ACCESS PATHS IN THE "ABE" STATISTICAL QUERY FACILITY

by

Anthony Klug

Computer Sciences Technical Report #474

May 1982

Access Paths in the "Abe"

Statistical Query Facility

Anthony Klug

Computer Science Department

University of Wisconsin

Abstract

An increasingly important part of information processing today involves the taking of counts, sums, averages, and other statistical or aggregate quantities. The "Abe" query language is designed to make formulation of complicated aggregations simple. Access path selection in Abe finds efficient ways to execute these complicated queries. Access paths for Abe queries perform "aggregate joins", that is, they compute aggregate quantities at the same time as they join subqueries with parent queries. This can be done using index scans or merging scans depending on how many "partitions" need to be accessed.

This work was supported in part by NSF Grant MCS8102864

1. Introduction

An increasingly important part of information processing today involves the taking of counts, sums, averages, and other statistical or aggregate quantities. While we have witnessed tremendous overall advances in relational database technology in the last decade, the relational technology for processing these statistical queries still requires more work. The statistical facilities in many relational query languages are difficult to use or do not provide as much power as might be desired. In addition, the query processing facilities in these systems have not incorporated enough intelligence in access path selection for efficient processing of statistical queries.

In this paper we present some elements of the "Abe" statistical query facility being developed at the University of Wisconsin. We concentrate on the access paths for efficiently executing Abe queries.

In the next section we describe briefly the Abe language. The Abe query language is powerful, yet simple. It is a pure relational calculus language [Ullm] with a friendly full-screen user interface.

In section 3 we describe the access paths in the Abe system. We show that the access patterns in computing a query having aggregates are virtually identical to those encountered in computing a simple join. We then present

three scan procedures for computing aggregates: one is like a file (or segment) scan; one is analogous to an index scan; and one is similar to a merging scan. Cost formulas for these access procedures are given, and some example access paths are compared.

In section 4 we examine query language features and access paths in System-R and Ingres. We will see that each system essentially uses just one strategy in evaluating aggregates.

2. Abe Background

Space limitations do not permit us to give an extensive description of the Abe query language. We will give a brief introduction to the language and a few examples. More examples can be found in [Klug81].

Aggregate Functions

The concept of an aggregate function is quite simple. An aggregate function takes a set of tuples (a relation) as an argument and produces a single simple value (usually a number) as a result. The aggregate functions we consider are: ave, count, max, min, and sum. Ave, max, min, and sum take an additional argument which identifies the attribute over which to perform the specified operation. Count requires no such parameter. For example, "sum_{salary}(employee)" returns the sum of the salary

attribute of the employee relation. By explicitly specifying the attribute over which to perform the aggregation, we remove the need to introduce the concept of "duplicate tuple". This simplifies the set of concepts used.

Embedding Aggregate Functions

There are two basic ways to repeatedly evaluate an aggregate function over similar sets of tuples within a query language: One way is to use a partitioning operator with the aggregate function applied to each partition; the second way is to repeatedly call (at least in principle) a subquery with the subquery's free variables having different values each time. Partitioning fits naturally into algebraic languages, and the concept of a subquery with free variables fits naturally into calculus-like languages. See [Klug] for more details.

Relational Calculus

In relational calculus, a query is constructed by specifying a source of output (the target list) and by giving a formula defining the properties desired of the output (the qualification). Both tuple relational calculus and domain relational calculus can be defined [Ullm]. In the former, variables refer to entire tuples, while in the latter variables refer to attributes of tuples. Domain relational calculus (without aggregates) is defined as follows:

The set of constants and the set of variables form the <u>terms</u>. The set of <u>formulas</u> is defined as follows: If R is a relation of degree n, and $t_1,\ldots,t_n$ are terms, then $R(t_1,\ldots,t_n)$ is a formula. If t_1 and t_2 are terms, and θ is '=', or '<', then $(t_1 \theta t_2)$ is a formula. If ψ, π are formulas, then the negation $\sim\psi$ and the disjunction $(\psi \vee \pi)$ are formulas. If ψ is a formula, and x is a variable then the quantification $(\exists x)\psi$ is a formula. If $t_1,\ldots,t_n$ are terms and if ψ is a formula, then $\{t_1,\ldots,t_n : \psi\}$ is an <u>alpha expression</u> of degree n. A <u>query</u> is a <u>safe</u> alpha expression having no free variables, that is, every variable is either within the scope of a quantifier or appears in the target list. ("Safe" means that all variables are constrained by the qualification to have a finite range. See [Ullm] for a formal definition.)

We have noted that an aggregate function takes a set of tuples as an argument and produces a single simple value as a result. The calculus has a class of syntactic objects, alpha expressions, corresponding to sets of tuples, and it has a class of syntactic objects, terms, corresponding to single simple values. Hence, to extend relational calculus to have aggregate functions, we need only make the following addition to the definitions:

If α is an alpha expression, and f is an aggregate function, then $f(\alpha)$ is a term.

Free variables within the alpha expression serve a function analogous to the use of group-by clauses in other languages.

The Abe query language is the result of a restriction and an enhancement of relational calculus:

(1) Relational calculus is restricted to a manageable subset, the <u>conjunctive queries</u>. A conjunctive query is one in which all alpha expressions have the form:

$$\{t_1,\ldots,t_n : (\exists y_1,\ldots,y_k)(c_1 \;\&\ldots\&\; c_m \;\&\; C)\}$$

where each c_i (a <u>conjunct</u>) is an atomic formula $R(u_1,\ldots,u_n)$, and C is a boolean combination of atomic formulas of the form $u_1 \Theta u_2$ (u_1, u_2 terms). (This is a generalization of the conjunctive queries of [ChMe] and the tableaux of [AhSu].)

(2) The conjunctive query subset is enhanced with a user-friendly interface without sacrificing the mathematical precision of the underlying formalism.

Our philosophy, in contrast to the approaches taken with QBE [Zloo] and other "friendly" query languages, is to leave nothing "implicit" and to accurately follow the relational model in every respect (i.e., no concept of dupli-

cates). The friendly interface is similar to the QBE interface: At any time one conjunctive alpha expression is displayed on the screen. (This is the current level.) Each subquery has a name and can be made the current query by "opening" it. Every relation appearing in a conjunct of the current level is represented by a table on the screen. Each conjunct $R(u_1,\ldots,u_n)$ is represented as a row in the table for R. Bound variables, free variables, constants, and subquery names are given mutually distinct display modes using color, underlining, etc.

We give below two examples of Abe queries. These examples, and others in this paper, use the relational schema of Figure 1. Since color is not available here, we use the following typographic conventions for distinguishing bound variables, free variables, constants, and subquery names:

object	display type
bound variable	underlined
free variable	double underlined
constant	no display enhancements
subquery	upper case

Example 1. For each department in the marketing division, list the department name and a count of its employees. The calculus expression of this query (where the subquery is placed on a separate line for readability) is:

```
division (dvname, manager, budget)
department (dname, division, manager, budget)
employee (ename, salary, dept, seniority, recruiter)
```

Figure 1. Schema for a Company Database.

$$\{d, count(\bullet) : (\exists m,b)\ department(d,\text{"marketing"},m,b)\}$$
$$\{e : (\exists s,n,r)\ employee(e,s,d,n,r)\}$$

The Abe version (where each box represents one query level) is given in Figure 2.

Example 2. Sum employee salaries over departments, and then average these sums over divisions. Print division names and the averages.

The calculus expression is:

$$\{v, ave_2(\bullet) : (\exists m,b)\ division(v,m,b)\}$$
$$\{d, sum_2(\bullet) : (\exists m',b')\ department(d,v,m',b')\}$$
$$\{e,s : (\exists n,r)\ employee(e,s,d,n,r)\}$$

The Abe version is given in Figure 3.

3. Access Paths in Abe

One of the best features of the relational approach is that users do not have to be concerned about physical structures when formulating a query. Instead, the query processor assumes the burden of finding the best access path from

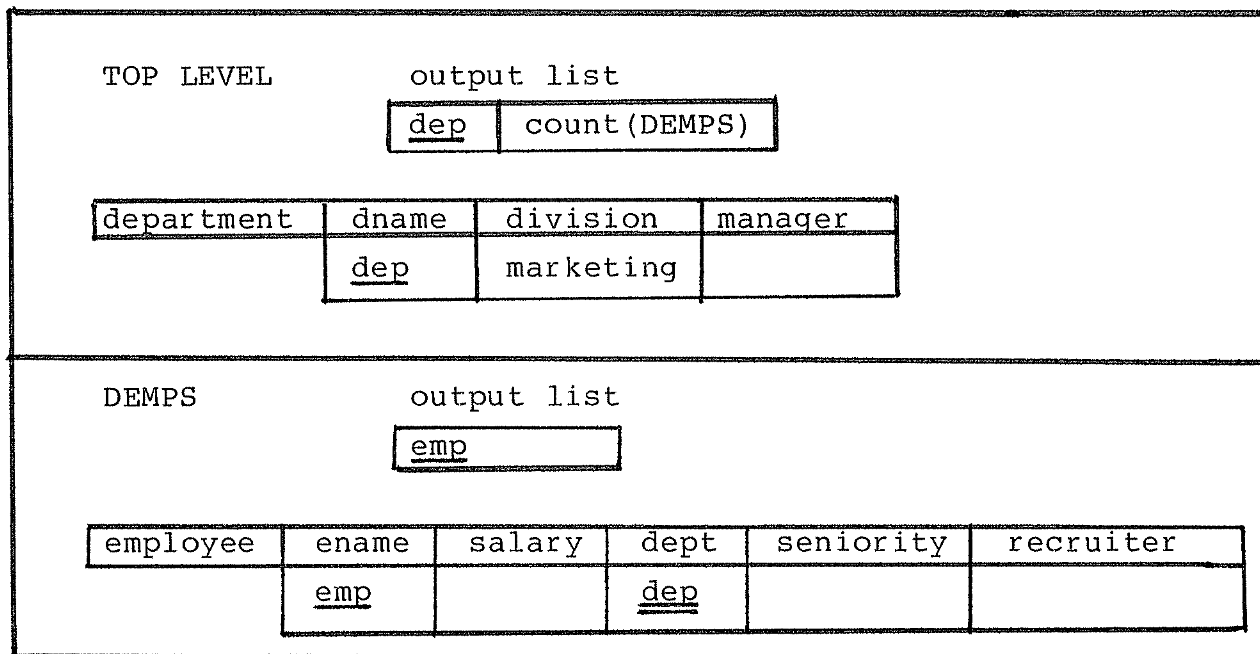

Figure 2.

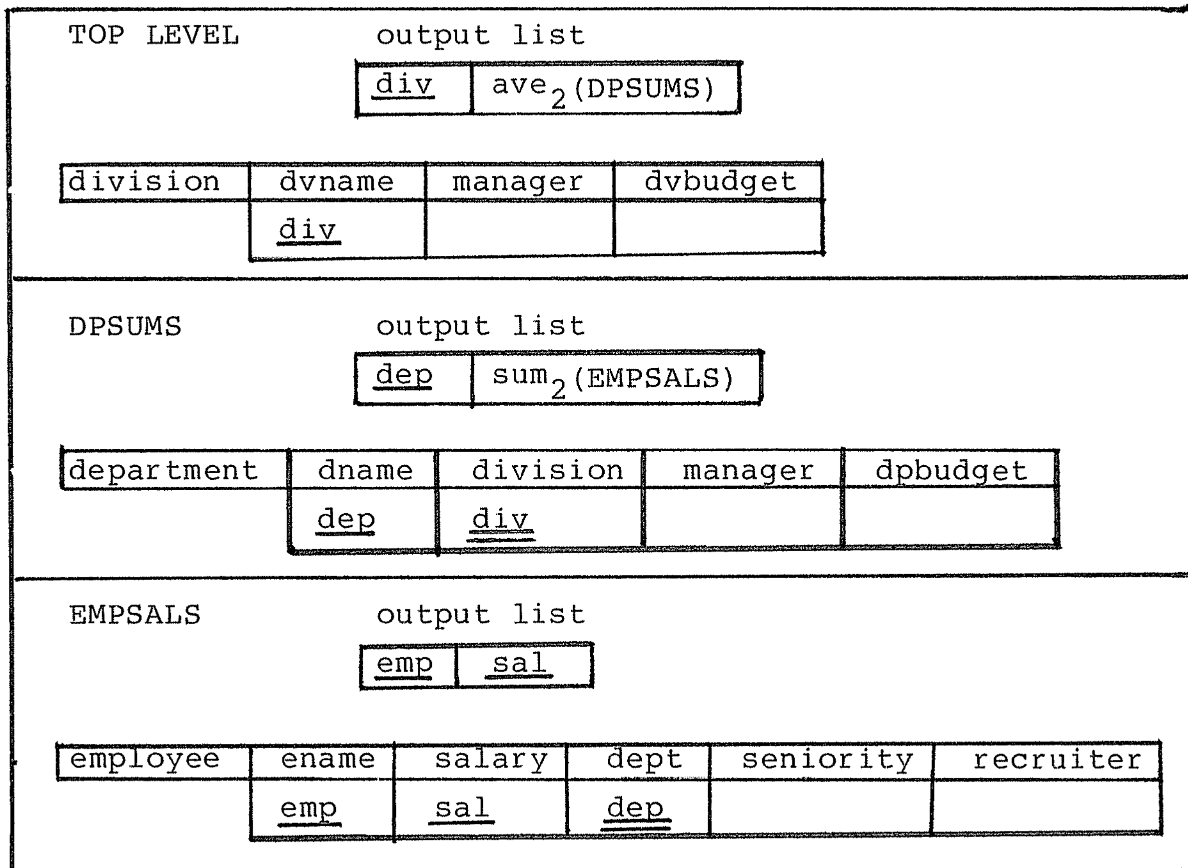

Figure 3.

among the many possible ways to implement a query. We extend the approach to access path selection taken in System-R [GACL]: Access paths are constructed from a basic set of procedures and functions according to cost estimates determined from database statistics kept in the schema concerning numbers of tuples, numbers of pages, numbers of distinct values, etc. We assume that the file system supports clustered files and indices. For ordinary joins, index scans and merging scans are possible, just as in System-R Our contribution is this: We introduce access paths which compute an aggregate value at the same time as they join the

subquery with the outer query. Consider the query in example 1. If there are few departments in the marketing division, and if there is an index on the dept attribute of employee, then there is no need to compute counts over the entire employee relation. An index scan can access just those employees in the required partitions. If most departments are in the marketing division, most department partitions of the employee relation will be used, and it will pay to sort the employee relation by department and simultaneously scan both relations, computing the employee counts as the scans proceed.

Abe Access Paths

Access paths are constructed from a small set of stylized *access procedures* and *access functions*. The first five of these are similar to access paths provided by System-R [GACL] [LoNi].

procedure sort(oldfile, b, newfile, sortfields, outputfields)
The "outputfields" of tuples in "oldfile" satisfying the boolean "b" are placed in "newfile" and are sorted according to "sortfields."

procedure fscan(file, b, var, ap)
The given "file" is scanned sequentially. For every tuple t satisfying the boolean "b", variable "var" is bound to t and access path "ap" is called. (The variable can be referenced in booleans of "ap".)

procedure iscan(index, index_range, b, var, ap)
The given "index" (over attribute A of file F, say) is used to access tuples in F whose A-values are within "index_range." For every one, say t, of these tuples which also satisfies the boolean "b", variable "var" is bound to t and access path "ap" is called.

<u>procedure</u> mscan(f1, b1, var1, f2, b2, var2, mergedoms, ap)
Files "f1" and "f2" are assumed to be sorted on "mergedoms." Both files are simultaneously scanned. One pass is made through each file. For every pair t1,t2 of tuples from "f1" satisfying boolean "b1" and "f2" satisfying boolean "b2", respectively, and having equal "mergedoms" values, the access path "ap" is called with variable "var1" bound to t1 and variable "var2" bound to t2.

<u>procedure</u> output(destination, value_list, b)
If the boolean "b" is satisfied, the value list, made up of attribute qualified variables and constants, is output to "destination" as a tuple. The destination may be a file or the user's terminal, denoted "TERMINAL."

For each of the above three scans, there is a corresponding scan which computes an aggregate. An additional procedure is used to accumulate partial aggregate quantities.

<u>function</u> aggfscan(f, b, var, scan_id, agg_fn, ap)
An internal variable, say x, is initialized according to the value of "agg_fn" on the empty set. For every tuple t in file "f" satisfying boolean "b", access path "ap" is called with variable "var" bound to t. The called path is expected to increment x with the accumulate procedure (see below). The final value of x is returned.

<u>function</u> aggiscan(index, index_range, b, var, scan_id, agg_fn, ap)
An internal variable, say x, is initialized according to the value of "agg_fn" on the empty set. The given "index" (over attribute A of file F, say) is used to access tuples in F whose A-values are within "index_range." For every one, say t, of these tuples which also satisfies boolean "b", variable "var" is bound to t and access path "ap" is called. The access path is expected to modify internal variable x by using the accumulate procedure defined below. The final value of x is returned.

<u>function</u> aggmscan(f1, b1, var1, f2, b2, var2, mergedoms, agg_fn, scan_id,
ap1, ap2)
Files "f1" and "f2" are assumed to be sorted on "mergedoms." Both files are simultaneously scanned. For

every tuple t1 from "f1" satisfying "b1", an internal variable x is initialized to the value of "agg_fn" on the empty set. Then for every tuple t2 from "f2" satisfying "b2" and having "mergedoms" values equal to those of t1, access path "ap1" is called with "var1" bound to t1 and "var2" bound to t2. It is expected that x is modified during this call. Before advancing the scan on "f1" to the next qualifying tuple, access path "ap2" is called. It can reference the final value of x with the expression "scan_id.VALUE".

procedure accumulate(scan_id, value)
: The internal variable associated with "scan_id" is updated to reflect the addition of "value" to the input set of the aggregate function associated with "scan_id." For example, if "agg_fn" is "max", then this procedure is equivalent to "x = max(x,value)". If "agg_fn" is "ave," then x is actually a structure holding both a sum and a count.

Two other constructs are added for a limited programming capability:

begin . . . end
: A sequence of access procedures can be executed by enclosing them within a begin-end block.

variable = access_function
: A variable can be associated with the returned value of a function. This could be used for naming the value before testing it with a boolean expression. We will use such equalities in the examples to improve clarity.

Each of the above functions and procedures has an associated cost. We give their cost functions in Figure 4.

Discussion of Costs

We have assumed that the dominate costs are for I/O and not for CPU utilization. (Weights for CPU can easily be added.) In the formulas, NP() denotes the number of pages in a file; SEL() (a number between 0 and 1) denotes the selectivity of a boolean expression (determined by consulting information in the schema); and NT() denotes the number

procedure or function	cost formula
sort	NP(oldfile) + NP(newfile) + 2*NP(newfile)*$\log_9$(NP(newfile)) (assuming 10 buffers available and > 10 pages to sort) OR 2*NP(oldfile)*$\log_9$(NP(oldfile)) if oldfile=newfile
fscan	NP(file) + SEL(boolean)*NT(file)*cost(acc_proc)
iscan	SEL(index_range)*NP(FILE(index)) + SEL(index_range & boolean)* NT(FILE(index))*cost(acc_proc) (if FILE(index) is clustered on the indexed attr.) OR SEL(index_range)*NT(FILE(index)) + SEL(index_range & boolean)* NT(FILE(index))*cost(acc_proc) (if not clustered)
mscan	NP(f1) + NP(f2) + NMATCHES(f1,b1,f2,b2,mergedoms)*cost(acc_proc)
output	cost(value_list) + size(value_list)/page size OR cost(value_list) if destination=TERMINAL
aggfscan	NP(file) + SEL(boolean)*NT(file)*cost(acc_proc)
aggiscan	SEL(index_range)*NP(FILE(index)) + SEL(index_range & boolean)* NT(FILE(index))*cost(acc_proc) (if FILE(index) is clustered on the indexed attr.) OR SEL(index_range)*NT(FILE(index)) + SEL(index_range & boolean)* NT(FILE(index))*cost(acc_proc) (if not clustered)
aggmscan	NP(f1) + NP(f2) + NMATCHES(f1,b1,f2,b2,mergedoms)*cost(ap1) + SEL(boolean1)*NT(f1)*cost(ap2)
accumulate	cost(value)
begin-end	sum of costs of procedures within begin-end.
assignment	cost of right-hand side

Figure 4. Cost Formulas for Basic Procedures and Functions

of tuples in a file. FILE() denotes the file of records indexed by the given index. NMATCHES denotes the expected number of tuples in the join of f1 and f2.

<u>sort</u>: If we are sorting a permanent relation into a temporary file, we must read NP(oldfile) pages. The number of pages in newfile is determined by the formula:

```
NP(newfile) = NP(oldfile)*SEL(b)*
   (#output fields)/(#oldlfile fields)
```

For simplicity, we assume all fields are the same length. We also assume that sorts use 10 buffers, giving a cost of 2*NP(newfile)*$\log_9$(NP(newfile)) when an in-core sort is not possible.

<u>fscan</u>, <u>aggfscan</u>: Scanning the file costs NP(file), and for each of the SEL(b)*NT(file) qualifying tuples, we execute "ap".

<u>iscan</u>, <u>aggiscan</u>: I/O for traversing the index tree is not considered. If the file is clustered on the indexed attribute, then the number of file pages touched is SEL(b)*NP(file). Otherwise, the number of pages touched is SEL(b)*NT(file) (one I/O to get each tuple). For each of the SEL(b)*NT(file) qualifying tuples, "ap" is executed.

<u>mscan</u>, <u>aggmscan</u>: One scan is made through each file for a

cost of NP(f1) + NP(f2). The number of matches, NMATCHES, is calculated from NT(f1), NT(f2), SEL(b1), SEL(b2), and other schema information such as key and foreign key constraints.

output: If the destination is a temporary file, the cost is the fraction of a page that one record occupies. If the destination is TERMINAL, we can assign a cost of zero since all access paths must output exactly the same set of tuples.

Selecting Access Paths

In access path selection, two things must be done: Access paths that implement the query must be generated, and the costs of these access paths must be determined. Here, we only illustrate how different paths may be best according to the query and the available file structures. Exactly how access paths are generated and how the search space is limited is beyond the scope of this paper.

Examples

The database statistics we use to evaluate the example access paths are given in Figure 5.

Example 3. Consider the query:

$$\{d, ave_2(\bullet) : (\exists v,m,b)\ department(d,v,m,b)\ \&\ 50K<b<200K\}$$

$$\{e,s : (\exists n,r)\ employee(e,s,d,n,r)$$

relation	number of pages	number of tuples	clustering attribute
division	5	50	dvname
department	50	250	dpname
employee	250	2500	ename

attribute	range
dvbudget	[500K, 2500K]
dpbudget	[50K, 250K]

index	attribute	relation
dpbudg	dpbudget	department
dpdiv	division	department
empdep	dept	employee

Figure 5. Example Database Statistics

This query computes the average employee salary by department for departments with a budget between $50K and $200K. In this query the index on dpbudget is of no use since most pages of the department relation are expected to be accessed. A nested loops access path could use an fscan on department which calls an aggiscan on employee (Figure 6). The fscan retrieves department tuples with a budget between 50K and 200K. When such a department is found, variable V1 is bound to it and the aggiscan and output statments are executed. To execute the aggiscan, an internal variable, say x, is initialized to (0,0) (zero sum and zero count), and the identifier ′S1′ is associated with this scan. Then the empdep index is used to find employee tuples whose dept attribute values match V1.dname. For every one of these,

```
fscan(department, 50K<dpbudget<200K, V1,                       (1)
   begin                                                       (2)
      X = aggiscan(empdep, dept=V1.dname, - ,                  (3)
             V2, S1, ave,
             accumulate(S1, V2.salary)                         (4)
          )
      output(TERMINAL, V1.dname, X)                            (5)
   end
)
```

Figure 6. Access Path for Example 3 (Nested Loops)

the variable V2 is bound to the tuple, and the accumulate statement is executed, the effect of which is to add one to x.count and V2.salary to x.sum.

The cost of this access path is given by the formulas in Figure 7.

A merging scan access path would sort the employee file on dept into a temporary file, and then merge the two files (Figure 8). In this path, the sort creates a temporary file with employee tuples sorted on the dept attribute. The aggmscan then scans the department and templ files. When a new department tuple is bound to variable V1, an internal variable, say x, is initialized to (0,0). Then all associated employees are scanned. Each employee tuple is bound to variable V2 and the accumulate is executed. When the entire group of employees has been scanned for a given department, the output statment is executed.

The cost formulas for this path are given in Figure 9.

```
cost       = cost(#1)                                    = 1925
cost(#1)   = NP(department) + SEL(50K<dpbudget<200K)*    = 50 + 0.75*
                NT(department)*cost(#2)                    250*10
cost(#2)   = cost(#3) + cost(#5)                         = 10
cost(#3)   = SEL(dept=V1.dname)*NT(employee) +           = 0.004*2500
             SEL(dept=V1.dname)*NT(employee)*cost(#4)
cost(#4)   = 0
cost(#5)   = 0
```

Figure 7. Cost Formulas for Example 3 (Nested Loops)

```
begin                                                          (6)
   sort(employee, - , templ, dept, (dept,ename,salary))        (7)
   aggmscan(department, 50K<dpbudget<200K, V1,                 (8)
      templ, - , V2, dname=dept, ave, S1,
      accumulate(S1, V2.sal),                                  (9)
      output(TERMINAL, V1.dname, S1.VALUE)                     (10)
   )
end
```

Figure 8. Access Path for Example 3 (Sort/Merge)

```
cost       = cost(#6)                               = 1284
cost(#6)   = cost(#7) + cost(#8)                    = 1084 + 200
cost(#7)   = (1 + 1)*NP(employee) +                 = 250 + 150 +
             2*NP(employee)*log9(NP(employee)         2*150*log9(150)
cost(#8)   = NP(department) + NP(templ) +           = 50 + 150 + 0
             NMATCHES(department,50K<dbbudget<200K,
                      templ,-,dname=dept)* cost(#9)
cost(#9)   = 0
cost(#10)  = 0
NP(templ)  = 250*1*(3/5) = 150
```

Figure 9. Cost Formulas for Example 3 (Sort/Merge)

In this example, we see that the merging scan is cheaper. This is because almost all "dept" partitions of the employee relation are accessed. After sorting templ, the pages are access only once, while in the index scan, employee pages may be accessed many times.

Example 4. We modify the last example query so that the restriction is more selective:

$$\{d, ave_2(\bullet) : (\exists v,m,b)\ department(d,v,m,b)\ \&\ 50K<b<60K\}$$
$$\{e,s : (\exists n,r)\ employee(e,s,d,n,r)$$

The fscan/aggiscan and the sort/aggmscan paths for this query are analogous to the ones above and will not be reproduced here. The cost formulas for the fscan/aggiscan are given in Figure 10.

The cost for the sort/aggmscan path does not change since the selectivity term for dpbudget is multiplied by a zero factor. Thus in this case the fscan/aggiscan is much less expensive than the sort/aggmscan path. The reason is that only 1/20-th of the "dept" partitions of the employee file are accessed, so that the high cost of sorting the entire employee file is mostly wasted. Also note that an iscan/aggiscan path would be worthwhile here, and its cost would be only 0.05*250 + 0.05*250*10 = 138.

```
cost      = cost(#1)                                      = 175
cost(#1)  = NP(department) + SEL(50K<dpbudget<60K)*       = 50 + 0.05*
                NT(department)*cost(#2)                     250*10
cost(#2)  = cost(#3) + cost(#5)                           = 10
cost(#3)  = SEL(dept=V1.dname)*NT(employee) +             = 0.004*2500
            SEL(dept=V1.dname)*NT(employee)*cost(#4)
cost(#4)  = 0
cost(#5)  = 0
```

Figure 10. Cost Formulas for Example 4 (Nested Loops)

Now we consider a more complicated set of queries in which there is a nesting of aggregates.

Example 5. In this example, we consider the following generic query:

```
{v, ave2(•) : (∃m,b)(division(v,m,b) & C)}
         |
{d, sum2(•) : (∃m´,b´)(department(d,v,m´,b´)
         |                           & C´)}
         |
   {e,s : (∃n,r) employee(e,s,d,n,r)}
```

This query sums employee salaries over departments and then averages these sums over divisions. The terms "C" and "C´" denote possibly empty restrictions on division and department budgets, respectively. We will consider three cases: (1) both C and C´ empty; (2) C having a 0.25 selectivity and C´ empty; and (3) C empty and C´ having a 0.25 selectivity.

We consider three access paths for evaluating this query. The first using two nested loops is given in Figure

11. The second path using two merging scans is given in Figure 12, and the third path, given in Figure 13, uses a merging scan for the outer aggregate and a nested loops for the inner aggregate.

Next, we construct formulas for the costs of the three paths. These are given in Figures 14 through 16. They are derived by a direct application of the basic cost formulas. The actual costs for the three versions of the query are given in Figure 17, where the underlined costs are the cheapest among the three paths. For each of the three versions of the query, a different access path is best. Two aggmscans are best when there is no selectivity and all partitions are accessed. Two aggiscans are best when there is selectivity at the top level. In that case, only a few partitions of the department relation are needed, and, consequently, only a few employee partitions are needed. When there is selectivity only on the department relation, it is best to use an index scan to get the employee partitions. Since all divisions are needed, it is best to use a merging scan at the outer level.

4. Comparisons with Other Systems

In this sections we examine the statistical query

```
fscan(division, C, V1,                                      (11)
   begin                                                    (12)
      Y = aggiscan(dpdiv, division=V1.dvname,               (13)
             C´, V2, S1, ave,
             begin                                          (14)
                Z = aggiscan(empdep, dept=V2.dname, - ,     (15)
                               V3, S2, sum,
                               accumulate(S2, V3.salary)    (16)
                    )
                accumulate(S1, Z)                           (17)
             end
          )
      output(TERMINAL, <V1.dvname, Y>, - )                  (18)
   end
)
```

Figure 11. Example 5, Path 1 -- Two Nested Loops

```
begin                                                           (19)
   sort(employee, - , temp1, dept, (dept,ename,salary))         (20)
   aggmscan(department, C´ , V1, temp1, - , V2, <dname,dept>,   (21)
                sum, S1,
                accumulate(S1, V2.salary),                      (22)
                output(temp2, <V1.division, S1.VALUE>, - )      (23)
   )
   sort(temp2, - , temp2, division)                             (24)
   aggmscan(division, C, V1, temp2, - , V2,                     (25)
                <dvname,division>, ave, S1,
                accumulate(S1, V2.sum),                         (26)
                output(TERMINAL, <V1.division, S1.VALUE>, - )   (27)
   )
end
```

Figure 12. Example 5, Path 2 -- Two Merge Scans

```
begin                                                                    (28)
   sort(department, C´ , templ, division, (division,dname))             (29)
   aggmscan(division, C, V1, templ, V2, <dvname,division>,               (30)
                ave, S1,
                begin                                                    (31)
                   Y = aggiscan(empdep, V2.dname, - ,                    (32)
                                V3, S2, sum,
                                accumulate(S2, V3.salary)                (33)
                       )
                   accumulate(S1, Y)                                     (34)
                end,
                output(TERMINAL, <V1.dvname,S1.VALUE>, - )               (35)
   )
end
```

Figure 13. Example 5, Path 3 -- Nested Loops within Merge Scan

```
cost(Path 1)    = cost(#11)
cost(#11)       = 5 + SEL(C)*50*cost(#12)
cost(#12)       = cost(#13) + cost(#18)
cost(#13)       = 0.02*250 + 0.02*SEL(C´)*250*cost(#14)
cost(#14)       = cost(#15) + cost(#17)
cost(#15)       = 0.004*2500 + 0.004*2500*cost(#16)
cost(#16)       = 0
cost(#17)       = 0
cost(#18)       = 0
```

Figure 14. Cost formulas for Path 1.

cost(Path 2) = cost(#19)
cost(#19) = cost(#20) + cost(#21) + cost(#24) + cost(#25)
cost(#20) = 250 + 150 + 2*150+$\log_9$(150)
cost(#21) = 50 + 150 + NMATCHES1*cost(#22) + SEL(C´)*250*cost(#23)
cost(#22) = 0
cost(#23) = 0.1
cost(#24) = 2*NP(temp2)*$\log_9$(NP(temp2))
cost(#25) = 5 + NP(temp2) + NMATCHES2*cost(#26) + SEL(C)*50*cost(#27)
cost(#26) = 0
cost(#27) = 0
NP(temp1) = 250*1*(3/5) = 150
NP(temp2) = 50*SEL(C´)*(2/4)
NMATCHES1 = 2500 (every employee has a unique department)
NMATCHES2 = 250*SEL(C´) (every qualifying dept. has a unique div.)

Figure 15. Cost formulas for Path 2.

cost(Path 3) = cost(#28)
cost(#28) = cost(#29) + cost(#30)
cost(#29) = 50 + NP(temp1) + 2*NP(temp1)*$\log_9$(NP(temp1))
cost(#30) = 5 + NP(temp1) + NT(temp1)*cost(#31) + 50*SEL(C)*cost(#35)
cost(#31) = cost(#32) + cost(#34)
cost(#32) = 0.004*2500 + 0.004*2500*cost(#33)
cost(#33) = 0
cost(#34) = 0
cost(#35) = 0
NP(temp1) = 50*SEL(C´)*(2/4)
NT(temp1) = 250*SEL(C´)

Figure 16. Cost formulas for Path 3.

Case	Costs		
	Path 1	Path 2	Path 3
C = empty C' = empty	2755	1412	2678
C = (500K<dvbudget<1000K) C' = empty	693	1412	2678
C = empty C' = (50K<dpbudget<100K)	880	1333	693

Figure 17. Cost Table

facilities of two relational systems: System-R and Ingres.

System-R: Query Language

The general form of an SQL query block [ABCE] [CAEG] is:

```
 SELECT    <target list>
 FROM      <relation list>
[WHERE     <boolean>]
[GROUP BY <field list>]
[HAVING    <boolean>]
```

In the target list there may be aggregate expressions of the form "agg_fn(attribute)". (Arithmetic expressions are also allowed.) The GROUP-BY clause indicates that the qualifying tuples are to be partitioned by the given fields and the

aggregate is to be applied to each partition.

Example 6. Find the average salary for employees by department.

```
SELECT dept, AVG(salary)
FROM employee
GROUP BY dept
```

From the syntax we can see that nested aggregation, that is, the output of one aggregate being the input of another, is impossible in SQL without building temporary relations in the course of several separate queries. Since access path selection cannot cut across queries, some access paths, possibly the least costly ones, will not be available.

When an aggregate is needed only in the WHERE-clause, a subquery may be used:

Example 7. Find departments which have more than 10 employees.

```
SELECT dname
FROM   department D
WHERE  10 < SELECT count(*)
            FROM   employee
            WHERE  dept = D.dname
```

System-R Access Paths

Access path selection in System-R is done on a query block basis [GACL]. That is, the optimizer determines an order in which to evaluate the query blocks, and then access

paths for individual query blocks are chosen separately. System-R access paths for query blocks are constructed from index scans, merging scans, and sorts.

Queries with GROUP-BY clauses are always evaluated by using a sort (if needed) and a sequential scan [LoNi]. Since only one relation is generally involved, other access paths are irrelevant. Correlated subqueries are always evaluated by a "tuple substitution" [WoYo] procedure. In fact, even if the nested query is equivalent to a simple join, System-R still uses tuple substitution. For example, assuming the employee has a manager attribute, the following query:

```
SELECT ename
FROM   employee X
WHERE salary >
    (SELECT salary
     FROM   employee
     WHERE  ename =
         (SELECT manager
          FROM   employee
          WHERE  ename = X.manager))
```

is evaluated by tuple substitution even though is equivalent to a simple join [GACL].

<u>Ingres</u>: <u>Query</u> <u>Language</u>

QUEL, the query language of Ingres, is a tuple calculus language [HeSW] [SWKH]. QUEL uses the concepts of "simple aggregate" and "aggregate function". A simple aggregate corresponds to an aggregate term in the calculus whose

subquery has no free variables. An aggregate function corresponds to an aggregate term in the calculus whose subquery has one or more free variables. Aggregate functions also include an explicit group by clause, called the "by-list".

Example 8. The QUEL version of example 1 is:

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

retrieve (d.dname,
          cnt=count(e.ename by d.dname
                 where e.dept=d.dname))
        where d.division = "marketing"
```

Ingres Access Paths

Aggregate function processing in Ingres proceeds according to the following five main steps [Epst]:

Given an aggregate function "agg_fn(a_expr BY by-list [WHERE qual])",

(1) Create a temporary relation "templ" to hold the aggregate results.

(2) If the aggregate function has a qualification, project the BY-list into "templ".

(3) If the aggregate is multivariable, join and project the qualifying tuples into a relation "temp2".

(4) Perform the aggregation on "temp2", storing the values in "templ" by sorting or hashing.

(5) Link "templ" to the outer query on the BY-list attributes.

Consider these steps applied to the QUEL version of example 4:

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

retrieve (d.dname,
          ave=avg(e.salary by d.dname
                  where e.dept=d.dname))
where 50K < dpbudget and d.dpbudget < 60K
```

To process this query, the Ingres query processor will: (1) create a temporary relation "templ"; (2) project department(dname) into templ; (3) join department to employee and store the result in temp2; (4) hash or use sorting to get average salaries from temp2 into templ; and (5) join templ to departments having budgets between $50K and $60K.

Under no circumstances would this access path be cheaper than an index scan or even a merging scan.

We verified by experiment that Ingres does not consider selectivities in outer queries. On the standard Ingres demo database, we ran the following two queries:

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

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

retrieve (d.name, d.manager,
          cnt=count(e.name by d.manager
            where d.manager=e.manager))
    where d.floor = 0
```

There is only one department where floor = 0, but the second query actually had higher CPU and I/O costs than the first query.

5. Summary and Conclusions

The Abe statistical query facility includes a simple but powerful language and access procedures for efficient execution of aggregation queries. The Abe query language consists of a friendly interface to conjunctive relational calculus. An Abe query looks like a tree of QBE tables.

Grouping operators are not used in Abe. Instead, free variable subqueries are the sole means for expressing complicated aggregations. Although the use of free variables in a query language has been criticized [ChBo], we believe that with Abe's simple structure, free variables will be easy for users to understand.

Access paths in Abe compute aggregates while simultaneously joining the subquery with the outer query. This "aggregate join" can be evaluated by using either an index scan or a merging scan. Which kind of scan is best depends on what fraction of the subquery partitions will be needed in computing the query.

We believe the features of Abe described in this paper indicate a promising approach to processing statistical queries. The Abe interface has been implemented, and the implementation of the access paths described in this paper is underway.

References

[ABCE] Astrahan M.M., Blasgen M.W., Chamberlin D.D., Eswaran K.P., Gray J.N., Griffiths P.P., King W.F., Lorie R.A., McJones P.R., Mehl J.W., Putzolu G.R., Traiger I.L., Wade B.W. and Watson V. "System R: Relational Approach to Database Management" ACM-TODS 1, 2, pp.97-137 (1976)

[AhSu] Aho A.V. Sagiv Y. and Ullman J.D. "Equivalences among Relational Expressions" SIAM J. Comptng. 8, 2, 218-246 (May 1979)

[CAEG] Chamberlin D.D., Astrahan M.M., Eswaran K.P., Griffiths P.P., Lorie R.A., Mehl J.W., Reisner P., and Wade B.W. "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control" IBM Journal of Research and Development, 1, pp.560-575, and in IBM Research Report RJ 1798

[ChBo] Chamberlin D.D., Boyce R.F. "SEQUEL: A Structured English Query Language" ACM SIGMOD Workshop on Data Description, Access and Control, 1974

[ChMe] Chandra A.K. and Merlin P.M. "Optimal Implementation of Conjunctive Queries in Relational Databases", Proc. 9-th Annual Symp. on Theory of Computing, May, 1976, 77-90

[Epst] Epstein R. "Techniques for Processing of Aggregates in Relational Database Systems" Electronics Research Laboratory UCB/ERL M79/8, University of Calif. Berkeley

[GACL] Griffiths P., Astrahan M.M., Chamberlin D.D., Lorie R.A. and Price T.G. "Access Path Selection in a Relational Database Mangement System", ACM-SIGMOD 1979 International Conference on Management of Data

[HeSW] Held G.D., Stonebraker M.R. and Wong E. "INGRES -- A Relational Data Base System", NCC 1975

[Klug] Klug A. "Equivalence of relational algebra and relational calculus query languages having aggregate functions", to appear, JACM; also Univ. of Wisc. Tech. Rep. #386

[Klug81] Klug A. "Abe -- A Query Language for Constructing Aggregates-by-Example" Workshop on Statistical Database Management, Menlo Park, Calif., December 1981

[LoNi] Lorie R.A. and Nilsson J.F. "An Access Specification Language for a Relational Data Base System" IBM J. Res. Develop., 3, pp.286-298 (1979)

[SWKH] Stonebraker M., Wong E., Kreps P. and Held G. "The

Design and Implementation of INGRES", ACM-TODS 1, pp.189-222 (1976)

[Ullm] Ullman J.D. "Principles of Database Systems", Computer Science Press 1980

[WoYo] Wong E. and Youssefi K. "Decomposition -- A Strategy for Query Processing" ACM Trans. Database Sys., 1, pp.223-241 (1976)

[Zloo] Zloof M.M. "Query-by-Example; a data base language" IBM Sys. J. No. 4, 1977, pp.324-343