NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

TRANSLATION OF THE DATA FLOW QUERY LANGUAGE FOR THE MULTIMODEL, MULTIBACKEND DATABASE SYSTEM

by

Nancy C. Free

September, 1994

Thesis Advisor: C. Thomas Wu

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704

1 AGENCY USE ONLY (Leave blank)	2 REPORT DATE Sep 1994	3 REPORT TYPE AND DATES COVERED Master's Thesis

4 TITLE AND SUBTITLE TRANSLATION OF THE DATA FLOW QUERY LANGUAGE FOR THE MULTIMODEL, MULTIBACKEND DATABASE SYSTEM	5 FUNDING NUMBERS
6 AUTHOR(S) Free, Nancy C	
7 PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000	8 PERFORMING ORGANIZATION REPORT NUMBER
9 SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10 SPONSORING/MONITORING AGENCY REPORT NUMBER

11 SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited	12b. DISTRIBUTION CODE A

13. ABSTRACT (minimum 200 words)

This thesis involved the design and translation of the Data Flow Query Language (DFQL) for the Multi-Lingual, Multi-Backend Database System (MDBS). The MDBS is a database system that can effectively support multiple data models and their corresponding data manipulation languages. The problem was the MDBS interfaces are text-based, and not very user-friendly.

The approach taken to solve this interface problem was to design and translate the DFQL for implementation on the MDBS. DFQL was designed to improve and extend SQL, the data manipulation language associated with the relational data model. It uses a graphical interface based on the data flow paradigm. This translation would extend the MDBS by allowing a graphical interface to be implemented, whereas currently a user can only access the system with text-based interfaces.

The result of this thesis is the development of the DFQL to ABDL translator. The subsequent implementation of this translator on the MDBS would be a user-oriented enhancement to the current system. In addition, further improvements to the MDBS should be made, such as allowing the use of additional data types (currently constrained to string and integer) and the ability to create views. These changes would allow all the benefits from DFQL, such as orthogonality, language extensibility and incremental querying to be achieved and made available to the user.

			15. NUMBER OF PAGES 85
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19 SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

Approved for public release; distribution is unlimited.

TRANSLATION OF THE DATA FLOW QUERY LANGUAGE FOR THE MULTIMODEL, MULTIBACKEND DATABASE SYSTEM

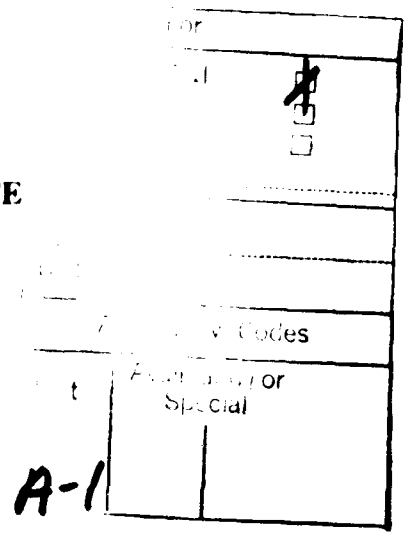
Nancy C. Free
Captain, United States Army
B.S., Kansas State U[illegible], 1984

Submitted in partial fu[illegible]llment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1994



Author: _____
Nancy C. Free

Approved by: _____
C. Thomas Wu, Thesis Advisor

_____
David K. Hsiao, Second Reader

_____
Ted Lewis, Chairman
Department of Computer Science

ABSTRACT

This thesis involved the design and translation of the Data Flow Query Language (DFQL) for the Multi-Lingual, Multi-Backend Database System (MDBS). The MDBS is a database system that can effectively support multiple data models and their corresponding data manipulation languages. The problem was the MDBS interfaces are text-based, and not very user-friendly.

The approach taken to solve this interface problem was to design and translate the DFQL for implementation on the MDBS. DFQL was designed to improve and extend SQL, the data manipulation language associated with the relational data m odel. It uses a graphical interface based on the data flow paradigm. This translation would extend the MDBS by allowing a graphical interface to be implemented, whereas currently a user can only access the system with text-based interfaces.

The result of this thesis is the development of the DFQL to ABDL translator. The subsequent implementation of this translator on the MDBS would be a user-oriented enhancement to the current system. In addition, further improvements to the MDBS should be made, such as allowing the use of additional data types (currently constrained to string and integer) and the ability to create views. These changes would allow all the benefits from DFQL, such as orthogonality, language extensibility and incremental querying to be achieved and made available to the user.

DISCLAIMER

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

The reader is cautioned that computer programs developed in this research may not have been exercised for all cases of interest. While effort has been made, within the time available, to ensure that the programs are free of computational and logic errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

TABLE OF CONTENTS

ACKNOWLEDGMENT

I would like to thank my thesis advisor, Professor C. Thomas Wu, for his patience and assistance, but mostly for his sense of humor. I would also like to thank LCDR Bill Deiners for all of his experienced help with the Multi-lingual, Multi-backend Database System. Even after he graduated, his long distance assistance proved to be invaluable. I must also thank CPT Thierno Fall, Senegalese Army, for all of his patient help with the Macintosh computers and the Prograph programming language.

Finally, I wish to thank my husband, Eddie, for making these past two years memorable. It was better because we did it together!

I INTRODUCTION

A. MOTIVATION

Database systems constitute a significant portion of computer technology today, mainly due to their increasing role in all areas of our society. There are three predominant database models in use today. These are hierarchical, network, and relational database models. A fourth, and relatively new model, is the object-oriented database model. Finally, a fifth model is the functional model, often used in Artificial Intelligence programs. Each of these models has strengths and weakness', although each is more suited to a specific type of database application. For example, the relational model is well suited for business/financial applications where relationships can be viewed as a table; this database is probably the most widely used today. On the other hand, the hierarchical model would be a better choice for maintaining a database for design applications. There are many situations where the advantages of more than one model would be beneficial to the implementation of the database. However, most organizations cannot afford to institute more than one model. Not only are the direct costs of procuring another, separate, system usually prohibitive, but there are also the training, administration and storage costs to consider. One prototype solution to this problem is the Multi-lingual, Multi-backend Database System (MDBS) - a system which can support many different data models and their corresponding data language (Demurjian, 1987). Currently, this system allows a user to access data from any of the following four

database models relational, hierarchical, network, and functional as well as from the MDBS kernel data language.

Another problem with most database systems today is the lack of a user friendly interface. And the MDBS is guilty of this downfall as well. Although the current interface allows fairly easy access to the user's database model choice, once the user reaches that model, the specific data language of that model must be used. A graphical/visual interface to the relational model has been developed to enhance the ease of use of the relational database model (Clark, 1991). This Data Flow Query Language (DFQL) utilizes a dataflow paradigm to implement and expand the Structured Query Language in a much more user friendly and graphically intuitive environment.

Since the relational database is the most popular and widely used data model today, I am focusing my research on designing and translating the DFQL for implementation on the MDBS. Although DFQL will be translated into the kernel language of the MDBS (the Attribute Based Data Language), and therefore can be used to access any of the supported data models, its basis is in the relational model and its language, SQL. The ultimate goal is to significantly improve the user interface for interacting with the MDBS.

B. BACKGROUND

1. Multi-lingual, Multi-backend Database System

The Multi-lingual, Multi-backend Database System (MDBS) was developed by Demurjian and Hsiao at The Ohio State University and currently, at the Naval

Postgraduate School's Laboratory for Database Systems Research. The concept behind this system is to provide a single database computer to manage multiple data models and execute all database operations written in the corresponding data languages (Demurjian, 1987). The MDBS has demonstrated the ability for at least four different data models and languages to not only coexists on a single system, but also to interact and share data (Demers, 1994).

The key to this interaction is the simple kernel data model of the MDBS. This kernel data model and language are the attribute-based data model (Hsiao, 1970) and the attribute-based data language (Banerjee, 1977). This model has been shown to support several different data models and their data languages. The MDBS performs the interaction between two source databases (relational, hierarchical, etc.) by two different types of mappings into the kernel data model. The first is the *data-model transformation* which transforms the source database into the kernel database. The second is the *data-language translation* which takes an operation in the source data language and translates it into an equivalent operation in the kernel language (Demurjian, 1987). These two mappings allow for the complete interaction of different data models using the data language of choice. For example, a network database could be queried using SQL transactions.

2. The Data Flow Query Language

The Data Flow Query Language (DFQL) was developed by Clark and Wu at the Naval Postgraduate School in the early 90's. The purpose of this graphical interface to a

relational database was to improve and extend the standard query language associated with the relational model, the Structured Query Language (SQL) (Wu, 1991). There are many documented problems with SQL, many of which focus on the ease of use issues and extension issues (Elmarsi, 1989). DFQL uses a graphical environment which allows a user-friendly interface to the database. It also provides for easy extensibility of the language by allowing the user to create new operators in terms of the existing ones (Clark, 1991). The implementation language used for DFQL is Prograph, an object-oriented language, in an *Apple Macintosh* environment. DFQL is based on a dataflow structure, as is Prograph. In addition, Prograph is object-oriented which provides many powerful features to improve modularity and maintainability of the DFQL code.

The purpose of this thesis is to design and develop a translation of the Data Flow Query Language to be implemented on the Multi-backend Database System. Through my research, I also plan to determine the portability and ease of translation of the current system.

C. THESIS ORGANIZATION

In Chapter II, the Multi-lingual, Multi-backend Database System is discussed in detail. This will include the functionality and organization of the system, as well as a thorough description of the attribute-based data language (ABDL). In Chapter III, the Data Flow Query Language is examined in detail. Chapter IV discusses the design and translation decisions made in translating DFQL's SQL statements into ABDL for further

implementation on the MDBS. Chapter V discusses problems encountered, analysis and solutions determined in the translation. This chapter will also cover those problems concerning portability and the ease of translation of DFQL. Finally in Chapter VI, a summary discussion of the work and research is provided, conclusions made and recommendations for further work in this area.

II THE MULTI-LINGUAL, MULTI-BACKEND DATABASE SYSTEM

In this chapter, a more detailed examination of the Multi-lingual, Multi-backend Database System is provided. The motivation behind designing and implementing such a system is reviewed, with a discussion of its advantages and functionalities. This is followed by a more technical review of the system's structure and methodology.

A. MOTIVATION AND ADVANTAGES

1. Motivation for a Multi-lingual Database System (MLDS)

Since the evolution of the database management system (DBMS), design and implementation philosophy has focused on mono-lingual database systems; that is, a database computer with a single data model and its corresponding data manipulation language (i.e. the relational data model and SQL). And when an organization is determining which model to use, the data model which offers those functionalities best suited for the organization's particular data is chosen.

The key word here is 'best suited'. Considering the increasing database computing requirements in business and government organizations today, just one database system rarely fits the required application perfectly. Therefore, these organizations settle for the one DBMS which suits their needs best. This results in one of two main alternatives. The first is to have the users "work around" those functionalities they need in the DBMS, but are not there. Or the organization can purchase each of the monolingual database systems it needs separately. Either alternative results in additional costs to the

organization: employees' time spent "working around" the current system, or purchase and training costs for additional DBMSs. So Demurjian and Hsiao proposed a multi-lingual database computer that has the capability and flexibility to support a variety of data models and their corresponding data manipulation languages on a single database computer (Demurjian, 1985).

Another motivating factor can be seen in an analogy provided by Demurjian (Demurjian, 1987) in a comparison with Operating Systems. Operating Systems were originally mono-lingual, designed to support a single programming language, much like a current database system supports a single data model and it's data manipulation language. As computers and operating systems have evolved , many varied programming languages can be supported by a single operating system. Therefore, an operating system can execute and support a user's program in different languages and data structures, as well as handle resource allocation. And by Demurjian's analogy, a single DBMS should be able to provide management of varied data models and access to these models via their corresponding data manipulation language. Thus, the proposal for a *multi-lingual database system* (MLDS) is defined as a single system to support many different data models and their data languages, much like an operating system supports different programming languages.

2. Advantages of the MLDS

One of the most valuable features is the ability to "migrate" existing databases into the MLDS (Holste, 1986). This can be especially beneficial for organizations

currently operating an existing mono-lingual DBMS. Once the existing database is migrated into the MLDS, data manipulations may be performed using the original data language. In addition, the same database may be interfaced using one of the other supported data models, taking advantage of it's particular features and capabilities. The result is the ability to access current data in a wider variety of transactions.

Another practical advantage of the MLDS is the ability to reuse previously developed database transactions (Demurjian, 1987). Since database transactions written in different data languages can be run in the MLDS, transactions written in a specific data language on another database system can also be run in the MLDS. This means that no transaction conversions are required when migrating to the MLDS, and old transactions can be reused as is. In addition, new transactions can be written in any of the supported data languages.

A third advantage to this system can be realized in the economy and effectiveness of hardware upgrades. As technology progresses, or with increased data requirements, it is inevitable that hardware upgrades will be necessary. And as a single system, an upgrade to the MLDS will benefit all of the supported data models. This is contrasted with upgrading *separate* DBMS, each of which supports a single data model, resulting in increased expense and effort.

With the ability to support different data models in one system, the MLDS provides a user with an environment to explore the other data models. Thus, the strengths

of one model may be explored and those desirable features can be utilized for different applications. And all of this may be done with one database computer.

Two other advantages of the MLDS can be better defined as enhanced functionalities. The first of these focuses on the availability of the system's native data model and data language. This is the kernel data model (KDM) and the kernel data language (KDL). This is the base data model, and it is implemented in the MLDS as the attribute-based data model and language (ABDM/ABDL). Basically, all of the conventional data models are be transformed into equivalent databases structured in the kernel model. In addition, each of the data languages are translated into the kernel data language. Although this is the system's kernel model, because it is a high-level model and language, it also acts as an additional data model in which the user can explore and use as desired. These transformations and translations are performed by the MLDS, and are essentially transparent to the user (Holste, 1986). The technical details of this facility are further discussed later in this chapter.

The other enhanced functionality allows the MLDS to be used as a rapid-prototyping environment to develop and experiment with new data models, languages and interfaces (Demurjian, 1987). An example is the work proposed in this thesis; that of providing a more user-friendly interface to the system. With a multi-lingual environment, a proposed model and/or language can be fine-tuned and tailored as necessary, without having to develop an entire database system for the new model.

B. STRUCTURE AND CONFIGURATION

1. Structure of the MLDS

A diagram of the multi-lingual database system structure is provided at Figure 1. The user interacts with the *Language Interface Layer (LIL)* utilizing a *user-chosen data model (UDM)* and with transactions written in the corresponding *user-chosen data language (UDL)*. The LIL is where the user interfaces with the system. It includes querying the user for the data model to be used and requesting file names for databases and queries. The LIL passes the user transaction to the *kernel mapping system (KMS)*.

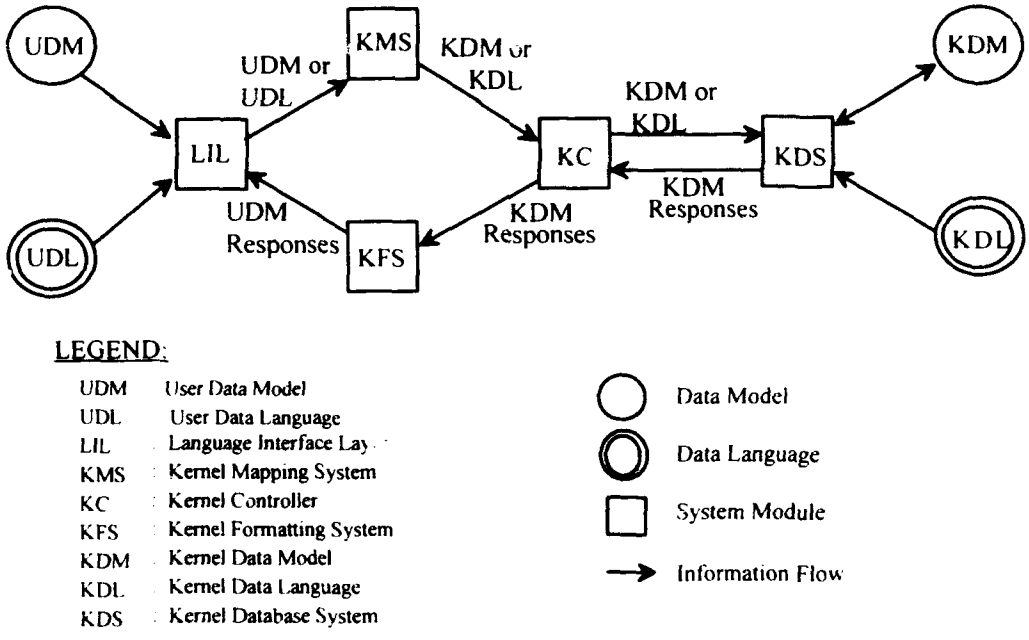


Figure 1. The Multi-Lingual Database System

The KMS is basically the heart of the system. The KMS handles both database definition and database manipulation requests. For database creation, the KMS transforms the UDM database definition into an equivalent *kernel data model (KDM)* definition. This definition is then sent to the *kernel controller (KC).* The KC sends this transformed database definition to the *kernel database system (KDS),* which issues the commands to define the new database in KDM form. The KDS notifies the KC, which in turn notifies the user via the LIL, that the database definition is processed, and the data may now be loaded.

UDL transactions are also routed through the KMS for translation into an equivalent *kernel data language (KDL).* This KDL transaction is forwarded to the KC, which forwards it to the KDS for execution. The KDS sends the result of the transaction in KDM form back to the KC. Then, the results are sent to the *kernel formatting system (KFS)* for transforming them back into UDM form. After this transformation, the results are returned to the user via the LIL.

The kernel controller plays an important role in the system in that it handles all interfaces to the backend system. Basically, it's task is to simulate the operational environment required by the UDM and UDL (Demurjian, 1987). The KC is responsible for overseeing the execution of the KDL transactions so that the integrity of the database is preserved. It also performs exception handling if an error is detected in the backend. During retrieval transactions, the KC properly structures the KDS responses and passes this data and control to the KFS (Demers, 1994).

This structure, minus the KDS, KDM and KDL, is repeated for each supported data model and language. Each data model/language has its own LIL, KMS, KC and KFS, called the *language interface*. And each of these interfaces share the kernel system: the KDS, KDM and KDL. Therefore, from the user's perspective, there are several different data models/languages with which to access the data. However, from the system's perspective, there exists only one data model and language with which to manipulate the data. This structure highlights another advantage: duplicated data is reduced. Although the data can be accessed using any or all of the supported data models, it is only stored once, in the kernel data model.

2. Configuration of the Multi-backend Database System

The Multi-backend Database System (MDBS) was designed with performance enhancement in mind, as opposed to the performance degradation often found in mainframe-based system designs (Hogan, 1989). It accomplishes this goal by utilizing off the shelf micro-computers working in parallel. Database functions are moved to these computers, called the *backend systems.* Another computer acts as the controller to interface with the backends and/or the user. A diagram of the MDBS is provided at Figure 2.

The backend computers have their own hard disk subsystems, and are responsible for processing user queries. The base data is distributed fairly equally over the backends, which thereby reduces the search space of each backend. The system works as follows. When the controller receives a transaction, it is transmitted to all of the backends.

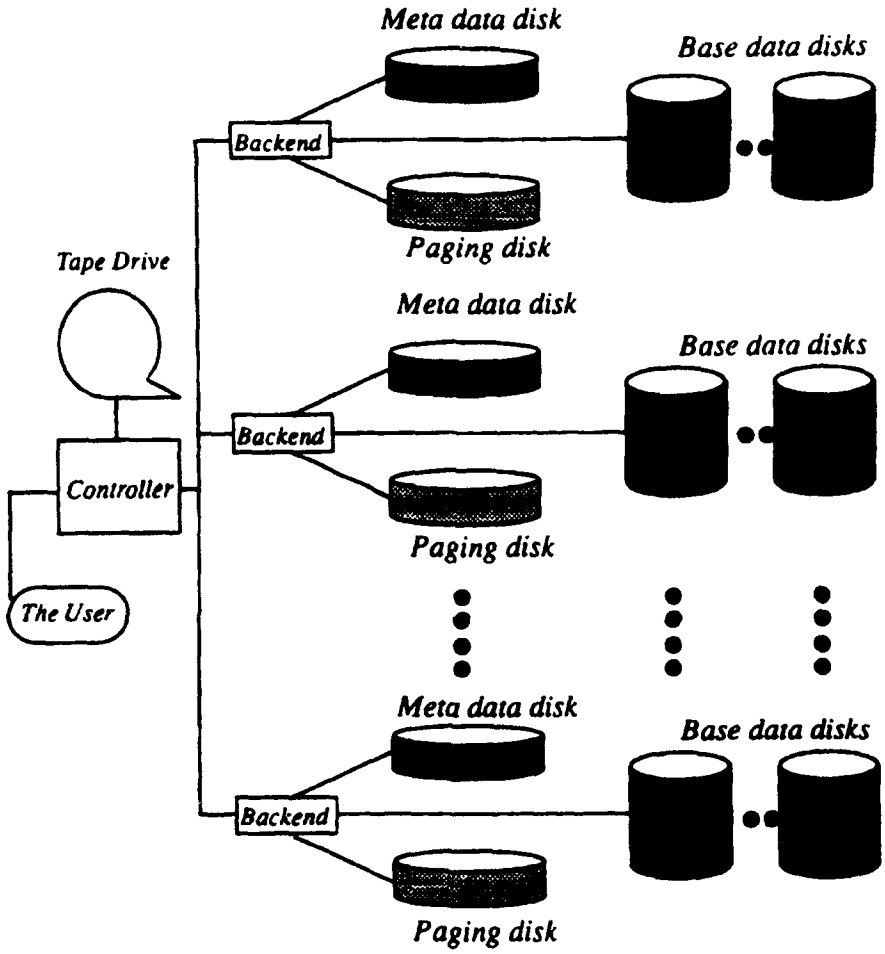


Figure 2. Hardware Configuration of the MultiBackend Database System

Communication is handled using a standard Ethernet Local Area Network. This query is then executed simultaneously on each backend, with the result being passed back to the controller. This parallel aspect of the backend structure results in improved response time since each backend works simultaneously and independently to complete the transaction (Hall, 1989). Since there is only one controller, it has the potential of becoming a bottleneck in the system. However, this potential is minimized by allowing the controller to perform a minimum of functions, placing the heaviest burden on the backends (Holste, 1986).

The configuration of the MDBS also allows for a high degree of extensibility. As the amount of data being stored grows, performance may decrease. All that is needed to improve both response times and system capacity is to add additional backend systems. In fact, studies have shown that by doubling the number of backends can nearly double the speed and capacity of the MDBS (Hall, 1989). And since these backend systems are off the shelf purchases, the availability, affordability, and maintainability all add up to a very cost effective performance enhancement.

C. THE ATTRIBUTE-BASED DATA MODEL

As previously discussed, the Attribute-Based Data Model (ABDM) and the Attribute-Based Data Language (ABDL) are the kernel data model and language for the MBDS. The ABDM was chosen as the kernel model because it stores the meta data and the base data separately, introduces equivalence relations which partition the base data into mutually exclusive sets called clusters, and allows the clusters to be distributed

across the backends, thereby enhancing the system's perfomance (Bourgeois, 1992). The corresponding data language, ABDL, provides a simple but semantically rich and complete language. It is designed to allow traditional languages (i.e. SQL, DL/1, etc.) to be translated into ABDL. This translation is key to mapping the multiple data models/languages into a single data model/language (ABDM/ABDL).

1. ABDM Constructs

In the ABDM, each *database record* is a set of attribute-value pairs. An *attribute-value pair* consists of the attribute name and its corresponding value. The attribute-value pair is enclosed by a pair of angled brackets, with the attribute name first, followed by the corresponding value. For example, <DLOCATION, Houston> is an attribute-value pair where DLOCATION is the attribute name with the corresponding value of Houston. A *record* consists of a set of attribute-value pairs and an optional textual field called the *record body*. The first attribute-value pair in a record identifies the *file* which contains the records, followed by the rest of the pairs which make up the record. A *template* name may be used instead of file name in the record. The entire record is enclosed by parenthesis, as shown below:

(<TEMP, Dept_Locations>, <DNUMBER, 1>,
<DLOCATION, Houston>)

Within the record, no two attribute-value pairs have the same attribute name.

Another important construct in the ABDM is defined for indexing purposes. Certain attribute-value pairs of a record, or a file, are called the *directory keywords* of the record, or file. These are kept in a *directory*, and are used to index and identify the

records, or files. All of the other pairs not kept in the directory are called *non-directory keywords*. Searching for records or files under the keywords can significantly reduce the search space when any of the ABDL operations are performed on the database.

2. ABDL Operations

There are five primary database operations supported by the ABDL. These are RETRIEVE, RETRIEVE-COMMON, INSERT, DELETE, and UPDATE. All database queries in any of the supported data manipulation languages are translated into some combination of these operations. Each of these operations are combined with qualifications (used to specify the part of the database on which to operate) to form a *request* in the ABDL. *Transactions* are two or more requests grouped together and surrounded by square brackets.

Before any of these operations can be performed, the qualification of the database is defined through *keyword predicates*, or simply *predicates*. A predicate consists of a three-tuple: (attribute name, relational operator, attribute value). A *query* is formed by combining these in disjunctive normal form as shown below.

(TEMP = Dept_Locations and DLOCATION = Houston) or
(TEMP = Dept_Locations and DLOCATION = Stafford)

This query will locate all the records of the Dept_Locations file which have as a value for DLOCATION, Houston or Stafford. Each of the primary operations is described in further detail below. Greater emphasis is given to the first two operations, because these are key to the translation from DFQL into ABDL.

a. The RETRIEVE Request

The RETRIEVE request is used to retrieve information from previously defined databases. It does not affect the contents of the database in any way. This request specifies which records to retrieve using a qualification consisting of a query, a list of output attributes, and an optional BY attribute (used to order the output). An example of a RETRIEVE would be:

```
[RETRIEVE ((TEMP = Works_on) and (HOURS >10))
          (ESSN, PNO)  BY PNO]
```

This request returns the Employee SSN and Project Number of those employees who work on projects more than 10 hours. The result will be sorted by Project Number.

Aggregate operations are also allowed on RETRIEVE requests. These include *SUM, COUNT, AVG, MIN,* and *MAX which may be performed on one or more* attribute values. This allows the attribute value to be an aggregate of values from multiple records. For example, the query

```
[RETRIEVE ((TEMP = Employee) and (DNO = 5))
 (AVG (SALARY))]
```

will return the average salary of all employees in department 5.

b. The RETRIEVE-COMMON Request

This RETRIEVE-COMMON request involves the merging of two files based on given common attribute values. Basically, this is a transaction of two or more RETRIEVE requests, separated by the COMMON attributes, and is processed serially.

For example, the following single query would provide all of the department names and locations:

[RETRIEVE (TEMP = Department) (DNAME)
COMMON (DNUMBER, DNUMBER)
RETRIEVE (TEMP = Dept_Locations) (DLOCATION)

Although in this example the attribute names in the COMMON clause are the same, they are not required to be. However, they must be common attribute values.

c. The INSERT Request

The INSERT request inserts a new record into a previously defined database. The request consists of the operational word INSERT followed by all of the attribute-value pairs in the database record, and surrounded by parenthesis. For example,

[INSERT (<TEMP = Dept_Locations>, <DNUMBER = 4>,
<DLOCATION = Houston>)]

inserts a new record for Department 4 at Location Houston.

d. The DELETE Request

The DELETE request can be used to delete either an entire file or specific records in a file in a previously defined database. For example, the query

[DELETE ((TEMP = Works_on) and (PNO = 1))]

deletes all of the records in Works_on file in which the Project Number is 1.

e. The UPDATE Request

The UPDATE request modifies one or more records of a previously defined database. The query of the UPDATE specifies those records to be modified, and

a modifier is used to specify how they are to be updated. For example,

[UPDATE ((TEMP = Works_on) and (PNO = 2))
(HOURS = HOURS + 5)]

which increases the hours of those employees who worked on Project 2 by 5 hours.

III THE DATA FLOW QUERY LANGUAGE

A. BACKGROUND

The Data Flow Query Language (DFQL) was proposed and implemented by Wu and Clark in 1991 as a graphical/visual interface query language to the relational model based on a dataflow paradigm (Clark, 1991). It retains the capabilities of existing query languages while providing an additional facility to easily extend the language. This facility allows users to create new operators in terms of existing primitive operators. In addition to the relationally complete operators, DFQL also includes aggregate functions, thus providing query facilities beyond first-order predicate logic. The following goals are met by DFQL for a visual database interface (Wu, 1991):

- Employ a fully graphical environment as a user-friendly interface to the database.
- Sufficient expressive power and functionality, including relational completeness.
- Ease-of-use in learning, remembering, writing, and reading the language's constructs.
- Consistency, predictability, and naturalness (in both syntax and function).
- Simplicity and conciseness of features.
- Clarity of definition and lack of ambiguity.
- Ability to modify existing queries to form new ones incrementally.
- High probability that users will write error-free queries.
- Operator extensibility - allow users to create new operators in terms of existing ones.

DFQL achieves these goals by using the following approaches (Wu, 1991):

- Complete faithfulness to relational algebra and maintenance of the requirements for operational closure.
- Elimination of range variables from queries.
- Elimination of nesting in query constructs.

The implementation of operational closure is key to implementing large and complex queries in DFQL, as the output from each operator is always a relation. However, this fundamental characteristic of DFQL is not so easily translated into ABDL. This problem and its analysis is discussed further in Chapter V.

In addition to the above implementation approaches, the dataflow paradigm allows the user to treat relations as abstract entities operated on by relational operators. Therefore, users can compose their queries knowing the operators of relational algebra but without concern for the details of how the operations are carried out.

DFQL is implemented on a Macintosh II, using the programming language Prograph. The Macintosh allows DFQL to take advantage of the visual interface environment, while Prograph provides a visual dataflow structure similar to the dataflow paradigm approach taken for *DFQL. This dataflow structure allows a user to define a* query by graphically connecting specified DFQL operators. The operator arguments flow from the bottom or "output node" of the operator into the top or "input node" of the next operator (see Figure 3). Once the required input data becomes available, the operator

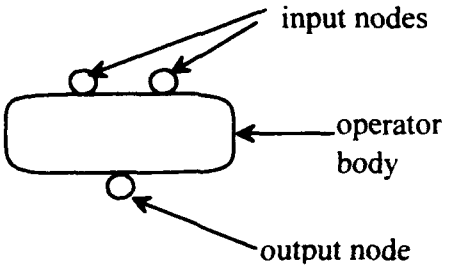


Figure 3. DFQL Operator Construction

23 may fire to execute the specific operation. The flow between operators is defined by the user by drawing lines between input and output nodes of the operators.

B. DFQL OPERATORS

1. Basic Primitive Operators

Basic operators in DFQL are derived from the requirements for relational completeness, that is, it has the expressive power of first-order predicate calculus (Wu, 1991). This means that the following operations must be implemented: (with the DFQL operator name) selection (select), projection (project), Cartesian product (join), union (union), and difference (diff). In addition to implementing these five operations, DFQL implements a sixth one, group count (groupcnt). This operator provides for simple aggregation. It also offers a solution for the problem of universal quantification in SQL; SQL only supports this concept indirectly. With these six basic primitives, DFQL supports orthogonality, which in turn makes it easier to use both syntactically and semantically (Wu, 1991).

A discussion of each of the operators is provided below. Also included are illustrations of each of the operators. In addition to these primary basic operators, other DFQL objects are included in the illustrations (Cince, 1993). These objects are represented by a line drawn underneath the text, with an output node under the line. These additional objects are generally defined as follows:

- textual objects - conditions, attribute lists, or some other syntactic item to be used in the operation.
- relation - name of a relation, or an instance of the specific relation, fed as input to an operator.

a. Select

This operator implements the relational algebra operation *selection*, or in relational algebra notation $\delta_{<condition>}(<relation>)$. It retrieves tuples from the relation which meet the condition, while removing duplicate rows. The result is also a relation, maintaining operational closure. The DFQL implementation is provided in Figure 4.

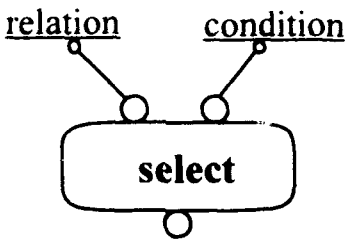


Figure 4. DFQL Select Operator

b. Project

This operator implements a relational algebra *projection*, or in notation form, $\pi_{<attribute\ list>}(<relation>)$. The attribute list contains those attribute names which are to be retrieved from the relation separated by commas. The resulting relation contains only those attributes specified in the attribute list. This operation also eliminates duplicates in the result. The DFQL implementation is provided in Figure 5. The *project* operator in DFQL can also be used to change attribute names in the resulting relation. In

the attribute list an equality condition such as *Dept_Num* = *DNUM* can be used to change the attribute name *DNUM* to *Dept_Num*.

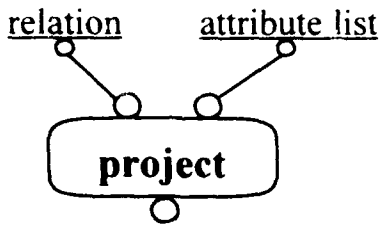


Figure 5. DFQL Project Operator

c. Join

The DFQL *join* operator implements the relational algebra operation *theta-join*. Notationally, this is < *relation*1 > *< *condition* > < *relation*2 > . The resulting relation is made up of all the attributes from <relation 1> and <relation 2>. The DFQL implentation is provided in Figure 6. If no condition is given, the join essentially becomes a cartesian product of the two relations. The join condition, if specified, uses basically the same syntax as the WHERE clause in SQL. However, if both relations have the same named attribute used in the condition, range variables must be used. These are limited to r1 (for <relation 1>) and r2 (for <relation 2>). For example, a join on *DNUM* in two relations would be specified as *r1.DNUM* = *r2.DNUM*.

The result of the DFQL operation *join* retains all of the attributes of the input relations. Therefore, when two attributes do share the same name, special handling must occur. In the resulting relation, one attribute name stays the same (from <relation 1>) whereas the attribute name from <relation 2> is appended with a "1". Using the above example, the output relation would have an attribute named *DNUM* and one named *DNUM1*.

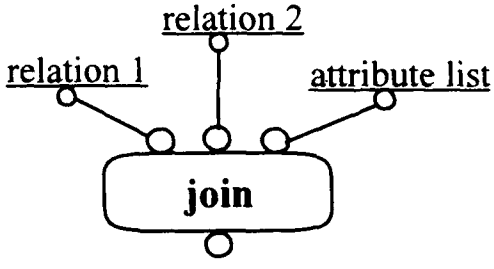


Figure 6. DFQL Join Operator

d. Union

DFQL implements the relational algebra operation *union*, with notation $< relation1 > \cup < relation2 >$, as illustrated in Figure 7. This operation combines the tuples of both relations, and eliminates duplicates, to provide the resulting relation. Since the tuples are combined in this operation, both input relations must be union compatible. By this, we mean that both relations must have the same number of attributes, and the corresponding attributes must have compatible data types.

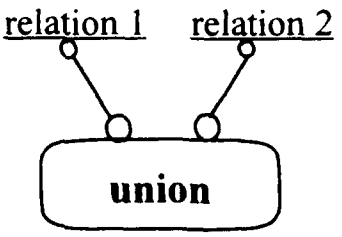


Figure 7. DFQL Union Operator

e. Difference

This operator implements the relational algebra operation *difference*. The notation is $< relation1 > - < relation2 >$. This operation returns a relation which contains all the tuples in relation 1 but not in relation 2. And as in the *union* operator, both relations must be union compatible.

f. Group Count

The *groupcnt* operator, although not directly based on relational algebra, is provided as a primitive operator to add some simple aggregation capabilities to theprovided as a primitive operator to add some simple aggregation capabilities to the language. Basically, *groupcnt* counts the number of tuples in a particular user-specified grouping. An illustration is provided in Figure 8. The grouping attributes can either be one attribute or a comma separated list of attributes. The result is a relation with the grouping attributes listed in the same order specified along with the attribute name

provided as the count attribute. This count attribute is an integer representing the number of tuples in the specific grouping, and is provided for each grouping attribute.

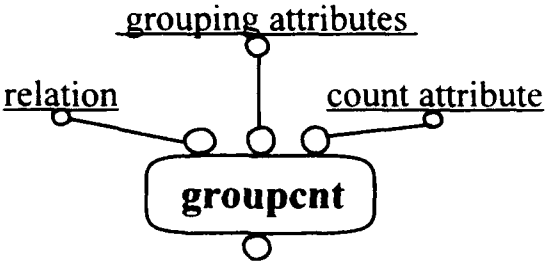


Figure 8. DFQL Group Count Operator

2. Non-Basic Primitive Operators

There are several other primitive operators implemented in DFQL, which perform special operations. Most can be implemented by user-defined operations using the basic primitives. However, DFQL builds them into the language to take advantage of the underlying DBMS which my already provide the operation. This makes the system easier to use, and slightly reduces the overhead required to interpret user-defined operations. A list of these operations and brief descriptions is provided below.

- *Intersect* - Relational algebra operation of *intersection*.
- *GroupALLsatisfy* - Used for simple universal quantification.
- *GroupNONEsatisfy* - Opposite of above. Gives the grouping attributes only if none of the tuples satisfy the condition.
- *GroupNsatisfy* - Specifies exactly how many of the tuples in the group must satisfy the given condition.
- *GroupMin* - Finds the minimum value of the specified attribute according to the grouping attributes.
- *GroupMax* - Similar to above, but finds the maximum value.

- *GroupSum* - Adds all of the aggregated attribute's values in each group, based on the grouping attributes.
- *GroupAvg* - Calculates the average of the given aggregate attribute.

In addition to the above, DFQL provides a *display* operator which prints the results of the query to the screen.

These operators have not been translated into ABDL for two reasons. First, with the basic primitives available, the user can define the other operations as needed. Secondly, the underlying DBMS is the ABDM. And as described in Chapter II, there are only five basic query types provided by the ABDL and from which all transactions can be defined. Therefore, there underlying DBMS does not already provide any of these operations by themselves, however they can be built from the basic operations.

C. CHARACTERISTICS OF DFQL

DFQL provides many improved features as a graphical interface to the relational database model. These characteristics are based on the dataflow structure and orthogonality DFQL employs in its implementation. Each of these characteristics combine to provide the user the ability to easily express simple, as well as complex, queries intuitively. Each of theses characteristics is examined in more detail below.

1. Extensibility

This important benefit of DFQL allows users to create new operators in terms of existing ones. The user may extend the query language either through the use of the DFQL primitive operators and/or the user's own previously defined user-operators. This

extensibility is achieved without decreasing the orthogonality of the language. This is because user-defined operators are constructed by combining the DFQL primitives which have already been coded so to ensure that orthogonality is maintained (Wu, 1991). In addition, the user-defined operators are constructed so as to support operational closure, and therefore maintain their compatibility with other operators. This extensibility also allows for encapsulation. As users define their own operators, the details are encapsulated within the operation. This means that once a complex query has been correctly written and converted into a user-defined operator, that operator may be easily used without any need to know its internal structure.

2. Dataflow Structure

The visual features of DFQL are based on the dataflow paradigm. The dataflow diagram allows the user to represent complex queries in an intuitive manner. This is because relations are visualized as objects (in this case, relations) flowing from one operator to another. This provides a levels of abstraction which contributes to the ease-of-use of DFQL (Wu, 1991).

3. Incremental Querying

Another key feature of DFQL which also supports ease-of-use is the ability to incrementally construct queries. Incremental construction allows a user to build a query part by part while determining the results of each part. This feature is related to the dataflow structure in that each dataflow represents an actual relation that can be displayed as a partial result (Wu, 1991). These partial results can be returned from any point in the

query to help the user verify or debug the query. Incremental querying is possible because of operational closure of all DFQL operators. The user knows that the output from any operator will always be a relation. And since one of the inputs to an operator must be a relation, the result of one operator is easily combined with another to form more and more complex queries.

4. Visual Interface

The visual interface is probably the main motivation behind DFQL, and therefore represents a critical feature of the language. All of the features mentioned above are possible because of the visual interface. As stated by Wu and Clark (Wu, 1991), "Allowing the user to interactively manipulate the DFQL query on the computer screen give a spatial or two-dimensional representation of the query that is lacking from any textual query language." The visual interface encourages the user to take advantage of the other ease-of-use features provided by DFQL. Screen real estate can become an issue, especially as queries become more complex. However, this problem exists with any visual interface. And the benefits of the visual environment easily outweigh the problems.

D. CONCLUSION

The technical features and structural characteristics of DFQL have been presented in this chapter. The advantages of DFQL can be seen from its visual interface, its graphical dataflow structure, and its set of primitive operators. These combine to make

DFQL a unique query language, one with the ability to easily express both simple and complex queries in an intuitive manner (Clark, 1992). In the next chapters, the translation of DFQL to ABDL is discussed. The problems associated with this translation are discussed with relation to the features inherent to DFQL, as well as the ease of translation.

IV TRANSLATION

As presented in Chapter II, the MDBS provides a unique and valuable database system. Allowing users to create and interact with a variety of databases using their choice of database models and languages was shown to be highly flexible as well as cost effective. And in Chapter III, the many benefits of the graphical interface language DFQL were described in detail. In order to implement DFQL as an improved interface to the MDBS, it must first be translated into the system's kernel data language, that is, the attribute-based data language (ABDL).

This chapter explains the translation and the mappings required to generate the correct ABDL syntax from the DFQL code, i.e., where SQL statements were previously generated, now ABDL commands are generated. The actual code is provided in Appendix B. The user interface was not changed at all in order to maintain true portability and standardization. Since DFQL is currently implemented using the Prograph programming language, all translations were also made in Prograph, and the DFQL syntax remains the same. Problems encountered in the translation will be discussed in the next chapter, along with recommendations and solutions. The problems were primarily due to the objective of maintaining the user interface of DFQL (i.e., ensuring portability), and the functionality of the MDBS (i.e., those functions provided by the kernel data model and language - ABDM/ABDL).

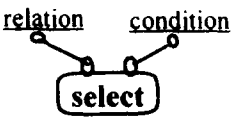
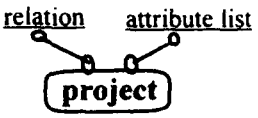
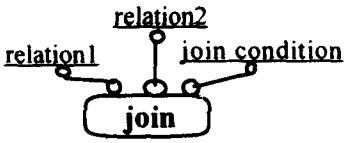
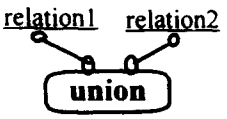
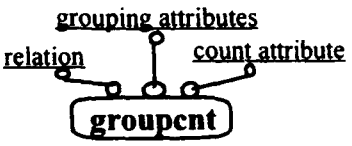
DFQL	ABDL
	[RETRIEVE (TEMP = relation) and (condition) (attributes)]
	[RETRIEVE (TEMP = relation) (attribute list)]
	[RETRIEVE (TEMP = relation1)(attributes) COMMON (join condition attribute) RETRIEVE (TEMP = relation2)(attributes)]
	[RETRIEVE(TEMP = relation1)(attributes)] [RETRIEVE(TEMP = relation2)(attributes)]
	[RETRIEVE (TEMP = relation) (COUNT(grouping attributes))]

Figure 9. DFQL to ABDL Translation

The first input is the same as in DFQL. The relation is provided by the user, from which the relation name is extracted. In ABDL each relation name is part of the first attribute-value pair (as discussed in Chapter III), and it identifies the file or *template* which contains the records of that relation. Hence, the *attribute* TEMP identifies the *value*, or relation name. This is the qualification of the ABDL operator and is used, in this case, to specify the relation on which to operate. Therefore, the predicate (TEMP = relation) will locate the records of the relation on which the Select will operate. This first predicate is combined in disjunctive normal form with the condition(s), which is the second input provided by the user.

The third item required for this query is a list of attributes. This list is not provided by the user because the Select operator should return all of the attributes in the relation. However, in ABDL the list of attributes desired in the output must be explicitly stated. Therefore, in the translation, additional code was required in order to generate this list of attribute names. This code is provided in Appendix B, Figures 15 - 18. Basically, the list of attributes is obtained from the relation object and then the name of each attribute is selected and put into another list. This list is then provided to be part of the generated ABDL statement.

The result of this ABDL statement will be all the records in the relation which meet the user-supplied condition. These records will be output using the same attribute-value pair which defines the ABDL database. An example using the sample database in Appendix A is provided in Figure 10.

A note should be made at this point about the case used in the ABDL statement. DFQL generates relation names, attribute names, etc. in lower-case. And since the MDBS is case-sensitive, the same information in the ABDL statements must also be in lower case. This requires that the database be loaded with relation names and attribute names in the same case in order to be compatible with DFQL. By preserving the case of the schema, the user interface remains standard, and portability is maintained.

QUERY:

[RELATION (TEMP = Works_On) and
(pno = 1) (essn, pno, hours)]

OUTPUT:

<TEMP, Works_On>, <essn, 123456789>, <pno,1>, <hours, 33>
<TEMP, Works_On>, <essn, 453453453>, <pno,1>, <hours, 20>

Figure 10. Example ABDL Select

2. The Project Operator

The Project operator allows for the most direct translation of all of the basic primary DFQL operators. The two inputs provided by the user are the same as the two used in the ABDL statement. The relation name is again extracted from the relation and combined in the first predicate with TEMP. Next, the attribute list provided by the user is passed directly to the ABDL statement (see Figures 19 - 20). No other manipulation of the relation is required. The result of this statement is a list of all records from the relation

with only those attributes specified in the attribute list. An example query and output is provided in Figure 11 to show how the ABDL project statement works.

QUERY:

[RETRIEVE (TEMP = Department) (dname, mgrssn)]

OUTPUT:

<dname, Research>, <mgrssn, 333445555>
<dname, Administration>, <mgrssn, 987654321>
<dname, Headquarters>, <mgrssn, 888665555>

Figure 11. Example ABDL Project

3. The Join Operator

The translation from the DFQL Join to ABDL requires the use of the RETRIEVE-COMMON operator from ABDL. This translation requires additional manipulation of the code in order to generate the correct ABDL statement from the user provided inputs. This code is provided in Appendix B, Figures 21 - 24.

Two relations are input by the user, and both relation names are extracted. Each is paired separately with the attribute TEMP to locate the records of the relations to be joined. The list of attributes from each relation must also be retrieved, however, as in the Select operator, this is not provided by the user. The same code which was used to generate this list in the Select operation is used here to generate lists of attributes for both relations. The join condition attributes are the two attributes obtained from the input condition. These two attributes are combined with COMMON to form the join predicate. The two relations are then merged on the common attribute values, resulting in an

equi-join. The output is a list of all records from the two relations which meet the join condition, in attribute-value pairs. An example of this query is provided in Figure 12. As shown in this example, the two common attributes do not have to have the same name.

QUERY:

[RETRIEVE (TEMP = Department) (dname, dnumber)
COMMON (dnumber, dnum)
RETRIEVE (TEMP = Project) (pname, pnumber)]

OUTPUT:

<dname, Research>, <dnumber, 5>, <pname, ProductX>, <pnumber, 1>
<dname, Research>, <dnumber, 5>, <pname, ProductY>, <pnumber, 2>
<dname, Research>, <dnumber, 5>, <pname, ProductZ>, <pnumber, 3>
<dname, Admin >, <dnumber, 4>, <pname, Computer>, <pnumber, 10>
<dname, Admin >, <dnumber, 4>, <pname, Newbenefit>, <pnumber, 30>
<dname, Hdqtrs >, <dnumber, 1>, <pname, Reorganize>, <pnumber, 20>

Figure 12. Example ABDL Join

4. The Union Operator

The translation from the DFQL Union operator to ABDL results in a modified relational union. The modification is that ABDL does not provide for the removal of duplicate rows, if any exist. This is because each relation is retrieved separately through the RETRIEVE query. The syntax is similar to the Select operation, except that there is no condition necessary for the retrieve. The relation is input by the user, and the attribute list is extracted from the relation object to form the query. Because this is a combination of two relations using two separate RETRIEVE queries, the relations are not required to be union compatible. An example of this query is in Figure 13. The source code is provided in Appendix B, Figures 25 - 28.

QUERY:

[RETRIEVE (TEMP = Department) (dname, dnumber)]
[RETRIEVE (TEMP = Dept_Locations) (dnumber, dlocation)]

OUTPUT:

<dname, Research>, <dnumber, 5>
<dname, Administration>, <dnumber, 4>
<dname, Headquarters>, <dnumber, 1>
<dnumber, 1>, <dlocation, Houston>
<dnumber, 4>, <dlocation, Stafford>
<dnumber, 5>, <dlocation, Bellaire>
<dnumber, 5>, <dlocation, Sugarland>
<dnumber, 5>, <dlocation, Houston>

Figure 13. Example ABDL Union

5. The GroupCnt Operator

The GroupCnt operator counts the number of records which meet the grouping attribute requirement. The count attribute, which is used in DFQL to identify the result relation which stores the result of the count, is not used in the translation. The translation code is provided in Appendix B, Figure 29. Since ABDL does not produce a relation as output at this time, there is no need for a new relation name. The output from this query is the grouping attribute, and the number of records in with that attribute. An example is given in Figure 14.

QUERY:

[RETRIEVE (TEMP = Project) (COUNT (pnumber))]

OUTPUT:

< pnumber, 6>

Figure 14. Example ABDL GroupCnt

B. GENERAL NOTES

A couple of additional notes need to be made with regards to the code used from DFQL and for the translation. These involve some or all of the operator translations, and should be reviewed along with the code provided in Appendix B.

For the Select, Project, and Join operators there was quite a bit of additional code generated by DFQL which was not used in the translation to ABDL. Most of this additional code focused on the creation of "views", that is, the generation of temporary relations. As stated in Chapter III, DFQL generates these temporary relations in order to maintain operational closure. Therefore, a relation is always the output of every operator. Currently, ABDL does not support the ability to create views. One reason is due to the lack of communication between different requests in a transaction. That is, one request of a transaction cannot use the results of a previous request (Demurjian, 1987).

When the MDBS was initially developed, one major goal was to demonstrate that the key aspects of different data models could be supported in a single system (Demurjian, 1987). Since it is a research tool, the designers believed this to be sufficient. However, in order to be able to support more advanced data models and languages in the future, additional facilities such as creating views become essential. The additional code which is used by DFQL, and not used in the translation to ABDL was left in place so that when the additional facility of view creation is implemented, the additional translation work required will be minimal.

V ANALYSIS OF TRANSLATION

The implementation of DFQL on the MDBS requires what is basically an integration of the two systems. The first aspect of that implementation has been presented here - the translation of DFQL into the kernel language ABDL. Both DFQL and MDBS were designed to improve database interaction. The focus of this thesis research is to integrate the two systems, and thereby significantly improve the user interface for the relational database on the MDBS. However, since both of these systems were designed and implemented totally independently, each brought its own features and functional characteristics which had to be considered in the integration. And the fact that both systems are prototypes, and experimental, had an effect on the translation. For example, in the MDBS several SQL features were not implemented because the effort required to support them outweighed the current benefits, and detracted from the primary goal of demonstrating the feasibility of the system (Demurjian, 1987). Therefore, operations which seem simple in DFQL were not able to be implemented the MDBS as it is currently configured. And as a result, some of the beneficial features of DFQL were not able to be implemented.

In the rest of this chapter, the problems encountered in the translation of the two systems will be examined in more detail. Most of these were due to features which were not implemented in MDBS, but are instrumental to the effective operation of DFQL. Possible solutions and recommendations will also be discussed.

A. PROBLEM ANALYSIS

One of the most critical features not currently implemented on the MDBS is the ability to define views, or the creation of temporary relations in SQL. This feature was not implemented because it did not impact on the data-model transformations and the data-language translations of the original system (Demurjian, 1987). Again, this was because the MDBS was originally intended as a research tool, as opposed to a commercial product. The crucial issue with this inability to define views is the lack of support for operational closure.

Since the MDBS does not create views, the results of a query are those tuples or records which meet the requirements of the query. Therefore, temporary relations are not created. In DFQL, these views, or temporary relations, would contain the output from defined operations. And as described in Chapter III, this allows for easier extensibility of the language, because one input to all operations is a relation, and the output is always a relation. Since the MDBS does not support this feature, operations must be performed one at a time, and cannot be connected to other operators in order to extend the language. For example, a user cannot perform two *projects* on two relations, take that output and *join* it in a series of operations because the output from the *projects* are not relations (required as input to the *join*), but the records which meet the requirements of the *project* query. This significantly reduces the benefits of DFQL. It not only affects the extensibility of the language, it also reduces the benefits of incremental querying, as described in Chapter III.

The inability to create views in the MDBS causes another problem in the translation of DFQL - that of translating the *difference* operator. This operator is not translated into ABDL because temporary relations cannot be created and because the operator requires union compatibility. The temporary relations are necessary in order to obtain two relations which are union compatible, usually through *project* operations. These temporary relations are then compared to subtract those tuples which are in one relation, but not the other. However, since these temporary relations cannot be obtained, there is nothing to compare. Therefore, *difference* is not translated into ABDL.

There are several other operations which are limited on the MDBS which in turn limit the DFQL operations. One of these involves the *join* operation. The ABDL implementation of *join* only allows an *equi-join* and the joining of two relations at a time. Most joins required tend to be *equi-joins*, and therefore this is not considered a significant limitation. However, any limitation on the user results in less flexibility to manipulate the database. This also applies to the limitation of joining only two relations. Again, many times only two relations need to be joined, however, with large databases, this could be a severe limitation. Also, since operational closure cannot be implemented, the language cannot currently be extended to perform additional *joins* on the result of previous ones.

Another limitation involves the *union* operator. In the MDBS, the elimination of duplicates is not supported in addition to the inability to support a *union* of two different *select* operations (again, due to the inability to create temporary relations). Basically, the

translation to ABDL calls for two separate RETRIEVE statements, or in relational terms, two *select* operations in sequence. This basically accomplishes a mathematical set union, as opposed to a relational union, since duplicates are not eliminated. Also, since relations are not produced as output, users cannot use *projects* on relations to separate out desired attributes which would then be input to the *union* operator.

Other problems discovered during the translation were not as significant as those described above, however, they did impact the translation process. These included syntax limitations in ABDL which were not documented, but nonetheless caused considerable effort to be expended in the debugging of code. For example, the system is case sensitive. All code generated in DFQL must match exactly the code required for the MDBS and ABDL. Also, all data items (i.e., attribute names and values) are limited to 15 characters. For testing purposes, this is not a problem because test databases are relatively small. However, large and complex databases require abbreviations in order to be implemented, and this reduces ease-of-use and user comprehension.

One potential problem with DFQL is the way in which it is implemented. It is currently implemented on a Macintosh computer, using the Prograph programming language. As discussed in Chapter III, the Macintosh allows DFQL to take advantage of the visual interface environment. But more importantly to the constructs of DFQL, Prograph provides a visual dataflow structure which greatly enhances ease-of-use. Although these benefits are significant, they are limited in that the Prograph environment

is currently available only on a Macintosh system. To have real portability, the system should have the ability to be accessed from other platforms.

B. RECOMMENDATIONS

One major improvement to the MDBS would be to implement the ability to create views. This one change could provide solutions for many of the problems discussed above. Most importantly, it would allow for operational closure which is an instrumental feature of DFQL. This one enhancement would allow the major benefit of DFQL to be realized on the MDBS. Extensibility would be significantly improved by allowing the full implementation of all DFQL operators, as well as user-defined operators. The ability to create views would be an additional benefit for the entire system. This is because the kernel model and language (ABDM/ABDL) would be upgraded to support this feature, and therefore be available to all data models used on the system. Essentially, this change would make the MDBS, with DFQL implemented, a viable database system for more advanced data models, and much more complex databases.

An alternative method of creating operational closure would be to have DFQL handle the creation of temporary relations. By this, we mean that DFQL would take the results from one query, restructure it and send it back to the MDBS in the form of a file creation, with the new data inserted. The MDBS could then act on this new relation in whichever operation prescribed by the user in DFQL. This solution puts the burden on DFQL to ensure operational closure is maintained. This is not an optimal solution for several reasons. First, the updated DFQL would lack the orthogonality, which is

currently one of its key features, and one of the goals DFQL set to originally accomplish. It would make user defined operators much more difficult to construct, thereby losing the benefit of extensibility. Changing DFQL would also make the system much slower by significantly increasing the communication time between the two systems. And restructuring the code in DFQL would significantly reduce its maintainability and portability. Additionally, none of the other data models implemented in the MDBS would benefit from the change if it were made to DFQL only.

Other recommendations to the MDBS involve enhancements to the relational operations currently allowed. These would include providing all the functionality of a *join* operation, not just an *equi-join*, as well as allowing the joining of as many relations as necessary. The problem with the *union* operator would be corrected by the ability to create temporary relations. Also, it would be possible to implement the *difference* operator. The ultimate goal of these solutions would be to have a full set of relational operators available to the user, as well as all of the advantages of DFQL.

When these additional features are implemented in the MDBS, and it is set up to support more advanced data models, the syntactic constraints should also be removed. The most important of these would be the removing the limitation of *string* and *integer* data types. At a minimum, a *real* data type should be added, as well as a *date*. Additionally, the 15 character minimum constraint should be removed. Although it works fine for test purposes, in actuality most databases require much more flexibility in allocating space for attribute values. These changes would make the system much more

flexible and user friendly. As a result, larger and more complex models could be effectively loaded and manipulated by the system.

Finally, the change to the programming environment for DFQL is not currently an issue which can be resolved at this point. Although it is possible to write all code in C++, the environment would not necessarily be a graphic one. However, when new software for Prograph is made available for different environments (i.e. DOS, Windows, Unix), DFQL can be implemented as necessary. This would enhance the portability of DFQL, and make it much more widely available.

VI CONCLUSION

A. DISCUSSION

This thesis involved the design and translation of the Data Flow Query Language (DFQL) for the Multi-lingual, Multi-backend Database System (MDBS). DFQL is a user friendly, graphical interface designed for the relational data model using a dataflow paradigm. The MDBS is a database system which can effectively support multiple data models and their corresponding data manipulation languages.

The background information on both systems was discussed. In addition, their strengths were presented to illustrate the benefits to be gained from combining the two systems. The design and translation were then explained in detail, including the DFQL code. This was followed by the problems encountered during the translation, and recommended solutions.

The ABDL statement generation in DFQL was fairly straightforward. Once we gained an understanding of the Prograph programming language and the data flow paradigm, the required changes in code required were relatively simple. In addition, the debugging features of DFQL, specifically incremental construction and execution, allowed for a much more expeditious translation than what is normally available in other programming environments.

The problems encountered with the MDBS were described in Chapter V. This system has many outstanding features including the effective management of database systems growth, database performance, data sharing and resource consolidation (as

presented in detail in Chapter II). It is truly unique in the database community. The concept was to combine these two systems, and thereby realize benefits much more than just the sum of the two.

B. FUTURE RESEARCH

The obvious next step in this area of research would be to develop the communications required to connect the two systems. This interconnection between DFQL and the MDBS would include the design and implementation of DFQL and MDBS socket interfaces. The design of these sockets would include the capability to pass the ABDL strings generated by DFQL correctly to the MDBS for execution. Then the output from the execution would be similarly passed back to DFQL, and formatted and displayed according to the user's needs.

Once this step is completed, an in-depth human factors analysis could be conducted on the DFQL interface implemented on the MDBS backend. This analysis would be similar to the one proposed in (Wu, 1991). It could quantify the ease-of-use aspects as well as the flexibility provided by the MDBS. This could be a stand alone analysis, and/or one performed in comparison with other stand alone data models, and query languages.

Other aspects of future research concentrate solely on the MDBS. As discussed in Chapter V, it would be extremely beneficial to implement the ability to create views in the MDBS. This would add significant flexibility and extensibility to the system. When this modification is made, the basic set of DFQL primitives translated into ABDL could

be greatly expanded to include the non-basic primitive operators, as well as additional user-defined operations.

Although the ability to create views is one of the most important changes which could be made to the MDBS, the system overall should be revamped. Some of the syntactic requirements should be removed (i.e., the limitation of string length to 15). In addition, the documentation of the system has become outdated due to the many changes by research students over the past few years. Although many of these changes may have been small, if they are unknown, many unnecessary hours can be spent in debugging the system when it fails. An analysis of the current system, with complete documentation, would be extremely beneficial for all future research.

C. SUMMARY

The results of this thesis show that the implementation of DFQL on the MDBS would be a enhancement to the current system. A graphical interface is much more user-friendly than a text based one, and the data flow paradigm aids in the ease-of-use. In addition, the MDBS supports multiple database models, providing flexibility and cost-efficiency to the user. However, as also demonstrated, further enhancements need to be made to the MDBS to truly realize all of the benefits from DFQL.

APPENDIX A. SAMPLE DATABASE

This appendix contains the sample database used in queries and examples throughout this thesis. The database is *Company Database* from the textbook DATABASE SYSTEMS (Elmarsi, 1989).

EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERRSSN	DNO
John	B	Smith	123456789	09-JAN-55	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	08-DEC-45	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	19-JUL-58	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	20-JUN-31	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	15-SEP-52	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	31-JUL-62	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	29-MAR-59	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	10-NOV-27	450 Stone, Houston, TX	M	55000	null	1

DEPARTMENT

DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
Research	5	333445555	22-MAY-78
Administration	4	987654321	01-JAN-85
Headquarters	1	888665555	19-JUN-71

DEPT_LOCATIONS

DNUMBER	DLOCATION
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

PROJECT

PNAME	PNUMBER	PLOCATION	DNUM
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP
333445555	Alice	F	05-APR-76	DAUGTER
333445555	Theodore	M	25-OCT-73	SON
333445555	Joy	F	03-MAY-48	SPOUSE
987654321	Abner	M	29-FEB-32	SPOUSE
123456789	Michael	M	01-JAN-78	SON
123456789	Alice	F	31-DEC-78	DAUGHTER
123456789	Elizabeth	F	05-MAY-57	SPOUSE

WORKS_ON

ESSN	PNO	HOURS
123456789	1	33
123456789	2	8
666884444	3	40
453453453	1	20
453453453	2	20
333445555	2	10
333445555	3	10
333445555	10	10
333445555	20	10
999887777	30	30
999887777	10	10
987987987	10	35
987987987	30	5
987987987	30	20
987987987	20	15
888665555	20	null

APPENDIX B. SOURCE CODE

This appendix contains the source code for the translation of DFQL to ABDL. The only code provided are those portions of the operations which required a change in the current DFQL code. The order of the operations are SELECT, PROJECT, JOIN, UNION, and GROUP COUNT.

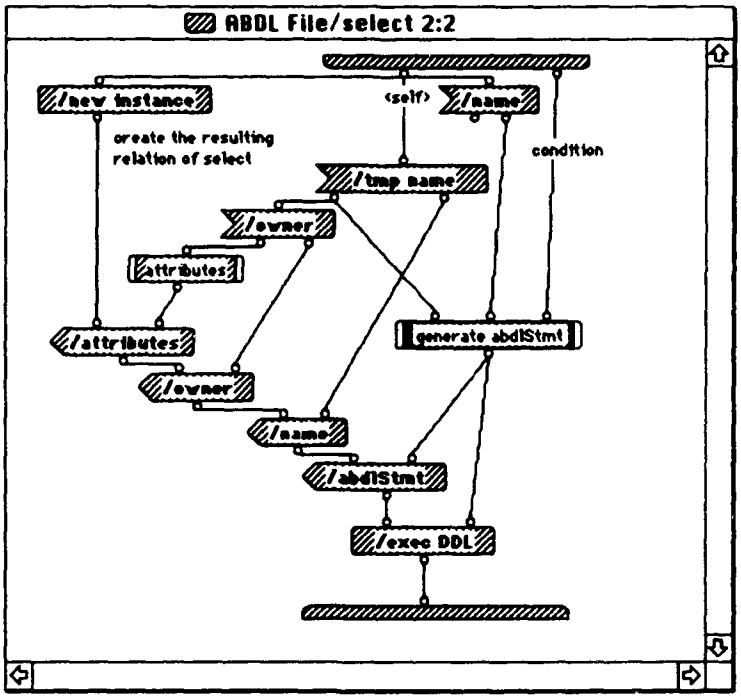


Figure 15. DFQL Generation of ABDL Select

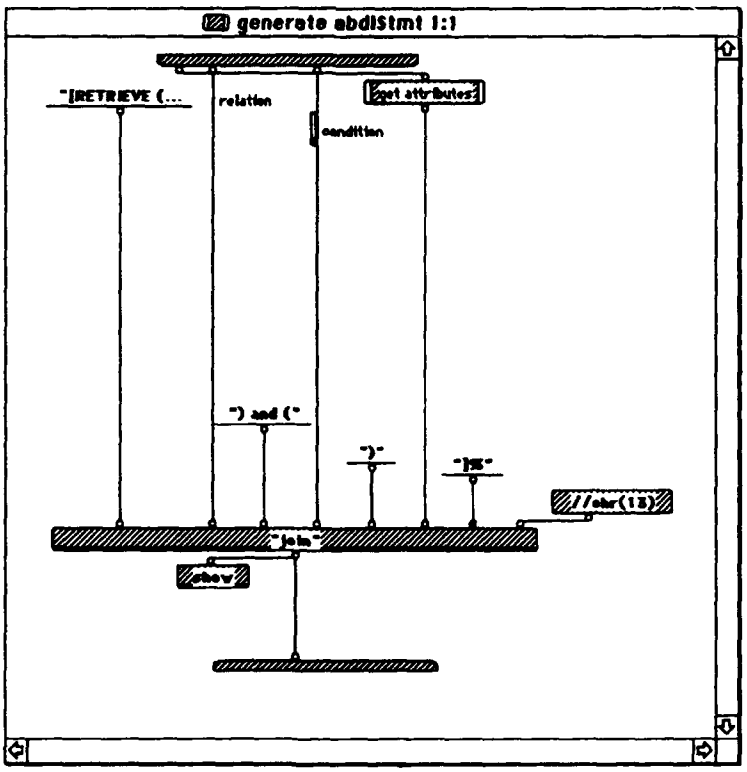


Figure 16. Generate ABDL Select Statement

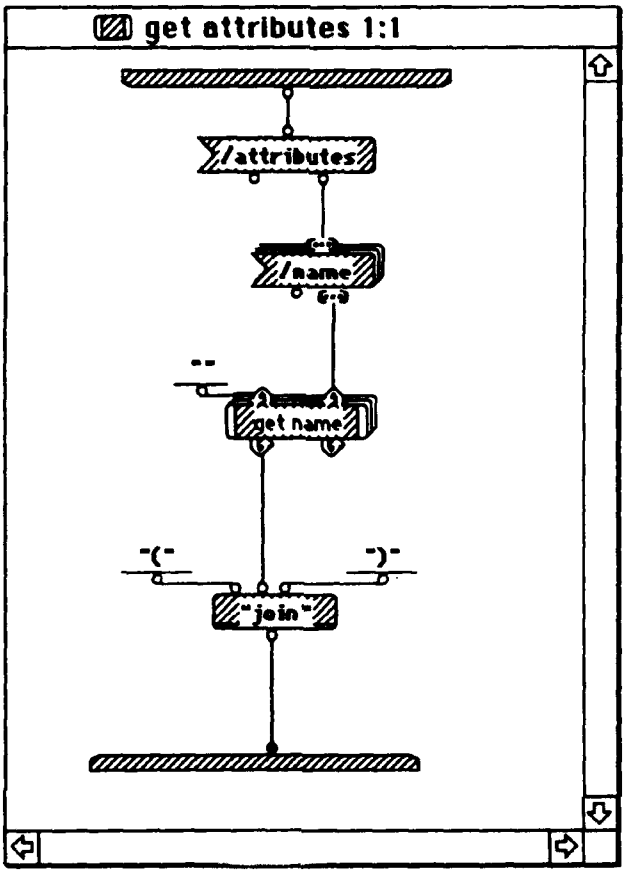


Figure 17. Generate List of Attributes

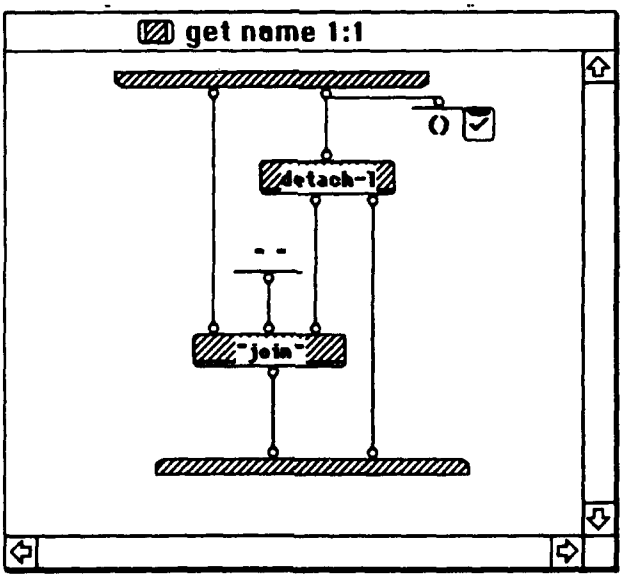


Figure 18. Generate Attribute Names

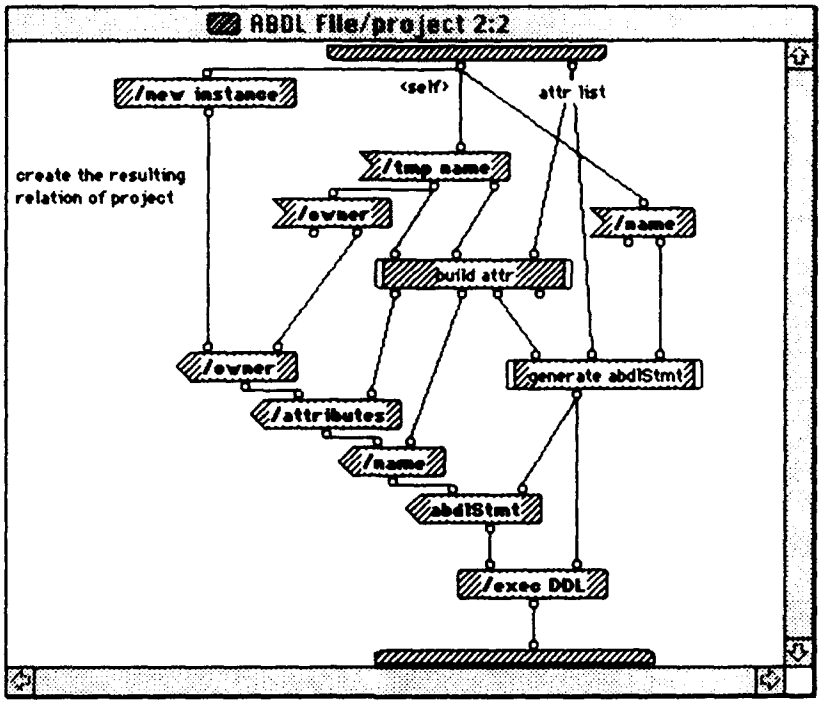


Figure 19. DFQL Generation of ABDL Project

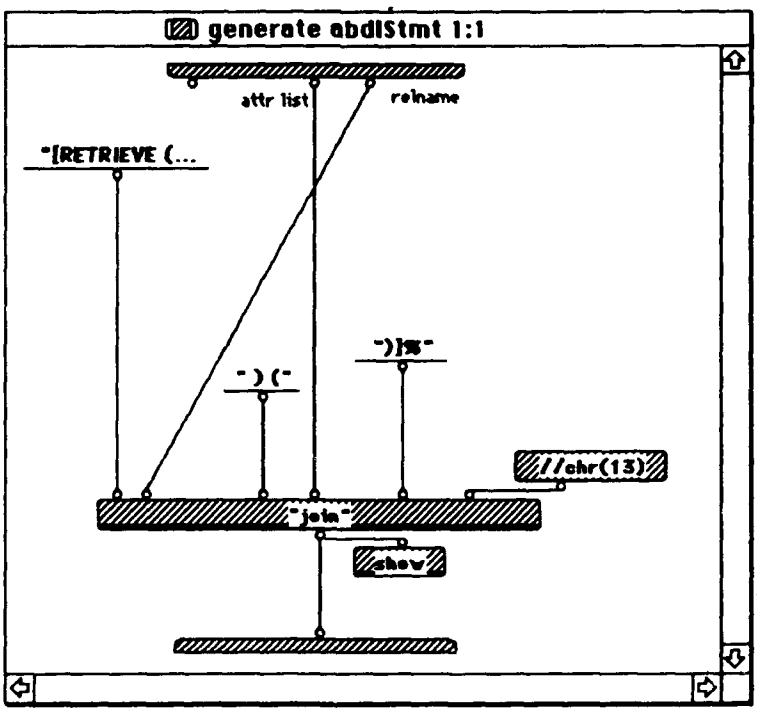


Figure 20. Generate ABDL Project Statement

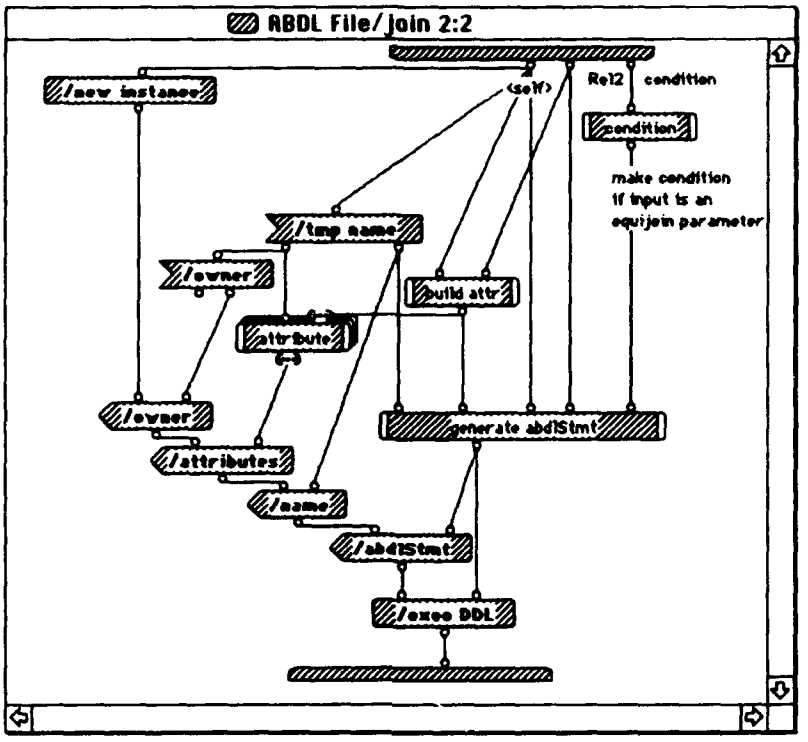


Figure 21. DFQL Generation of ABDL Join Statement

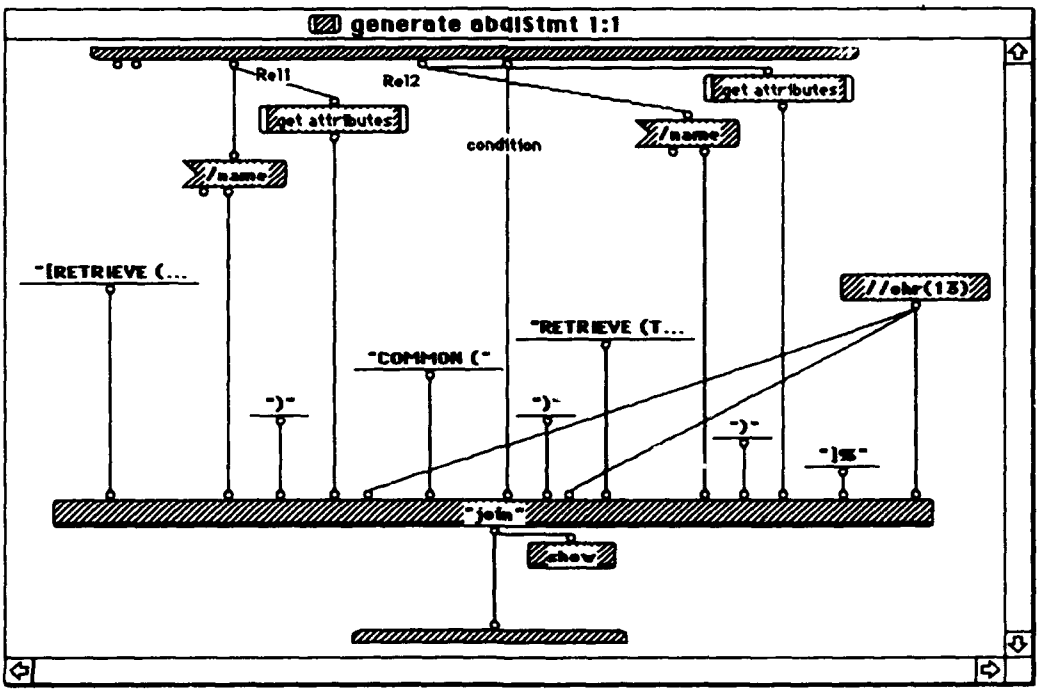


Figure 22. Generate ABDL Join Statement

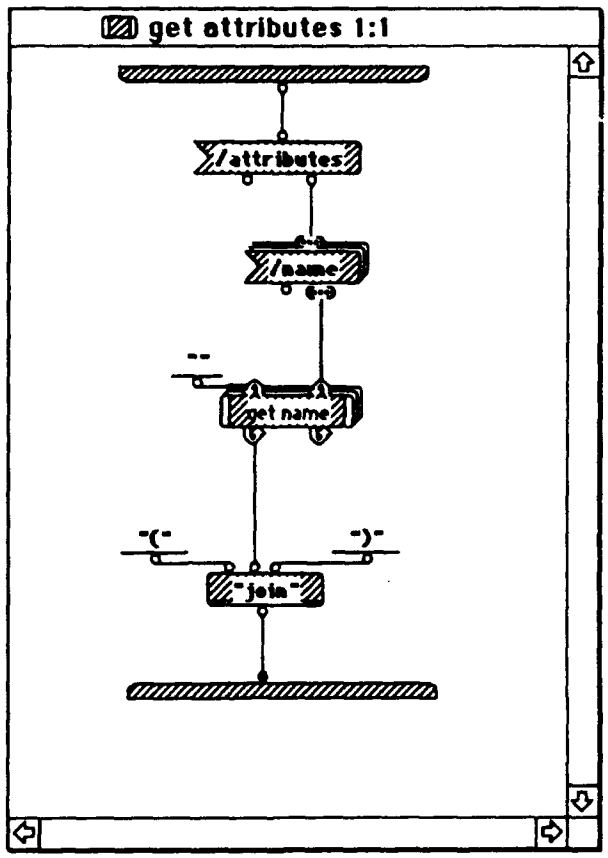


Figure 23. Generate List of Attributes

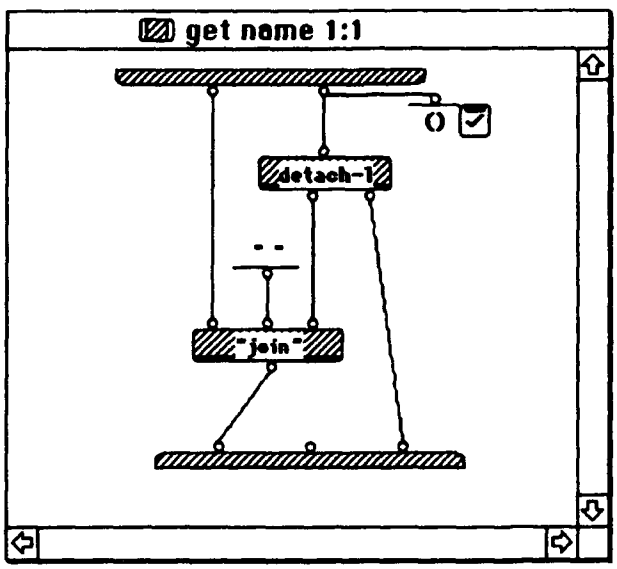


Figure 24. Generate Attribute Names

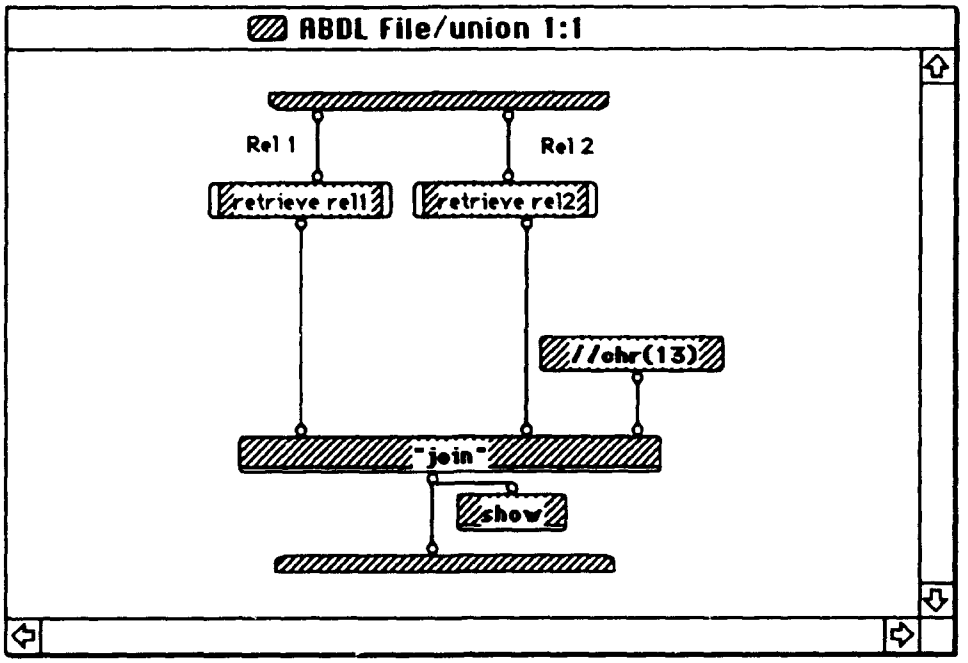


Figure 25. DFQL Generation of ABDL Union

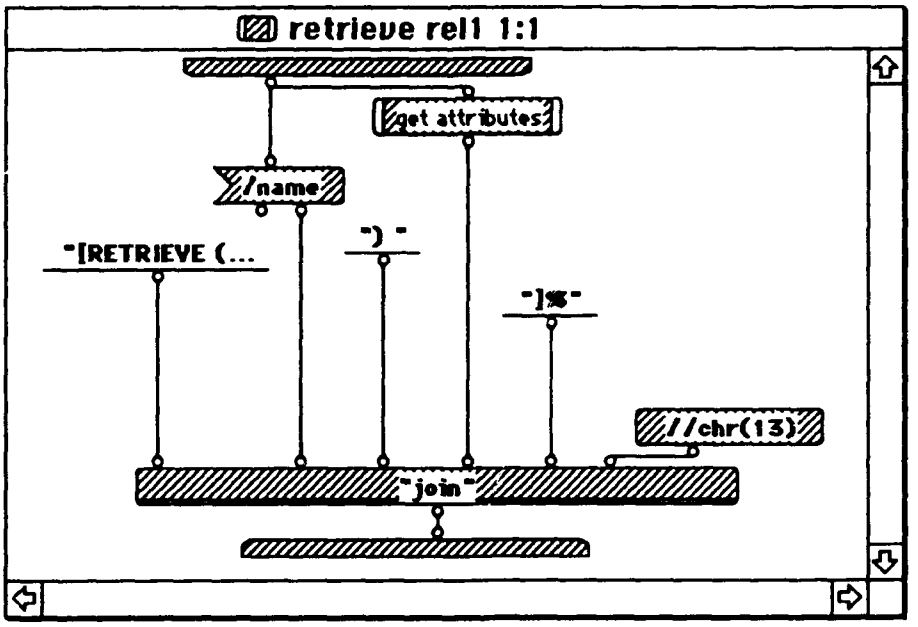


Figure 26. Generate ABDL Union Statement

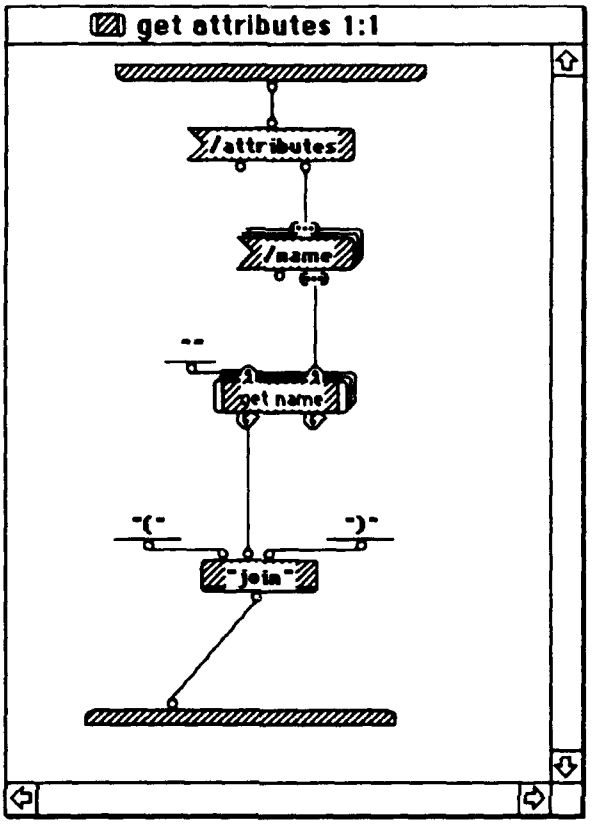


Figure 27. Generate List of Attributes

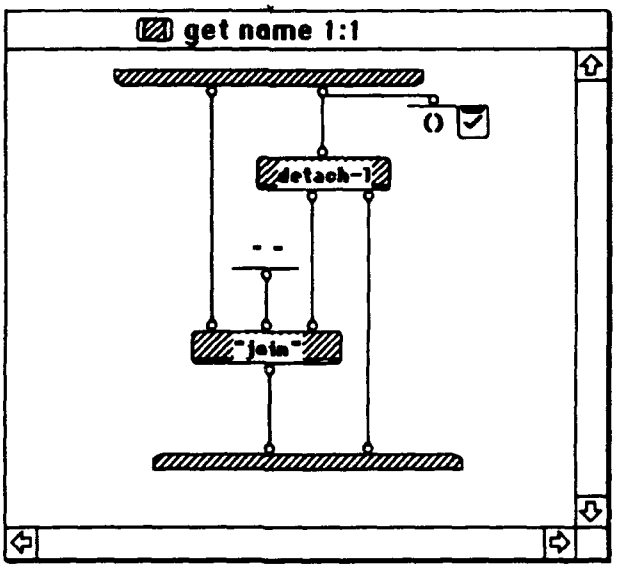


Figure 28. Generate Attribute Names

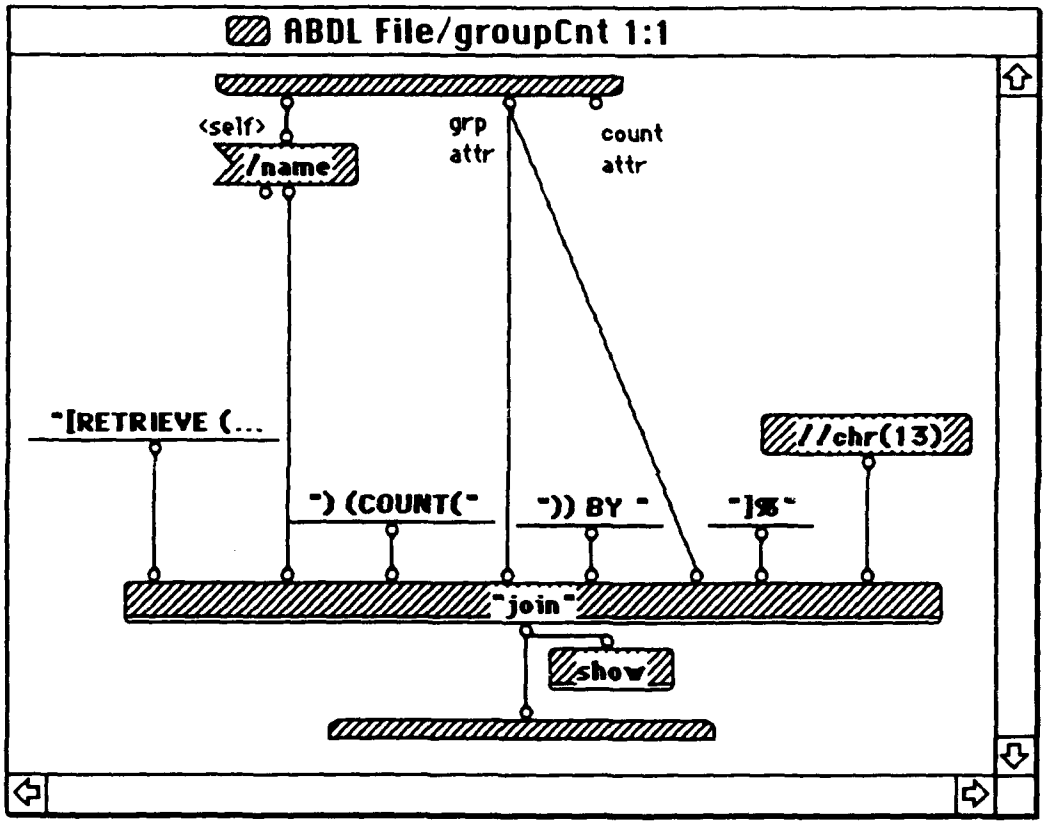


Figure 29. DFQL Generation of ABDL Group Count

LIST OF REFERENCES

Banerjee, J. and Hsiao, D. K., *DBS Software Requirements for Supporting Relational Databases*, Technical Report, Ohio State University, 1977.

Cince, Turgay, *Design and Implementation of a Query Editor for the Amadeus System*, Master's Thesis, Naval Postgraduate School, 1993.

Clark, Gard J., *DFQL: A Graphical Dataflow Query Language*, Master's Thesis, Naval Postgraduate School, 1991.

Demers, William A. and Rogelstad, Jon M., *The Design and Implementation of a Functional /Daplex Data Interface for the Multimodel and Multilingual Database System*, Master's Thesis, Naval Postgraduate School, 1994.

Demurjian, Steven A., *The Multi-Lingual Database System - A Paradigm and Test-Bed for the Investigation of Data-Model Transformations, Data-Language Translations and Data-Model Semantics*, Dissertation, Ohio State University, 1987.

Demurjian, Steven A. and Hsiao, D. K., "New Directions in Database-Systems Research and Development," *Proceedings of the International Symposium on New Directions in Computing*, 1985.

Elmarsi, R. and Navathe, S., *Fundamentals of Database Systems*, The Benjamin/ Cummings Publishing Company, Inc., 1989.

Hall, James E., *Performance Evaluations of a Parallel and Expandable Database Computer - The Multi-Backend Database Computer*, Master's Thesis, Naval Postgraduate School, 1989.

Holste, Steven T., *The Implementation of a Multi-Lingual Database System - - Multi-Backend Database System Interface*, Master's Thesis, Naval Postgraduate School, 1986.

Hogan, Thomas R., *Interconnection of the Graphics Language for Database System to the Multi-Lingual, Multi-Model, Multi-Backend Database System Over an Ethernet Network*, Master's Thesis, Naval Postgraduate School, 1989.

Hsiao, D. K. and Harary, F., "A Formal System for Information Retrieval from Files," *Communications of the ACM*, vol. 13, no. 2, 1970.

Macy, Griffin N., *Design and Analysis of an SQL Interface for a Multi-Backend Database System*, Master's Thesis, Naval Postgraduate School, 1984.

Rollins, Richard E., *Design and Analysis of a Complete Relational Interface for a Multi-Backend Database System*, Master's Thesis, Naval Postgraduate School, 1984.

Wu, C. Thomas and Clark, Gard J., *DFQL: Dataflow Query Language for Relational Databases*, Unpublished Paper, Naval Postgraduate School, 1991.

INITIAL DISTRIBUTION LIST

		Number of Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Dudley Knox Library Library, Code 052 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Computer Technology, Code 37 Naval Postgraduate School Monterey, California 93943-5002	1
4.	C. Thomas Wu, CS/Wq Department of Computer Science Naval Postgraduate School Monterey, California 93943-5002	2
5.	David K. Hsiao, Code CS/Hq Department of Computer Science Naval Postgraduate School Monterey, California 93943-5002	1
7.	Ms. Doris Mlezko Code P22305 NAWCWPNS Point Magu, CA 93042-5001	2
8.	CPT Nancy C. Free PO Box 3180 Ft. Leavenworth, KS 66027	2