ABE

A QUERY LANGUAGE FOR CONSTRUCTING AGGREGATES-BY-EXAMPLE

by

Anthony Klug

Computer Sciences Technical Report #475

May 1982

A Query Language for Constructing Aggregates-by-example

Anthony Klug

University of Wisconsin

1. Introduction

Many queries written today involve taking counts, sums, averages, etc. We call these statistical queries. Although this class of queries is very important, up to now it has not been given adequate attention by query language designers. Syntax and semantics for statistical queries are often confusing and not well defined. Expressive power for writing complicated statistical queries is generally poor. What is needed is a query language that not only has clear and simple syntax and semantics, but is also very powerful for constructing statistical queries. In this paper, we present such a language -- Abe.

The following example is indicative of the kind of query we have in mind. It refers to a database containing information about company divisions, departments within divisions, items sold by departments, and employees.

> Consider divisions having budgets exceeding $500K. Let DV be one of these. Consider a department in DV to be "old" if it has more high seniority employees than low seniority employees. For each old department within DV, add up the salaries of the high seniority employees. Compute the average of these sums over all such departments within DV. Do this same computation for all divisions with budgets over $500K.

Clearly, this is a complicated query. No existing high-level query language we are aware of could handle this query without resort to temporary files and/or programming language coding. Yet, using Abe, this query can be

formulated easily.

Abe is a relational query language. It is similar in a number of ways with Zloof's Query-by-Example (QBE) [Zloo]. Although Abe is more powerful than QBE, its definition is actually simpler. Abe is also more powerful than QUEL, the query language of Ingres [SWKH], and SQL, the query language of System-R [ABCE] (at least as far as statistical queries are concerned). It achieves this power, not through a maze of ideas such as grouping, partitions, partition selectors, and duplicates, but through the simple idea of a subquery with parameters.

In the next section we outline the necessary underlying data model definitions. Then in section 3, we give in just a few paragraphs, the complete semantics of Abe. Section 4 contains a number of examples of Abe queries. Finally, in section 5 we discuss how the user and the underlying database system interface with Abe.

2. Data Model Definitions

In this section, we outline basic relational data model and aggregate function concepts.

A relation is a finite two-dimensional table with columns labeled by distinct attributes. Each attribute A_i has a domain $D(A_i)$, and entries in a column labeled by A_i must be elements of the domain $D(A_i)$. The relation scheme

(or format) for a relation R with attributes $A_1,\ldots,A_n$ is $R(A_1,\ldots,A_n)$. The order of the rows in a relation has no significance, and we can view a relation $R(A_1,\ldots,A_n)$ as a finite subset of the cross product $D(A_1)\times\ldots\times D(A_n)$. A database is a finite set of relations. The relational schema for a database is the set of relation schemes for the relations it contains. The following relational schema will be used in our examples:

```
division (dvname, manager, budget)
department (dname, division, manager)
employee (ename, salary, dept, seniority, recruiter)
item (iname, color, price)
sells (dname, iname)
```

An aggregate function, is a function which takes a set of tuples as input, and produces a single value (usually numeric) as output. We do not associate, and we do not need to associate, with aggregate functions the concept of partition. In Abe, there is no concept grouping such as is found in QUEL and SQL. Abe also does not use the concept of duplicate tuple[1].

The aggregate functions we will consider in this paper are: count, min (minimum), max (maximum), sum, and ave

[1] The problem is not that "duplicate tuple" cannot be formally defined. The concept of bag [RoLe], a "set" with duplicates, does this. One problem is that bags are usually added as an afterthought to the relation model. Another problem is that the elegance of the relational model is severely degraded by adding bags.

(average). All of these except count are given a column on which to operate in addition to the actual input set of tuples. Thus to sum a set of salaries, we do not say "sum the bag of salaries", we say "sum the salary column of the employee table."

In some cases, the value an aggregate function should take on the empty set is clear. In particular[2]:

$$\begin{aligned} count(\emptyset) &= \emptyset \\ sum(\emptyset) &= \emptyset \end{aligned}$$

3. Specification of Abe

In this section, we specify the formation rules, and the semantics of Abe. The presentation will be concise, and the reader may wish to review this section as the examples are presented in section 4.

(1) A query (or subquery) consists of an output list, a condition box, and zero, one, or more relation tables. Queries and subqueries are given names; the first one the user sees is always called "top level."

(2) An output list is a table containing one row and several possibly labeled columns. Items specifying

---

[2] If A and B are disjoint sets, $sum(A \cup B) = sum(A) + sum(B)$. We can also argue that $min(\emptyset) = +\infty$, and $max(\emptyset) = -\infty$, since $A \subseteq B$ implies $min(A) \geq min(B)$ and $max(A) \leq max(B)$.

query output are entered into the output list.

(3) A condition box is a table containing one (wide) column and several rows. Boolean conditions on items in the relation tables are placed in the condition box.

(4) A relation table for a relation R has a column for each attribute of R and several rows.

(5) Relation tables and the output list are filled with elements. The condition box is filled with two elements connected by one of the operators =, $\neq$, <, $\leq$, >, $\geq$.

(6) There are several kinds of elements: constants, variables, fixed variables, and aggregates.

(7) An aggregate consists an aggregate function (count, min, max, sum, or ave), an output list label (unless the function is count), and the name of a subquery.

(8) Variables and fixed variables are arbitrary identifiers. Fixed variables can appear only in subqueries, and they correspond to some (regular) variable in some higher-level query. Fixed variables are similar to procedure parameters in a programming language. Constants are numbers or strings.

(9) Normally, variables, fixed variables, and constants (but not aggregates) appear in relation tables, and

variables and aggregates appear in output lists.

(1Ø) For conciseness, a relation table may contain a comparison operator (as in (3)) followed by an element, e.g., "> 1Ø". This is an abbreviation for a unique variable x at that point along with the condition "x > 1Ø" in the condition box.

The output of a query is a set of tuples. This output is determined by the following rules: The entries in the relation tables act as patterns, and the query is evaluated by looking in the database for matches to the patterns. Empty rows in relation tables are not considered. A variable can match anything. A constant matches only itself. When a subquery is evaluated, any fixed variables occurring in it have been previously bound to particular values, and in this evaluation, a fixed variable matches only the value to which it has been bound. A row is matched when there is a match for all of its (nonempty) entries in the corresponding relation in the database. A variable may appear in several places, but each occurrence must be matched to the same value.

When a match is found for all rows in the relation tables, the conditions in the condition box are examined. If a condition contains a variable, a fixed variable, or a constant, the corresponding value is used in evaluating the condition. If an aggregate is present, its subquery is evaluated (with all of its fixed variables bound to the

current values of the variables), and the aggregate function is applied to the resulting set of tuples.

If all conditions are satisfied, the output list is used to generate a row of output. If a variable appears in the output list, the value it matched is used. If an aggregate appears, it is evaluated as above, and its value is used. The resulting tuple is added to the output if it is not already present.

After a tuple has been output, more matches are searched for. There is no particular order in which this is done. When all matches have been found, the evaluation is complete.

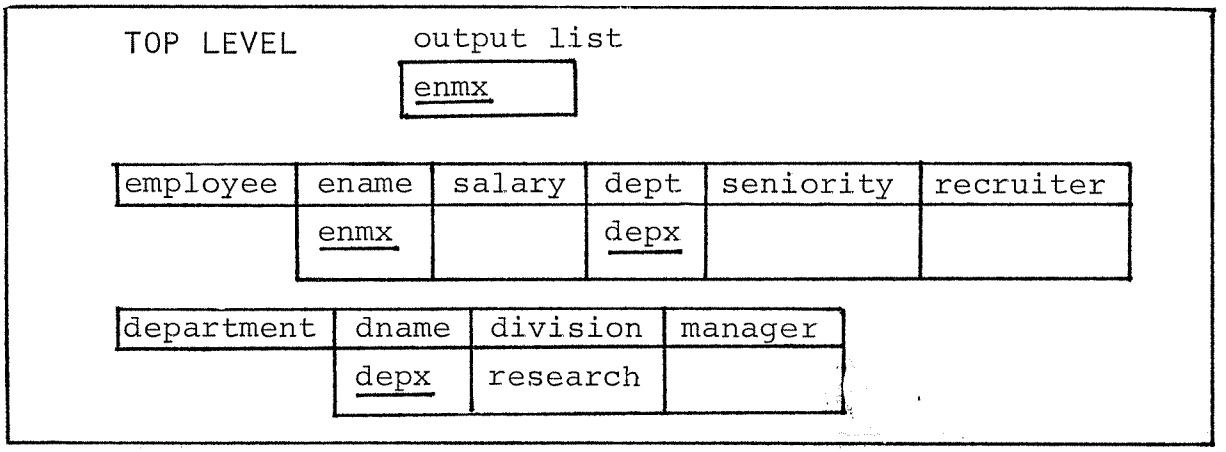
Note that although the user view of how an Abe query is evaluated involves repeated evaluation of subqueries (akin to tuple substitution of QUEL [WoYo]), repeated evaluation does not happen in reality. Section 5 describes the translation of Abe to relational algebra.

4. Examples of Abe Queries

In this section we give Abe formulations of progressively more complicated statistical queries. Note that the structure of Abe naturally induces users to formulate queries in a top-down fashion. The commands used to build these queries are discussed in the next section. To distinguish between variables, fixed variables, constants, and subquery names, we use the following key:

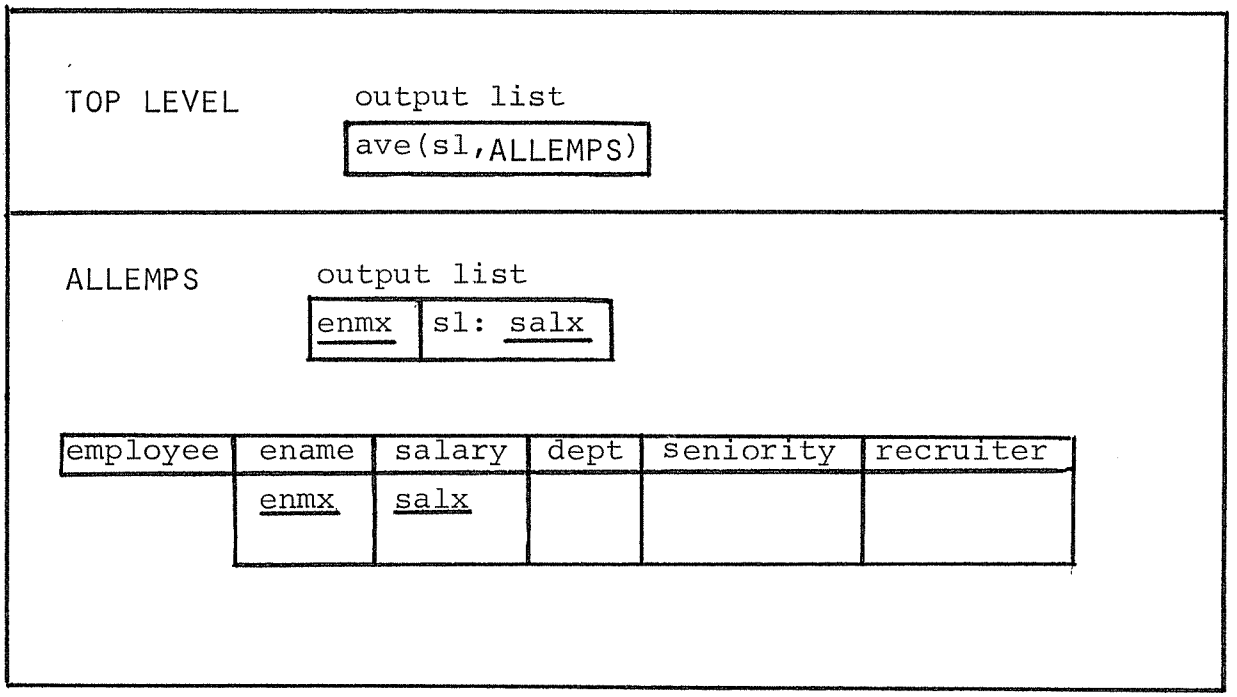
- variables are underlined
- fixed variables are underlined and in italics
- constants have no typographic enhancements
- subqueries are typed in block letters

Example 1. Get names of employee working in departments of the research division.



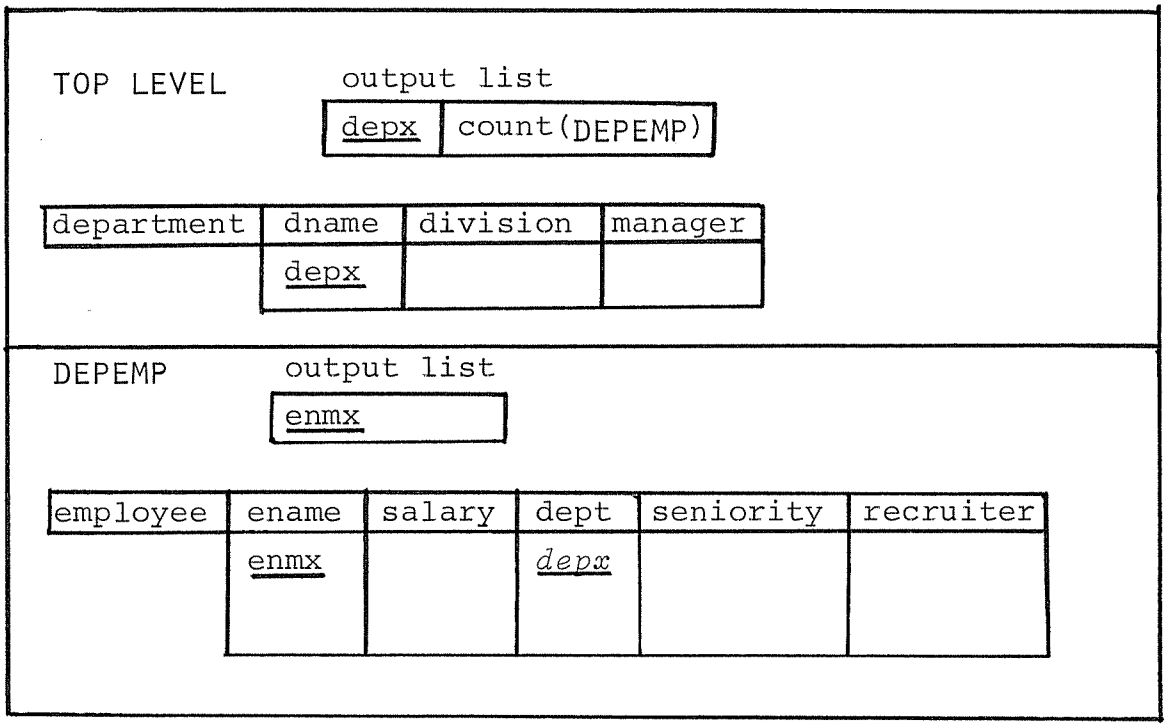
This simple query does not have any aggregates. It illustrates how variables are used to correlate (join) information in several tables.

Example 2. Print the average salary of all employees.



This query corresponds to a "simple aggregate" of QUEL, whereas the next example would correspond to a QUEL "aggregate function" (non-simple aggregate). In Abe, there is no need for such distinctions.

Example 3. List each department name and the number of its employees (a count of employees by department).



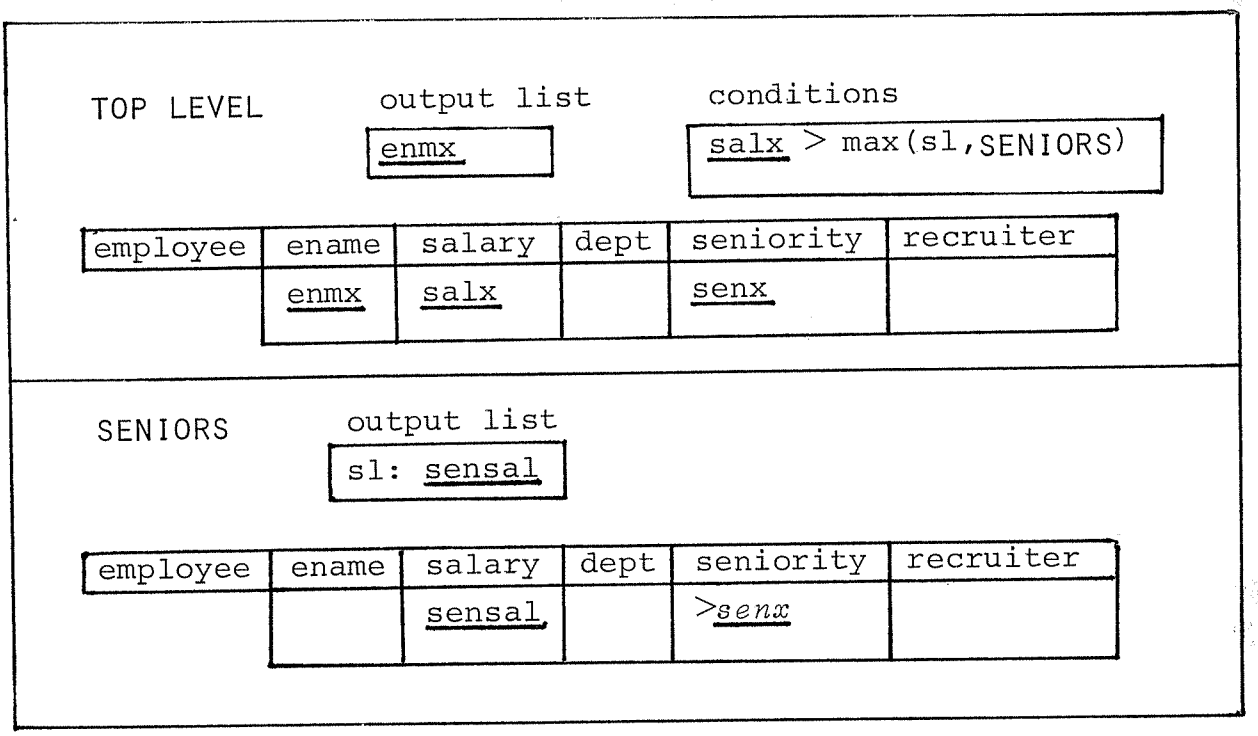
This query illustrates an important difference between the concept of aggregates as used in Abe and those in other query languages. Using SQL, most users would write the following:

```
SELECT DEPT, AVG(SALARY)
FROM EMPLOYEE
GROUP BY DEPT
```

It is possible that some departments have no employees (e.g., a ski shop during the summer), and these departments will not appear in the above SQL query. In fact, there is no natural way to get such departments to appear in the

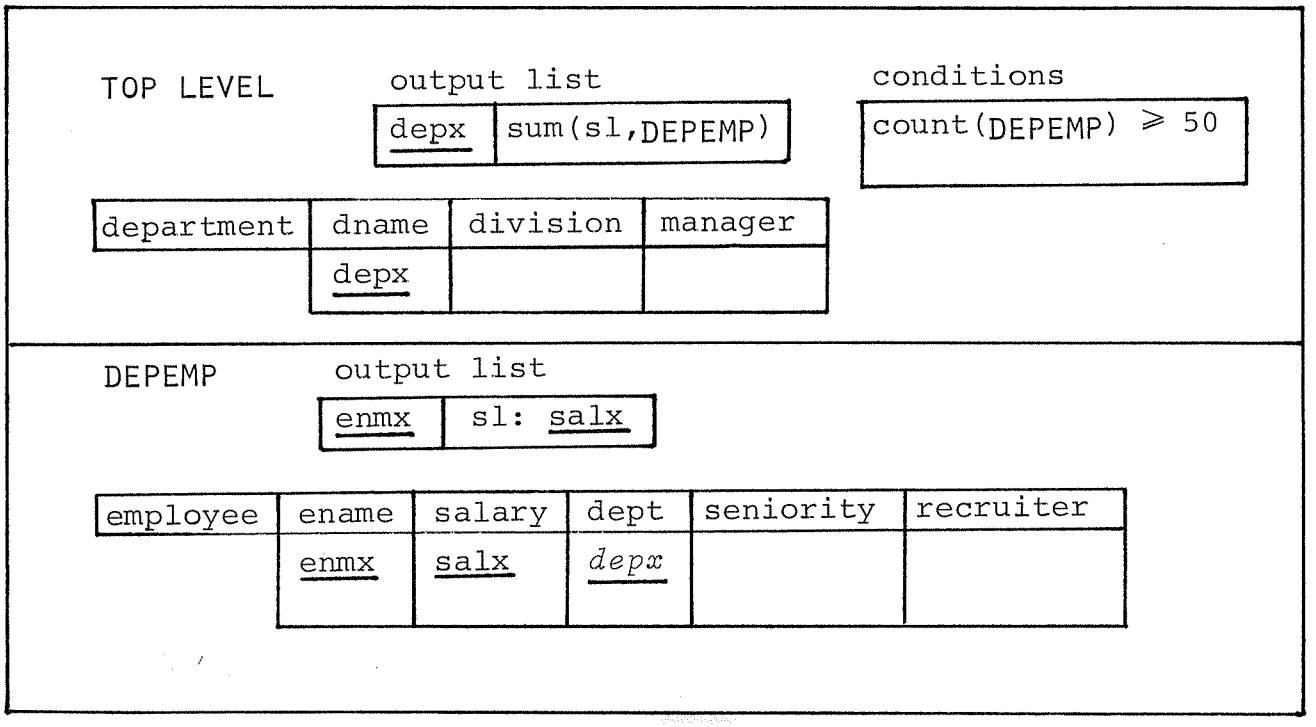
result of a query expressed by partitioning-type aggregates. In Abe, this example is simple to formulate.

Example 4. List names of employees who earn more than the maximum salary of employees with greater seniorities.



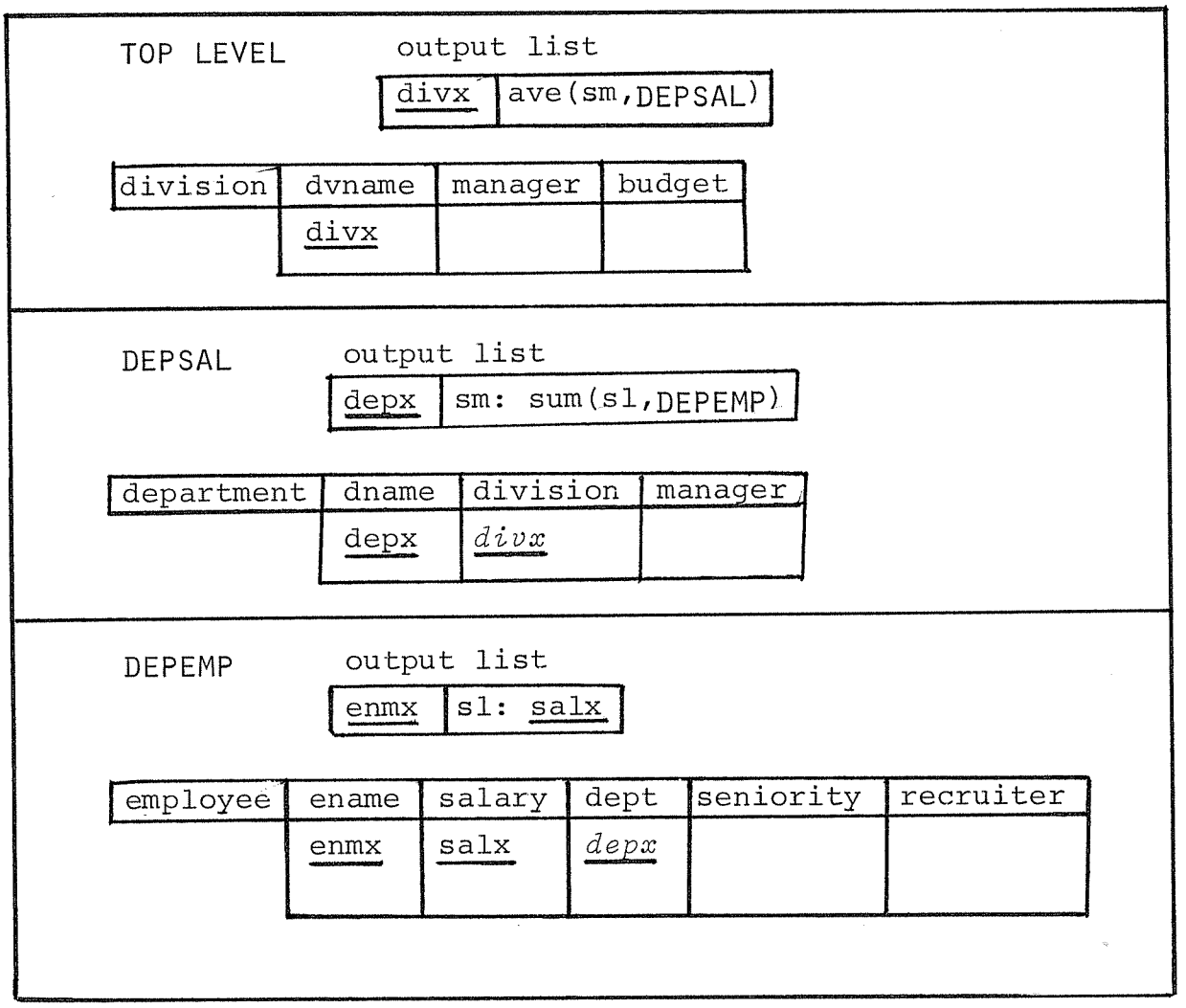
Not only does Abe avoid introducing the concept of "partition", it also can express queries which would need a generalization of the partition concept (or an artificial inequality join). In this case, for each seniority value, we need to form the set of employees having a greater seniority. This is not a partition but rather a nested grouping of employees.

Example 5. For each department in which at least 50 employees work, print the sum of the salaries of the employees in that department.



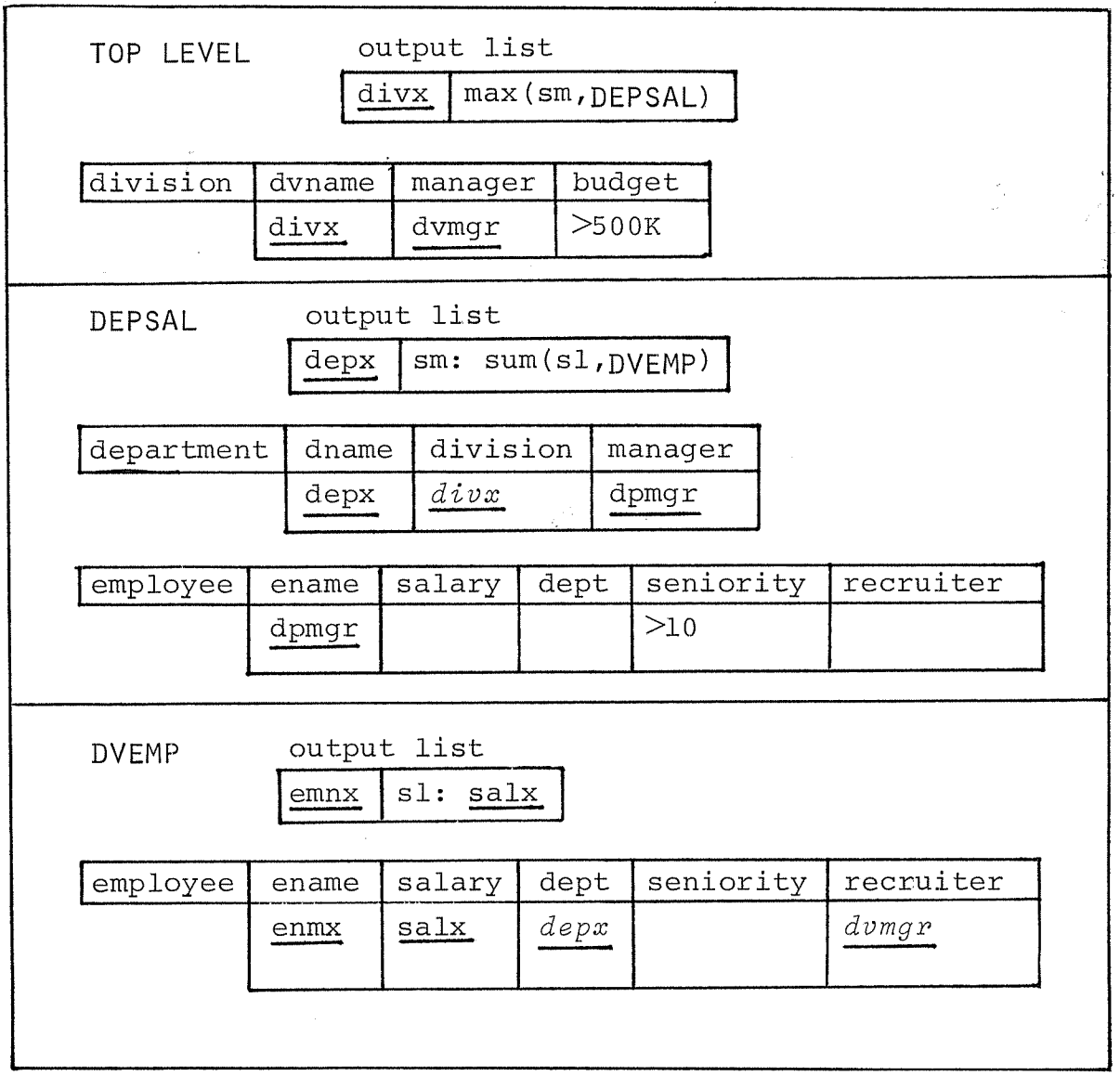
To formulate this query in SQL, both a GROUP BY and a HAVING clause would be needed.

Example 6. For every division DV, for every department DP in DV, compute the sum S of salaries of employees in DP; then average all the S's over all departments in DV. Print the name of DV and this average.



In previous query languages, this query could not even be expressed without the use of temporary relations or the writing of code.

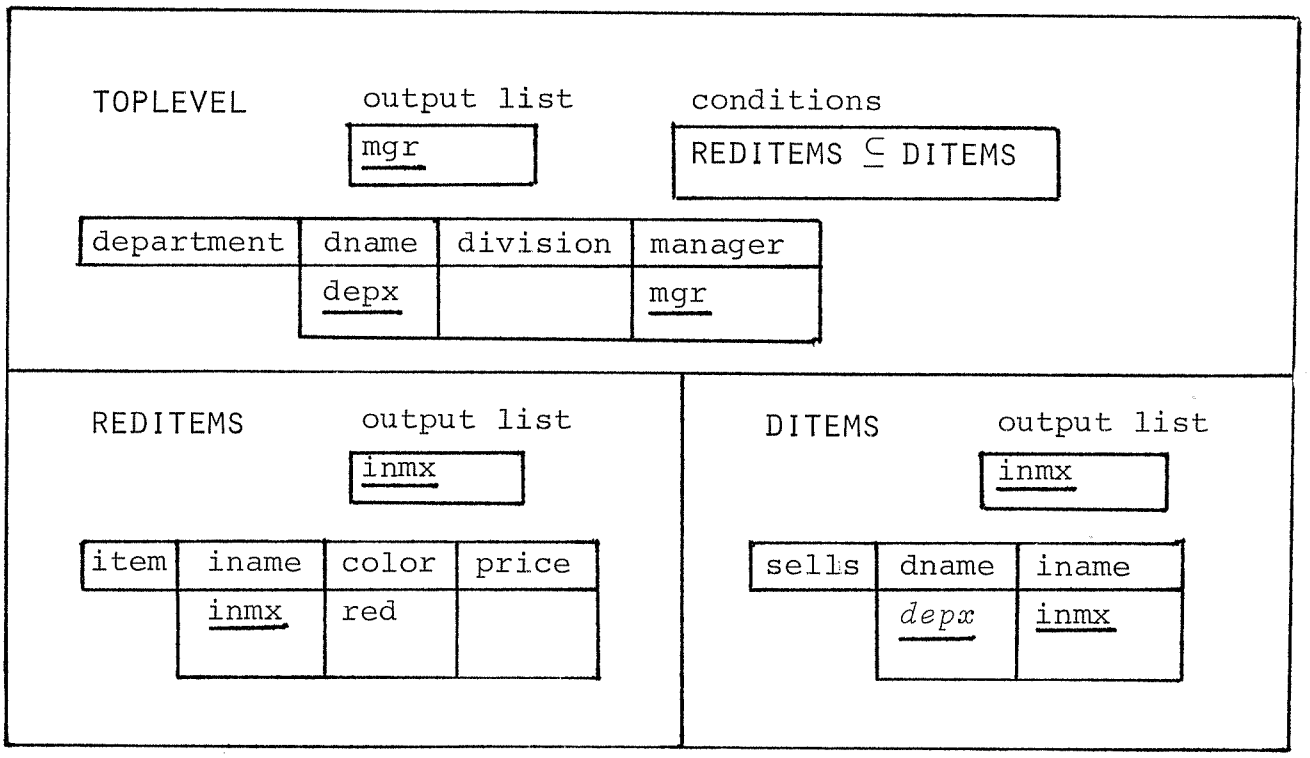
Example 7. Let DV be a division with a budget greater than $500K. For each department DP managed by an employee with a seniority greater than 10, compute the sum S of salaries of employees in DP who were hired by the manager of DV; then print the name of DV and the maximum of these sums.



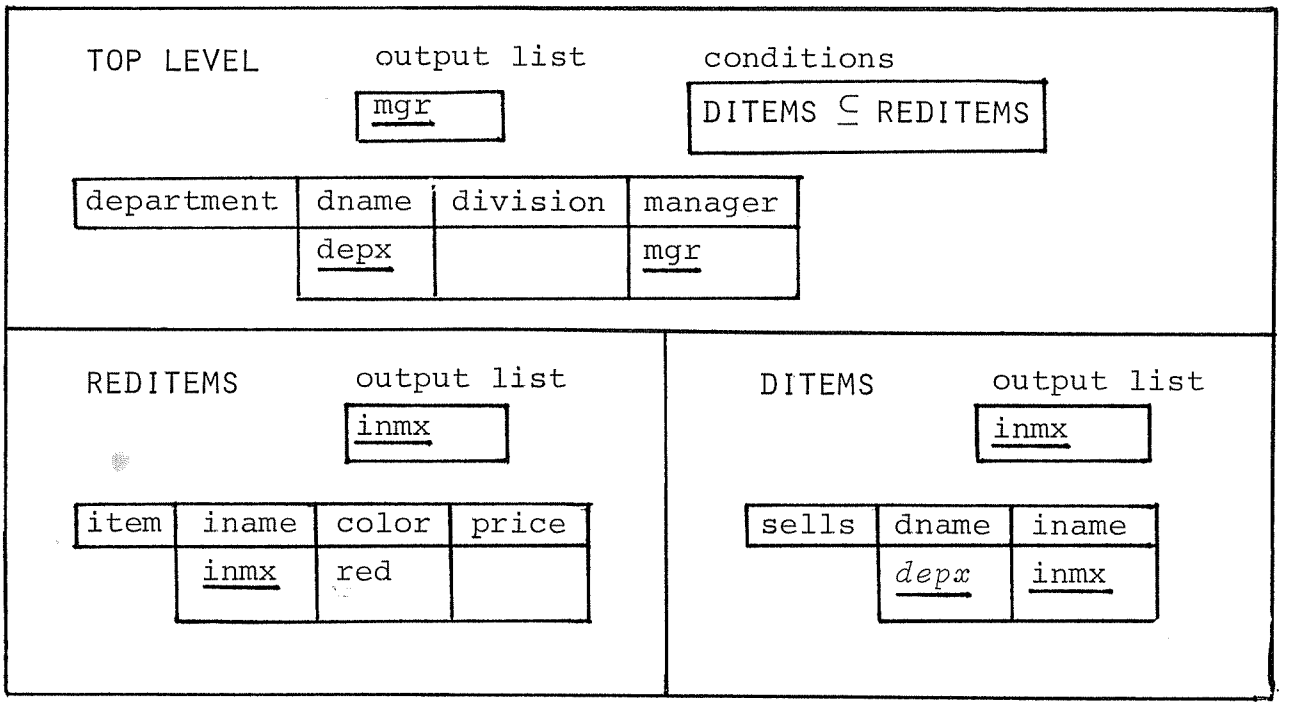
Increasing the complexity of the query by adding of a number of qualifications on "partitions" and "members of partitions" does not materially increase the complexity of the Abe query. Note that the manager of DV is not used in an immediate subquery of the top-level query, but in a second level subquery.

The next three examples do not deal with aggregate functions, but they illustrate a simple but powerful extension of the language defined so far. This extension allows queries to be expressed which involve "all", "only", or "no" qualifiers. In relational algebra, these are usually expressed using various combinations of division, set difference, and projection. To express these queries, Abe allows conditions in the condition box to be set-comparisons (set equality, set containment) between subqueries. The semantics are simple: If a set-condition appears in the condition box, evaluate both subqueries as for aggregates, and see if the specified relationship (equality or containment) holds.

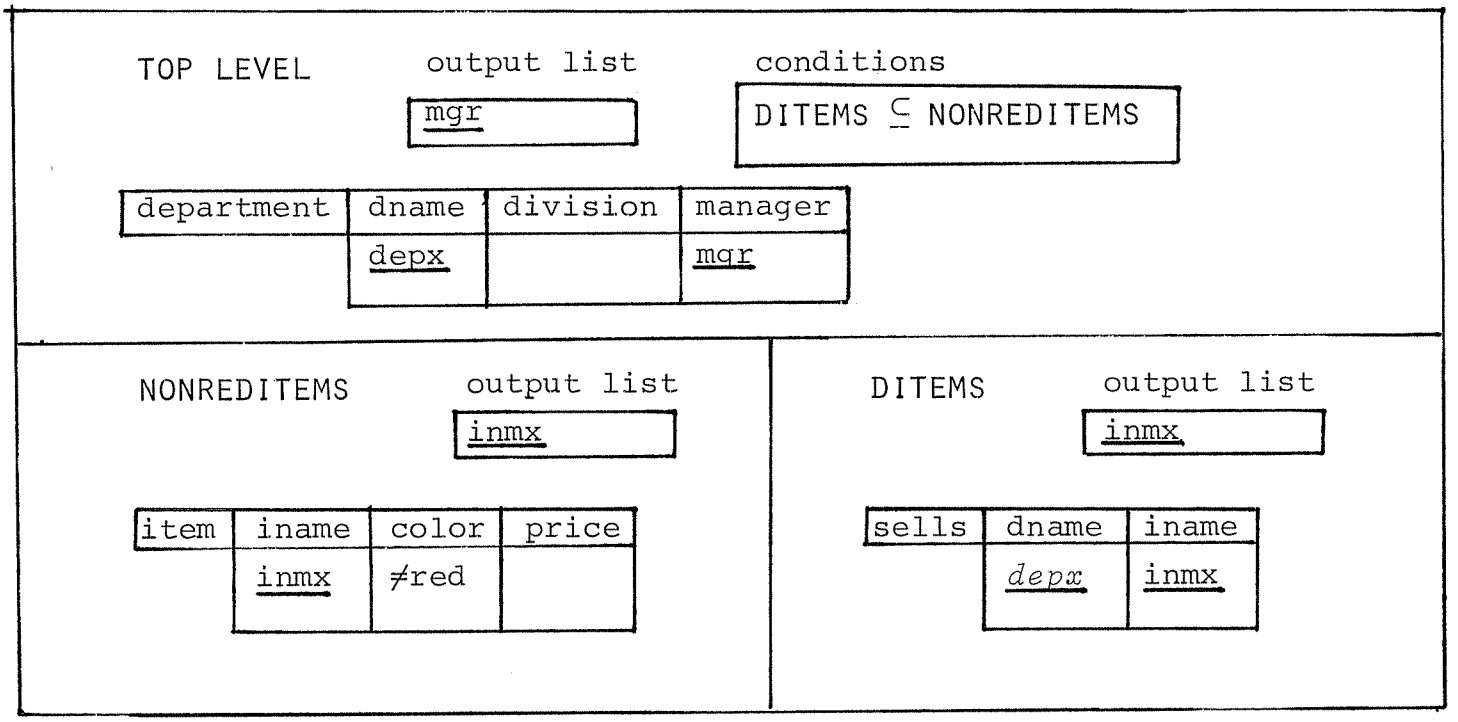
Example 8. List managers of departments which sell all red items.



Example 9. List managers of departments selling only red items.



Example 1Ø. List managers of departments selling no red items.



In QBE, this query would be formulated using a negation operator applied to an entire row of a relation table. This kind of negation is not needed in Abe.

5. User and System Interfaces

The current implementation of Abe runs from a number of intelligent terminals. The terminals are required to have cursor addressability and several display enhancements. Each type of element is assigned a different display type using color, type font, etc. At any time, the top level query or some subquery is the current query, and its tables are displayed on the screen. The cursor position determines

where the next element is entered, and is controlled by single key-stroke commands similar to full-screen editors. Other single key-stroke commands cause the various elements to be entered. Movement down the "query tree" is achieved by moving the cursor to a subquery name and using the "open" command. Movement up the tree is achieved with the "exit" command.

Abe is interfaced to System DM*, a DBMS supporting multiple views and multiple data models [Klug]. The interface language is relational algebra. The relational algebra operator for aggregate functions uses partitioning[3], and we denote it f<X,e> (partition input e on X and apply f to each partition). The basic idea in translating an Abe query into a relational algebra expression is as follows: Suppose a query Q has a subquery Q' which is translated to a relational algebra expression e', and suppose the fixed variables occurring in Q' are $X = X_1,\ldots,X_k$. If Q contains an aggregate f(h,Q') in its condition box, and if the relation tables of Q correspond to a relational algebra expression e (which is just a join), then the expression for Q itself consists of e joined with $e'\langle X,f_h\rangle$ on X. To illustrate, Example 3 of the previous section will be translated to the following join (the necessary projection is omitted):

---

[3] Note that we have only claimed that the idea of partitioning is not user-friendly. It is still an important implementation device.

department [dname = dept] (count<dept, employee>)

Future enhancements to Abe will include query optimization. Since no temporary relations or programming code is needed to express a complicated statistical query, the structure of the entire query is available at once to the system. This makes global optimization in the style of [AhSU] and [ChMe] possible.

6. 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. "Efficient Optimization of a Class of Relational Expressions", ACM TODS, 4, 435-454 (1979)

[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

[Klug] Klug A. "Multiple View, Multiple Data Model Support in System DM*" University of Wisconsin Technical Report.

[RoLe] Robinson L. and Levitt K.N. "Proof Techniques for Hierarchically Structured Programs" CACM 20, pp. 271-283

[SWKH] Stonebraker M., Wong E., Kreps P. and Held G. "The Design and Implementation of INGRES", ACM-TODS 1, #3, 1976, pp.189-222

[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