AN ABSTRACT OF THE THESIS OF

Richard Goodemoot for the degree of Master of Science in Computer Science presented on June 11 1986.

Title: A Query Facility for Allegro

Redacted for Privacy

Abstract approved: _____
Earl F. Ecklund, Jr.

Allegro is a network database management system being developed at Oregon State University. This project adds a user friendly query facility to the system.

The user is presented with pictorial display of the network records and a query interface modeled on the Query–By–Example system. By request the user may be shown the network sets of the query schema. When necessary the user may specify query navigation of the network schema. While implemented and functional, this facility should be considered as a feasibility study for a full query system on a network data base.

To provide the desired display this facility is implemented on a system separate from the main Allegro system and uses a communication interface to it. This facility is a Smalltalk implementation.

A Query Facility for Allegro

by

Richard Goodemoot

A THESIS

submitted to

Oregon State University

In partial fulfillment of the requirements for the degree of

Master of Science

Completed June 11, 1986

Commencement June 1987

APPROVED:

Redacted for Privacy

Adjunct Professor of Computer Science in Charge of Major

Redacted for Privacy on behalf of Walter Rudd

Chairman of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis is presented June 11, 1986

Typed by Richard Goodemoot

TABLE OF CONTENTS

LIST OF FIGURES

A Query Facility for Allegro

Chapter 1

Introduction

Allegro is a network database management system. Allegro has been developed and is currently being enhanced by the students of Dr. Ecklund at the Computer Science Department at Oregon State University. For query purposes, the current Allegro system provides only an embedded DML.

The purpose of this work is to develop the query facility for Allegro that is called *Allegro Query*. The first problem is to define what is needed in a network query facility. There is the constraint that Allegro is a network database management system. That is any schema to be queried has an underlying structure which must be considered. The author is impressed with the Query-By-Example user interface as being intuitive and easy to use. He considers TTY style line-at-a-time interfaces a relic of the old days. With this mind set, the author chose to attempt a visual query facility for Allegro that is modeled on the Query-By-Example user interface.

This paper is organized as follows. This introduction has a section on Allegro as a network database and a section on Allegro Query. Chapter 2 is a brief discussion of query systems in general. Chapter 3 is a discussion of query on a network database. Chapter 4 gives the user interface to Allegro Query. Examples of Allegro Query are in chapter 5. Chapter 6 describes some of the internal structure of Allegro Query. Chapter 7 lists possible future work to improve Allegro Query.

1.1 Network Database Management and Allegro

Network database management[11,12,16] normally refers to systems following the series of proposals of the Data Base Task Group (DBTG) of the Conference on Data Systems Languages (CODASYL)[10]. There is a Data Definition Language (DDL) that describes the logical network structure (schema). There is a Data Storage Definition Language (DSDL) the definition of which is in the process of being split from the DDL. There is a Subschema Data Definition Language (Subschema DDL) for defining views on the schema. Finally there is a Data Manipulation Language (DML) that provides a set of commands for manipulating data in the database. The Allegro system definition has subsets of several of these languages. There is SDDL/SDSDL (S for subset) to describe the schema. There is no view definition. There is a DML whose Find statements are close to the record selection portions of CODASYL Find without the for-update and retaining phrases.

The network model uses the terms record, set, and fields. A *record* usually models an entity of the data model. The data of the database is stored in record instances. The attributes of an entity are the *fields* of the record used for the entity. The *network set* represents a many to one relationship of one record type to another record type. It is not a mathematical set. The 'one' is called the owner record type while the 'many' is(are) called the member record type(s). Attributes of relationships modeled by a network set will be attached to one of the record types. A many to many relationship is handled with a pair of sets and a link record type (Figure 1.2.1).

At the DML level a set is implemented by some type of pointer scheme from an owner instance to (at any instant) one of the member record instances. Which of the members is handled by the concept of currency. There are several types of currencies. Allegro has current of each record type, current of each set, and current of run-unit (the last record touched). The user of a network DML must normally be aware of currencies. To look ahead, Allegro Query is an exception to this need.

1.2 Allegro Query

There are two primary concerns in the use of a visual query facility for a network database. First the user must be shown the schema he is interested in querying. It is not enough that the user simply be shown some representation of the schema. The user must be given the freedom to see a representation of his schema as he would diagram it. That is, the user must be able and should be encouraged to position the schema representation as he sees fit.

Since Allegro is a network database, Bachman diagrams[9] (figure 1.2.1) are used to describe Allegro schemas. Four types of 'things' in a network schema may be described by these diagrams. An *entity* is a particular object being considered. An *entity class* is a group of entities sufficiently similar in terms of their attributes to be considered collectively. The named rectangles in the diagram are entity classes. That is rectangle 'BOOK' stands for a collection of books. An *entity set* is a different type of entity grouping. It associates a group of entities from one entity class (subset of the class) with one entity from a different entity class. It is a one to many relationship between members of separate entity classes. A *set class* is a group of entity sets sufficiently similar in terms of their attributes to be described collectively. The lines with an arrow in the diagram represent a set class. The named line 'WRITTEN_BY' stands for the class of entity sets each of which links a book to the intermediate entities, members of BOOK_AUTH.

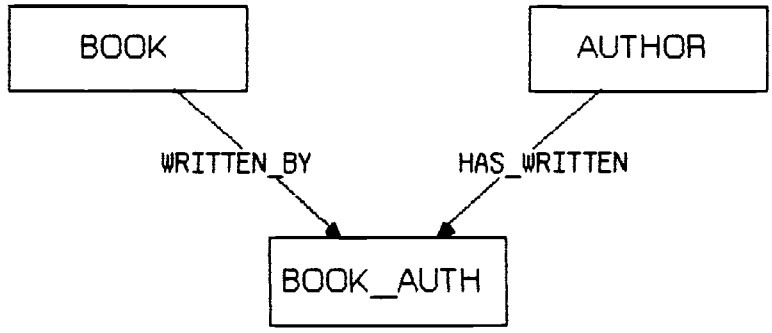

Figure 1.2.1. The schema BOOK_AUTHOR as an illustration of Bachman diagrams.

The records and sets of an Allegro schema are displayed in Bachman diagram form. The user may position the records of his schema when and where he sees fit. Allegro Query will remember the users last positioning of the schema records query to query and session to session of Allegro Query itself. Allegro Query normally displays only the records of the network schema. The sets of the network schema are displayed when the user requests them.

The second primary concern for a visual query facility is that the user be able to specify his desired navigation[7] of the schema to answer his query. It is intended that the user will navigate his schema with limited awareness that he is doing so. In Allegro Query the user does not specify query navigation with the initial specification of his query. If Allegro Query can find a unique path to answer the query, it will do so without bothering the user for a navigation specification. If it can not find a unique path, Allegro Query will present a list of schema sets to the user and request that he select the sets to be used. The user also has options that limit the schema structure available for Allegro Query to consider. These options can significantly decrease the occasions that Allegro Query must ask the user to navigate his query. The details of these two primary concerns and a discussion of query navigation are in Chapter 3.

Allegro Query uses a direct function call interface to the functions of Allegro. This is the interface defined for the original kernel of Allegro, the work of Myra Lane Uy.[4] This is the same interface used by another portion of the Allegro system, the superdbcs and DML C macros. This is shown in figure 1.2.2.

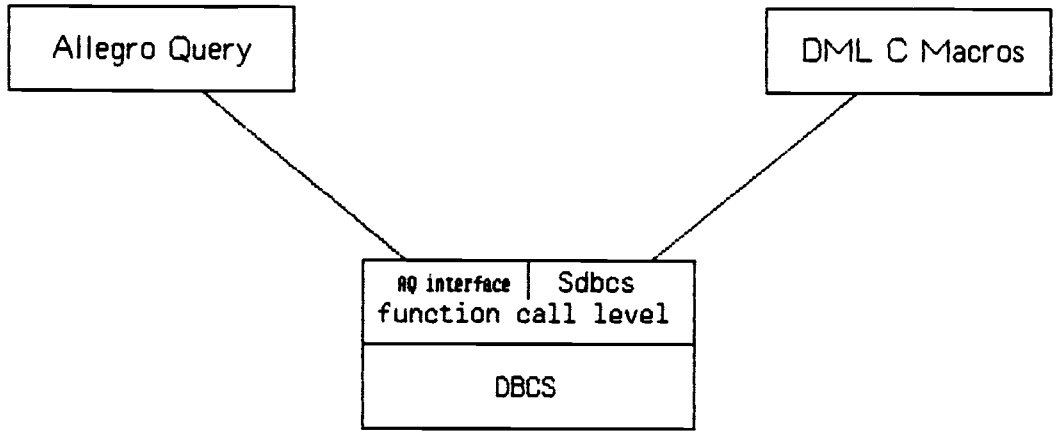

Figure 1.2.2 Tha Allegro Query Interface to Allegro.

The decision to model Allegro Query on the appearance of Query-By-Example resulted in the requirement of a complex display interface. After much searching and several false starts, the author settled on Smalltalk as an available system able to provide the desired displays. This query facility is implemented in Smalltalk on the Textronix 4404 system.

Allegro itself is a UNIX on VAX implementation. A small portion of Allegro Query is a user level part of Allegro on the VAX. The main portion of the Allegro Query facility has a serial communication line interface to access the portion of itself that is integrated with Allegro (figure 1.2.3).

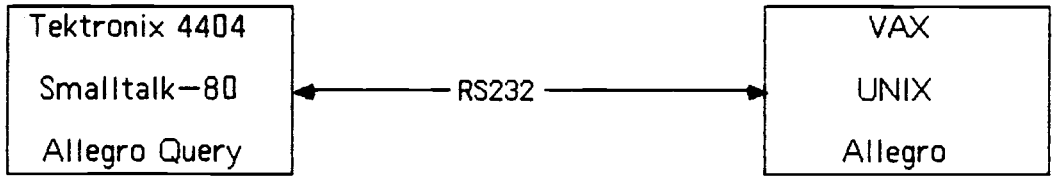

Figure 1.2.3. Allegro Query System Configuration.

Chapter 2

Query Languages

2.1 Overview.

A query language may be defined as a high level computer language for the retrieval and modification of data in databases. It is usually interactive, on line, and able to support queries that are not predefined.[6] Some query languages, for example ISBL[12], are retrieval only at the high level and have their modification facilities embedded in the host programming language. Allegro Query is a retrieval only language in that data modification capabilities have not been defined.

Vassilou and Jarke[6] classify query languages as building from two different backgrounds; one the need for simple end user interfaces, the other from theoretical considerations. The first group emphasis usability, that is the minimization of effort required to do useful queries. The second group emphasis functional capabilities, that is what can be done with a query system. Both of these groups are considered as evolving to include considerations from the other background.

Allegro Query is oriented to the usability group. In particular Allegro Query is intended to make the interface with a network database user friendly. Specifically this means the navigation of the network schema is made user friendly. The author is not aware of any other high level query language that navigates a network database. As a query language separate from the navigation, Allegro Query is unremarkable when compared with other modern query languages.

Vassilou and Jarke[6] continue with their query language classification by lumping both of the above groups of query languages with the term *previous*

generation query languages. They have a grouping *new generation query languages* some of whose characteristics are:

Use of more senses.
Direct manipulation of objects.
Use of examples as a natural deduction process.

Query-By-Example (QBE) is mentioned as a language which has features of a new generation language. Allegro Query also has features of a new generation query language. Allegro Query has a two dimensional display similar to QBE and a pointer driven menu interface. It uses more of the visual sense than command line languages and uses other than the keyboard (a mouse) to indicate the area of user interest. Allegro Query directly manipulates the record type entity classes of the network database. It uses examples in the same manner that QBE does. Allegro Query also has a ability to do incremental queries by cut and paste modification of request entries.

The expression of queries in a high level language often gives no consideration to the amount of work the database system will have to do to answer the query. There may be more than one way to retrieve the requested data, some ways more efficient than others. Often it is possible to algebraically manipulate the query so that the less efficient operations are taken over less data. For example, in relational algebra queries, selection and projection may be moved inside a join such that the join is done over less data. When a join over extensive data is necessary, a sort of the separate relations on the join attributes reduces the data accesses required to do the join. In most current query languages the expression of a query indicates the relations used to answer it and implicitly a path to use to answer it. The Universal Relation concept basically hides knowledge of the component relations. Ullman[12] gives an example (P319) where different answers could be produced by different paths through the component relations and where the knowledgeable user may have to use tuple variables to direct the query.

Network databases are usually discussed in terms of programmed low level retrievals. Considerations of query optimization do not apply as the retrieval path is explicitly specified by the query and data selections are coded directly into the retrieval specifications. Allegro Query, being an

attempt at a higher level query interface, does have path navigation and data selection problems. These are discussed in Chapter 3.

2.2 Query—By—Example.

Query-By-Example (QBE)[1,2,3] is labeled a domain calculus language[12], since it uses sample elements which are variables over the domain of some attribute. Zloof[1] makes a distinction between sample elements and variables. This distinction does not appear to have been maintained in later descriptions of QBE. In his papers, Zloof deemphasizes the predicate calculus basis of his language in favor of definition by illustrative examples. He considers this form of definition "... in our opinion - more appealing to the causal user, which is one of the major aspects of Query-By-Example."[1] This aspect is the primary reason why the appearance of Allegro Query is based on the appearance of QBE. This flavor of QBE is given in the conclusion of Zloof's first QBE paper[1].

> "In this paper we presented the data access portion of the Query By Example Language. We conclude that the unique features of this language are as follows:
>
> 1. The user has the perception of manual table manipulation.
> 2. The user has a pre-established *frame of reference*, ie the tables.
> 3. The user can easily pre-identify the relations to be used, resulting in an early reduction of the scope of the data base.
> 4. As opposed to linear-type languages where the user is constrained to one degree of freedom, here the user has multi-degrees of freedom in that the sequence of filling in the tables and rows within the tables is immaterial. This implies that given a data base the system does not constrain the user's thinking process in any way while he/she is formulating the query. Take Q7 as an example. If the user's thinking process wishes to first choose a manager and then compare his/her salary to the salary of his/her employees, the query would be the same whatever the row order is, thus the system is capable of capturing the different ways different users approach the problem.
> 5. The sequence of the following steps is also immaterial.
>
>   a) filling in the constant elements,
>   b) linking the variables,

c) specifying the output by the P. function (projection), and
d) grouping.

6. It follows from 4 and 5 that Query By Example allows the user to divide the query into decoupled segments, making it declarative and highly non-procedural. In contrast, most linear-type and other languages require the user to first specify the information to be outputted and then structure the query accordingly.
7. Due to the decoupling features inherent in Query by Example, it can handle rather complicated queries without relinquishing its simplicity. This is in contrast to other languages where a lengthy and complicated query has to be artificially divided into multiple steps and then taken one at a time."

Q7, referenced in the quote, is presented in section 4.2 below.

Six examples from Zloof's papers[1,3] are used to illustrate QBE. Q2 asks the paper's department store database "What colors of ink are available?"

type	item	color	size
	ink	p.black	

Figure 2.2.1. Q2 of [1].
The colors of ink available.

In this query 'p.' stands for 'print' the attribute in which it appears. This is QBE projection. 'ink' is a constant element. Its presence under item specifies that all tuples considered will have the value of the item attribute equal to 'ink'. This is the query selection. 'black' is an example element. It states that the user is interested in the color of the item with black as an example of his interest. In this query the example element serves no query semantic purpose and may be omitted.

Q4 shows a link between two relations to find the suppliers of items sold by the toy department.

Sales	Dept	Item
	toy	pen

Supply	Item	Supplier
	pen	p.gm

Figure 2.2.2. Q4 of [1].
A link between relations.

Q8 and Q9 show first an 'and' of which departments are selling both items and then an 'or' condition of which departments are selling either item.

Sales	Dept	Item
	p.toy	pen
	toy	pencil

Figure 2.2.3. Q8 of [1].
A department selling pens and pencils.

Sales	Dept	Item
	p.toy	pen
	p.hardware	pencil

Figure 2.2.4. Q9 of [1].
A department selling pens or pencils.

Entries can get considerably more complex. Constant entries may have relational operators. Full row entries may be negated to mean that the rows which answer the query do not do whatever is indicated. There are aggregate operators, arithmetic expressions in operands, and condition boxes.

Two more examples from Zloof[3] show the use of arithmetic operations and the condition box.

EMP	Name	Sal	Comm
	Jones	s1	s2

Output	Name	Earnings
	p.Jones	p.(s1 + s2)

Figure 2.2.5. Figure 20 of [3].
The use of arithemetic expressions.

EMP	Name	Sal
	p.	s1
	Jones	s2
	Nelson	s3

Conditions
s1>(s2+s3)

Figure 2.2.6. Figure 21 of [3].
The use of a condition box.

QBE has update capabilities in the language. Zloof[2] describes how QBE can be used for data descriptions, integrity constraints, etc.

2.3 Other Query Languages.

The examples for the other query systems to be discussed and most of the material has been taken from Ullman[12]. These systems are presented as a contrast to Allegro Query. There was awareness of, but no consideration of, these systems in the design of Allegro Query. It is this type of interface that Allegro Query is avoiding.

The example database used has three relations:

```
members( name, address, balance)
orders( order_no, name, item, quantity)
suppliers( sname, saddress, item, price)
```

The first example is to print the names of members with negative balances. It references only one relation. The second example is to print the supplier name, items, and prices of all suppliers that supply at least one item ordered by member Brooks. This example requires a join. The last example is to print the suppliers that supply every item ordered by Brooks. The last example specifies a 'for all' quantifier on the items ordered by Brooks. It stresses all the query languages.

SQL is often the language of reference for command line query languages. In addition to retrieval functions SQL has aggregate functions and update actions. It may be a stand alone language or embedded in PL/1. SQL uses keywords to translate from command line input to relational form.

In SQL the first example may be written:

```
select name
from members
where balance < 0
```

The 'select ... from ...' is projection from the named relation. The 'where' clause can be very complex. Here it is a simple relational operator.

In the second example the 'where' clause represents an item ordered by member Brooks:

```
select unique sname, item, price
from suppliers
where item in
        select item
        from orders
        where name = 'Brooks'
```

Alternatively the second example can read like the join it represents.

```
select unique sname, suppliers.item, price
from suppliers, orders
where name = 'Brooks' and suppliers.item = orders.item
```

In the third query the 'where' clause is specifying set membership:

```
select sname
from suppliers t
where           /* The set of items for supplier of given name */
    (select item
    from suppliers
    where sname = t.sname)
contains        /* The items ordered by Brooks. */
    (select item
    from orders
    where name = 'Brooks')
```

The value after the initial 'suppliers' 't' is an example tuple giving meaning to 't.sname' that folows.

QUEL is an example of a tuple relational calculus language. QUEL is the query language of INGRES running under UNIX. It can be used stand alone or embedded in a 'C' program. The three example queries are shown for comparison. From the first example:

```
range of t is members
retrieve (t.name)
where t.balance < 0
```

The join of the second example can be read directly:

```
range of t is orders
range of s is suppliers
retrieve (s.sname, s.item, s.price)
where t.name = 'Brooks' and t.item = s.item
```

The third example from Ullman takes approximately 14 lines to state the query:

```
range of s is suppliers
range of i is suppliers
retrieve into dummy (s=s.sname, i=i.item)
range of t is dummy
delete t  where t.s = s.sname and t.i = s.item
range of r is orders
retrieve into junk (s = t.s, i=t.i)
        where r.name = 'Brooks' and r.item = t.i
retrieve into sups (s = s.sname)
range of u is sups
range of j is junk
delete u  where u.s = j.s
sort sups
print sups
```

It uses dummy relations with deletion and sort actions over the dummy relations.

The last example language is ISBL. ISBL by itself is only a query language. To obtain aggregate operators, update facilities, etc the language is embedded in PL/1. It is presented as an example of a relational algebra language. There are operators in ISBL for the relational algebra operators; '+' for union, '−' for set difference, '.' for intersection, ':' for selection, '%' for projection, and '*' for the natural join. 'list' prints the value of a relational algebra expression. Dummy relations are defined with the assignment operator '='. 'n!' specifies expression evaluation by name. There are other operators which do not show up in our examples.

In ISBL the first example is:

```
list members: balance < 0 % name
```

The second example may be written by defining a delayed evaluation natural join:

```
os = n!orders * n!suppliers
list os: name = 'brooks' % sname item price
```

or directly by:

```
list orders*suppliers: name = 'brooks' %sname item price
```

The 3rd example is also a convolution in ISBL:

```
s = n!suppliers % sname                 /* the set of suppliers */
i = n!suppliers % item                  /* the set of supplied items */
b = n!orders; name = 'brooks' % item    /* items ordered by Brooks */
ns = (n!s * n!i) - (n!suppliers % sname, item)
        /* pairs of suppliers and items not supplied by supplier */
nsb = n!ns . (n!s * n!b)
        /* set of supplier-item pairs such that supplier doesn't
        supply the item, and Brooks ordered the item */
s - (nsb % sname)
```

To continue the comparison the same three queries are given in QBE. In QBE all the user has to do is fill in the attribute positions as necessary. The first and second examples are simple:

members	name	address	balance
	p.		<0

Figure 2.3.1 The first of the sample queries in QBE.

orders	order_no	name	item	quantity
		Brooks	banana	

suppliers	sname	saddr	item	price
	p.		p.banana	p.

Figure 2.3.2 The second of the sample queries in QBE.

The third example is not given in Ullman[12]. Either Q9 or Q18[1] serve as a template for the query, so it would be writen:

orders	order_no	name	item	quantity
.		Brooks	all x	

suppliers	sname	saddr	item	price
	p.		p. [all x]	p.

Figure 2.3.3 The last of the sample queries in QBE.

The brackets represent set grouping. Here the brackets state that all of the sample elements x are a subset of the items in the supplier tuple. The possible additional elements are represented by '.' in the brackets. Set notation was not covered in the description of QBE above.

From this sample it would appear that QBE allows the most user-friendly writing of queries.

Chapter 3

Query on a Network Database

Query on a network database is normally discussed in terms of a low level data manipulation language. For Allegro the DML is a subset of the basic parts of CODASYL DML[10]. For the FIND statement, Allegro uses only the record selection expression of the CODASYL statement. The FOR UPDATE and RETAINING... phrases are not used. There are six formats of the CODASYL record selection expression. Allegro has portions of five of these formats. Formats 1, 3, 5 were provided by Uy[4]. Formats 2, 4 were provided by Swasdichai[5].

format 1

```
{ ANY       } record-name-1  USING {identifier-1}...
  DUPLICATE
```

Allegro has the portion of this format known as find by CALC key, that is the identifiers supplied are those needed to locate a record whose location mode is CALC.

format 2

```
DUPLICATE WITHIN set-name-1  USING {identifier-2}...
```

format 3

```
  NEXT
  PRIOR
{ FIRST      } [record-name-2] WITHIN set-name-2
  LAST
  integer-1
```

The Allegro documentation does not specify identifier-3 (alternative to NEXT, etc). Allegro also has no notion of realm.

format 4

data-base-key-identifier-1

Allegro has the keep list form of this format. Swasdichai[6] indicates the reference is by keep list position not by key comparison.

format 5

OWNER WITHIN set-name-3

The GET statement, according to Allegro documentation[4], is not the GET statement of CODASYL.

GET [record-name-1] [identifier-1]...

Allegro is specified to return the current of record named, if one is named, whether that record is current of run unit or not.

Allegro has some additional auxiliary retrieval DML[6]. It also has update DML[4,6].

The retrieval functionality of a network database, even of the subset implemented in Allegro, is sufficiently rich that for a schema of any complexity there normally will be several ways to program a given query. The programmer must choose the retrieval operations he will use to answer the query. That is , the programmer must navigate[7,12] his way through the network schema from whatever database entry he has to the data that will answer his query. The overall navigation process is described in textbooks on database.[11,12] For Allegro a description of one navigation on the standard demonstration schema (figure 1.2.1) is in Uy's thesis[4]. This example is given here using the psuedo DML of Uy:

```
find the CALC record called BOOK based on its key
get BOOK record
print information from BOOK record
find the first BOOK_AUTH record in the WRITTEN_BY set
while current BOOK record is not current of WRITTEN_BY set
      find owner of current BOOK_AUTH record in HAS_WRITTEN set
      get AUTHOR record
      print information from AUTHOR record
      find next record in WRITTEN_BY set
```

To look ahead, Allegro Query requires no navigation for this query.

With Allegro Query, query is on a visual level rather than in a 'C' program level. When necessary the user must still navigate his query. However the user navigates his query without writing DML statements. As indicated in Chapter 1, this is done with the use of a visual display and a method to navigate, when necessary, by simply selecting schema sets. The rest of this chapter focuses on these actions omitting the details. The details of the user interface are in the next chapter.

To place Allegro Query on a visual level the user is shown a display which has one display object for each record type in the schema. The contents of each display object are discussed in Chapter 4. The user may remove record displays that are of no interest to him for the current query or set of queries. Removed records are removed from consideration by Allegro Query processing. Such a trimmed display may survive session to session of Allegro Query. Removed records are easily restored. The user normally sees only the record displays of the schema. There is a menu option that will display the set structures of the schema. Sets that connect to records which have been removed will not be displayed and are also removed from consideration to answer the query.

An example will help to explain what trimming the schema accomplishes for the user. This example shows where it is easy to trim too much. Consider the standard Allegro example schema, BOOK_AUTHOR. Augment it with a record type which carries position held information for the various authors (figure 3.1).

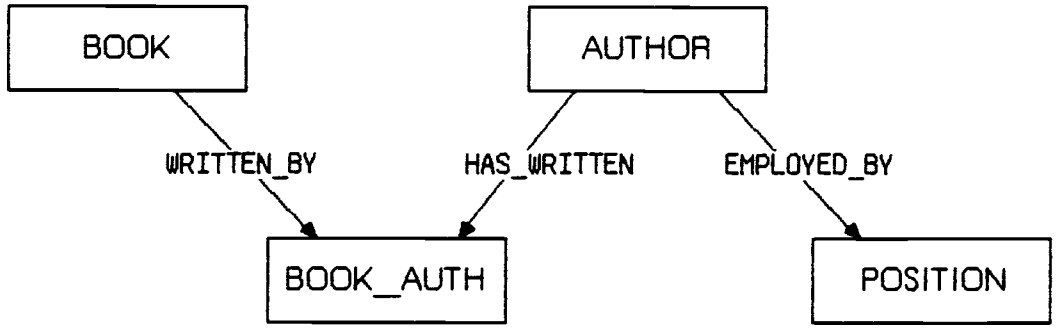

figure 3.1. BOOK_AUTHOR schema augmented for trimming example.

The user wishes to enter a book title and have the system respond with the authors. For this query, author position information is superfluous and may be trimmed. The user removes the record 'POSITION' from the screen. This removes the record from availability for Allegro Query processing. The set 'EMPLOYED_BY is also removed from availability for Allegro Query processing. If displayed, the set is also removed from the screen. So far, all is well. Next the user decides the BOOK_AUTH record contains no useful information. So he removes it from the screen. If displayed, the two sets connected to BOOK_AUTH also disappear. Record removal removes that record and any connecting sets from consideration during the path search. If BOOK_AUTH is removed, Allegro Query will be unable to answer the query without asking the user to navigate. This is one of the reasons the user should understand the schema to use Allegro Query. While in the normal case the query of this example will proceed silently, the user still should navigate the schema in his thinking about the query.

Since Allegro is a network database, the user is expected to be familiar with the Bachman diagramming technique[9] used for network databases. It is assumed that the user has a data structure diagram vision of the schema he is using. While not necessary, the user can and should position the schema records he has not removed so that they correspond to his vision of the schema. Allegro Query makes no attempt to guess the user's vision of the schema. Once the user has positioned the schema records as he likes, Allegro

Query will remember that position query to query and session to session. Such positioning may be changed at will.

Each schema record is displayed in a QBE like format. The user interacts with the display in a QBE like manner, that is the query information that applies to a field of a record is interactively placed in that field. Do not assume from the use of a QBE like format that Allegro Query is a domain calculus language where the variables range over the domain of an attribute. Allegro is a network database where the data available in any schema record depends on instances of the path into instances of that record. The data available is that which will have the QBE like selection and projection specifications applied to it. The details of this interface are in section 4.2.

In the initial statement of a query the user makes no statement of the navigation necessary to answer a query. When the query answer is requested, Allegro Query will search for a path through the schema sets that accesses all the records that participate in the query. The search will detect the availability of more than one path or of no paths. If there is only one path, the query proceeds silently. If the user has exercised his option to trim the schema available to the current query, this will happen more often than it might first appear.

If there is more than one path possible to answer a query, Allegro Query does not attempt to determine all the paths. Instead it presents the user with a list of schema sets and asks him to select the sets that should be used to access the records that participate in the query. This is the case where the user must navigate his query. All the user has to do is select the sets that are to be used. The user does not have to state 'find first in ...', etc. Allegro Query will generate the DML necessary to follow a path through the specified sets.

What does the user need to know to use the visually displayed schemas of Allegro Query? As mentioned above, he needs to understand the one-to-many concepts of network databases. He needs to be able to visualize retrieval following set paths and know that it is possible to retrieve both ways along a set path. He needs to know that Allegro FIND format 1 is limited to the calc

key concept. If he visualizes a record, other than a system record, as the starting point for a query, he must provide a calc key for that record. For schemas that have both calc keys and a system record (the student_rec schema in the examples chapter), he needs to realize that the normal mode of Allegro Query will prefer calc key retrieval as less data is retrieved when calc key retrieval is possible. The designed 'retrieve for local query' option will reverse this preference. Finally the user does need to know the very simple Allegro Query language. Since this language is modeled on the well known QBE, the language usage is expected to be intuitive.

Allegro Query does little query optimization. Allegro Query retrieves all the data that occurs along whatever path is specified. The structure of a network database and, if necessary, the user's path selection has programmed for Allegro Query that which on a relational database would be the implementation of a join. There is no optimization here. Selections are not moved inside the retrieval path.

There is a positive reason for not moving selections inside the retrieval path. The author visualizes the situation where the user will be able to do local queries on the data once retrieved. Consider a situation where the user is interested in the salary history of the firm's employees. Assume a schema with the employee records as members of a singleton set and the employee records connected to several things one of which is that employee's salary history (figure 3.2). There are no calc keys specified. The initial retrieval will get all the employee and salary history records. The user then does a series of local queries fine tuning selection and projections until he has the data he wants. With such a scenario, Allegro Query can not move initial selection inside a retrieval path. Once the data is retrieved selections are applied in path order. That is selections will be applied to the owner records first thereby pruning member data that must be considered.

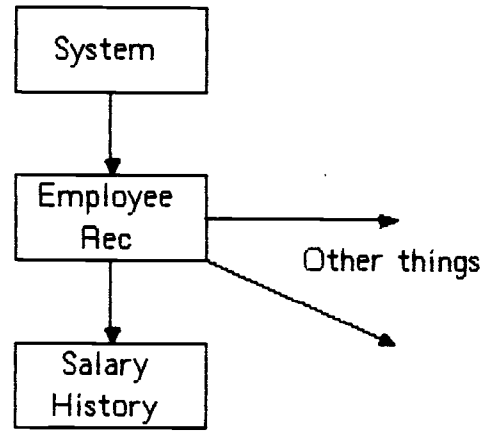

Figure 3.2
Example of a subset of a schema
that could be used for local queries.

The query answer is displayed in a manner intended to maintain user awareness that Allegro is a network database. When the user requests projection items from several schema records, the display will be in a generated record with a field for each projected item. This answer is not in the form of a full table as would be displayed for a relational database. Instead the set relationship is maintained by not repeating owner record data for member data along a set instance. This presentation can follow several sets in a path with the intermediate owners, if they are projected, repeated when their instance changes. An example of this display is in chapter 4.

Chapter 4

User Interface

4.1 Display Interface.

Smalltalk[13,14,15] terminology is used to describe the user interface.

4.1.1 One Time Actions.

To use Allegro Query the user should have a standard Smalltalk image with the category Allegro-Query added by 'file in'. After the 'file in' the Query class method initialize should be sent. These are actions in the nature of system generation.

A query instance is obtained by evaluating 'Query open' (without the quotes) in any view which supports the standard 'do it' function. The author normally uses the system browser then cancels the change. Upon evaluation of 'Query open' Smalltalk will have the user frame the view referred to as the Schema Name View (figure 4.1.1.1). This view may be framed at the minimum size the view will allow. With a Smalltalk image 'save' the state of a query instance survives session to session. Closing the Schema Name View releases the query instance.

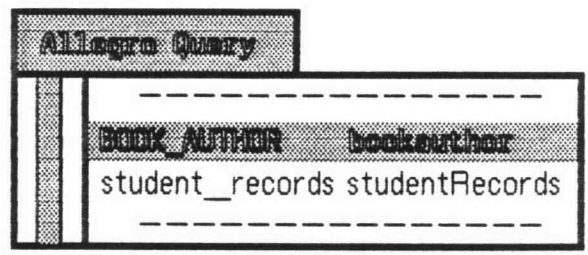

Figure 4.1.1.1. Schema Name View.

The Schema Name View contains the names of the Allegro schemas that have been made known to Allegro Query. Allegro Query does not attempt to ask Allegro for all schemas known to it. The display of a schema name contains the schema name as known to Allegro and the data file name expected by Allegro on DML 'open'. If there are more schema names than space in the view, the names will scroll. These names are maintained in a class variable and will be available to other query instances. The Schema Name View does, as do all views in Allegro Query, respond to the standard blue button menu.

The project design envisions that an Allegro schema is associated with an Allegro Query instance. This schema is used until the user is done with it. It is possible to have more than one query instance (with view) in existence. Since the user can control view placement, he may be able to keep the situation straight. The data of each query instance will be distinct. Although no provisions are made, the interface with VAX Allegro is discrete enough and of short enough duration that no problems should arise.

4.1.2. The Main Control Menu.

The Schema Name View also contains a yellow button menu (figure 4.1.2.1). This menu is what the user uses to control Allegro Query. The content of this yellow button menu varies with the content of the view and the state of the query instance. When there are no schema names or one has not yet been selected this menu presents only 'add schema name'. When a name entry has been selected, but not yet fetched (described below), this menu presents 'fetch schema' 'add schema name' and 'remove schema name'. When a schema has been fetched, this menu presents choices as shown in figure 4.1.2.1.

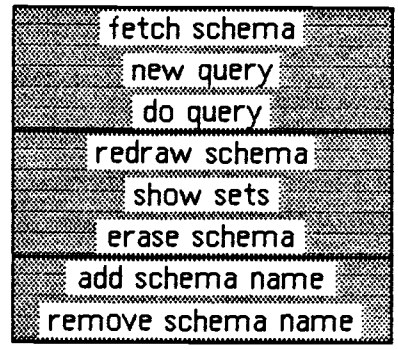

Figure 4.1.2.1. Yellow Button Menu of the Schema Name View.

The term fetched means a schema has been associated with the query instance. It means Allegro Query has communicated with the portion of itself in Allegro on the VAX to fetch those portions of the schema it needs. It also means this query instance is ready to do queries on this schema. A change in Schema Name View schema name selection will require another schema fetch. Schema fetch should also be a one time action. It need be repeated only if the schema definition changes for Allegro.

The schema records will have been displayed on schema fetch. If the user has moved the schema records to suit his image of the schema, his positioning of the records will be remembered and used on any redisplays of the schema.

Selection of 'new query' initializes the query portion (Section 4.2) of the query instance. If the schema is not displayed, 'new query' will display it. The use of 'new query' is not necessary when the user prepares a new query. Individual fields may be modified and 'accept'ed (see below).

The use of 'do query' is discussed below.

The action 'redraw schema' will maintain user record positioning and restore removed record views. It also makes all network sets available (but not displayed) for query processing. 'redraw schema' will restore the original state of the records affected by the 'resize field' or 'remove field' actions. Along with 'new query', 'redraw schema' serves to clean up the state of the

Allegro Query instance. It will often be appropriate after the user has responded to a request for query navigation (see below).

Selection of 'show sets' will display schema set lines for the sets currently available for query processing. If the user has not removed records or responded to a path navigation request, the display will include all the schema sets. Set lines, when drawn, are not views. They will be erased by 'redraw schema', by the display screen yellow button menu 'restore display', and by anything else that does restore the display; for example the activation of overlapping views.

The action 'erase schema' removes the schema from the screen. It does not affect the state of a query. Any (partial) query language request will be restored on 'redraw schema'.

When the user selects 'add schema name' he will be presented with a pair of fill-in-the-blank views. The first asks him for schema name (as known to Allegro). The second asks him for the VAX filename of the schema.

4.1.3. Record View.

On 'schema fetch', 'new query', or 'redisplay schema' a record view is displayed for each record in the schema. These records are constructed from schema data, particularly field size. The user is not asked to frame schema records. The appearance of a record view (see figure 4.1.3.1) is similar to that normally shown for a QBE query. The record name appears as the label of the record. It does not have its own column. There is a column for each field in the record with the field name at the head of the column. Lines for query request entries follow. The field size will be adjustable as defined below. A record view is composed of two subviews, the 'name view' and the 'request view'. When appropriate, an output data view will be tacked on below the request view during 'do query'.

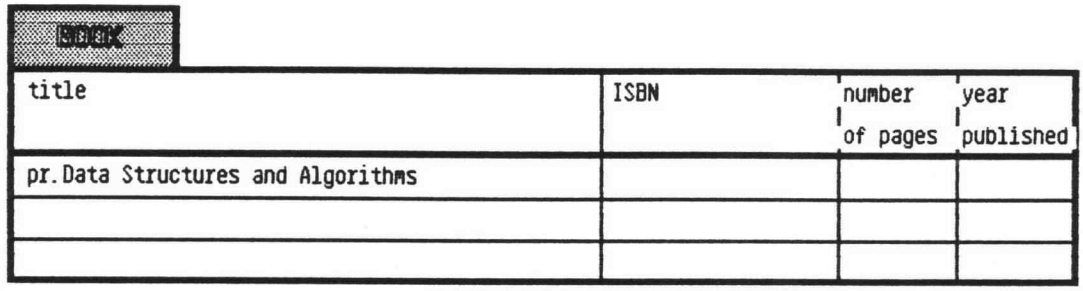

Figure 4.1.3.1 Record View 'BOOK' of the BOOK_AUTHOR schema with a query entry.

On schema fetch the initial record views for the schema will appear overlapped on one another with an offset for each record. There is no record placement algorithm. One of the thrusts of this thesis is that the user should place the schema records as he would draw them on paper. The record view blue button 'move' is used to do this. The user may also 'frame' the record view. The user may, but need not, remove the views of records he is not interested in for the current query. Use blue button 'close' to do this. Only the blue button 'move' and 'frame' actions have overrides to capture user record positioning.

The name sub view of a record view is non-interactive. Some attempt has been made to display readable field names. The underscores commonly used for 'C' have been removed and an attempt has been made to split the words of a field name over two lines at word boundaries.

The request sub view is the important view in preparing a query. The query language is discussed in section 4.2 below. This view has three query request lines where the number of lines is changeable. The request line on which a query is entered is unimportant. Some effort was expended to defeat scrolling for the request lines as it was felt that this would be less intimidating to the user. However scrolling does occur on a field/line by field/line basis for a useful reason. The fields are sized on the Allegro schema field size subject to a maximum and a minimum. The variable size character display and the added query operators can make a field too long for display on

a single field/line in the view. The default font used displays numbers quite large. A field that contains numerical data, eg social security number, will normally overflow its boundaries. The view will scroll in this case but without scroll bars. User control of the scrolling is by dragging the cursor through the top or the bottom of the field/line. Single sub view scrolled data (a query field/line) is received by the processing portion of Allegro Query as one line. The scrolling is an illusion.

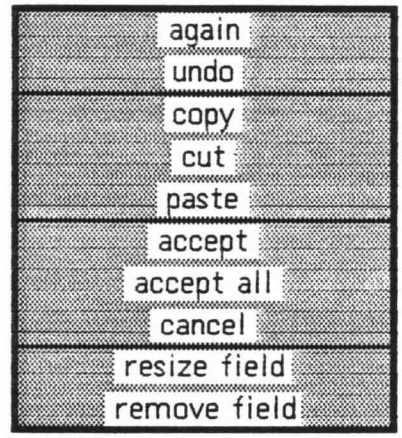

Figure 4.1.3.2 Yellow Button Menu of a record view.

The request view also has a yellow button menu (figure 4.1.3.2). This menu controls query request entries. Most of the items on the menu are the standard editing items. 'Accept' presents the current field/line entry to Allegro Query for processing. There are three additional entries in the yellow button menu; 'accept all', 'resize field' and 'remove field'. The availability of 'accept all' gives the user a chance to be lazy. He may make several field/line changes then do one 'accept all' rather than several 'accepts'. Even 'accept all' is not necessary if these are the initial entries of a query, that is there have been no previous accepts for this query. 'resize field' is intended to be used when the Allegro Query displayed field size is not visually satisfactory. 'remove field' will be useful for a large record whose display overflows screen boundaries.

4.1.4. The 'do query' Function and Navigation.

The 'do query' function of the Schema Name View is used to obtain the answer to a query. If the user has entered data he may use 'do query' directly, even if he has done no previous 'accepts' or 'accept all'. The accepts will be simulated. If the user has changed data and not used 'new query', he needs to use 'accept all' or 'accept' for each changed field. It is during 'do query' that several error situations, for example lack of a valid calc key, are discovered. The 'do query' function communicates with Allegro on the VAX for its data.

As discussed in Chapter 3, when Allegro Query can process a users query silently, it will do so. When Query detects the existence of more than one path or can find no path there is further user interaction to provide query navigation. When the user must navigate, the view frame symbol will appear. Upon completing the view frame, the user will see a view that is referred to as the user navigation view (figure 4.1.4.1).

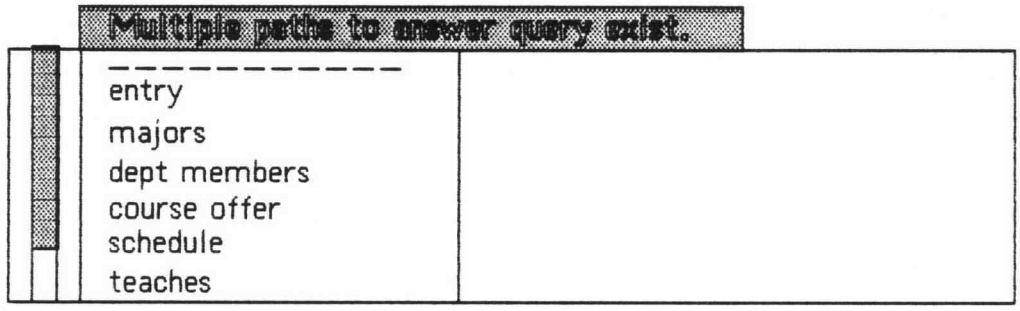

Figure 4.1.4.1. A typical view when the user has been asked to navigate a query. In this case the user has not yet selected a menu choice.

The view as shown is titled "Multiple paths to answer query exist." The view title alternatively may read "No path to answer query found." The left hand side of this view shows the sets of the schema with no initial selection. The sets scroll if necessary. The left hand side of this view has a yellow button menu associated with it. When initially displayed this menu shows four lines 'name sets', record paths', 'use BB close to', and 'abort query' with a divider line after the second item. The users choices are 'name sets' and 'record

paths'. The last two lines function as a no action reminder message. This menu will change depending upon the users selection.

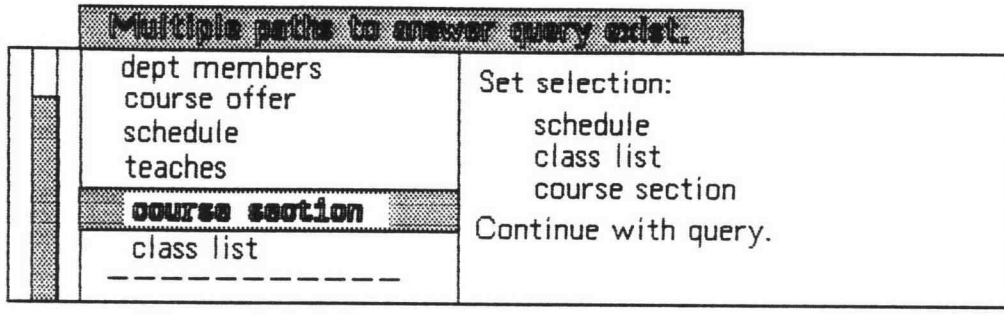

Figure 4.1.4.2. A typical view when the user has been asked to navigate a query. In this case the user has selected the 'name sets' menu choice.

The 'name sets' choice is the easiest choice for the user to use. Allegro Query design considers this as the normal user choice. With this choice the user specifies the sets that are available for Allegro Query to use to attempt to navigate the query. The use of 'show sets' in the Schema Name View may be helpful here. When 'name sets' is chosen the line 'Set selection:' appears in the navigation view right hand sub view. The user then selects as many sets as he wants Allegro Query to consider. Each set selected will be reflected in the right hand sub view. The order is not important. The selection of 'name sets' will have changed the yellow button menu. The choice 'do query' appears in place of the original first two choices. Selection of 'do query' starts the query over with the named sets available for processing. It also enters the phrase 'Continue with query.' in the right hand sub view.

The set choices mark sets in the same manner that record removal removes them. It takes use of 'redraw schema' to restore the full quota of network sets to Allegro Query processing. It is possible for the named group of sets to produce either of the results that cause an entry to the user navigation view. This has happened to the author when he specified a name value for the student record entry (of student_records schema in the examples chapter) forgetting that the calc key was social security number.

The path specified had included only the records necessary for calc retrieval but not for use of the system record.

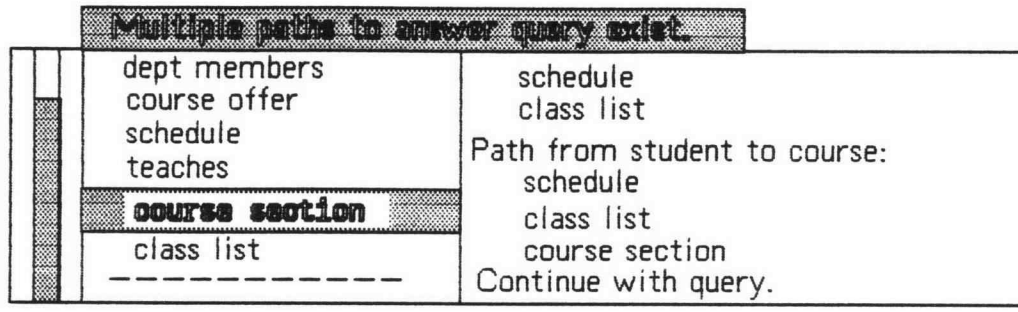

Figure 4.1.4.3. A typical view when the user has been asked to navigate a query. In this case the user has selected the 'record paths' menu choice.

The menu choice 'record paths' is intended for the really sticky cases. With this choice the user is asked to specify a path from the record Allegro Query intends to use to enter the schema to each other record in the query. A separate path is expected for each target record in the query. The sets should be named in order along each path. When all paths have been specified, Allegro Query continues with the query. The user's path choices are used as given. No checks are made.

The use of 'record paths' starts with the menu selection. That action will show the line "Path from <entry record> to <current record>:" with the appropriate substitution for the <...> in the right hand subview of the user navigation view. The user then selects sets as before. Each set selected will be reflected along the right hand side. Query detects when the target record has been reached by the set path. It will then move the current record and show the path line again until the target records are exhausted. Query resumes automatically when the last target path is complete. This menu choice has no optimization. If one record lies along the partial path of another, the user will still have to make two path specifications.

4.1.5. Output Data View.

The output data view may appear in two places. If all the projected fields are members of a single record the output data view is tacked on the affected record view as a sub view below the request view sub view. This is true even for a multiple record query that must follow paths to fetch data and process selections. If fields from multiple records are projected, the output is presented as a generated record (named 'dataOut') with name and output data subviews (figure 4.1.5.1). Only the projected fields are present. In both cases the output data view is scrolled on a full view basis, that is all fields scroll as a unit.

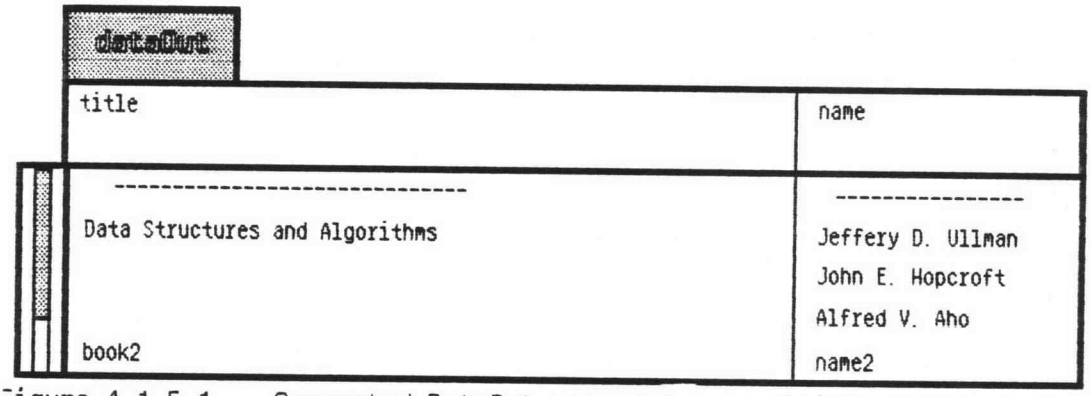

Figure 4.1.5.1. Generated DataOut record for a multiple record query. (Some data added to show the network structure appearance.)

The awareness that this is a network database is maintained in the output data views (figure 4.1.5.1). The output data view by design does not have a full table appearance. The awareness of one-to-many relationships is maintained by not duplicating the owner data along a chain path.

4.2 Query Language.

Entries in the query language deal with the several fields on one line of the request view. The language is illustrated using more examples from Zloof[1]. A typical Query-By-Example (QBE) entry, "Print the red items:", is

type	item	color	size
	p.pen	red	

Figure 4.2.1. Q1 of [1] in QBE.
A simple query.

where all but 'p.pen' and 'red' are shown by the QBE template. For Allegro Query the user will type 'p._pen' and 'red' in the slightly different template.

type			
item	color	size	
p._pen	red		

Figure 4.2.2. Allegro Query version of Q1.

again where all but 'p._pen' and 'red' is an object displayed by Allegro Query. The case of the initial 'p.' is not relevant. This entry is an example of a possible answer with the addition of a command function. 'p.' specifies printing of this field. 'red' is a constant element. The items selected to answer the query must have data element 'red' in the 'color' field. When the field is a string, the case of the constant element is important. '_pen' is the example element of this query. There might be a red pen in the query answer. There might also be a red dress in the query answer. There might not be a red pen in the query answer. For the example given '_pen' is unnecessary and may be omitted.

For many queries, the entire query language has been given. Consider Q5, "List the names, salaries, and managers of employees in the toy department:"

EMP			
name	sal	mgr	dept
p.	p.	p.	toy

Figure 4.2.3 Q5 of [1] in Allegro Query.
Another simple query.

The example elements have been omitted. If in this case it was also desired to print the department, QBE would use 'p.' in the record name field. Allegro Query follows the Smalltalk idea of labeling the display object by using the record name in the view label. There is no column to enter record data. To obtain an entire record print without entering 'p'. in each field, enter 'pr.' in any field.

Q3, "Find the department(s) that sells an item(s) supplied by the supplier Parker:" illustrates a QBE link between separate relations. Allegro is a network database. The network does the linking. Links as shown in Q3 are not used in the query. It is assumed that SALES/SUPPLY has a network similar to BOOK_AUTHOR[4] with 'ITEM' the connecting record.

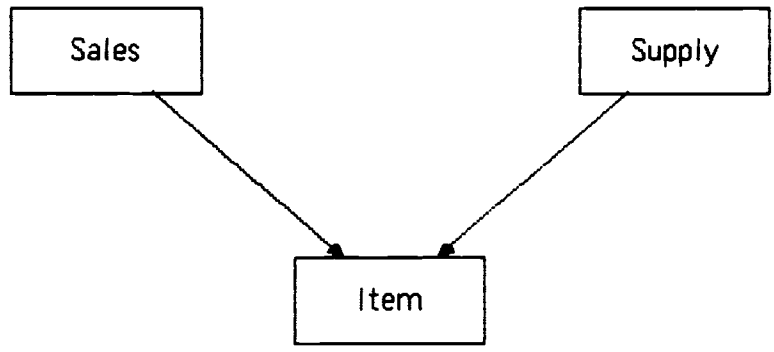

Figure 4.2.4. A possible network
for Q3 and Q6 of [1].

In Allegro Query this query is written as:

Figure 4.2.5. Q3 of [1] in Allegro Query.
The schema does Allegro Query linking.

With this network, we do Q6 by placing a 'p.' in these records and the connecting ITEM record. No example elements are necessary.

Q7 is entered as shown in Zloof[1] except that the underline becomes a prefix '_'. "Find the name(s) of any employee(s) who earn more than his (their) manager(s)."

EMP			
name	sal	mgr	dept
p._Jones _Peter	>_10K _10K	_Peter	

Figure 4.2.6. Q7 of [1] in Allegro Query.
The use of example elements to establish conditions in the data.

"If Peter is an example of such a manager and if Peter earns 10K (as an example) then Jones is an example of an employee who earns more than 10K (indicated by the > operator) and, therefore, more than his manager."[1]

The above query introduces several more elements of the Allegro Query query language. Applied to constant elements are the relational operators; '<', '<=', '>', '>=', with omission for equal and '!' for not equal. Both omission representing equality and '>' are used in the query above. Three synonyms '=', '==', and '!=' are accepted. The last two are for the Allegro 'C' programmers. When the constant element contains a '.', an explicit

equality operator '=' (or '==') rather than omission should be specified. If the explicit operator is omitted and a '.' is present, the query parse will try to treat that which proceeds the '.' as a command function. The standard demonstration schema, BOOK_AUTHOR, has data with a '.' in it.

The above example also shows the use of and'ed conditions both on the same line and on separate query lines. The conditions on the first line postulate an employee who both makes more than some (example) amount and has some (example) manager. The conditions on the second line specify that the example manager makes the example amount. This is and'ed with the first line. The use of 'or' was shown with Q9 given in section 2.2.

The above specifies the design of Allegro Query language. There are sort operators in QBE as in figure 6 of Zloof[3]. They may be specified similarly for Allegro Query but were not as they are not needed to demonstrate query on Allegro. Similarly built-in functions like 'cnt.' or 'sum.' may be specified. There is no provision for a condition box nor are there expressions in a query field. Allegro Query also lacks a provision for set theoretic constructs. The queries of figure 20 and 21 of Zloof[3], (see section 2.2) are not currently expressible by Allegro Query. Again these are functions that are not needed to demonstrate a higher level query on Allegro. Allegro Query could follow QBE in the specification of these items with possibly { all _x, .} as the set notation.

In the network database of Allegro some schema are specified with calc keys. For such a schema the query user must be aware of the database and know what data may be expected to be on a calc key. Allegro Query will detect if a user query lacks an equality constant in at least one of the fields specified as a calc key as it will be unable to construct a retrieval for the query. The implementation can detect the error of missing calc key only after attempting to retrieve data from Allegro (on the VAX). The standard schema used to demonstrate Allegro projects is a schema with calc keys.

Three example queries[12] were presented in section 2.3 above. For comparison the first two of these are given in Allegro Query. The schema of these examples in network form is:

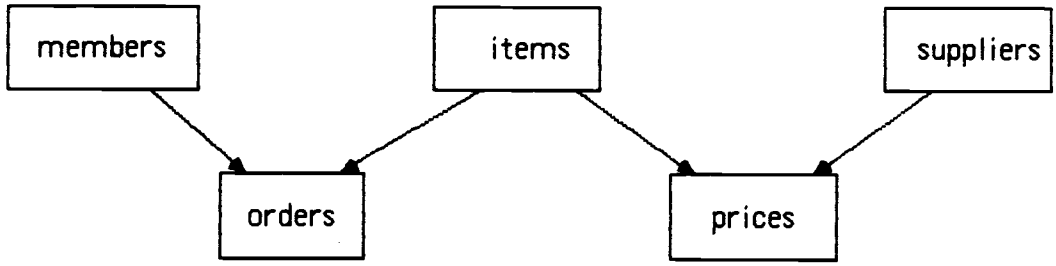

Figure 4.2.7. Network schema for three query examples.

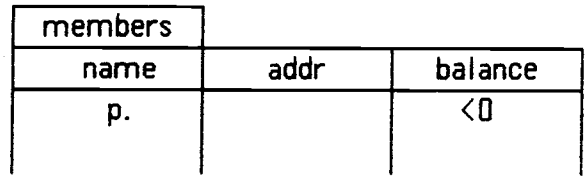

Figure 4.2.8. Example 1 in Allegro Query.

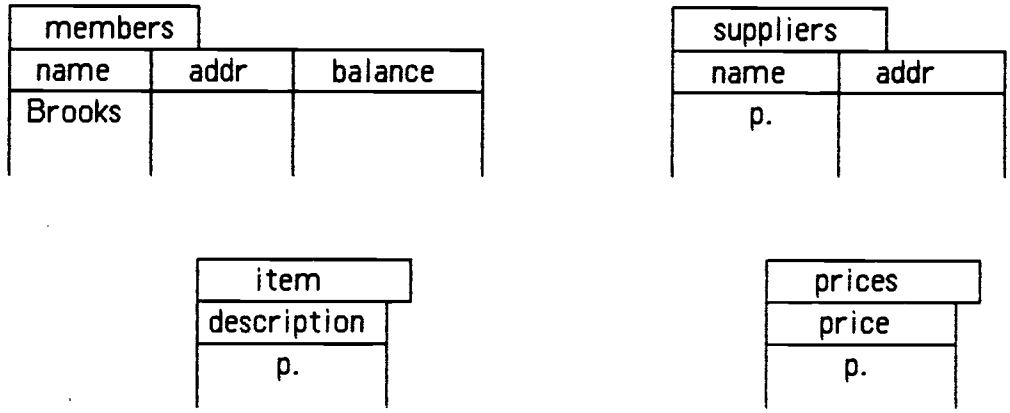

Figure 4.2.9. Example 2 in Allegro Query

The third example will need specification of Allegro Query set theoretic functions to write it. When so specified, the third example would be expected to look much as the QBE example does.

Chapter 5

Examples of Allegro Query.

The figures of this chapter are actual examples of Allegro Query obtained by the Smalltalk function 'copy display'. These are full page figures following the chapter text.

The schema BOOK_AUTHOR is a schema used by Uy[4] and in most of the work on Allegro. It is the schema used in most of the discussion in the text of this report. The Allegro program sample2 is a question and answer implementation of two queries on BOOK_AUTHOR. The first of these two queries prepared in Allegro Query is shown as figure 5.1.

At some previous time the system generation actions resulting in the Schema Name View (middle right) with the schema names shown had been done. Earlier in the session, we also had logged in to proper directory on the VAX. The schema name selection was changed to the highlighted BOOK_AUTHOR and 'fetch schema' selected. This displayed three overlapping record views. The record views AUTHOR and BOOK_AUTH were moved with blue button 'move' to obtain the triangle shown. 'show sets' was selected as that is the way the author likes to use Allegro Query. The sets are superfluous for a schema as simple as BOOK_AUTHOR. The request line shown in BOOK (highlighted) and also in AUTHOR were filled in. 'do query' (Schema Name View yellow button) was used directly not bothering to do any 'accepts'. The result is the dataOut view as shown where the top and bottom boundaries show that the entire data is seen without scrolling the view.

A more complex schema, student_rec, is used for figure 5.2 and figure 5.3. This schema includes six data records, a system record, and nine sets with multiple paths everywhere. This schema has not been entered in Allegro. The communication line is redirected to dummy files when using schema

student_rec. The schema and data files used were hand prepared from a study of structures in Allegro routine dbcs, the formats in Uy[4], and what actually does transmit for BOOK_AUTHOR.

Figure 5.2 shows a simple query for a student transcript. The full schema and set lines are shown. There are entries in four of the schema records including a calc field in the student record. Due to font size for numbers it is necessary to scroll 'student ss nbr' to see the full entry. The entry is typed without concern for the scrolling. When 'do query' is selected, Allegro Query detects the possibility of multiple paths to answer the query and responds with the Navigation View. In this view the 'name sets' menu option was selected, followed by three sets from the list shown on the left, then 'do query' from the Navigation View menu. With only these three sets to search, Allegro Query has a unique path. It proceeds silently to yield the dataOut view shown. The lack of a bottom boundary indicates that only part of the data is shown. The rest of the output data may be seen by scrolling the view or in the case of limited data by enlarging the view vertically.

Figure 5.3 shows the same student transcript query. This time the option to trim the schema by removing the department and faculty records has been used. 'do query' was selected as in the previous figure. This time the result appears without having to specify a navigation path.

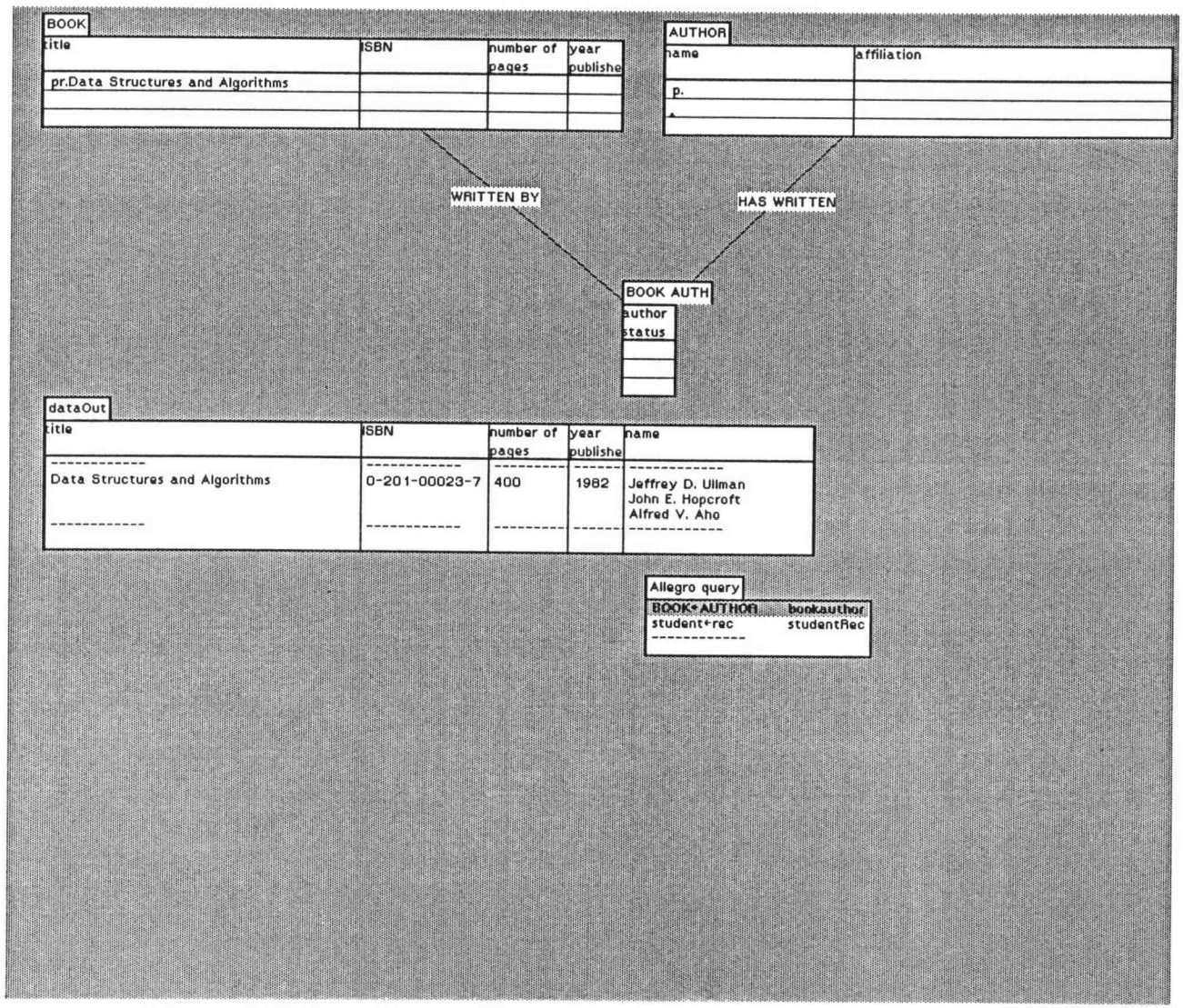

Figure 5.1.1 Screen Dump of BOOK_AUTHOR example.

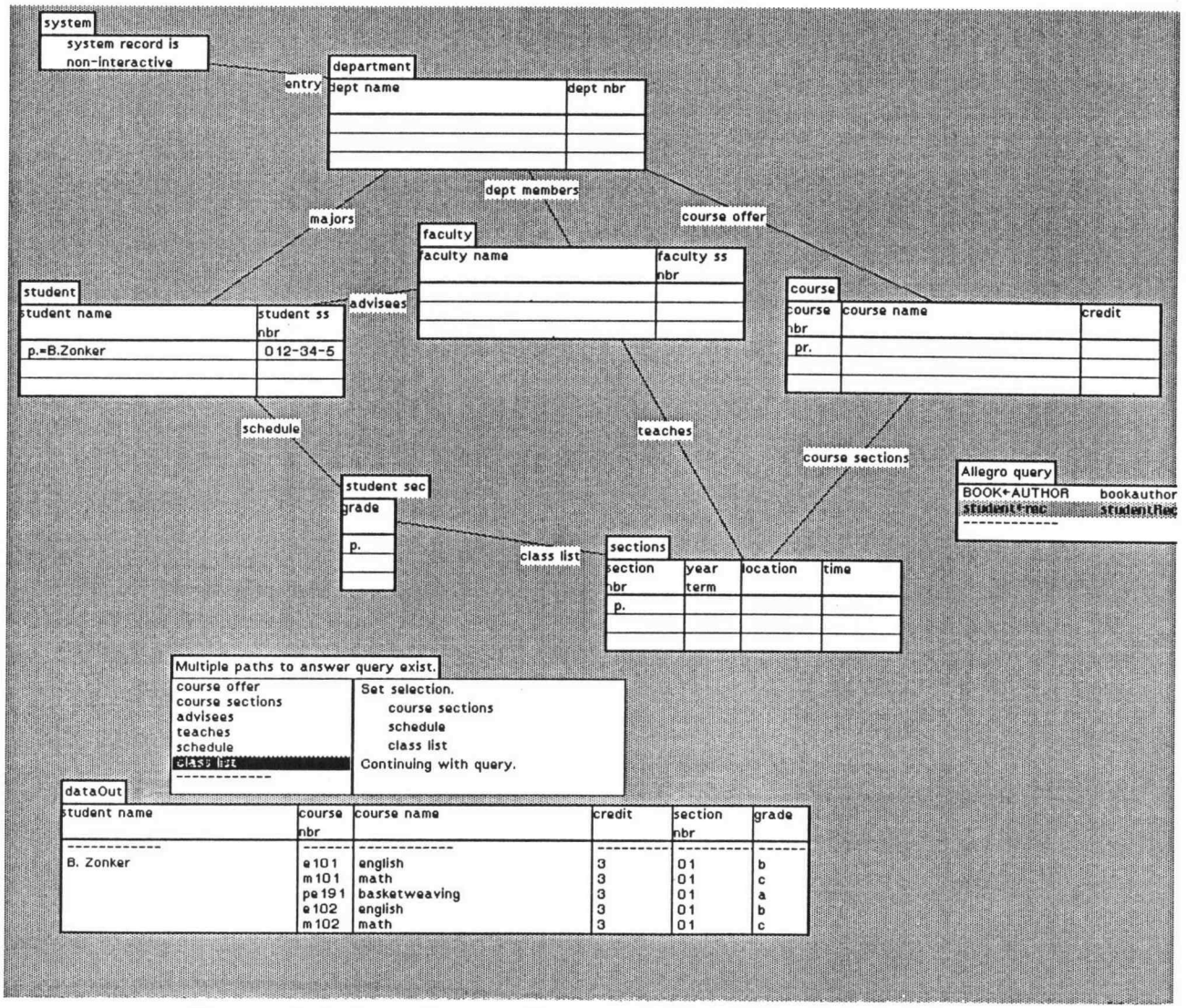

Figure 5.1.2 Screen Dump of 'student_rec' example.

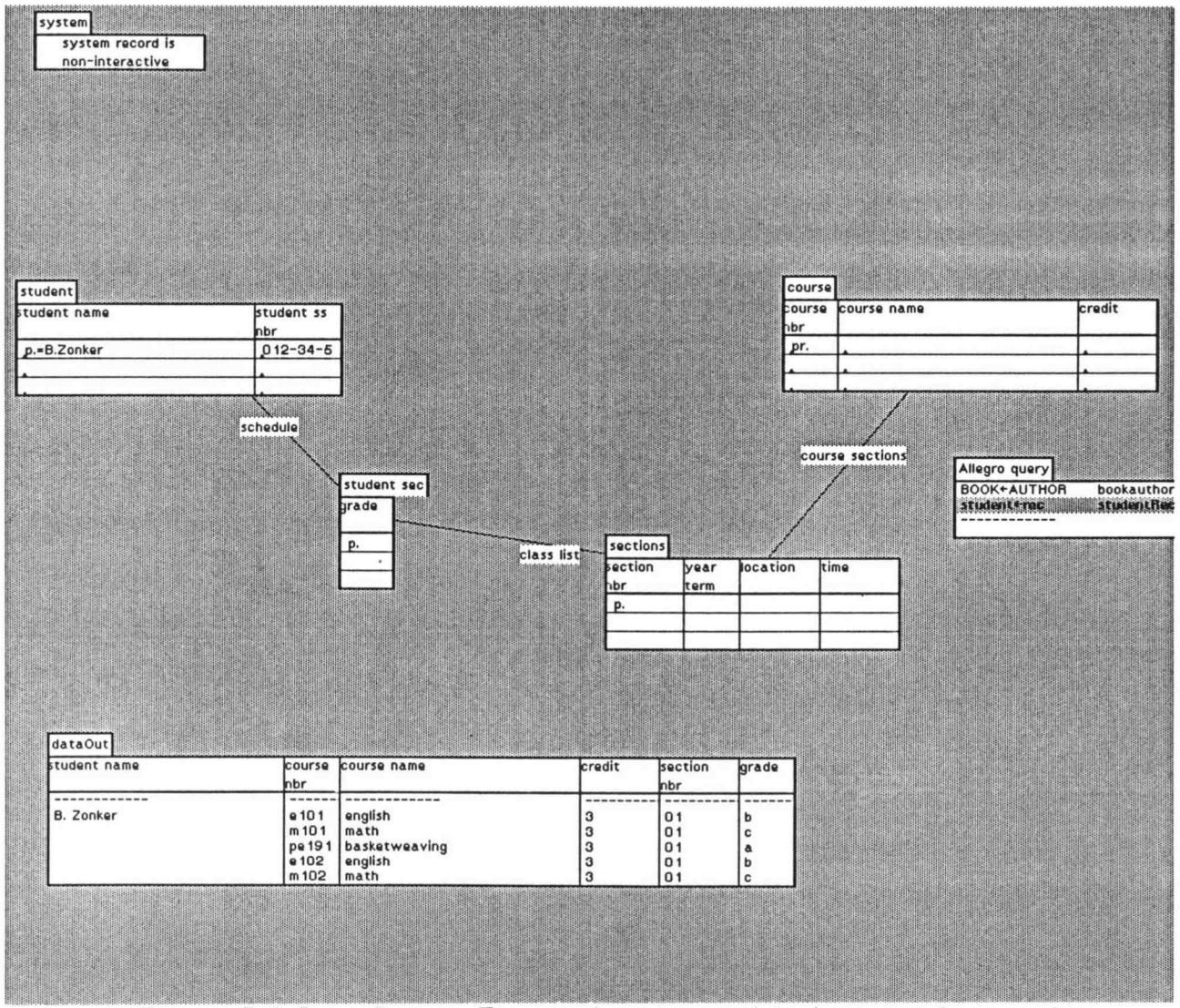

Figure 5.1.3 Screen Dump of trimmed 'student_rec' example.

Chapter 6

Allegro Query Internal Structure

6.1 Query Processing.

The implementation of Allegro Query follows the model/view/controller triad of the Smalltalk display implementation.[15] The structure of the model section of Allegro Query follows Allegro's main data structures. At the top there is a query instance which can process a series of queries. One of the instance variables of the query instance is an instance of class Schema. An instance of class Schema has several instance variables. Most of the variables are collections of various things. Two of the instance variable collections are a collection of schema records and a collection of schema sets. The elements of the collection of schema records contain instances of class Record. One of the instance variables of a record instance contains a collection of instances of class Field. An instance of class Field again contains several things one of which is a collection of instances of FieldModel. These FieldModel instances contain the user query for that field. Another variable of the Field instance contains the data received from Allegro for that field. Another variable contains the data that remains after selection processing.

This section is a brief outline of the code of Allegro Query on the query side. Within Smalltalk, Allegro Query exists as a series of class definitions. Each class defines the object it represents with a series of variable definitions and the specific methods which operate on those variables. The order following is class by class as the reader would see them if he had done a 'print out' of the category 'Allegro Query'. The actual query processing happens to be near the end.

The class OutputController is used to scroll all the field output data views of a record in lock step on one record scroll bar.

The class FieldController handles the yellow button control actions for the request views. It disables scroll bars on individual field lines. Actually the standard editing actions are handled by the Smalltalk defined superclasses of FieldController. This class is dealing with the sensitive ParagraphEditor.

The class RecordBBController contains overrides and extensions to SystemStandardController. The record view open, close, and move actions are here. It appears to the author that Smalltalk view controlling is intended to work with one view at a time. There are multiple views for the network schema. There was considerable trouble with view scheduling early in the project until it was decided to write this override class and do the scheduling explicitly. This class is playing with fire and reminds us of that every time we touch it.

The class Field defines an object which parallels Allegro's field structure. Each field in a schema has an instance here. This is mostly a data structure. There is not much processing here.

The class FieldModel has instances which contain the models of the paragraphs displayed as each field/line of the request subview of the record view. The user edits query input until he is satisfied with it. When he selects 'accept' or one of its equivalents his input is deposited in an instance of FieldModel. The parse of the user input is the main processing of this class.

Class Query has the overall processing for a query instance and the instance of a Schema Name View associated with it. The Schema Name View yellow button menu selections are received here. The interface to Allegro for 'fetch schema' is the main processing of this class. The schema name add and remove are here. Most of the rest of the yellow button functions are passed on to the selected instance of class Schema (see below).

The class QuerySet defines an object which parallels Allegro's set structure. There is an instance for each set of the schema. Set line display is here. The high use method 'pathCheck:' is here.

Similarly class Record has an instance for each record in the Allegro schema. The method 'checkInQuery' is here. It is a series of double loops through Field instances to FieldModel instances to determine that record's participation in the query.

Class Schema is the big class of Allegro Query. Its main function is to do the query processing that results when 'do query' is selected. This class is discussed protocol by protocol, the Smalltalk breakdown of a class. The protocol 'schema display' implements those SchemaNameView yellow button display actions that were passed on by class Query. The protocol 'process query external' implements 'accept all' and 'do query'. The protocol 'process query' contains some of the internal Schema class methods for processing the query. The 'data display' protocol contains more of the internal methods for 'do query'. The methods relating to query paths are in a separate class (see below). There are other less significant protocol sections.

The method 'doQuery' starts the significant action of this class. After initialization the records that participate in the query are collected. This uses method getRecords and can get involved with 'accept all' and a second pass through getRecords. 'getRecords' loops through the record instances checking a variety of flags. The result of 'getRecords' is a collection of instances of class Association that represent the records which participate in the query. The entry to the schema, either a calc record or the system record will be first in the getRecords collection. Each association represents one record and a currently nil path to the record. If there is more than one record in the query the query needs path data via method 'getPath:'. This is discussed below.

After any path work 'do query' continues with method 'doQuery2'. The record associations now have path data filled in. The method 'generateAllegroQuery' is used to generate the commands to the Allegro section of Allegro Query that will be sent across the communication line. It is also discussed below (section 6.2). The query is written by 'writeQuery', then Allegro Query waits to read the returned data with 'readQueryData'. The method 'readQueryData' converts the data from the communication line to the various collections which are instance variables of the various field

instances. It prepares the presentation of data along chain lines which maintains the perception of the network data structure. The structure of the output data view is prepared by checking which fields are to be projected. Finally 'doQuery2' uses method 'processQueryData' to apply the query selections and fill in the data view.

The method 'processQueryData' is currently a data collection copy. It will expand greatly if query selections are implemented (future work). When implemented the several methods that will exist here may have to be split off from class Schema for the same reason as class SchemaPath (next).

The class SchemaPath is logically a part of class Schema. It was split from class Schema when Smalltalk 'oops' limitations were encountered. The entry to the methods of SchemaPath is 'getPath:' It initializes then starts a depth first search of the schema network by calling method 'followPath:of:' at the query entry node, either the appropriate calc record or the system record. The method 'followPath:of:' assembles the out sets of the current node then recursively follows each. The used out sets are taken from those out sets marked available as explained in Chapters 3 and 4. Unless a node that has been visited before is encountered, the algorithm will search the entire available schema. If it encounters a previously visited node, there are multiple ways to get to this node and any node back along the current path and any node along the path that resulted in the original visit to the current node. If any of these nodes or any node that is later visited from these nodes represents a record used in the query the algorithm backs out the search then asks the user for a path selection. If the search backs out without having found multiple paths it checks to see if all query used records have been visited. If no path to all the used records is found, the user is also asked for a path selection.

The eleven methods in the 'user path' protocol of SchemaPath are those used to interact with the user navigation view. When the user selects the 'name set' method of path specification Allegro Query initializes all the schema sets to not available then marks available only those the user specifies. The use of 'do query' starts query processing over ignoring previous processing. When the user selects 'record paths' method of path specification,

his selections simply fill in the path sections of the queryRecords associations described above. In this case the query resumes.

The class NameView is used in the display of field names in the record views. Its main processing is to attempt to present the Allegro field name reasonably.

The class RecordView is all class level protocol. It generates and displays the various schema record views.

6.2 Interface to Allegro.

At the outset it is acknowledged that this is the weak portion of the Allegro query facility. This has not been the area of emphasis of the project. It works on the schema tested. A more complex schema has been tested through retrieval command generation and supplied data return. The following first considers the code used to retrieve data then the physical interface. The terms user side and VAX side are used with obvious meaning.

Allegro Query obtains two forms of data from VAX side Allegro, schema description and database data. For each usage (and for any usage of Allegro DML) the first action is to open the schema desired. As described in the user interface above, the user of a query instance has selected a schema to use. Allegro Query uses the selected schema information to provide Allegro DML OPEN with schema name and data filename. The open action makes schema description tables available to the Allegro DBCS. When the user side request is schema fetch, the VAX side of query ships portions of the schema tables to the user side, then exits.

The code used to retrieve data was intuitively designed to walk specified paths in a network schema. It starts from whatever entry point into the database has previously been found and proceeds to whatever records are necessary to answer the query. At this point the path has been specified. Allegro Query codes the retrieval by generating a collection of retrieval statements from path information. The method used, generateAllegroQuery in class Schema, is a depth first pass down the paths in the collection

queryRecords. Each generated statement consists of two chain pointers and one of four pidgin DML statements. The VAX side reads these statements and makes the appropriate procedure calls. There is a procedure for each of the four pidgin DML statements. These procedures make the DML calls necessary to perform its function at the Allegro DBCS function call level. The procedure to implement GET returns data to the user side. At the end of the retrieval statements the VAX side of query exits.

The two statement chains, in statement order are termed 'next' and 'sub'. The 'sub' chain is used when the retrieval moves from one schema record to the next along a path. Its value, if not nil, is always the next retrieval statement. This chain is followed recursively on the VAX side. The 'next' chain is used when there are actions following the first at a record i.e. a GET and a query used set path out of the record, or more than one query used set path out of the record. When not nil and the 'sub' chain is in use the value of the 'next' chain is a forward reference. The recursion of method generateAllegroQuery handles the forward reference without a fix-up pass.

The four pidgin DML statements are:

```
calc record_name calc_key
follow set_name
owner set_name
get record_name
```

Of these statements only 'follow' should require an explanation. On the VAX side 'follow' is implemented with FIND FIRST followed by as many FIND NEXT's as are necessary. After each FIND the sub chain, if not nil, is followed.

For the query entry (figure 5.1):

```
pr.Data Structures and Algorithms
p.
```

in records BOOK and AUTHOR respectively of the schema BOOK_AUTHOR.

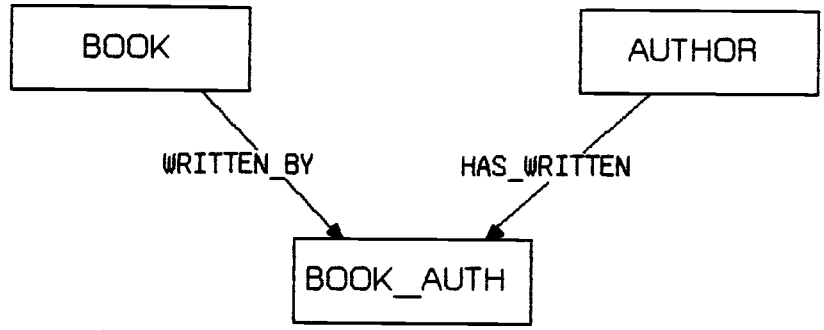

Figure 6.2.1.. The schema BOOK_AUTHOR.

the retrieval code is:

```
5 @cmd start@
2 0 1 calc BOOK Data Structures and Algorithms
1 2 0 get BOOK
1 0 3 follow WRITTEN_BY
1 0 4 owner HAS_WRITTEN
1 0 0 get AUTHOR
```

The first line and first column are implementation items. The second and third columns are the 'next' and 'sub' chains.

The physical interface is coded to use the 4404 remote function, a serial line RS232 connection. The serial interface was chosen due to the remote function's ability to be used from a program. The user prepares the connection using remote, then key F1 to temporarily leave remote, and either enter or return to Smalltalk.

The VAX side connection is prepared by logging in on the VAX then moving to the appropriate directory. This is done by the user. Both of the Allegro Query functions, schema fetch and data retrieval, send the command line 'query' as the first line. Each then waits until the VAX query function has responded showing its presence. They then send schema identification and their request. The VAX side exits after returning data. Data transfer is by blank and new line delimited character streams. The underscore is used to handle blanks embedded in the data. Any underscore character in the data will be represented as blanks on the user side.

Chapter 7

Future Work.

This section will show Allegro Query, as implemented and described above, to be only a feasibility study. While most of the items that follow were in the original design, it was with awareness that they would be deferred to future work. A full query system is not a one person project. The emphasis has been on the visual display and query navigation. The presentation is in a suggested order of priority.

The query language selection operator code is not implemented. This includes the relational and logical operators of section 4.2. The design is to apply query selections along retrieval paths. While this has not been investigated, it is expected that the selection operator code will be straight forward. The method processQueryData will expand to several methods. The size problem we had with class Schema is expected to reoccur.

The definition of the query language should be filled out as described at the end of section 4.2. The current language is so simple that a formal parse system was not used. Since language entries are discrete for each field/line/record a formal parse will probably not be necessary even for the full language This should be done in conjunction with the selection logic above as the full language will complicate that logic considerably.

A forms mode is expected in current query implementations. The current dataOut record will not be sufficient. It would be best if this mode would take advantage of the display capabilities of Smalltalk as described in Chapters 18-20 of Goldberg.[13] The author has not investigated this area of Smalltalk.

The ability to do a local query should be added. This will be useful for changes in projection and selection in the query specifications. Two additions are required. Allegro Query must be told in some manner to get all the applicable data. A schema with a system record will be required as Allegro lacks a utility to retrieve all the calc keys of a record. In a user-unfriendly manner this could be left up to the user. An entry in any field of the record retrieves the whole record. All the user has to do is code one field of each data record from the system record to the desired record with a relational operator '>' and a minimum value constant. With the current implementation this would fetch the data. A more reasonable way for the user would be a command option in the desired record, for example 'all.'. The second change is the addition of a 'do local query' menu option adjacent to 'do query' in the Schema Name View. The problem with this feature is expected to be Smalltalk size limitations.

The path navigation interface would be more user friendly if it had an option to display example paths to answer a particular query. The examples would be set line displays of actual paths with the shortest path shown first. The user would have a next/previous capability to see other navigation paths until he has the path he wants. This will be somewhat harder than the current implementation. In addition to path identification, a better method for redrawing set lines would be needed.

The record view yellow button options 'resize field' and 'remove field' are not implemented. The implementation of these will probably involve record view display. This is always a touchy area full of surprises.

Error processing of the query entry on 'accept' (or its variations) is not implemented. Currently Allegro Query expects clean query request input. The syntax is simple, so this should not be much of a problem.

There is no consideration for Allegro errors or communication errors The case of immediate importance is missing calc key; for example if a requested book is not entered into the data of the book record of the BOOK_AUTHOR schema.

Currently scrolling in the request fields occurs more often than it should, causing query entries to disappear from view. This is corrected by positioning the cursor in the view and dragging it through the top or bottom of the line. This is dealing with the ParagraphEditor. The code for ParagraphEditor is very long and it is only the tip of the iceberg. There are many temporary objects each with their own class definition.

With an added function in Allegro (VAX side), Allegro Query could implement an 'update schema names' Schema Name View menu item. This function would replace 'add schema name'. It would relieve the user of the burden of knowing the precise schema name and filename of a schema.

When the network schema in use has calc keys, the schema name view could carry a yellow button entry 'show fields'. The information to do this is currently present.

A name completion facility on query request constant elements would be useful. Some of the book names in BOOK_AUTHOR are rather long. The retrieval considerations for 'do local query' would get the necessary data. With the data this facility would be just an enhancement to the selection code.

The data out view is non-interactive. A user desire to cut items from the data output display and paste into a new query request has been noted. This should be little more than the addition of an editing menu.

There are several assumptions in the design of Allegro Query:

One member record type sets are assumed. The generalization to multiple member sets is not expected to be trivial.

Schema record calc keys are restricted to one field. This assumption should be reasonably easy to generalize.

Certain places in the code assume that schemas contain only one set between any two records. This assumption should yield to recoding.

In developing a user version of this feasibility study these assumptions would have to be removed.

Chapter 8

Summary

This thesis has prepared a query facility for the network database system Allegro. This facility is implemented in Smalltalk on the Tektronix 4404 system. It is about as distinct from the main Allegro system as it is possible to get, a result of the distinct machine and distinct language implementation.

In the section on Query-By-Example the flavor of QBE was showed by quoting Zloof's conclusion on the features of the language. Allegro Query has all of these features intended to be appealing to the causal user. Some of Allegro Query's features are:

1. The user has the perception of schema record manipulation as he manipulates display objects which represent the schema records. These record display objects are the frame of reference. The user can position these records. The user can see the schema sets if he wants.
2. The user can identify the records to be used by removing those he does not want. This is an reduction in the scope of the query.
3. Allegro Query is declarative in nature. The user has freedom in the order in which he will specify his query. There are a few constraints concerning what is and what is not on the same line but these are intuitive. Some complicated queries are simple to state.
4. The user does have the constraint that any path specification occurs after the rest of the query is entered, but in many cases he does not have to specify path navigation.

BIBLIOGRAPHY

[1] Zloof, M.M. Query-By-Example. AFIPS Conference Proceedings, National Computer Conference 44, 431-438 (1975).

[2] Zloof, M.M. Query-By-Example, The Invocation and Definition of Tables and Forms. Proceedings of the International Conference on Very Large Data Bases, Boston Ma. Sept. 22-24, 1975 pp. 1-24.

[3] Zloof, M.M. Query-By-Example: A Data Base Language. IBM System Journal V16,4 1977.

[4] Uy, M.L.Y. A Network Data Base Management System. M.S. Thesis. Department of Computer Science, Oregon State University, 1983.

[5] Swasdichai, C. Extension to Allegro, A Data Base Management System, Research Paper, Department of Computer Science, Oregon State University, 1984.

[6] Jarke, M. and Vassiliou, Y. A Framework for Choosing a Database Query Language. ACM Computing Surveys V17,3 (Sept 85).

[7] Bachman, Charles W. The Programmer as Navigator. Communications of the ACM V16,11 (November 1973).

[8] Bachman, Charles W. Integrated Data Store. DPMA Quarterly, Jan 1965 pp 61-80.

[9] Bachman, Charles W. Data Structures Diagrams, Data Base 1,2 (1969) Quarterly Newsletter of ACM SIGBDP pp 4-10.

[10] CODASYL COBOL Committee, Journal of Development, Ottawa, Canada. Canadian Government Publishing Centre, Supply & Services Canada 1981.

[11] Date, C.J. An Introduction to Database Systems. 3rd ed. Addison-Wesley, 1981

[12] Ullman, J.D. Principles of Database Systems. 2nd ed. Computer Science Press, 1982.

[13] Goldberg, Adele and Robson, David. Smalltalk-80 The Language and its Implementation. Addison-Wesley, 1983.

[14] Goldberg, Adele Smalltalk-80 The Interactive Programming Environment. Addison-Welsey, 1984.

[15] 4404 Artificial Intelligence System Introduction to the Smalltalk-80 System Part No 070-5606-00. Tektronix, Inc. Beaverton, Or. 1984.

[16] Taylor, R.W. and Frank, R.L. CODASYL Data-Base Management Systems. ACM Computing Surveys V8,1 (March 1976).

[17] Reisner Phyllis Human Factors Studies of Database Query Languages: A Survey and Assessment. ACM Computing Surveys V13,1 (March 1981).