ELSEVIER

Information & Management 27 (1994) 1–15

INFORMATION & MANAGEMENT

Research

# DFQL: Dataflow query language for relational databases

Gard J. Clark, C. Thomas Wu *

*Naval Postgraduate School, Department of Computer Science, Code CS, Monterey, CA 93943, USA*

**Abstract**

This paper proposes a new query language, DFQL, which has been designed to mitigate SQL's ease-of-use problems. DFQL provides a graphical interface based on the dataflow paradigm in order to allow a user to construct queries easily and incrementally for a relational database. DFQL is relationally complete, maintains relational operational closure, and is designed to be easily extensible by the end user. A prototype DFQL system has been implemented.

*Key words:* Relational algebra; Query languages; SQL; Visual interface; Dataflow diagram; DBMS

## 1. Introduction

The relational model for database management was introduced in 1970 by E.F. Codd [6]. Compared to other actually implemented models, namely hierarchical and network, the relational one has the simplest and most uniform data structures and is founded in mathematics. These

**Gard J. Clark** is a Lieutenant on active duty in the Navy. He is currently serving as the Strategic Weapons Officer on USS TENNESSEE (SSBN-734)(GOLD). His research interests include nontraditional database query languages and user interfaces. Clark earned a BS degree from the United States Naval Academy in 1985 and an MS degree in Computer Science from the Naval Postgraduate School in 1991 where he received the Grace Murray Hopper Award for Computer Science. Clark has also conducted research in databases and rapid prototyping at Los Alamos National Laboratory in the ADP Division.

**C. Thomas Wu** is an Associate Professor of Computer Science at the Naval Postgraduate School in Monterey, California. Before joining the Naval Postgraduate School in 1985, he was an Assistant Professor of Computer Science at the Northwestern University in Evanston, Illinois. He received his Ph.D from University of California, San Diego. His current interests are database query languages, database design, object-oriented programming, and user interface.

* Corresponding author.


*SSDI* 0378-7206(94)E0053-7

features of the relational model make it a tool for many database implementations. However, its standard query language Structured Query Language (SQL) [5] has some serious problems. Several of these are discussed in a two part article "Fatal Flaws in SQL" [8,9]. Our research primarily addresses "The Psychological Mix-up" or human factors aspect of the language [10]. We extend this from ease-of-use issues, to such problems as the expression of universal quantification in SQL [20]. In general, we believe that a new database query language is required and propose a graphical/visual interface to the relational model based on a dataflow paradigm. This DataFlow Query Language (DFQL) retains all of the power of current query languages and is equipped with an easy to use facility for extending the language. DFQL includes operators (e.g. aggregate functions) beyond the minimum, providing query facilities beyond the benchmark of first-order predicate logic. DFQL meets the following goals for a visual database interface [3,25]:

- A fully graphical environment
- Sufficient expressive power and functionality
- Ease-of-use in learning, remembering, writing and reading
- Consistency, predictability, and naturalness
- Simplicity and conciseness
- Clarity of definition and lack of ambiguity
- Ability to modify existing queries
- High probability that users write error-free queries
- Operator extensibility

The approaches taken in DFQL to attain the above goals include:

- Use of relational algebra especially the operational closure
- Elimination of range variables
- Elimination of nesting

There have been several research efforts directed towards the design of visual database querying systems. Notable ones are [1,2,4,11,15, 18,20,21,27,30]. However, none of these efforts utilize a dataflow approach. The dataflow paradigm provides several advantages. Perhaps the most important is the ability of the user to treat relations as abstract entities manipulated by relational operators. This allows the user to compose queries strictly in the realm of relational algebra, rather than having to know the details of the operations.

DFQL has been implemented on a Macintosh II/ci running version 6.0.7 of the Macintosh Operating System using the Prograph language of The Gunakara Sun Systems (TGSS) of Halifax, Nova Scotia, Canada. The backend DBMS is ORACLE for Macintosh version 1.2. DFQL has also been operated on a remote ORACLE DBMS (version 6.0) running on a Digital Equipment Corp. Micro-VAX via a DECNET Ethernet connection. All features discussed in this paper have been implemented. Prograph is a pictorial object-oriented programming environment [28]. Its visual dataflow structure is very similar to the approach taken for DFQL. The fact that Prograph is object-oriented allowed the use of many powerful features of the object-oriented paradigm which also greatly improved the modularity and maintainability of the resultant code.

## 2. Deficiencies of SQL

Query languages for the relational database model can be divided into three basic types: relational calculus, relational algebra, and a combination of these. In relational calculus, the user provides a predicate calculus expression that defines the characteristics of the tuples to be retrieved. Tuple variables are used to make the logical connections between instances of relations being joined. In relational algebra the user specifies a sequence of relational operations to be performed on the relations. Both the relational algebra and the relational calculus have equal expressive power [7]. SQL is a mixture of both relational algebra and calculus with some other additional ideas (such as nesting to provide a block structure). However, SQL tends more toward a calculus based approach, because it is mainly a declarative language. The user specifies the required output in one statement. Date comments [13], "When the language [SQL] was first

designed, it was specifically intended to differ from the relational calculus (and, I believe, from the relational algebra). ... As time went by, however, it turned out that certain algebraic and calculus features were necessary after all, and the language grew to accommodate them." The result is that SQL is a strict implementation of neither relational algebra nor calculus.

### 2.1. Difficulty in comprehending SQL

SQL is primarily a declarative query language; its use is straightforward for simple queries, but for more complex queries the logical expression can become complicated. This problem is exacerbated when the complex query involves universal quantification. Another problem is that it may not always present the clearest representation of the query. The logical, declarative method of expression may be more compact, but if humans think of complex problems in a sequential fashion the declaration may not be as easy to formulate or interpret. Indeed Codd uses the algebraic approach to introduce operators because "upon first encounter, that approach appears easier to understand." The lack of procedural nature of SQL is compensated by embedding it into a procedural, 3rd generation host programming language. This allows the user to use the host language to accomplish operations that are difficult or impossible to code in the query language. Human factors studies show that, for complex queries, users perform better when using a procedurally oriented rather than a declarative language such as SQL [26]. Ease-of-use issues for database query languages have been of concern for quite some time (see Schneiderman [23]).

### 2.2. Difficulty in expressing universal quantification

The idea of universal quantification is expressed in English by the phrase "for all." This idea is often required to formulate database queries but is supported only indirectly in SQL. One of the problems in using universal (or existential – "there exists") quantification is that the meaning of these operations is not completely intuitive, especially to persons who are not experienced in predicate logic. The ideas represented by these operators are difficult to use correctly even when the user has experience [19]. This is compounded because SQL's lack of a specific "for all" operator forces one to use "negative logic" (the existential quantifier NOT EXISTS to achieve the result of universal quantification). The following example illustrates the difficulty of expressing the idea of universal quantification. Consider a database with the following relations (key attributes are underlined):

COURSE (CNO, TITLE, INO)
ENROLL (CNO, SNO, GRADE, TESTSCORE)
INSTRUCTOR (INO, INAME, PAY)

The desired query is: "list the names of all the instructors who gave all A's in at least one of the courses they taught." In SQL the query could be expressed as:

```
(1) SELECT DISTINCT INAME
(2) FROM INSTRUCTOR, COURSE, EN-
    ROLL E1
(3) WHERE INSTRUCTOR.INO =
    COURSE.INO
(4)   AND COURSE.CNO = E1.CNO
(5)   AND NOT EXISTS
(6)     (SELECT *
(7)     FROM ENROLL E2
(8)     WHERE E2.CNO = E1.CNO
(9)       AND E2.GRADE ! = 'A');
```

A direct English translation of the above SQL query is: "Retrieve the names of instructors (line 1) who taught courses (line 3) which had students enrolled (line 4) in which there was at least one of these courses in which there was not any student who received a grade that was not an 'A' (lines 5–9)." This translation describes only the basic semantics of the SQL statement. In order to determine how it will function, a knowledge of the differences between relation names and range variables and their scoping rules is also required. This example query would not be possible without the correct use of range variables; the linkage between the "inner" and "outer" SELECT statements depends entirely on understanding and

correctly employing range variables to represent the ENROLL relation.

### 2.3. Lack of orthogonality

"Orthogonality in a programming language means that there is a relatively small set of primitives that can be combined in a relatively small number of ways to build the control and data structures of the language." [22] SQL does not present a simple, clean, and consistent structure to the users; there are numerous examples of arbitrary restrictions, exceptions, and special rules. An example of an unorthogonal construct in SQL is allowing only a single DISTINCT keyword in a SELECT statement even if the SELECT statement contains other nested SELECT's. Lack of orthogonality increases the number of special rules to be memorized by the user, decreases its readability, and in general decreases the usability.

### 2.4. Unnecessary complexity due to nesting construct

SQL permits a nesting structure of the form

SELECT ⟨attribute list⟩
FROM ⟨relation list⟩
WHERE attribute IN (SELECT ...)

This format allows for a block structure construct. In fact, it is from this construction that the term "Structured" is derived. The original purpose of this nesting construct was to allow the specification of certain types of queries. However, with the introduction of some relational algebra and calculus operations into SQL the need for the "IN subquery" construct was eliminated. While all queries specified using the nesting construct should be directly translatable into queries using an equi-join, if duplicate rows are allowed in relations (SQL docs), one will obtain different results when performing the two. This non-equivalence is a unnecessary extra burden. There is also a problem in the optimization of nested queries by the DBMS. Although work has been done to demonstrate the translatability of nested queries into their non-nested counterparts [16], most optimizers perform poorly, if at all, in optimizing levels of nesting in SQL queries.

### 2.5. Lack of functional notation

The use of functions in programming languages allows the abstraction of operational detail to whatever level is appropriate. Complex queries that provide an intermediate result for a higher level query could hide this result from the user through the use of functional notation. This concept is universally adopted in all modern programming languages, but not in SQL. DFQL supports user-defined functions (operations).

## 3. Dataflow programming languages

Dataflow diagrams have been used as an aid in systems analysis and design for some time. Dataflow diagrams provide an easy way of describing a network of functional processes that are interacting; the flow involves the transfer of data from one process to another [29]. Dataflow programming languages take these ideas and make them directly executable. In other words, rather than using the dataflow diagram as a tool in designing a program, the diagram becomes the program.

### 3.1. Dataflow diagram

Dataflow diagrams have a strong tie to the depiction of directed graphs: the processes, data stores, and terminators are the nodes and the dataflows are the arcs. Fig. 1 is an example of a simple dataflow diagram that depicts a query processor running on top of a backend DBMS. In this example, data (the user's query) flows from the USER to Process #1. The parsed query is then passed to the DBMS, which is external to the query processor. The external DBMS then returns a result, which is formatted by Process #2 and then passed back to the USER. The sequence is specified only through the availability of data for each process. All processes that have

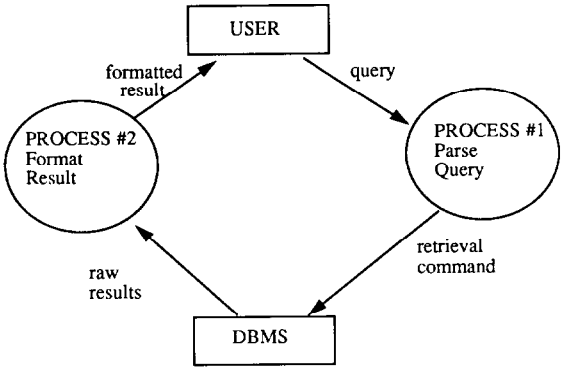



Fig. 1.

data available may theoretically execute simultaneously.

### 3.2. *Visual dataflow programming*

Shu [24] defines a visual programming language as one "which uses some visual representations (in addition to or in place of words and numbers) to accomplish what would otherwise have to be written in a traditional one-dimensional programming language." Dataflow diagrams are inherently visually oriented. A graphical dataflow programming language allows construction of diagrams that are directly translatable into executable code. Davis and Keller discuss the advantages of using a graphical representation for dataflow programs as [14]:

- A dataflow graph conveys the mental image which suggests the data dependencies and flows.
- Dataflow programs are easily composable into larger programs.
- Dataflow programs avoid describing a specific execution order.
- Graphs can be used to attribute a formal meaning to the program.

The two dimensional representation and the use of the value oriented computation method also help to increase the understandability of graphical dataflow programs [17]. Fig. 2 shows a program fragment written in Prograph. This fragment represents the equation $y = mx + b$. The values for m, x, and b are expected as inputs, and the value for y is generated as the output.

## 4. Description of DFQL

DFQL is a visual relational algebra for the manipulation of relational databases. It has sufficient expressive power and functionality to allow the user to express database queries. As such, DFQL is relationally complete and includes an implementation of aggregate functions. A facility is provided for the user to create DFQL operators, thus allowing extensibility. Orthogonality has been stressed in its design. DFQL has been developed as a token model graphical dataflow language. The use of the token model implies that each of the defined operators are designed to operate on a stream of tokens over their lifetime. Our language does not allow iteration or recursion; each operator will execute once over the life of a given query. Queries are defined by the user connecting the desired DFQL operators graphically. The arguments for the operators flow from the bottom or "output node" of the operator to the top or "input node" of the next operator. Operator execution is initiated by the presence of the requisite input data. All fireable operators could be executed simultaneously. In our implementation, only one operator is executed at a time, since the system is being run on a single processor. The structure of DFQL queries directly mimics that of standard dataflow diagrams.

All DFQL operators have the same basic appearance. Each operator has three types of com-

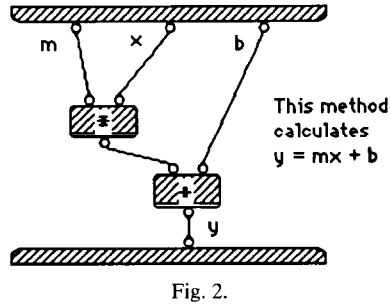



Fig. 2.

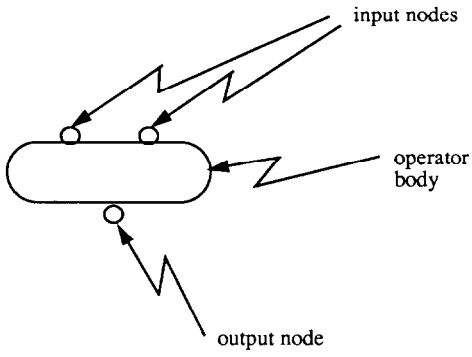



Fig. 3.

ponents: the input nodes, the body, and the output node. A sample operator (with no name) is shown in Fig. 3.

One output node may be connected to other operators's input to pass the intermediate result along. The functional paradigm is fully supported by the DFQL notation. The inputs to each operator, or function, arrive at the input nodes of the operator and the result leaves from the output node. All operators of DFQL implement operational closure: output from each operator is always a relation. Two broad categories of DFQL operators are defined. A *primitive operator* has been defined directly in the native DFQL language. They are basic, non-basic, and display operators. A *user-defined operator* has been constructed by the user from primitives and possibly other previously created user-defined operators.

### 4.1. Basic DFQL Operators

DFQL provides six basic operators derived from the requirements for relational completeness and also the requirement to provide a form of grouping or aggregation. A query language that is relationally complete has the expressive power of first-order predicate calculus; five relational operations must be implemented: selection, projection, union, difference, and cartesian product. These are thus part of the basic set of DFQL operators. Provision is also made for simple aggregation by including groupcnt (group count) as a basic operator. The basic operators

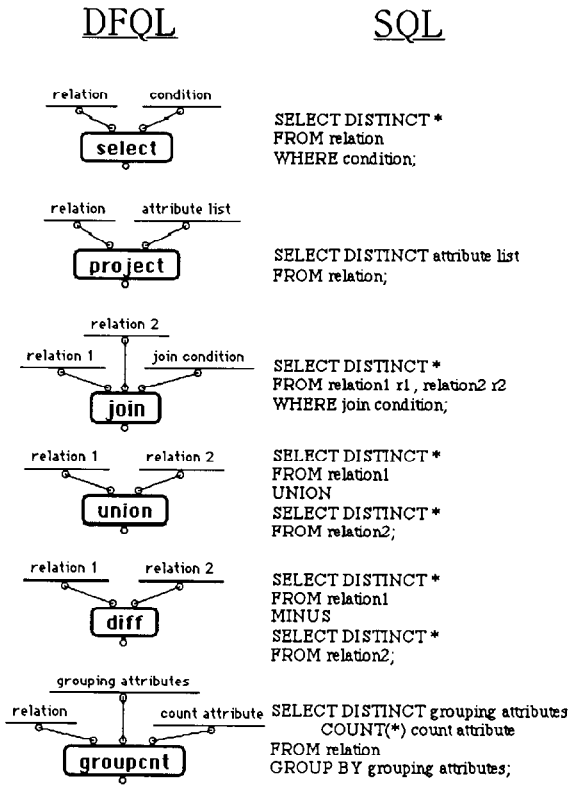



Fig. 4.

and a corresponding translation into SQL are shown in Fig. 4.

A special notation provides textual input to the DFQL operators. Text entered by the user is displayed as an object with the text attached to an output node (see Fig. 5). The text object can be interpreted in two different ways. If the text is the name of a relation, the output can be an instance of that specific relation. If the text represents a condition, a list of attributes, or textual input to another DFQL operator, then it is passed on to that operator as a textual argument.

#### 4.1.1. Select

The relational algebra notation for the select operation is $(\sigma_{\langle \text{condition} \rangle}(\langle \text{relation} \rangle)$; the $\langle$*condi-*

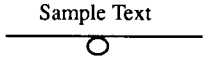
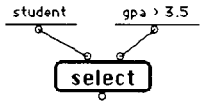



Fig. 5.



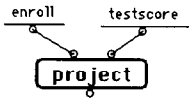

| SNO | SNAME | ADDR | PHONE | GPA |
|---|---|---|---|---|
| S1 | STU #1 | ROOM 1 | 111-1111 | 3.85 |
| S3 | STU #3 | ROOM 3 | 333-3333 | 3.75 |

Fig. 6.

*tion*⟩ specifies which tuples should be retrieved from the given ⟨*relation*⟩. The resulting relation has no duplicate rows (a *proper* relation). An example of the DFQL select operator is shown in Fig. 6. This example retrieves all tuples in the STUDENT relation whose GPA is greater than 3.5. The specification of the ⟨condition⟩ uses the same syntax as SQL. The schema definition and data for the sample relational database used in the following illustrations are given in Appendix A.

*4.1.2. Project*

The relational algebra notation for the project operation is $\pi_{\langle \text{attribute list} \rangle}(\langle \text{relation} \rangle)$; the ⟨*attribute list*⟩ specifies attributes to be retrieved from the ⟨*relation*⟩. The ⟨attribute list⟩ consists of attribute names to be retrieved, separated by commas. The result (as always) is a proper relation. The resulting relation is composed only from the attributes listed in the attribute list. Fig. 7 shows an example of the project operator (note that duplicate rows would result if they were not eliminated by the operator).



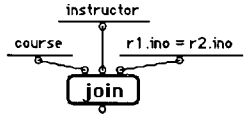

| TESTSCORE |
|---|
| 70 |
| 72 |
| 82 |
| 83 |
| 85 |
| 90 |
| 91 |
| 92 |
| 93 |
| 94 |
| 95 |
| 98 |

Fig. 7.

The project operator can also be used to change the names of the attributes in the resulting relation. For example, if we substituted "QUIZGRADE = TESTSCORE" in the attribute list, the result would have the same values, but the attribute would be named QUIZGRADE.

*4.1.3. Join*

The relational algebra notation for the join operation is $\langle \text{relation1} \rangle *_{\langle \text{condition} \rangle} \langle \text{relation2} \rangle$; the result has the attributes from both ⟨relation1⟩ and ⟨relation2⟩, and its content is the subset of the tuples of the cartesian product of ⟨relation1⟩ and ⟨relation2⟩ that satisfy the condition. The condition other than equality of attribute values may be used as arguments to the operator. The join condition is specified using basically the same syntax as the WHERE clause in SQL. Range variables in the condition are limited specifically to r1 (for ⟨relation1⟩) and r2 (for ⟨relation2⟩). (These range variables need to be used only if the condition is specified on attributes with the same name in both of the input relations, for example, r1.cno = r2.cno.) Any ⟨condition⟩ that is a tautology will result in the cartesian product of ⟨relation1⟩ and ⟨relation2⟩. Perhaps the most common use of the join is a special case commonly referred to as the equi-join, which specifies an equality condition between certain attributes of ⟨relation1⟩ and ⟨relation2⟩. An example of an equi-join is given in Fig. 8, where the COURSE and INSTRUCTOR relations are joined based on INO. The output relation produced contains all attributes from both the COURSE and INSTRUCTOR relations and conceptually is pro-




| CNO | TITLE | INO | INO1 | INAME | PAY |
|---|---|---|---|---|---|
| CS05 | COURSE # 5 | I1 | I1 | INST #1 | 100000 |
| CS10 | COURSE #10 | I2 | I2 | INST #2 | 50000 |
| CS20 | COURSE #20 | I2 | I2 | INST #2 | 50000 |
| CS15 | COURSE #15 | I3 | I3 | INST #3 | 47380.78 |
| CS25 | COURSE #25 | I3 | I3 | INST #3 | 47380.78 |
| CS30 | COURSE #30 | I3 | I3 | INST #3 | 47380.78 |

Fig. 8.

duced by selecting tuples from the cartesian product of COURSE and INSTRUCTOR where COURSE.INO = INSTRUCTOR.INO. The result of this join is a relation containing tuples for all of the courses taught combined with the information about the course instructor.

Special handling occurs when an attribute with the same name exists in both input relations. An alternative to retaining all attributes would be to discard one of the duplicated attributes as redundant. We have chosen to retain all attributes from both relations. Since the output relation cannot have columns with identical attribute names, DFQL provides a method of handling this. Our solution is to append a "1" to the attribute name of the second attribute. Thus INO and INO1 are output. A special case of the equi-join is the *natural join*, where one of the attributes with common names used in the equality condition is automatically removed from the result. This is not implemented as a primitive.

*4.1.4. Union*

The relational algebra notation for the union operation is ⟨relation1⟩ ∪ ⟨relation2⟩. The relational union is similar to but not as general as mathematical set union; the relational union requires that ⟨relation1⟩ and ⟨relation2⟩ be *union compatible*. This means the attributes must be of the same data type and in the same order. Union does not create any additional columns in the output. Relational union produces a relation containing all of the tuples of both input relations (without duplication). The example in Fig. 9 produces a relation containing the names of all the students and instructors from the example database. We first project the names of the instructors and students and then take their union: INAME and SNAME are union compatible and

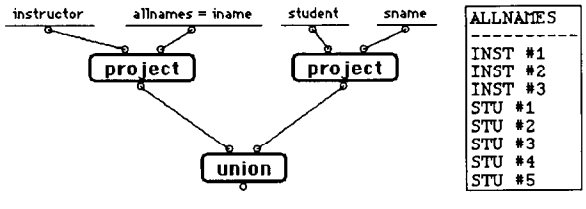



Fig. 9.

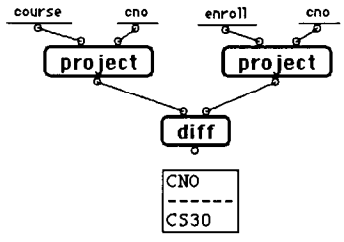



Fig. 10.

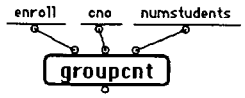
renaming changes the result relation column name to "ALLNAMES," otherwise the default column would have been INAME.

*4.1.5. Diff*

The relational algebra notation for the difference operation is ⟨relation1⟩–⟨relation2⟩. As with relational union, diff requires that ⟨relation 1⟩ and ⟨relation2⟩ be union compatible. Relational difference returns the relation that contains all tuples that occur in ⟨relation1⟩ but not in ⟨relation2⟩. An example is given in Fig. 10. It returns tuples representing courses that have no one enrolled in them.

*4.1.6. Groupcnt*

Groupcnt (short for group count) is a basic operator that provides simple aggregation capabilities. Groupcnt returns the number of distinct values in a particular set of attributes of relation of grouping attributes, and a name for the result of the count. The count is an integer representing the number of distinct values in a specified set of attributes in the input relation that belong to each value of a grouping attribute. As a special case, we allow the keyword "ALL" as an argument for the grouping attribute list. When this is specified, groupcnt simply counts all tuples in the input relation and produces a single attribute relation (using the attribute name specified for the count column) with a single tuple in it. In Fig. 11, groupcnt is used to produce a relation listing each course and how many students are enrolled in it. The result is produced by grouping the ENROLL relation by CNO with the counting attribute NUMSTUDENTS.




| CNO | NUMSTUDENTS |
|---|---|
| CS05 | 4 |
| CS10 | 3 |
| CS15 | 3 |
| CS20 | 2 |
| CS25 | 2 |

Fig. 11.

### 4.2. Non-basic primitives operators

Most of the additional primitives perform operations that are low level. Several could be specified as user-defined operators. However, in our protoype, we take advantage of built-in functions of the underlying DBMS. An example of this is the intersect primitive. Specifying an operator as a primitive slightly reduces the overhead required to interpret the query. Fig. 12 shows the additional non-basic primitives.

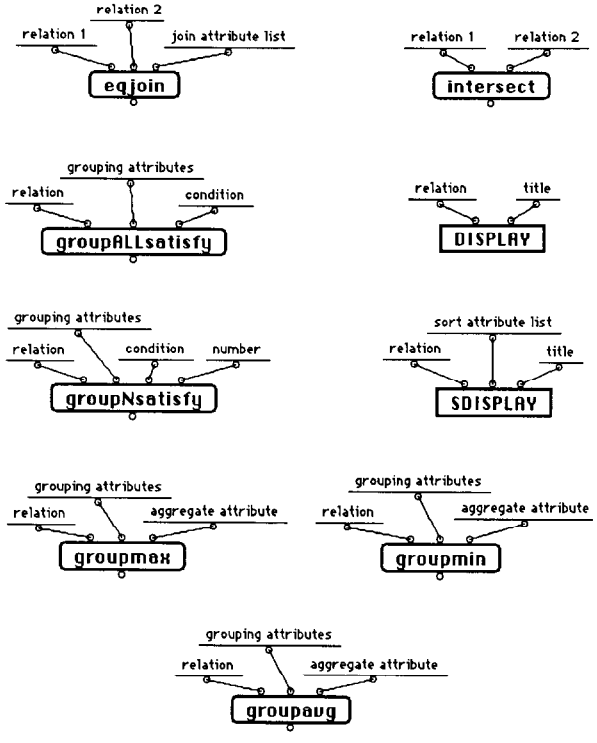



Fig. 12.

#### 4.2.1. Eqjoin

The eqjoin operator is provided primarily to aid in the construction of user-defined operators. It takes two relations ⟨relation 1⟩ and ⟨relation2⟩ and a list of attributes (⟨attribute1⟩, ⟨attribute2⟩, etc.) that occur in both relations. The DFQL join condition is specified without explicitly including equality statements.

#### 4.2.2. GroupALLsatisfy

This operator provides a simple way of introducing universal quantification. The three inputs are the name of the relation, a list of grouping attributes, and a condition statement that must be satisfied by all tuples in each group. The grouping attributes consists of their names separated by commas. For each group that meets the condition, an output tuple is generated, consisting of the grouping attributes. The result contains only those groups, as specified by their grouping attributes, where all the tuples in that group satisfy the given condition.

#### 4.2.3. GroupNsatisfy

GroupNsatisfy takes an extra argument that allows the user to specify how many of the tuples in the group satisfy the condition for that group to be included in the result. This fourth argument to groupNsatisfy must consist of a relational operator (⟨, ⟩, =, ⟨ =, ⟩ =, ~ =) and a number.

#### 4.2.4. Aggregate operators

Three aggregate operators are provided; they cannot be formed from any combination of other operators. Groupavg is used to calculate the average of an attribute of a group of tuples in the input relation. Groupmax produces the maximum and groupmin produces the minimum value of a specified attribute in each group. The three input arguments are a relation, a list of grouping attributes, and the attribute name for aggregation. The result consists of the grouping attributes with an additional column containing the aggregate values. This result attribute bears the same name as the aggregation attribute with the operation name prepended to it.

*4.2.5. Intersect*

This operator implements the relational algebra operation of intersection of relations that are union compatible. The result is a relation containing only those tuples that occurred in both ⟨relation1⟩ and ⟨relation2⟩.

*4.3. Display operators*

The display operators produce no output relation. They are provided to allow output to the screen. The display operators have square corners (and no output node) and names are displayed in all capital letters. Different colors could be used to differentiate types of operators.

*4.3.1. DISPLAY*

This operator takes a relation and a text string to be used as a title. It causes the input relation to be printed out as a table with its title printed as the header.

*4.3.2. SDISPLAY*

SDISPLAY is used to produced a sorted printout; it takes a relation and a sorting attribute list, plus a title. The attribute list should specify "ASC" or "DESC" for each attribute (for "ascending" or "descending"). The order of the attribute is important. The "major" order column is listed first, etc. The default ordering is ascending.

*4.4. User-defined operators*

With user-defined operators, the user can construct operators that look and behave exactly like the primitives. The user can create operators for situations that are unique to his query needs. A simple example of how a user-defined operator is constructed involves the select and project operators. In DFQL select and project are implemented as separate primitives. However, select and project often occur in pairs: first the selection is made and then a projection is performed. As an example, retrieve the SNO from the ENROLL relation where that student obtained at least one A grade. This query could be coded in DFQL as show in Fig. 13.

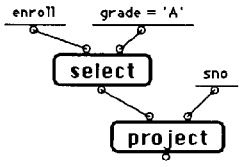



Fig. 13.

Since such combinations occur frequently, a single operator could be defined as shown in Fig. 14.

The shaded gray rectangle at the top is called the "input bar." There are three "incoming nodes" on the input bar, hence the new operator will have three input nodes. The result relation for the new operator flows out of the connected output node on the "output bar" in the diagram. Once the specification of the internals of the operator is completed, the user must provide a name for the new operator. For this example, the new operator is called *selproj*. Abstraction and encapsulation are modern techniques that are accepted universally in the field of software engineering but have not been put into practice in query languages. As another example of a user-defined operator we include Fig. 15. This demonstrates the capability to define arbitrarily complex subqueries as user-defined operators. In fact user-defined operators may contain other previously defined user-defined operators to any level of recursion.

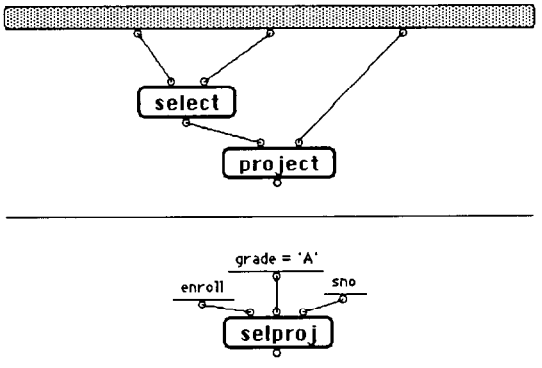



Fig. 14.

## 5. Query construction with DFQL

Here, we comment on some of the techniques used in DFQL query construction and on the benefits of the DFQL approach. All DFQL queries exist as a dataflow program in which text objects and operators are connected by dataflow paths. Execution of the query can be visualized as flowing from the top of the diagram to the bottom. When the input arguments to an operator are available, that operator may execute, producing its output which will then flow on to its connected operators. Since text objects have no inputs, they may fire at any time. Execution of the query continues until all input has been exhausted. Since DFQL does not allow recursion or iteration, each operator will fire exactly once during the life of a query. The results are displayed for the user by the DISPLAY and SDISPLAY operators. An example of a DFQL query is given in Fig. 16. The diff operator returns the SNO of students who did not receive any 'A' grades. There are several other ways that the same query could be posed. One would be to use groupNsatisfy with the condition "GRADE = 'A'" and the count condition " = 0".

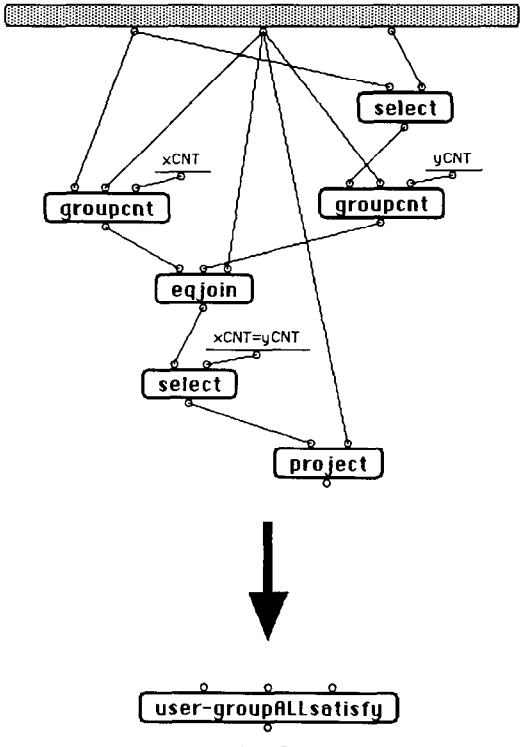



Fig. 15.

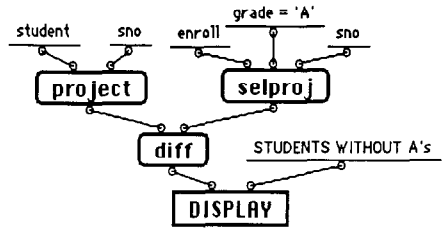



Fig. 16.

### 5.1. Incremental queries

The ability to build complex queries in an incremental manner greatly simplifies their formulation. In complex queries it is easy for the user to lose track of the process in the definition. The example query "List the names of instructors who taught CS10," can be broken into constituent parts as shown in Fig. 17. First all of the 'CS10' tuples are selected from the COURSE relation. The result can be displayed to ensure that there appears to be a correct partial answer. Next, the partial result is joined with the INSTRUCTOR relation to add information to the partial result. The new partial result can also be displayed. Finally, INAME is projected from the previous partial result to get the final solution. Although this example is extremely simple, its value should be obvious. Many large query errors are semantic and not syntactic; the DBMS will provide a result, but it will be erroneous. In a complex query, it is difficult to tell where an error has been introduced. DFQL allows the user to set a flag on any of the operators in a query to show the intermediate result. For example, in Fig. 18, the join operator is highlighted indicating that the user has flagged this operator.

Execution will then stop at the flagged operator, and the intermediate result will be displayed. If that partial result was incorrect, the user can go back and look at earlier partial results. Multiple display operators can also be used to report

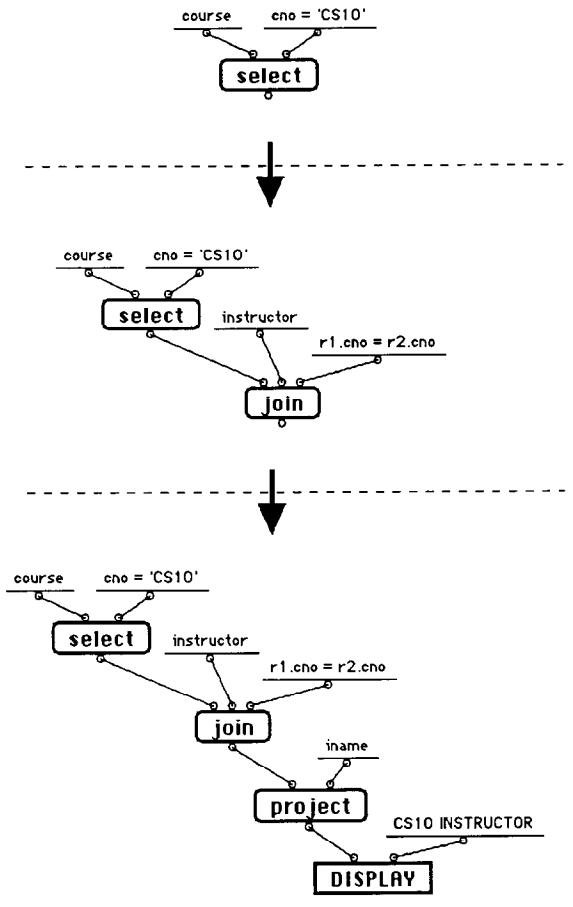



Fig. 17.

intermediate results at different locations in the query as shown in Fig. 19.

There is no easy way to simulate this approach with complex queries in SQL. It is possible with

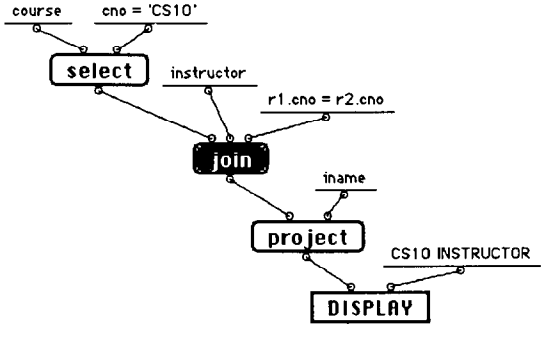



Fig. 18.

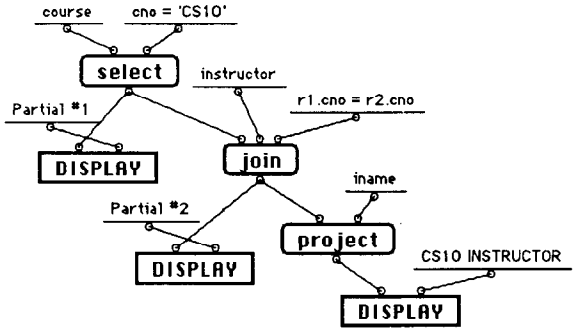



Fig. 19.

SQL to use a temporary view as an intermediate result, but this feature alone is not capable of fully supporting the notion of incremental querying.

### 5.2. Universal quantification

DFQL provides a unique solution to universal quantification by starting with elementary counting operations and building on them to satisfy the requirements of universal quantification. If all tuples in a relation or a group must satisfy some criteria, we first count the number of tuples that meet the criteria and then compare this number with the total number of tuples under consideration. If these two numbers are equal, then the universal quantifier has been satisfied. The actual implementation of this idea is incorporated by the user-groupALLsatisfy operation, shown in Fig. 15. The counting idea can be extended to supply other useful quantification type operators, such as groupNsatisfy. We believe that the concept of counting tuples is much simpler than understanding the logical idea of universal and existential quantification.

### 5.3. Nesting and functional notation

DFQL implicitly provides a nesting capability. Unlike SQL and block structured languages, however, no nesting constructs are required. Thus, DFQL requires no range variables or scoping rules, while good understanding of both range variables and scoping rules is necessary to code complex queries in SQL. The idea of nesting is

provided in DFQL by having subqueries (i.e. user-defined operations) execute first, providing arguments for later query operators.

### 5.4. Graph structure of DFQL query

There is a large body of techniques that have been developed for the optimization of relational algebra expressions. Most SQL interpreters/compilers are not capable of performing optimization across levels of a nested query, but if the same query is expressed as a series of relational algebra operations it can then be optimized. By using a graph structure of relational operators, the query can be more easily optimized globally than can be combinations of partial queries in a textual block structured language. In fact, the work of Dadashzadeh in converting SQL queries into relational algebra graphs for optimization purposes, results in structures quite similar to DFQL queries [12]. By using a graphical, relational algebra approach to query formulation, we believe that the user is provided with a much more consistent and straightforward interface to the database.

## 6. Conclusion

DFQL was created in order to provide an improved interface to the relational model of database management. It presents a new way of visualizing database queries. DFQL's dataflow structure and orthogonality greatly aid the user in the composition of complex queries. It also allows users to extend the language by the creation of user-defined operators, which can be used to simplify queries by introducing levels of abstraction, effectively hiding detailed query operations. We conducted a simple human factors experiment, in which DFQL compared favorably to SQL in query writing.

### 6.1. Advantages of DFQL

DFQL's advantages accrue from the combination of its visual representation, its dataflow structure, and its operator set. It can be used to express both simple and complex queries in an intuitive manner.

#### 6.1.1. Power

DFQL is relationally complete, and extends first-order predicate logic by the inclusion of grouping operators in both comparison functions and aggregation. The provided set of primitive operators gives the user the capability of coding any desired query.

#### 6.1.2. Extensibility

The user may extend the DFQL language by coding user-defined operators from the set of provided primitive operators and other already defined user-defined operators.

#### 6.1.3. Ease-of-use

A dataflow diagram has the capability, especially when using levels of abstraction, to represent even complex problems in an intuitive manner. Relations are visualized as objects flowing from one operator to another. This contributes to its ease-of-use. Providing the computer with a dataflow style query graph also enhances its ability to optimize the query.

The ability to form and modify queries incrementally is one of DFQL's most important ease-of-use features. Intermediate query results can be displayed by use of multiple display statements. Partial results can be returned from any point in a given query and used to help verify or debug the query. Since the output of an operator must be a relation, the result of a DFQL operator may always be combined with another DFQL operator to form a more complex query. The combination of all of these features definitely aids the user in the construction of correct queries.

#### 6.1.4. Visual Interface

The key to the implementation of DFQL is the ability for the user to build and modify the DFQL dataflow style queries easily and interactively. DFQL encourages the user to construct queries incrementally, use intermediate results, and take advantage of all of the benefits provided by the dataflow approach.

### 6.2. Shortcomings of the DFQL concept

#### 6.2.1. Interface problems

One of the most important requirements for a successful implementation of DFQL is the provision of an adequate user interface. The size of the display limits the number of visual objects that can be on the screen at any one time. There may be a corollary here to the "no more than one page of code per procedure" rule commonly touted in programming language circles. However, by using reasonable sized visual objects an average sized screen becomes cluttered. An attempt is made at mitigating this problem by allowing the drawing area to be scrolled both left and right and up and down. This allows more DFQL code to be "on the system" but is cumbersome: the user loses the advantage of being able to sit and look at the query as a whole. Another visual problem occurs when there are very many dataflows in a single query. The dataflow lines may become multiply crossed leading to a difficult to follow DFQL diagram. A solution to both of these problems lies in utilizing user-defined operators to their fullest. Text items take up an inordinate amount of space on the screen at any level of abstraction. However, it is difficult to come up with a more compact and convenient way to represent complex logical conditions, relation names, etc.

#### 6.2.2. Language problems

A problem stemming from DFQL's intense visual orientation is the ability to use DFQL in conjunction with other textual computer languages. DFQL queries could be compiled and inserted into textual programs as functions, however this provides no good way of actually looking at the DFQL code in the context of the program. Such an ability is a common attribute of most embedded query languages. The text translation of DFQL would still be a dataflow oriented object but the program in which it is embedded would be purely procedural/sequential.

## References

[1] Abiteboul, S., and Hull, R., "IFO: A Formal Semantic Database Model," *ACM Transactions on Database Systems*, v. 12, pp. 525–565, December 1987.

[2] Andyne Computing Limited, *GQL: Graphical Query Language GQL / User Demo Guide*, Kingston, Ontario, March 1991.

[3] Angelaccio, M., Catarci, T., and Santucci, G., "QBD*: A Graphical Query Language with Recursion," *IEEE Transactions on Software Engineering*, v. 16, pp. 1150–1163, October 1990.

[4] Bryce, D., and Hull, R., "SNAP: A Graphics-based Schema Manager," *Proceedings of the Second IEEE International Conference on Data Engineering*, pp. 151–164, February 1986.

[5] Chamberlin, D.D., and Boyce, R.F., "SEQUEL: A Structured English Query Language," *Proceedings of the ACM-SIGFIDET Workshop*, Ann Arbor, Michigan, May 1974.

[6] Codd, E.F., "A Relational Model for Large Shared Data Banks," Communication of ACM, 13:6, June 1970.

[7] Codd, E.F., "Relational Completeness of Data Base Sublanguages" in *Data Base Systems*, pp. 65–98, Prentice-Hall, 1972.

[8] Codd, E.F., "Fatal Flaws in SQL: Part I," *Datamation*, v. 34, pp. 45–48, 15 August 1988.

[9] Codd, E.F., "Fatal Flaws in SQL: Part II," *Datamation*, v. 34, pp. 71–74, 1 September 1988.

[10] Codd, E.F., *The Relational Model for Database Management: Version 2*, Addison-Wesley, 1990.

[11] Czejdo, B., and others, "A Graphical Data Manipulation Language for an Extended Entity-Relationship Model," *IEEE Computer*, v. 23, pp. 26–36, March 1990.

[12] Dadashzadeh, M., and Stemple, D., "Converting SQL queries into relational algebra," *Information and Management*, v. 19, pp. 307–323, December 1990.

[13] Date, C.J., "Where SQL Falls Short," *Datamation*, v. 33, pp. 83–86, 1 May 1987.

[14] Davis, A.L., and Keller, R.M., "Data Row Program Graphs," *IEEE Computer*, v. 15, pp. 26–41, February 1982.

[15] Kim, H., Korth, H.F., and Silberschatz, A., "PICASSO: A Graphical Query Language," *Software-Practice and Experience*, v. 18(3), pp. 169–203, March 1988.

[16] Kim, W., "On Optimizing an SQL-like Nested Query," *ACM Transactions On Database Systems*, v. 7, 1982.

[17] Kimura, T.D., "Determinancy of Hierarchical Dataflow Model: A Computation Model for Visual Programming," Washington University Department of Computer Science WUCS-86-5, March 1986.

[18] Miyao, J., et. al., "Design of a High Level Query Language for End Users," *1986 IEEE Workshop on Languages for Automation*, National University of Singapore, Kent Ridge, Singapore, 27–29 August 1986.

[19] Negri, M., Pelagatti, G., and Sbattela, L., "Short Notes: Semantics and Problems of Universal Quantification in SQL," *The Computer Journal*, v. 32, pp. 90,91, 1989.

[20] Ozsoyoglu, G., and Wang, H., "A Relational Calculus with Set Operators, Its Safety, and Equivalent Graphical Languages," *IEEE Transactions on Software Engineering*, v. 15, pp. 1038–1052, September 1989.

[21] Ozsoyoglu, G., Matos, V., and Ozsoyoglu, Z. "Query Processing Techniques in the Summary-Table-By-Exam-

ple Database Query Language," *ACM Transactions on Database Systems*, v. 14, pp. 526–573, December 1989.
[22] Sebesta, R.W., *Concepts of Programming Languages*, Benjamin Cummings, 1989.
[23] Schneiderman, B., "Improving the Human Factors Aspect of Database Interactions," *ACM Transactions on Database Systems*, v. 3, pp. 417–439, December 1978.
[24] Shu, N.C., *Visual Programming*, Van Nostrand Reinhold, 1988.
[25] Sockut, G.H., et. al,, GRAQULA: A Graphical Query Language for Entity-Relationship or Relational Databases, IBM Research Report RC16877 (#73833), 14 March 1991.
[26] Welty, C., and Stemple, D.W., "Human Factors Comparison of a Procedural and a Nonprocedural Query Language," *ACM Transactions on Database Systems*, v. 6, pp. 626–649, December 1981.
[27] Wong, H.K.T., and Kuo, I., "GUIDE: Graphical User Interface for Database Exploration," *Proceedings of the Eighth International Conference on Very Large Databases*, pp. 22–32, September 1982.
[28] Wu, C.T., "OOP + Visual Dataflow Diagram = Prograph," *Journal of Object Oriented Programming*, pp. 71–75, June 1991.
[29] Yourdon, E., *Modern Structured Analysis*, Prentice-Hall, 1989.
[30] Zloof, M.M., "Query-by-Example: A Data Base Language," *IBM Systems Journal*, v. 16, pp. 324–343, 1977.

## Appendix A: Example database

The schema and data for the database used by the examples in the text are included here. Most of the relationships between the data should be apparent. The intention is to represent a simple university database in which students are enrolled in courses taught by instructors. The relational schema is listed below. In the schema representation, keys are underlined.

COURSE (CNO, TITLE, INO)
ENROLL (CNO, SNO, GRADE, TESTSCORE)
INSTRUCTOR (INO, INAME, PAY)
STUDENT (SNO, SNAME, ADDR, PHONE, GPA)

Attribute definitions:

ADDR – Address
CNO – Course Number, unique to a single course
GPA – Grade Point Average
GRADE – Course Grade('A', 'B', 'C', etc.)
INAME – Instructor Name
INO – Instructor Number, unique to a single instructor
PAY – Instructor's Pay
PHONE – Phone Number
SNAME – Student Name
SNO – Student Number, unique to a single student
TESTSCORE – Numerical Grade for an exam in a course
TITLE – Name of a Course

The example data in the database is listed below.

COURSE

| CNO | TITLE | INO |
|---|---|---|
| CS05 | COURSE #5 | I1 |
| CS10 | COURSE #10 | I2 |
| CS15 | COURSE #15 | I3 |
| CS20 | COURSE #20 | I2 |
| CS25 | COURSE #25 | I3 |

ENROLL

| SNO | CNO | GRADE | TESTSCORE |
|---|---|---|---|
| S1 | CS10 | A | 92 |
| S1 | CS15 | C | 72 |
| S1 | CS20 | A | 93 |
| S2 | CS05 | A | 98 |
| S2 | CS10 | A | 95 |
| S2 | CS25 | A | 90 |
| S3 | CS05 | B | 85 |
| S3 | CS10 | A | 91 |
| S4 | CS05 | A | 93 |
| S4 | CS15 | B | 83 |
| S4 | CS25 | A | 94 |
| S5 | CS05 | C | 70 |
| S5 | CS15 | B | 82 |
| S5 | CS20 | A | 94 |

INSTRUCTOR

| INO | INAME | PAY |
|---|---|---|
| I1 | INST #1 | 100000.00 |
| I2 | INST #2 | 50000.00 |
| I3 | INST #3 | 47380.78 |

STUDENT

| SNO | SNAME | ADDR | PHONE | GPA |
|---|---|---|---|---|
| S1 | STU #1 | ROOM 1 | 111-1111 | 3.85 |
| S2 | STU #2 | ROOM 1 | 111-1111 | 3.40 |
| S3 | STU #3 | ROOM 3 | 333-3333 | 3.75 |
| S4 | STU #4 | ROOM 3 | 444-4444 | 2.85 |
| S5 | STU #5 | ROOM 5 | 555-5555 | 3.30 |