D. Thesis*, Case Western Reserve University, (September 1980).

Kooi1982a.
R.P. Kooi and D. Frankforth, "Query Optimization in Ingres," *IEEE Database Engineering* **5**(3) pp. 2-5 (1982).

Piatetsky-Shapiro1984a.
G. Piatetsky-Shapiro and C. Connell, "Accurate Estimation of the Number of Tuples Satisfying a Condition," *Proceedings of the ACM SIGMOD Conference*, pp. 256-276 (June 1984).

Richardson1987a.
J.E. Richardson and M.J. Carey, "Programming Constructs for Database System Implementation in EXODUS," *Proceedings of the ACM SIGMOD Conference*, pp. 208-219 (May 1987).

Shapiro1986a.
L.D. Shapiro, "Join Processing in Database Systems with Large Main Memories," *ACM Transactions on Database Systems* **11**(3) pp. 239-264 (September 1986).

Warren1977a.
D.H.D. Warren, L.M. Pereira, and F. Pereira, "PROLOG - The Language and its Implementation Compared with Lisp," *Proceedings of ACM SIGART-SIGPLAN Symposion on AI and Programming Languages*, (1977).

Yao1979a.
S.B. Yao, "Optimization of Query Evaluation Algorithms," *ACM Transactions on Database Systems* **4**(2) pp. 133-155 (June 1979).

Zanden1986a.
B.T. Vander Zanden, H.M. Taylor, and D. Bitton, "Estimating Block Accesses When Attributes Are Correlated," *Proceeding of the Conference on Very Large Databases*, pp. 119-127 (August 1986).