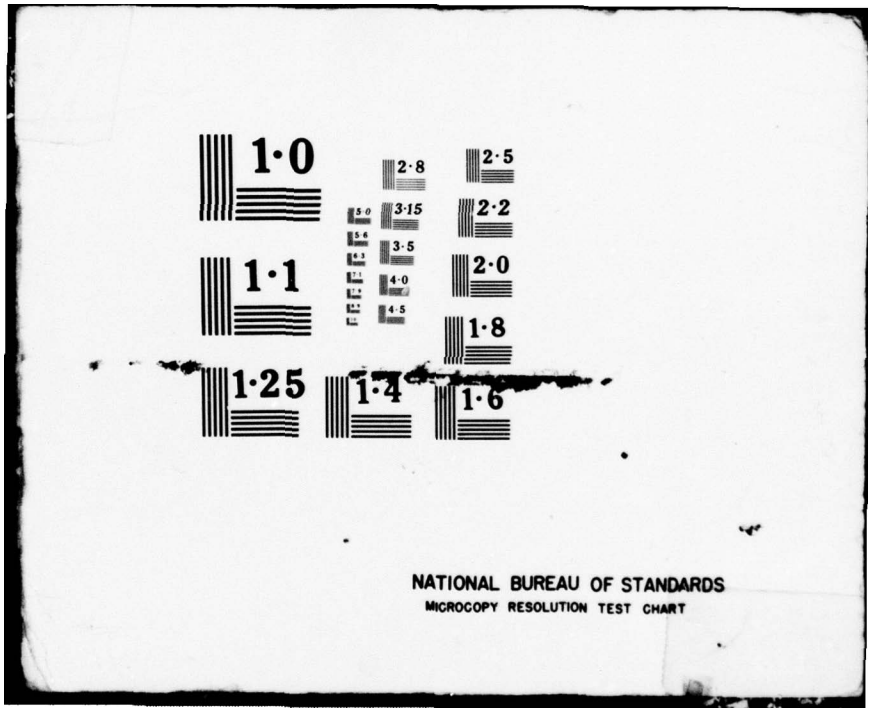
AD-A067 015 WHARTON SCHOOL PHILADELPHIA PA DEPT OF DECISION SCIENCES F/G 9/2
EXTENSIBLE DATA BASES.(U)
MAY 78 J W HAYWARD N00014-75-C-0462
UNCLASSIFIED 79-03-08 NL

1 OF 1
AD A
067015

END
DATE FILMED
6-79
DDC


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
4·5
NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

(12)

EXTENSIBLE DATA BASES

Jonathan W. Hayward

79-03-08

Department of Decision Sciences
The Wharton School
University of Pennsylvania
Philadelphia, PA 19104

Research supported in part by the Office of Naval Research
under Contract N00014-75-C-0462.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER (14) 79-03-08	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) (6) Extensible Data Bases.	5. TYPE OF REPORT & PERIOD COVERED (9) Technical Report. Apr 78 - Mar 79.
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s) (10) Jonathan W. Hayward	8. CONTRACT OR GRANT NUMBER(s) (15) N00014-75-C-0462
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Decision Sciences University of Pennsylvania Philadelphia, PA 19104	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR049-252 272
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research	12. REPORT DATE May 1978 13. NUMBER OF PAGES 61
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (11) May 78 (12) 68 p.	15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

databases, query languages, decision structures

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Traditional database systems only allow queries to be made about the data within the database, rather than about the data as well as the structure of the data. A set of primitive operations is proposed which allow a flexible means of entering and exploring the data and the structure of a database.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

408757

UNIVERSITY OF PENNSYLVANIA

THE MOORE SCHOOL

EXTENSIBLE DATA BASES

Jonathan W. Hayward

Presented to the Faculty of the College of Engineering and Applied Science (Department of Computer and Information Science) in partial fulfillment of the requirements for the degree of Master of Science in Engineering.

Philadelphia, Pennsylvania

May, 1978

Thesis Supervisor

Graduate Group Chairman

ABSTRACT

EXTENSIBLE DATA BASES

Jonathan W. Hayward

Peter Buneman

Traditional data base systems only allow queries to be made about the data within the database, rather than about the data as well as the structure of the data. A set of primitive operations is proposed which allow a flexible means of entering and exploring the data and the structure of a database.

Extensible Data Bases

Jonathan W. Hayward
May 1978

1.0 INTRODUCTION - SOME QUESTIONS THAT CANNOT BE ANSWERED

Many models have been proposed for database systems, and even more implementations of databases exist, but several problems still remain. There are three problems which I would like to discuss in this paper.

1 - There is a lack of data describing the content of a database within the database.
2 - Most database systems do not have provisions for denoting similarities between the different "entities" that the database describes.
3 - Most database systems only allow a static representation of the world.

This paper consist of seven chapters:

1 - Introduction
2 - Current data base models - covers the major models used in database systems to date, and defines the terms that are often used in regard to databases.
3 - Introduction to extend and restrict - a set of primitive operations is proposed for defining and accessing a database.
4 - Query languages - describes some minimal query languages that can be used for exploring the structure of a database, as well as the data within the database.
5 - Decision structures - describes decision structures, and relates them to extend and restrict.
6 - An alternative data manipulation language for SEED - describes a query language for a network database system that does not have the drawbacks of typical

network DML.
7 - Conclusion

1.1 Lack Of Data Describing The Data Base

There are two classes of information that may be unavailable to the user of a database. One is information concerning the structure and content of the database. The other class is information pertaining to the exact names of entities within the database. Natural language systems generally have both types of knowledge built into them, and are able to answer queries which assume the user has some knowledge of the database structure. But databases, or any query system built on top of databases do not have the ability to directly answer question about their structure. Consider the query "Tell me about a city" versus "tell me about Philadelphia". The latter question could be answered by many query systems, but the former question could not. Yet, if the former question could be answered, perhaps query systems would not have to be database dependent. The need for natural language query systems would not be as great, if database users could explore the structure of the database themselves.

1.2 Similar Record Classes

Data base systems generally are conceived with two constructs: 1 - Classes of objects can be defined (usually known as records); 2 - Functional dependencies can be created between the elements of one class of objects and the elements of another class of objects (known as sets). Most data bases can be described entirely using the 2 constructs above. However, two often needed constructs of databases cannot be described directly: Many to many relationships and similar record classes. "similar record classes" refers to sets of objects such as lawyers, doctors and students which share a common core of information ("people"). Many to many relationships are captured without much difficulty in confluencies, but there is no prefered method for implementation of similar record classes.

1.3 Flexibility

If a relationship has remained undefined in a database system, it is very difficult to add that relationship at a later time in all but relational database systems. Dynamic definition of relationships is not allowed. Yet, as new problem areas use a database as a common store of information, new relationships are "discovered" that were previously ignored. Old relationships that were thought to have existed may be proven false. It is interesting to note

that the "world" has not changed, just one's view of the world. The statement that there is a correct design for any given database is false in the sense that the design is limited to the view (both current and future) that the database designer sees.

Proponents of relational database systems use the argument above to illustrate the desirability of the relational model. However, the relational model captures the ability to create dynamic relationships at the expense of a total lack of descriptive information about the structure of the new relations.

1.4 Where Does The Problem Begin: Entities And Attributes

In looking at the question, "Tell me about Philadelphia" notice that there are not any quotation marks. yet, when we use programming languages, we use quotes often to distinguish literal data from names of data. In terms of data base models, the distinction is made by calling some items entities, and some items attributes. Our concepts about data base structure is heavily based on this distinction. Records represent entities and the have fields which describe the attributes of the entity represented by the record. Record classes represent sets of entities of the same type. Further complicating the problem in determining the difference between literal and non-literal

data in a query is the fact that attributes have names (or descriptions) as well as values associated with them.

If I asked someone that was familiar with data base systems to structure a model which contained information about all the people that live in Philadelphia, one common part would probably the definition of a "people" record which contains attributes such as name and social security number. Also suppose that the data base is completely invertible so that if I ask "Who is Howard", it is not necessary to specify the name of the attribute which contains "Howard". In the limited world of people that live in Philadelphia, such a system presents no problems. In the very statement of the problem, I have defined what entities we are talking about (people). However, if the problem scope was extended to include more detailed information about the residents of Philadelphia, the problem is no longer clear. If we wish to add occupation to the data base, it could be added as an attribute of each person, but that is not really adequate since an occupation could also be considered as an entity. As more information is added to the data base, it becomes less clear where the distinction between entity and attribute lies. In the case of answering "what is a doctor", one would like to make questions about the structure of the entities by treating the structure of the entity as an entity. But just as "doctor" can be considered an entity, it can also be considered an attribute

of a person. In natural language we easily switch our meanings from attribute to entity: There are no quote marks. We structure our knowledge in such a way that we can concurrently classify words, items, or whatever as either entities of one set, or attributes of another set.

1.5 What Help Is Available

Two options are generally available to the programmer of a query system. The information about the structure of the data base can be stored in the program as data, or as program code. In frame systems, for instance, the "world knowledge" is often within programs contained in the frames, but the frames are considered data.[6] Semantic nets are sometimes used to describe the nature of queries that are made. But again, rather than having information represented as establishes data structures, it is often encoded into a program. A simple dictionary of common names (as in the data dictionary) is an example of a minimal query system which is based on a data structure representing the relations between various record classes in the data base. All of these approaches are dependent on the structure of the data base. If the structure changes, the program must be changed.

INTRODUCTION - SOME QUESTIONS THAT CANNOT BE ANSWERED

In order to answer the question "tell me about philadelphia" i would like to propose that the query system should make queries to the data base system about the structure of the data base. With the information that is returned, more specialized queries can be made, until the initial question is answered. In such a system, the structure of the data base need not be defined in advance and need not be static for a query system to maintain it's usefulness.

I am going to propose a free format data base system. No restrictions will be placed on the structure of the system at any point in time. The structure will be determined by a combination of all previous definitions. I will show how network data base concepts such as "set ownership" can be introduced without violating the premise of free structure.

2.0 CURRENT DATA BASE MODELS

Network, hierarchical, and relational data base models carry many of the same concepts. The details of the restrictions (such as third normal form for relational models) place different constraints on a particular model for an application, but three concepts define the common ideas between them.

2.1 Records And Fields

Record classes are defined in all of these models. In relational terms, a record class is equivalent to a "relation". A relation, or record, is usually formed from a set of lower level attributes. The attributes contain the ultimate data of the data base. A particular record is the set of attributes representing one entity in the real world. For instance, a person record might contain the attributes name, social security number, address, city, age, etc. A collection of records (or relations) of this type (people) is called a record class. The individual attributes are stored in the "fields" of a record.

2.2 Set Ownership

If one had a significant amount of information about the city in a "PERSON" record, one might place the

information in every record in the data base. The problems that arise from placing this redundant information in the data base relate to data storage, update problems, and clarity.

One of the main objects of data base models has been to provide easy translation from the model into real online mass storage. Because of this desire, it is unreasonably wasteful to duplicate large amounts of data such as details about a city in every person record.

In order to update one piece of information about a particular city, it would be necessary to find all the records where the information was stored, and then to update them all. Guaranteeing that all of the data has been updated properly becomes a major problem.

The third reason for wanting to not repeat the data, and perhaps the most important, is the desire to say something about the structure of the real world in the structure of the model. It is not clear from defining one large person record containing a lot of information about a city, what particular parts of the record relate to the details of the city.

For these reasons, the information pertaining to objects which themselves can be considered entities (in addition to attributes), are broken out of the original record Since there are a large number of people that have

the same city information for any particular city. The "city record" owns a set of "people records". Note that a set is a function from the owned record to the owning records. The owned record are functionally dependent on the owning record.

2.3 Confluency

A functional dependency is only one type of relation possible between entities. It is possible to also have a one to one relationship. This represents no problem, because if a relation between entities is one to one, a larger aggregate entity made. If the relation ship is many to many however, it is not immediately clear how to represent it. This problem is solved by breaking a many to many relationship down into two functional dependencies. A common record class is formed between the two records which have the many to many relationship. This record class is called a confluency, and it has a functional dependency to each of the record classes that have the many to many relationship. As an example, suppose we wanted to describe the relationship between doctors and the cities that they serve. We believe that there is a functional dependency from patient to doctor (i.e. a doctor owns a set of patients) and a functional dependency from patient to city (i.e. a city owns a set of patients). Then the patient record class represents the confluency. It fully the

describes the many to many relationship between the doctor and city. It also interesting to note that the patient record class contains other information with it than just it's relation between doctor and city. In the real world, it usually turns out that whenever a confluency is created, some additional useful data can be added to the confluency. In many cases, there is no useful name that can be assigned to a confluency. If we want to describe the relation between students and classes, enrollment is the obvious confluency. Grade is the additional piece of information that can be a added to the enrollment. But, there is no useful name that can be attached to a particular instance of an enrollment. One reason for this may be that there is no real world object corresponding to the entity enrollment. Enrollment is only a conceptual entity.

Note that the term "confluency" is generally used only in regard to network database models. I am using it in a more general manner, to describe the generally accepted representation of a many to many relationship.

2.4 Hierarchical And Network Database Models

Network and hierarchical database systems have explicit definitions of set ownerships in the definition of the database. a network database system will allow any (non-recursive) arbitrary structure of set ownership.

Hierarchical database systems, only permit a record class to be owned by at most one record class. As the name implies, the structure produced is hierarchical. In a network database, the data is usually stored as a series pointer structure. In a hierarchical structure, the onwed sets of data are usually stored as repeating groups of information within the owning record. Pointer structures can also exist in a hierarchical database system.[4]

Retrieval of an owned record from most network and hierarchical data base systems requires that a context be established for all the owning records before a retrieval can occur. A context is generally used since a means is not provided for identifying a reference to a particular record within the database. Proponents of network and hierarchical databases argue that the lack of identifiers on records is a feature, since it requires the user to get the particular records he is interested in manipulating before making the update. This has the advantage of guaranteeing that you have the latest copy of the record, or that you know that the record has been deleted. Alternately, I feel that establishing a global context is against good structured programming techniques. In Chapter 6, I discuss a query language that we have devised at the University of Pennsylvania that allows one to remove the global contexts from queries made to network databases systems.[5]

2.5 Relational Data Bases

A relational data base is externally described by a set of record classes alone. Rules for constructing relational models such as third and second normal form, place restrictions on the types of functional dependencies that are allowed between fields of a particular record, and between fields in different record classes. A set of attributes is called the candidate key for any relation (or record class) if the candidate key uniquely determines an element of the relation. Confluencies are used implicitly, but not explicitly in relational data bases. Rules such as third normal form define the sets of a relational database in terms of functional dependencies, but the functional dependencies are not explicitly stated in the database definition. The operations for manipulating data in relational data bases are much cleaner that for network models. Since the candidate keys are part of the data of the records, no context setting or checking is required to find a particular record of interest. If the values of the candidate key fields are known, it is no problem to go directly to the record of interest. Note that when confluencies are formed, the candidate keys from the two relations which have the many to many relationship must be aggregated to form the candidate key for the confluency.[2,3]

2.5.1 The Smith's Work With Relational Data Bases -

The Smiths have seen the lack of this information in relational data bases, and have made some suggestions for a syntax to add more structure. They have two ideas. One is aggregation, and the other is generalization. Aggregation refers to the description of relations between record classes as in network data bases. Generalization refers to a new concept, which describe the similar aspects of several record classes. The Smith's have defined update rules which guarantee consistency of the data base when defined with their syntax.[7,8]

2.5.2 Generalization -

Lawyers, doctors and construction workers are all people. People have certain attributes in common, such as name, address, and social security number. Lawyers, doctors, and construction workers are different because they each have some specialized attributes different from people in general. Lawyers are licensed under a particular state's bar. Doctors have a specialty. Construction workers have a union that they belong to. The various record definitions could be listed as:

```
PEOPLE RECORD:
  NAME
  SS
  STREET ADDRESS
  CITY
```

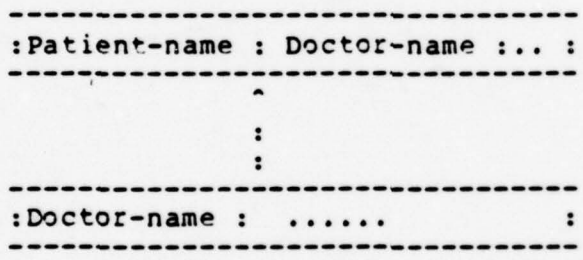
```
DOCTOR RECORD:
  SPECIALTY
  HOSPITAL
  PRIVATE PRACTICE

LAWYER RECORD:
  BAR
  LAW FIRM

CONSTRUCTION-WORKER RECORD:
  UNION
  LOCAL
  TRADE
```

The Smith's handle generalization by placing a field in the people record which might be called "profession". Profession would be limited to 3 possible entries ("doctor", "lawyer", "construction-worker") by the syntax of the data base definition. A record class defining "doctor", "lawyer", and "construction-worker" must be present. The fields of "people" may be inherited in "doctor", "lawyer", and "construction-worker", or they may not be inherited, depending on the desire of the data base designer, but at least the "key" fields for "person" must be present in "doctor", "lawyer", and "construction-worker". (The "key" fields are also given in the syntax of the database definition.)

2.5.3 Aggregation -

Aggregation is a method of defining what the "keys" to a particular record class are, and what fields within a record class contain data which are keys to other record classes, and consequently define a series of sets. For instance, suppose a doctor has a list of patients, and a patient has only one doctor. Then the person record defined above would have a doctor field added to it. In the definition of the doctor field, the Smith's would say "DOCTOR-NAME key DOCTOR", meaning that the field "doctor-name" contains data which is the key to a record in the "doctor" record class.

The Smith's would draw a diagram similar to the following:



Note that the representation of a Smith diagram is "upside down" relative to a Bachman diagram. The Smith's suggest that in the owned records, more fields can be defined as keys than just the fields which "point" to the owning record. Consequently, the Smith's are able to place a much finer type of restriction on set ownership and confluencies than are most network database system. Network database systems use only the "owning" records as keys for the

"owned", and they allow the keys to be duplicated. In a network database, for example, there would be no guarantee that the same patient does not have more than one occurence in the Patient record class. By declaring PATIENT-NAME as well as DOCTOR-NAME as a key to the the patient record, the Smith's have guaranteed that there will be no duplication.

2.5.4 Updates With Aggregation And Generalization -

All updates to a relational data base can be made quite easily, still guaranteeing the consistency of the data base because the links between different record classes are explicitly stated in the syntax of the data base definition. If a new doctor is being added, then it can be verified tnat he appears in the doctor record and the person record. Similarly, if a new "doctor-name" is added to a person, it can be verified that the doctor exists before the update is allowed.

2.5.5 Conclusions About The Smiths -

The Smith's ideas have: 1. introduced the idea of generalization, which has not been deal with explicitly before; 2. placed constraints on relational data bases which "link" record classes together, similar to the way that records are linked in network data bases.

Network data bases draw very close analogies to "aggregation". Diagrams describing aggregation usually look like Bachman diagrams turned upside down. An "aggregate" is very similar to a "confluency" in network data bases.

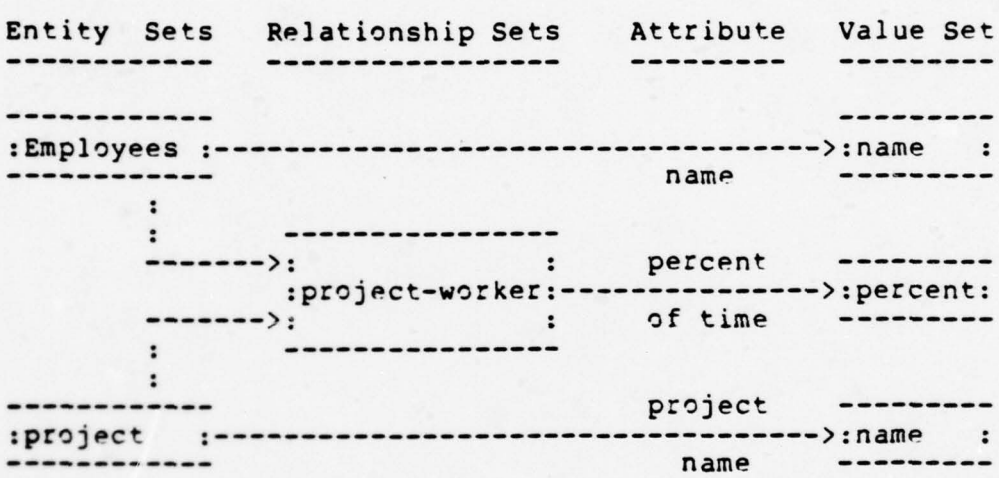
Generalization is something that can be modelled several ways in network and relational data bases. The Smith's are the first to look upon generalization as a new type of construct in data base descriptions.

2.6 Entity Set Model

One appealing break from the consistent ideas of records and fields that seems to be present in network, hierarchical, and relational database systems, is the entity-set model.[1] The entity set model defines 3 categories of sets of object: Entities, Relationships, and Values. The set of values is the ultimate data, i.e. "RED", "3", "HOWARD". The set of entities is connected to the set of values by a function (call the attribute) that maps a particular point in the set of entities to a number of points in the set of values. Entities, relationships, and value sets are further divided into subsets. For instance, 2 entity sets might be EMPLOYEE and PROJECT. Note that nothing stops an entity from being a subset of another set. For instance, MALE-EMPLOYEE is a subset of EMPLOYEE. Value sets are divided into similar classes also. There

might be a value set for NAME, DATE, FEET, etc. One is supplied with primitive functions which test if a particular entity is an element in a particular entity set. The set of relationships is connected to the set of entities, as well as the set of attributes, by a second function that maps a point in the relation set to a set of points in the entity set and attribute set.

Consider the following example:



The principles behind the entity set are quite amenable to the principles I will discuss in Chapter 3, but the implementation that is suggested for the entity set model is extremely close to a relational model, and lacks many of the types of information and flexibility I am interested in placing in the database.

3.0 INTRODUCTION TO EXTEND AND RESTRICT

I am going to define a set of operators which will allow the construction of a data base from something called the "null record". the null record is a structure with no fields. i want to incorporate the concept of generalization, and network data bases in the language. I want to break down the distinction between data and definition. The only record whose definition is known is the "null record". Other records will be constructed by adding data to the the null record. More and more data will be added, until a complete data base is defined. The result will be a structure which contains data about data about data about..... I do not want to think about implementation at this time. Implementation concerns would place too many restrictions on the thoughts I want to discuss.

The functions I want to define are "extend" and "restrict". They will take a record constructor as one of thier arguments, and produce a record constructor as the result. To make everything consistent, let us consider starting with the null record constructor instead of the null record itself. The point of dealing with constructors is to de-emphasize the traditional distinction between data and definition. A constructor is half way between the definition of a record (which initially defined the constructor), and the record that the constructor would

actually produce. Another point of dealing with constructors is that, like functions in a program, constructors have names which have no explicit connection with the function they perform. A reference to a function is distinct from a literal piece of data.

Let's suppose that we want to define a person record class. We can take the null record, and extend it in the following manner.

```
PERSON::= EXTEND ( NULL, NAME, STREET-ADDRESS )
```

A "PERSON" is now a record constructor which will create a record containing fields for a name and address when it is invoked. Notice that a person contains only information about the structure of the data base.

Some information is missing in this definition. Perhaps we would like to add city and social security number. A statement of the following form could be made to add these items:

```
PERSON::= EXTEND ( PERSON, CITY, SS )
```

The person constructor has been replaced with a new constructor that adds city and social security number.

People can be classified by the city that they live in. therefore, we might make the following type of statement:

```
PHILADELPHIAN::= RESTRICT ( PERSON, CITY = "PHILADELPHIA" )

BOSTONIAN::= RESTRICT ( PERSON, CITY = "BOSTON" )
```

Two new record constructors have been created. They are restrictions of "PERSON" in the sense tnat if a record is constructed from "PHILADELPHIAN", or "BOSTONIAN", the field "CITY" is restricted to one value (Philadelphia, for "PHILADELPHIAN"s, Boston for "BOSTONIAN"s). Note that more data has been added to the structure of the data base, but no records have yet been constructed.

I stated above that doctors were people witn a specialty. To create a "doctor", the following statement can be made.

```
DOCTOR::= EXTEND ( PERSON, SPECIALTY )
```

To create a lawyer, the following statement can be made:

```
LAWYER::= EXTEND ( PERSON, LAW-OFFICE )
```

To create a Philadelphian lawyer, tne following statement can be made:

```
PHILA-LAWYER::= RESTRICT ( LAWYER, CITY =
"PHILADELPHIA" )
```

or:

```
PHILA-LAWYER::= EXTEND ( PHILADELPHIAN, LAW-OFFICE )
```

Everybody from Philadelphia knows that all the citizenry must have a favorite "hoagie" to be a true Philadelphian. Noting this fact, the following statement can be made:

```
COMPLEAT-PHILADELPHIAN::= EXTEND ( PHILADELPHIAN,
HOAGIE )
```

Finally, we may feel the need to put some people in this data base.

```
PETER:.= RESTRICT ( PHILADELPHIAN, NAME = "PETER",
STREET-ADDRESS = "ELM STREET", HOAGIE = "CHEESESTEAK" )

JIM::= RESTRICT ( DOCTOR, NAME = "JIM", STREET-ADDRESS=
"CHESTNUT STREET", CITY = "PHILADELPHIA", SPECIALTY =
"PODIATRY" )

DICK::= RESTRICT ( BOSTONIAN, NAME = "DICK",
STREET-ADDRESS= "LIVINGSTON STREET" )
```

No records have been actually constructed!!! All that has been created are record constructors.

As the definitions of extend and restrict stand above, a seemingly pleasing structure has been created. In particular, the concept of a "*kind of*" has been introduced. A "LAWYER" is a kind of "PERSON". A "DOCTOR" is a kind of "PERSON". "DICK" is a kind of "BOSTONIAN". A "PHILA-LAWYER" is a kind of "PHILADELPHIAN". This class of knowledge is often missing in a data-base system. The Smiths have called it "generalization" and I have referred to the problem as "similar records". Extend and restrict are slightly more abstract than generalization. Restrict and extend result in a database in which the structure of data about data is the same as the structure of ultimate data.

As I mentioned before, a "PHILA-LAWYER" could be defined in several ways. The results of the different definitions would be the same. It is the fields of a

constructor, and the restrictions on the fields that determine if another constructor is of the same record class. For instance, I could create another person:

```
CARL::= EXTEND ( NULL, NAME, STREET-ADDRESS, CITY,
STATE, HOAGIE, LAW-OFFICE )

CARL::= RESTRICT ( CARL, NAME = "CARL", CITY =
"PHILADELPHIA" )
```

Carl is a "PERSON" (he has all the fields of a person constructor), a "PHILADELPHIAN" (he has a restriction on city = philadelphia), a "LAWYER" (he has a law-office), and Carl is a "PHILA-LAWYER".

If the question "what is Carl" is asked, the answers could be:

1. A constructor with name = Carl, city = Philadelphia, a street address, a state, and a law-office.
2. A Phila-lawyer
3. A Philadelphian
4. A lawyer
5. A person

If I was to ask "what is a Phila-lawyer", I would get a similar list of answers. Whatever question I ask about lawyers, Philadelphians Dick, Carl, Jim and Peter, I will always get one answer in common: "a person". "person" is the record constructor which has all the fields of the intersection of all constructors that contain any of the same fields in Dick, Carl, Jim, etc. For that reason, the person constructor is considered the "genotypical" constructor of all the other constructors mentioned above. The genotypical constructor need not have been explicitly

defined, but it will always exist for any record constructor. The genotypical constructor has an important constraint on it. In order to guarantee that no duplications exist in the data base, all restrictions of the genotypical constructor (explicit, or implicit in restrictions of another constructor) must be unique. In the example above, I cannot add anotner lawyer that lives in Philadelpnia at Elm street and whose name is Peter. If I did, an implicit restriction of person has occurred, duplicating another restriction or person.

It is interesting to consider wnat classes of responses exist in tne answers given to the hypothetical inquiry above. I am going to list the different classes by a "type number".

Type 1 - gives the explicit list of fields and restrictions on the fields of a particular constructor.

Type 2 - gives the constructors whose fields and restrictions are a subset of the field in question.

Type 3 - gives the constructors which contain the the fields and restrictions of the constructor in question.

Type 4 - gives the constructors which have any fields in common

The smallest non-null intersection of type 4 responses gives the genotypical constructor. The "data" (in the traditional sense) of the data base is a subset of the type 3 responses. In the case of a PERSON constructor, the presence of a

constructor represents "ultimate data".

So far, extend and restrict have been used in a limited sense. They have been used to illustrate the concept of similar record classes, but no problems of set ownership have been introduced.

Network data bases can be implemented quite easily using extend and restrict. We have only considered restrictions of fields that contain literal data. There is no reason to limit ourselves in this sense. The restrictions can just as easily be to other constructors. Suppose a doctor is to have a list of patients.

PERSON::= EXTEND (PERSON, DOCTOR-REF)

PERSON::= RESTRICT (PERSON, DOCTOR-REF = DOCTOR)

Note that DOCTOR on the right side of the "=" above is not in quotes. This is because the restriction is to a constructor, not a literal piece of data. By placing a restriction on DOCTOR-REF which is a constructor (the doctor constructor in this case), we have indicated that any further restrictions of person can only have a type 3 constructor of DOCTOR in the DOCTOR-REF field. The following restriction would be acceptable under these circumstances:

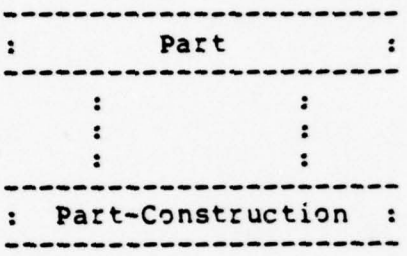
DICK::= RESTRICT (DICK, DOCTOR-REF = JIM)

The following restriction would not be acceptable:

```
DICK::= RESTRICT ( DICK, DOCTOR-REF = CARL )
```

In the latter case, since Carl is not a type 2 constructor of doctor, (ie Carl does not have the characteristics of a doctor), the new restriction cannot be allowed.

To illustrate a more complex situation, let's consider a "part file" for an automobile. The structure of file should pictorially resemble the following.




```
PART::= EXTEND ( NULL, PARTNUM, NAME, PRICE )

PART-CONSTRUCTION::= EXTEND ( NULL, PARTREF, FROMPART,
QUANTITY )

PART-CONSTRUCTION::=    RESTRICT    (PART-CONSTRUCTION,
PARTREF = PART, FROMPART = PART )
```

Suppose a door handle is made from two screws and a piece of metal. then:

```
SCREW::= RESTRICT ( PART, PARTNUM = "349", NAME =
"SCREW", PRICE = "0.00" )

METAL::= RESTRICT ( PART, PARTNUM = "201", NAME =
"METAL PIECE", PRICE = "0.01" )

HANDLE::= RESTRICT ( PART, PARTNUM = "750", NAME =
"DOOR HANDLE" )
```

All the individual parts are now defined, and the linking of the parts can be performed.

```
<unassigned>::= RESTRICT ( PART-CONSTRUCTION, PARTREF =
HANDLE, FROMPART = SCREW, QUANTITY = "2" )

<unassigned>::= RESTRICT ( PART-CONSTRUCTION, PARTREF =
HANDLE, FROMPART = METAL, QUANTITY = "1" )
```

Note that individual parts could have specialized fields attached to them. Screws have a length, diameter, and pitch. Metal plates have a set of dimensions describing length width and thickness. Using extend and restrict it is no problem to accommodate these special needs.

Individual linking records do not have a meaningful name. In effect, we would like to throw out the references to these items until it becomes necessary to look at an individual record. I have indicated this in the syntax by placing "<unassigned>" on the left side of the "::=". By using some of the primitives listed below, the constructor can be found and updated.

In defining a network of constructors, one often sequentially applies EXTEND, to create a linking field, and RESTRICT to limit the contents of the linking field. As an alternative to sequential application of EXTEND and RESTRICT, consider GENREC, an operator which combines EXTEND and RESTRICt. As an example:

```
PERSON::= GENREC ( PERSON, DOCTOR-REF=DOCTOR)
```

could be used to allow a doctor to have a list of patients.

PERSON::= GENREC (PERSON, DOCTOR-REF=DOCTOR)

could be used to allow a doctor to have a list of patients. The syntax is the same as for RESTRICT, except that a restriction might not be specified:

PHILADELPHIAN::= GENREC(NULL, NAME, SS, CITY='PHILADELPHIA')

Once the description of the data base has been entered, the natural question is "what is in the data base". I have mentioned 4 types of answers that talk about data in general, but it useful to be able to find a specific piece of data also. Reference names to all pieces of data are not always meaningful, as mentioned above. Sometimes we may be interested in searching for a particular value in a field. A set of primitives for locating records whoses references have been forgotten, or thrown away is needed. The primitives that are needed to find data are described below.

Select record - takes a constructor, type of query (2-4 from above), and a set of pairs of fieldnames and values. A list of record constructors that meets the type requirements, and also contains the specified values in the given fields, is returned.

Fields of - takes a constructor, and returns a list of pairs of fieldnames and values (This is essentially a type 1 inquiry. It is seperated from "select record" because it returns a list of pairs of fields and values, rather than a list of records.)

Literal - takes a value (from the values returned by fields of), and tests to see if it is literal data. If it is literal data, true is returned. Otherwise false is returned.

Constructor - tests whether a value is a record constructor or not, as in literal, and undefined.

The primitives outlined above classify all the important types of retrieval except those which intend to get the "ultimate" data from the database. Since restrict and extend do not provide any indication that a piece of data is complete, simply listing the the type 3 constructors is inadequate to look at the actual parts that go in the part file, for example. Looking at all the type 3 constructors would also reveal a great deal of information about the structure of the data base, as well as the constructors representing the actual parts. The constructors which do represent the parts will be referred to as "ultimate data". Notice that "ultimate data" can still be extended, restricted, and copied. For instance, one might want to use "JOE-STUDENT", who actually exists, as a model for new students that enroll in a university. For the purpose of differentiating between constructors that describe the data base, and constructors that contain ultimate data, the idea of a "candidate key" is borrowed from the relational model. In the relational model, a set of attributes of a relation is a candidate key if all the the attributes of the candidate key uniquely determine an instance of the relation. No functional dependencies can exist between the elements of the candidate key. The values

assigned to a candidate key uniquely determines an instance of data in the database. A relation may have more than one candidate key. I will refer to an arbitrarily chosen candidate key. as the "key" of a constructor

The concept of a genotypical constructor is very close to that of a key. In order for the elements of data stored in the database to be unique, the restrictions on the genotypical constructor must be unique. Unfortunately, the requirements of the genotypical constructor is unique may be too strong, "NAME" will probably appear as a field in many constructors, and would consequently be a genotypical constructor. But, "NAME" should not necessarily be unique.

To specify a less rigorous constraint, I would like to introduce the "GENKEY" operator. The GENKEY operator specifies the constraints on a set of constructors that constitute a similar record class. Using GENKEY it is possible to segment the database into a series of related constructors.

The GENKEY operator is similar to GENREC, with the exception that the restriction cannot be literal: the restriction can only be undefined or be another constructor. Any type 3 constructor is consequently a member of the similar record class which is defined by GENKEY.

Given a set of similar record classes and key fields, the database begins to take on a less arbitrary nature:

1 - If a constructor is ultimate data, then the key fields from the similar record class must be completely restricted. If key a field contains a reference to another constructor, the referenced constructor's key fields must also be completely restricted.

2 - The data base is segmented into similar record classes, rather than being an arbitrary set of definitions.

As an example, to set up a data base of courses, students, professors, and the courses they take and teach, the following instructions could be used:

```
PERSON::= GENKEY ( NULL, SS  )
PERSON::= EXTEND ( PERSON, NAME )
DEPARTMENT::= GENKEY ( NULL, DEPT, DEPTNAME )
STUDENT::= GENREC ( PERSON, MAJOR=DEPARTMENT )
PROFESSOR::= GENREC ( PERSON, INDEPT=DEPARTMENT)
COURSE::= GENKEY ( NULL, COURSE# )
COURSE::= GENREC ( COURSE, COURSENAME, PROF=PROFESSOR )
ENROLLMENT::=   GENKEY   (   NULL,    INCOURSE=STUDENT,
TAKESCOURSE=COURSE )
ENROLLMENT::= EXTEND ( ENROLLMENT, GRADE )
```

Grade is not a part of the key to enrollment, only one grade can be given for a particular enrollment.

In addition, a fifth type of constructor must be added to the 4 types outlined above:

Type 5 - gives those constructor's which contain the fields in question, and are "GENKEY"'s

The arguments of SELECT RECORD are no longer adequate since

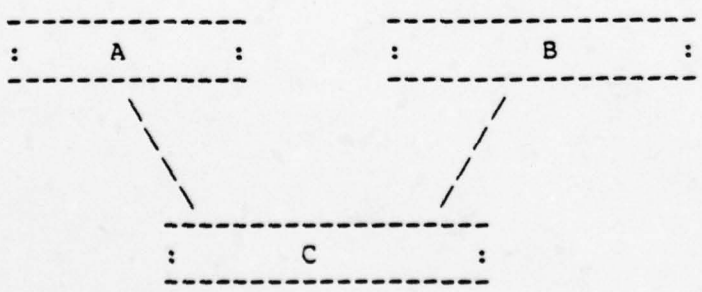
there is no manner of requesting only that ultimate data, or "non-ultimate" data be returned. A third argument specifying ultimate, non-ultimate, or all records must be be passed to SELECT RECORD.

4.0 QUERY LANGUAGES

In chapter 3, I discussed the 4 types of responses that could be made to the question "what is a <something>?". "what is a <something>" is the only question that can be asked in a "flat file" data base (with no links between record classes implicit or explicit) because only one record class is involved in a query. In network data bases more than one record class is typically involved in a query. For instance, to find out how many screws are in a door handle (from the example in chapter 3), both part 349 and 201 must be specified to get the constructor which says that 2 screws are required.

Another type of question might be "what parts is a screw used in?" or "what parts are used in a door handle?". in this case, we are asking to travel down and then back up a confluency. a third type of common question might be "who are Jim's patients": a straight forward case of set owner ship.

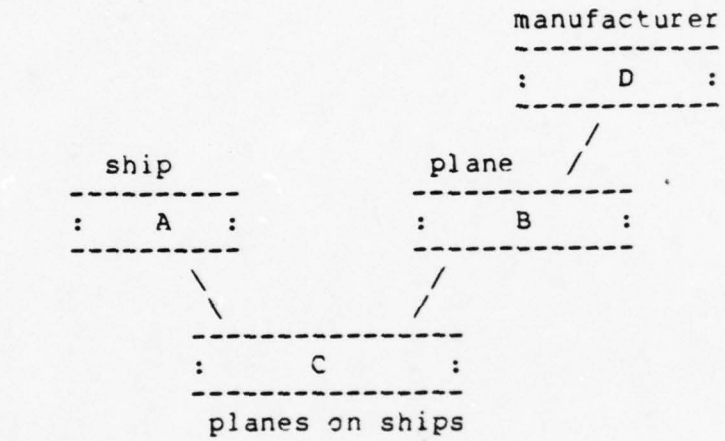
Given the following structure:



where an A and B determine one or more C's, the 3 types of queries discussed above are:

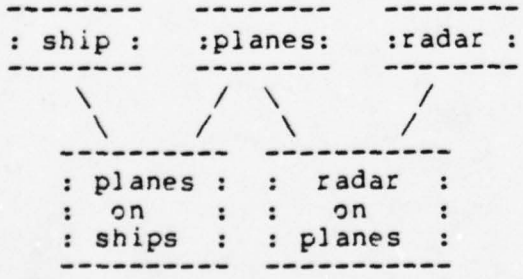
1. given A and B, what is C.
2. given A, what is B, or given B, what is A.
3. given c, what is A, or given C, what is B.

Most queries made to network data bases is one of the three above, or:



given A, what is D
or
what are the manufacturer (D) of the planes on ship "A"?

Another possible pattern might be:

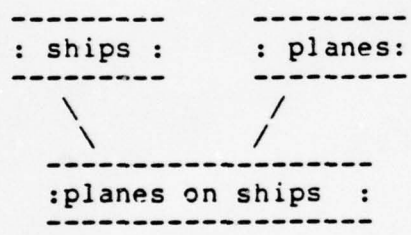


what are the "radar" on "planes" on "ships"?

4.1 The Data Dictionary

The data dictionary, developed at the University of Pennsylvania to interface the SEED Codasyl network database system with interactive users through a simple query system, is an example of how the types of queries shown above can be handelled. The data dictionary is simply a dictionary of common names for records, fields, and the links between records in a more complicated data base. An example of the record names in the test data base at The university of pennsylvania was I18NAME08 which stand for Shipname. The names have no common sense meaning. The first task of the data dictionary is simply to translate the common sense names into the corresponding names in the data base. The other purpose of the data dictionary was to guide inquiries, by having comments (or help) stored with every name. The structure of the data base is also stored in the data dictionary. Every entry has a "common name", "type", "comments", "system name", and "database". An entry is of type "record", "set", "field", or "database". The different types of entries had different information associated with them. A data base entry have a list of record classes. A record entry has a list of fields, a list of owned sets, and a list of owning sets. Sets have an owner record and an owned record class. It is possible to make queries about the structure of the data base and about the data in the data base with the data dictionary. Two words, SHOW and

DESCRIBE, are used to make the distinction between a query about the structure and about the data. The command DESCRIBE SHIP would tell about the structure of the record class ship in the data base. SHOW SHIP would list all of the records stored in the ship record class. SHOW JFK would show only the information in the record about the ship record with key JFK. When a command is given such as DESCRIBE SHIP, a context is which indicates that you are interested in ships. If you subsequently say SHOW JFK, it is assumed that JFK is a key to the ship record class, because JFK is not in the data dictionary as a common name. Suppose that the data base has the following structure:



If SHOW PLANES is given as a command as the command following SHOW JFK, (ie a command which has established the context of a particular ship), then the planes on the ship JFK are displayed. This is done by finding the shortest path to planes from ship, under the constraints that:

1. Only one confluency (immediately descendant to the record of current context) is allowed in any path.

This is a simplistic assumption, but it does allow many of the queries made to a network data base to be answered. Unfortunately, there is no method of indicating that more complex types of paths are to be followed.

4.2 A Data Dictionary For Restrict And Extend

The "two word" query system of the data dictionary does perform well enough to consider how it could be extended to the concept of restrict and extend. When dealing with a CODASYL network SHOW and DESCRIBE are adequate since there is only one level of description of data in the data dictionary. In a database created by EXTEND and RESTRICT, the ultimate data is differentiated from data describing the structure of the database, but there can be many levels of data definitions between the null record and the ultimate data that are not always of interest to a person exploring the database. "SHOW" can be used to request a display of ultimate data, but "DESCRIBE" is too vague. First, a method of differentiating the different levels of development in the definition of the database must be determined. Then a method for indicating the desired level of development to the data dictionary must be developed.

One method for determining the relative importance of certain record constructors in the database is to count the number of type 3 constructors that have been generated from

it. An ordering of constructors by this criterion would probably prove useful, especially once one has determined that a particular similar class of records is of interest. Secondly, the definition of the GENKEY's provide a starting point for the description of the classes of objects stored in the database. They can also be ordered by the number of type 3 constructors that have been generated from them.

We can divide the types of requests into 3 classes:

1 - Those requesting ultimate data
2 - Those requesting the constructors related to a GENKEY constructor.
3 - Those requesting a list of GENKEY constructors.

The two extremes of the above 3 requests (1 and 3) are "SHOW" and "DESCRIBE". The middle ground is still vague, because it is not clear whether we are interested in type 2 or type 3 constructors. For that purpose, I propose that two additional words be introduced which distinguish those requests.:

KINDS OF <x> to list all the type 3 constructors of <x>
WHAT IS <x> to list all the type 2 constructors of <x>

In addition, one might be interested in any of the similar classes to a particular record constructor (the type 5 constructors of a particular constructor):

SIMILAR <x> to list all the record classes which are similar to <x>

In order to provide the ability to traverse the sets that are defined in the database, I suggest another word be added:

LIST <x> to, after a SHOW <y>, list the ultimate <x>'s related to <y>

Suppose we have a database constructed from the student, enrollment, courses, departments and professors in Chapter 3, A set of queries and responses might look like the following:

```
DESCRIBE
The database contains information on:
   ENROLLMENT, PERSON, DEPARTMENT, COURSE
DESCRIBE PERSON
A "PERSON" has:
   NAME, SS (key)
KINDS OF PERSON
The kinds of "PERSON" are:
   STUDENT, PROFESSOR
SIMILAR STUDENT
"STUDENT" is similar to:
   PERSON, PROFESSOR
WHAT IS PROFESSOR
A "PROFESSIR" is:
   PERSON
DESCRIBE ENROLLMENT
An "ENROLLMENT" has:
   GRADE, INCOURSE (key), TAKESCOURSE (key)
SHOW ENROLLMENT
"ENROLLMENT":   GRADE       INCOURSE        TAKESCOURSE
                  A         Ø19-39-8389     BA8ØØ
                  B         Ø21-49-6562     BA8Ø1
                  A-        Ø35-61-7849     DS2ØØ
                  .......
                  ..
SHOW STUDENT Ø35-61-7849
NAME=Josepn Wharton, SS=Ø35-61-7849, MAJOR=DECSCI
LIST COURSE
"COURSE":     COURSE          COURSENAME
              BA8Ø1           Administration
              DS2ØØ           Intro to Decision Sciences
              .........
```

```
SHOW COURSE DS200
COURSE =DS200, COURSENAME=Intro to Decision Sciences,
   PROFESSOR=096-34-6892
SHOW PERSON 096-34-6892
NAME=William Duffy, SS=096-34-6892, DEPT=DECSCI
```

The ability to invert on any attribute should exist. For example, "SHOW JOSEPH WHARTON" should yield the same result as "SHOW PERSON 035-61-7849".

Such a simple language is adequate for many situations, particularly when a new user is shown a database that he is unfamiliar which. Note that the language outlined above for EXTEND and RESTRICT lacks the "helpful" information that is available with the data dictionary for SEED.

5.0 DECISION STRUCTURES

Certain decision sequences can be described easily as a series of questions to be asked. In planning a trip to another city, the basic information needed would be the routes that would be followed. Further information might also be desired concerning fuel consumption cost of transportation, location of overnight stays, and any other somewhat arbitrary pieces of information.

Many of the ideas expressed about restrict and extend were generated from thoughts about decision structures. For this reason, I would like to discuss some of the types of problems that are raised in attempting to describe a decision process, and note how closely structure created by extend and restrict come to implementing decision structure.

A very simple structure is proposed from which we can build a model to experiment with. We think of all problems having some overall knowledge attached to them. This knowledge is placed in a "PR-Block". A PR-Block is somewhat similar to a "frame". The major distinction is that PR-Blocks are meant to be defined by anyone. Only the designer of an AI program can generally design a frame that will fit into the program. The PR-Block is given a name. It could contain the following type of information:

1. Name

2. Help - text describing the use of this PR-Block
3. Entry Conditions - a program to be run when the PR-Block is entered
4. Data
5. Exit Conditions - a program to be run when the PR-Block is exited

The data may be a stored constant. For instance, in planning a meal, it might contain the name of the wine that was chosen. In a PR-Block for wine selection, the text would state that the data was a wine. The entry condition would ask for the name of the wine. The exit condition might check that the wine that was entered is in the wine cellar, and that is does not clash with the meal (i.e. red wine with fish). After the wine had been filled in, the PR-Block would have different information, and a somewhat different structure since it is now a completed decision, rather than a plan for a decision. A unique name for the decision would be stored in the PR-Block. The chosen wine would be stored. The name of the PR-Block from which this block was created would be stored, and new help text would be stored.

We can think of the PR-Blocks collectively as a dictionary of plans for different items. Each PR-Block then asks a specific question, or poses other plans which could be invoked. Plans are arranged in a hierarchy by this convention. Within the hierarchy, on each level, there are a number of decisions to be made in an arbitrary order. The

list at each level should not limit us to only those plans given, nor should it require us to enter them all. The exit conditions of the level that contains this list should check that either enough, or not too many subsidiary plans have been invoked on this level.

As an example, imagine a meal plan where an arbitrary number of wines is required, and one main dish is required. Other courses are optional. It might be represented as:

Plan name:	MEAL
Help:	DECISION STRUCTURE TO PLAN A MEAL
Entry condition:	none
Data:	WINE PLAN, MAINDISH PLAN, DESERT PLAN
Exit condition:	AT LEAST ONE WINE AND EXACTLY ONE MAIN DISH MUST BE CHOSEN

Plan name:	WINE
Help:	DECISION STRUCTURE TO PICK A WINE
Entry condition:	*ENTER A WINE NAME*
Data:	WINE NAME
Exit condition:	IF MAIN DISH IS FISH, WINE MUST BE WHITE; IF MAIN DISH IS MEAT, WINE MUST BE RED; WINE MUST BE IN THE WINE CELLAR;

Plan name:	MAIN DISH
Help:	DECISION STRUCTURE TO PICK A MAIN DISH
Entry condition:	ENTER A MAIN DISH
Data:	MAIN DISH NAME
Exit condition:	none

Plan name:	DESERT
Help:	DECISION STRUCTURE TO PICK A DESERT
Entry condition:	ENTER DESERT NAME
Data:	DESERT NAME
Exit condition:	none

Notice that in order to leave the wine plan, a decision must be made about the main dish. Since the order of making decisions is arbitrary, It will be important to be able to arbitrarily pick the next decision to be made, even in the middle of making any other decision.

In the main dish plan above, we might want to put an exit condition specifying the converse relations for the type of wine and meal that are already in the wine plan. A "lock out" condition could then occur. In actuality, we can solve the deadlock by allowing the exit conditions to act on data in the incompleted decisions. If we first decided on a wine, before leaving that decision, we would be forced to decide on a main dish. Upon leaving the main dish plan, the exit condition would check what wine was chosen and use the data in the partially completed wine plan. Finally, we could leave the wine plan.

"Ultimate decisions" are not the only decisions that are made. One can make decisions about plans as well. It is perfectly desirable to be able to have a set of plans that help to construct plans for different scenarios. For instance, one might use the "wine" plane to make up a series of more specialized plans for different types of wine.

Restrict and extend come very close to providing the a system for decision structures. The phrase "ultimate decisions" immediately reminds one of ultimate data.

Decision structures and extend and restrict share a large core of common concepts. In fact, some of the features of decision structures should probably be added to the structures created by restrict and extend. For instance, it should be possible to add help, or some kind of comments to a constructor, and to the fields and restrictions of a constructor. If help was available, the drawbacks of the data dictionary that is described for restrict and extend would be just about eliminated.

6.0 AN ALTERNATE DATA MANIPULATION LANGUAGE FOR SEED

Although network databases generally have fixed definitions of their structure, they do provide important features which relate to the efficiency for physical storage. Faced with the reality of such constraints, we have tried at the University of Pennsylvania to overcome the outward problems of Codasyl network database's by attaching front ends. One of the front ends is the data dictionary discussed in Chapter 4. Another is "Q", a communication query language for SEED. "Q" is not intended to answer the problems I have outlined in Chapter 1, but it does provide an interesting example of how one can change the view of a database system.

6.1 Why Is There A Need Of A "Communication Language"

Generally, computer systems have concentrated on having one or two languages (such as FORTRAN and COBOL) which are standard on on a given system. The standardization has lead to a number of support packages written in FORTRAN or COBOL which can be loaded only with other FORTRAN or COBOL programs. SEED is such a system. As understanding of programming languages has continued, one finds that special purpose languages have been developed that can be used for for production (as BLISS) or research (as POP10). However, support programs written in FORTRAN or COBOL cannot

generally be loaded with languages such as POP10 or LISP.

Development of network communications has worsened the situation. Until network communications became more important, the concept of machine independence was important to allow transfer of programs from one system to another more easily. FORTRAN and COBOL were the standard languages for machine independence. Even after network communications became important, one of the main uses was to transfer programs from one machine to another (FTP on the ARPANET for example) and machine independence was still important. However, now, computer networking is starting to emphasize the segmentation of program systems into various "tools" that are available at the sites on a network. The possibility of using many tools on different hosts means that a program cannot be loaded into one contiguous section of memory. Consequently, the concept of program independence is not as important; linking the independent operation of separate tasks becomes the main goal.

Both of these reasons lead to a concept in programming that is not fully understood: that of breaking apart a large task into smaller asynchronous components which synchronize activity by sending messages between themselves.

We have been faced with several research projects at the University of Pennsylvania which require a database to behave as a separate asynchronous component of a larger

system. DBLOOK of the SEED database system has been used to accomplish asynchronous operation in the past. Several problems become apparent with DBLOOK when it is used as an asynchronous task serving another task. DBLOOK is fairly intelligent, and to a person using DBLOOK, the results are satisfying. DBLOOK carries on an "implied" conversation. It lets the user figure out what it is reporting and requesting. For humans, the brevity of the output and input is an excellent feature, since it cuts out the information the user already understands. Programs which use DBLOOK do not have the same intelligence as humans, and have a much harder time carrying on the conversation. For instance, when DBLOOK displays a record, it is not explicitly clear where all the fields begin and end.

DBLOOK also has some limitations on its capability which make some queries difficult to perform. DBLOOK cannot give back values which are the result of computations on fields in the database. In addition, the CODASYL DML functions are not very appropriate once one has decided to access a database through a separate task. The DML definition was based on the ability to access a global area containing all the records easily (the UWA).

It is the intention of this project to try to overcome some of these problems by:

1 - designing a query language which is concise, allows

complicated queries to be processed simply.

2 - designing a control structure for executing the query which allows simple synchronization between the communicating tasks.

3 - reporting output in formats which contain all the information for a program to easily ascertain what the output means.

Of course all of these criteria are quite vague. Number 1 is especially vague, since that is the object of any query language. In considering what other criteria we might apply, we decided to adopt the following:

4 - the language should allow any query to be processed which does not require storage that increases more than linearly with the size of the query.

5 - the language should not allow any explicit control structures, such as do loops or conditional branching, yet should be able to selectively process portions of the database.

Statement 4 rules out any processing which would require sorting or merging.

Statement 5 eliminates the need for any functions such as "get first" or "get next". In the limited scope of a query language, it would be burdensome to require an explicit "get" for every record, since a "get" is generally necessary. In addition, we arbitrarily decided to limit

ourselves to exploring the database by defined set relationships.

We look at the database as a hierarchy by starting at one particular point in the database. Then, the particular fields that one wishes to access can be specified. Items called computed fields can be defined that are computed on the basis of other fields. Computed fields can be given a name for later reference, or used to restrict further processing. The functions that are allowed in computed fields are PLUS, MINUS, MULTIPLY, DIVIDE, EQUAL, GT, LT, GE, LE, AND, OR, NOT, and INT. Two more functions are provided which "reduce" portions of the tree. They are SUM and COUNT.

The control structure for executing queries is very straightforward. One has the option of entering a query, opening the database, processing a query, and aborting execution. If an error should occur, the system waits for a specific response to resynchronize itself with the controlling task. The system also informs the controlling task when the processing of a request has started and stopped.

The responses given back are straightforward. They fall into 4 categories:

1 - Errors

2 - Data

3 - Synchronization

4 - Resynchronization request

6.2 System Operation

When the "Q" system is ready, it will type "READY". At this point, commands can be entered. Legal commands are DBOPEN, PROGRA, RUN, VERIFY, DBCLOS, and EXIT.

DBOPEN will open a database. The name of the database must follow the DBOPEN command as the 7th through 12th characters on the line. The privacy key must start in the 14th character position.

PROGRA will allow the lines following it to be entered as a query. Syntax is checked as the query is enterred. To end the entry of the query, an "#" is typed as the first character on a line. To abort, an "@" is typed as the first character on a line. If a filename is specified after the PROGRA, then the file is used as the source of input for the query.

RUN will process the query. First the query is checked against the schema to make sure that all the items and sets are correctly defined. Then the database is accessed to process the query.

VERIFY will check a query against the schema.

DBCLOS closes the database.

EXIT stops the execution of the system.

The system responds to commands with the following keywords in the first 6 character positions of a line: START, DATA, DONE, SCHERR, RUNERR, SYSERR, CMDERR, CLRACK, ABOK, ENTER, SYNERR, FILE.

START indicates that the processing of the request has started.

DATA indicates that the rest of the line contains output from a print request in the query

ENTER indicates that the query system is waiting for a line of the query to be typed. ENTER will appear if the query is not input from a file.

FILE will appear if a query is input from a file. The remainder of the line contains a line of data as read from the file.

SYNERR indicates that an error in the syntax of the query exists.

DONE indicates that the processing of the query is complete.

SCHERR indicates that the query is in conflict with the schema.

RUNERR indicates that a run error has occurred.

SYSERR indicates that an error has been detected in the system's operation

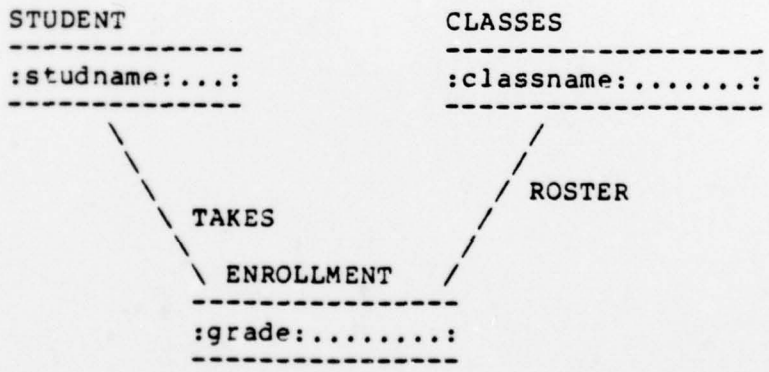
CMDERR indicates that the processing of a command is incomplete because of an error.

Any SCHERR will automatically cause a CMDERR at the end of the VERIFY process. After a CMDERR, SYSERR, RUNERR, or SYNERR the word "CLEAR" must sent back to the query system to indicate acknowledgement of the error. The query system acknowledges with "CLRACK".

To abort processing of a "RUN", or "PROGRA", an "@" can be typed. The system will respond with "ABOK" when it recognizes the abort request. (The system only checks for abort before printing a DATA statement).

6.3 Query Language

The query language will be explained in reference to the following database structure:



The language is designed around the concept of streams. One creates a stream by openning a set of parentheses preceded by a set or record name. For instance:

-STUDENT(...)

creates a stream of students. Operators can be applied to a stream to define elements of the stream:

-STUDENT(NAME:STUDNAME)

or to create a stream of streams:

-STUDENT(!TAKES(...))

In the former case, a stream of student names is created. In the latter case, a stream of enrollments for student is created. When defining a stream of streams, a "!" is used to indicate that the stream owns a set of items represented by the inner stream. A "^" indicates that the outer stream is owned by one item in the inner stream. "!" and "^" allow traversal of the Bachman diagram representing the schema. "-" is used to indicate that the stream that is being defined is simply a set of records, and is not related to any other streams. ("-" can only appear at the outside of an expression).

Items in the schema are referenced by placing the item name in the parentheses. If a name followed by a ":" precedes the item then the name is a user defined name for the item. In the example above, "NAME" is the user defined name for the item "STUDNAME" in the schema.

Once some items have been defined, they can be printed with a "$P". For example:

```
-STUDENT(NAME:STUDNAME, !TAKES(GRADE, ^ROSTER(CLASSNAME,
        $P NAME, $P CLASSNAME, $P GRADE)))
```

will print out the names of students, and the classes they are taking, and the grades they have in the classes.

Suppose we would like to know how many classes the students are taking. Then we could say:

```
-STUDENT(NAME:STUDNAME, !TAKES(GRADE) NUM:COUNT GRADE,
        $P NAME, $P GRADE)
```

The function count produce a count of the number of items in the stream given as an argument. Suppose we wish to get the average grade of all the students:

```
-STUDENT(NAME:STUDNAME, !TAKES(GRADE) S1:SUM GRADE,
  N1:COUNT GRADE, AVE:DIV S1 N1, $P NAME, $P AVE)
```

The function divide will divide S1 by N1 to produce the average grade. But, a problem could occur with the query above if a student is not taking any courses. A division by zero would occur. To eliminate certain portions of a stream, a restriction can be introduced:

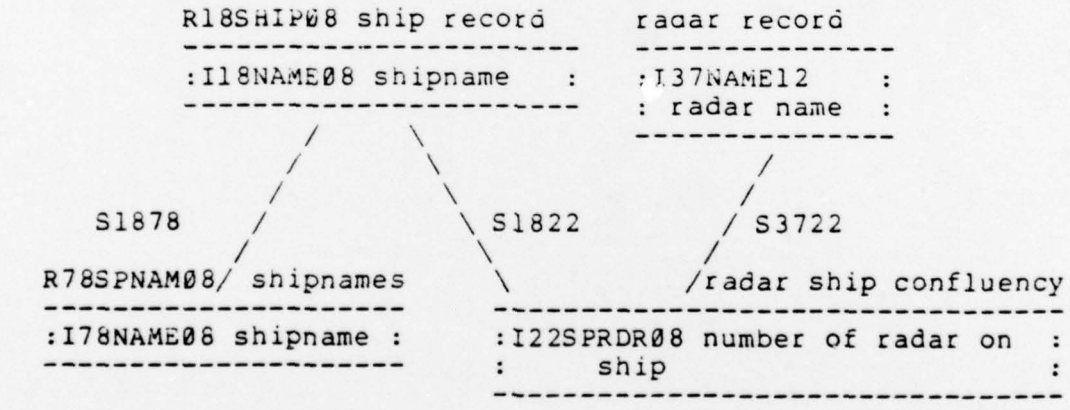
```
-STUDENT(NAME:STUDNAME, !TAKES(GRADE) N1:COUNT GRADE)
  $R GT N1 0 (S1:COUNT GRADE, $P NAME, AVE:DIV S1 N1,
        $P AVE)
```

The function after the "$R" is used to restrict any further processing of streams that contain no grade records. "$R" might also be used to look at the record of a particular

```
student:

   -STUDENT(NAME:STUDNAME)$R EQUAL NAME 'MARTIN MEYERSON'

          (!TAKES(GRADE, ^ROSTER(CLASSNAME),

               $P CLASSNAME, $P GRADE))
```

6.4 A Sample Execution

The following example is taken from the CTEC data base describing a naval scenario. The portion of the database which we are exploring involves ships and radar, and the confluency between them.




```
.RU Q

READY
DBOPEN ONRSUB CTEC
START  OF PROCESSING
DONE   QUERY RUNTIME:  0.013 SEED RUNTIME:  1.624
PROGRA Q0.DAT
START  OF PROCESSING
FILE   -R78SPNAM08(NAME:I78NAME08)$R EQUAL NAME 'CHICAGO'
FILE   (^S1878(SHIPNAME:I18NAME08,!S1822(NUMBER:I22SPRDR08,
FILE   ^S3722(RADARNAME:I37NAME12,$P SHIPNAME,$P RADARNAME,
FILE   $P NUMBER))))
DONE   QUERY RUNTIME:  0.875 SEED RUNTIME:  0.000
RUN
```

```
RUN
START  OF PROCESSING
DATA   SHIPNAME                          =CHICAGO
DATA   RADARNAME                         =SPS-10
DATA   NUMBER                            =                1
DATA   SHIPNAME                          =CHICAGO
DATA   RADARNAME                         =SPS-30
DATA   NUMBER                            =                1
DATA   SHIPNAME                          =CHICAGO
DATA   RADARNAME                         =SPS-43
DATA   NUMBER                            =                1
DATA   SHIPNAME                          =CHICAGO
DATA   RADARNAME                         =SPS-48
DATA   NUMBER                            =                1
DATA   SHIPNAME                          =CHICAGO
DATA   RADARNAME                         =SPS-52
DATA   NUMBER                            =                1
DONE   QUERY RUNTIME:  1.878 SEED RUNTIME:  0.998
PROGRA Q4.DAT
START  OF PROCESSING
FILE   -R18SHIP08(SHIPNAME:I18NAME08,!S1822(N:I22SPRDR08),
FILE   TOTALRAD:SUM N,NTYPE:COUNT N)$R GE NTYPE 2
FILE   ($P SHIPNAME, $P NTYPE, $P TOTALRAD,
FILE   !S1822(NUMBER:I22SPRDR08,^S3722
FILE   (RADARNAME:I37NAME12,$P RADARNAME,
FILE   $P NUMBER)))
DONE   QUERY RUNTIME:  0.936 SEED RUNTIME:  0.000
RUN
START  OF PROCESSING
DATA   SHIPNAME                          =DOWNES
DATA   NTYPE                             =                2
DATA   TOTALRAD                          =                2
DATA   RADARNAME                         =SPS-40
DATA   NUMBER                            =                1
DATA   RADARNAME                         =SPS-10
DATA   NUMBER                            =                1
DATA   SHIPNAME                          =TRUETT
DATA   NTYPE                             =                2
DATA   TOTALRAD                          =                2
DATA   RADARNAME                         =SPS-10
DATA   NUMBER                            =                1
DATA   RADARNAME                         =SPS-40
DATA   NUMBER                            =                1
DATA   SHIPNAME                          =BOWEN
DATA   NTYPE                             =                2
DATA   TOTALRAD                          =                2
DATA   RADARNAME                         =SPS-10
DATA   NUMBER                            =                1
DATA   RADARNAME                         =SPS-40
DATA   NUMBER                            =                1
@
ABOK   ABORT RECOGNIZED
EXIT
```

```
END OF EXECUTION
CPU TIME: 11.24     ELAPSED TIME: 4:25.83
EXIT
```

7.0 CONCLUSION

I have attempted to demonstrate the drawbacks to the traditional thoughts on database systems, and how they might be alleviated by allowing dynamic creation of structures describing data, as well as real world entities. Although restrict and extend are exciting concepts within their own right, I find convincing database users that they should allow flexibility in their design more important than trying to force restrict for and extend to be used an application.

In talking with several users of large database systems across the country, (AMOCO, EXXON), I find that there is an annoying tendency to believe that the structure of the real world can be captured in a fixed description of a database. Until database designers discover that there is no truly correct design for a database, I believe that the concept of a database will be useless. The overhead of setting up the data, and the programs to use the data will far outweigh the supposed benefits changing to an implementation that uses a database management system. The real work to be done in database research should be aimed at trying to homogonize the view that anyone person sees over a set of different databases that describe similar information.

Bibliography

1 - Chen, Peter Pin-Shan, "The Entity-Relationship Model -- Towards a Unified View of Data", ACM Transactions on Database Systems, (March 1976), pp 9-36.

2 - Codd, E. F., "A Relational Model of Data for Large Shared Data Banks", Communications of the ACM, 13, June 1970, pp 377-387.

3 - Codd, E. F., "Further Normalization of the Database Relational Model", In Data Base Systems, pp 33-64, Edited by Randall Rustin, Englewood Cliffs, NJ, Prentice Hall, 1972.

4 - Date, C. J., An Introduction to Database Systems, Reading, Massachusetts, Addison Wesley Publishing Co., 1977.

5 - Hayward, J., Sangal, R., and Buneman, P., "Q - A Communications Query Language for SEED", Working Paper 78-05-02, The Department of Decision Sciences, The Wharton School, University of Pennsylvania, May 1978.

6 - Minsky, M., "A Framework for Representing Knowledge", In The Psychology of Computer Vision, edited by P. H. Winston, NY, Mcgraw-Hill, 1975.

7 - Smith, J. M., and Smith, D., "Database Abstractions: Aggregation", Communications of the ACM, 20 (June 1977), pp 405-413

8 - Smith, J. M., and Smith, D., "Database Abstractions: Aggregation and Generalization", ACM Transactions on Database Systems, 2, (June 1977), pp 105-133.

DISTRIBUTION LIST

Department of the Navy - Office of Naval Research

Data Base Management Systems Project

Defense Documentation
Cameron Station
Alexandria, VA 22314
12 copies

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, IL 60605

Office of Naval Research
New York Area Office
715 Broadway - 5th Floor
New York, N.Y. 10003

Dr. A.L. Slafkosky
Scientific Advisor (RD-1)
Commandant of the Marine Corps
Washington D.C. 20380

Office of Naval Research
Code 458
Arlington, VA 22217

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217
2 copies

Office of Naval Research
Branch Office, Boston
495 Summer Street
Boston, MA 02210

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA 91106

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D.C. 20375
6 copies

Office of Naval Research
Code 455
Arlington, VA 2217

Naval Electronics Lab. Center
Advanced Software Technology Div.
Code 5200
San Diego, CA 92152

Mr. E. H. Gleissner
Naval Ship Research and
Development Center
Computation and Mathematic Dept.
Bethesda, MD 20084

Mr. Kim Thompson
Technical Director
Information Systems Division
OP-911G
Office of Chief Naval Operations
Washington, D.C. 20350

Prof. Omar Wing
Columbia University
in the City of New York
Dept. of Electrical Engineering
and Computer Science
New York, N.Y. 10027

Commander, Naval Sea Systems Command
Department of the Navy
Washington, D.C. 20362
ATTENTION: PMS30611

Captain Richard Martin, USN
Commanding Officer
USS Francis Marion (LPA-249)
FPO New York 09501

Captain Grace M. Hopper
NAICOM/MIS Planning Branch
OP-916D
Office of Chief of Naval Research
Washington, D.C. 20350

Bureau of Library and
Information Science Research
Rutgers - The State University
189 College Avenue
New Brunswick, N.J. 08903
ATTENTION: Dr. Henry Voos

Defense Mapping Agency
Topographic Center
ATTN: Advanced Technology Div.
Code 41300
6500 Brookes Lane
Washington, D.C. 20315

DISTRIBUTION LIST

Department of the Navy - Office of Naval Research

Data Base Management Systems Project

Defense Documentation
Cameron Station
Alexandria, VA 22314
12 copies

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, IL 60605

Office of Naval Research
New York Area Office
715 Broadway - 5th Floor
New York, N.Y. 10003

Dr. A.L. Slafkosky
Scientific Advisor (RD-1)
Commandant of the Marine Corps
Washington D.C. 20380

Office of Naval Research
Code 458
Arlington, VA 22217

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217
2 copies

Office of Naval Research
Branch Office, Boston
495 Summer Street
Boston, MA 02210

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA 91106

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D.C. 20375
6 copies

Office of Naval Research
Code 455
Arlington, VA 2217

Naval Electronics Lab. Cener
Advanced Software Technology Div.
Code 5200
San Diego, CA 92152

Mr. E. H. Gleissner
Naval Ship Research and
Development Center
Computation and Mathematic Dept.
Bethesda, MD 20084

Mr. Kim Thompson
Technical director
Information Systems Division
OP-911G
Office of Chief Naval Operations
Washington, D.C. 20350

Prof. Omar Wing
Columbia University
in the City of New York
Dept. of Electrical Engineering
and Computer Science
New York, N.Y. 10027

Commander, Naval Sea Systems Command
Department of the Navy
Washington, D.C. 20362
ATTENTION: PMS30611

Captain Richard Martin, USN
Commanding Officer
USS Francis Marion (LPA-249)
FPO New York 09501

Captain Grace M. Hopper
NAICOM/MIS Planning Branch
OP-916D
Office of Chief of Naval Research
Washington, D.C. 20350

Bureau of Library and
Information Science Research
Rutgers - The State University
189 College Avenue
New Brunswick, N.J. 08903
ATTENTION: Dr. Henry Voos

Defense Mapping Agency
Topographic Center
ATTN: Advanced Technology Div.
Code 41300
6500 Brookes Lane
Washington, D.C. 20315