EXAMINATION AND MODELING OF A PROTOTYPE INFORMATION SYSTEM

by

RICHARD CARL AKEMANN

S.B., Massachusetts Institute Of Technology
(1971)

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF
SCIENCE
at the
MASSACHUSETTS INSTITUTE OF
TECHNOLOGY
June, 1973

Signature of Author ....
Alfred P. Sloan School of Management, Mar. 19, 1973

Certified by ....
Thesis Supervisor

Accepted by ....
Chairman, Departmental Committee of Graduate Students

EXAMINATION AND MODELING OF A PROTOTYPE INFORMATION SYSTEM

by
RICHARD CARL AKEMANN

SUBMITTED TO THE ALFRED P. SLOAN
SCHOOL OF MANAGEMENT ON MARCH 19, 1973
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

Management Science continues to be plagued by a lack of generalized models for management information systems. We have models for other computerized functions: compilers, assemblers, and operating systems; but we have no generalized model for information systems.

This paper takes one information system, Janus, developed at the Cambridge Project at MIT, and using it as a prototype, adds certain features to create a more powerful, sophisticated information manager. The paper finishes with a revised model of Janus and extrapolates from this to propose a model for a generalized information system.

The paper concludes acknowledging the trade-offs between speed and efficiency on the one hand and power and flexibility on the other, but offers documentation that the trade-offs can be reasonably successfully managed for the general case.

Thesis Supervisor: Stuart E. Madnick

Title: Associate Professor of Management

Acknowledgements

The author gratefully acknowledges the helpful cooperation of Jeffrey Stamen and the rest of the Janus staff at the Cambridge Project: Pam Hill, Bob Wallace, and Dorothy Shuford, whose cheerful assistance were instrumental in collecting material for this paper. The incisive comments of Professor Stuart Madnick of the Sloan School, and Professor John Donovan and John Melber of the Electrical Engineering Department at MIT were helpful in achieving greater clarity and directness in this paper's presentation.

Finally, and most importantly, the author would like to thank Stuart Madnick whose time, energy, interest, and constant cooperation were inspirational to the author and largely responsible for the steady, progressive completion of this paper.

Table of Contents

Introduction page 8

Chapter One "A Who's Who of Information Systems" page 10

Chapter Two "Just How Janus Got Started" page 19

Chapter Three "A Brief Description of Janus" page 25

Chapter Four "Miscellaneous Modifications to Janus" page 38

Chapter Five "Automatic Relation Defining" page 62

Chapter Six "Network Relation Defining" page 84

Chapter Seven "Sample Session Using Revised Janus" page 102

Chapter Eight "Revised Model for an Information System" page 112

Conclusion page 121

Bibliography page 123

List of Figures

Figure 3.1 "A Janus Database" page 26

Figure 3.1a "A Single Janus Dataset" page 27

Figure 3.2 "Model of Janus" page 33

Figure 4.1 "Sorted Attribute Storage" page 40

Figure 4.2 "Sorted Attribute Storage plus Original Attribute" page 43

Figure 4.3 "Pattern Tree Attributes" page 44

Figure 4.4 "Summary of Storage Alternatives" page 47

Figure 4.5 "Janus Model Modifications for New Storage Strategies" page 49

Figure 4.6 "Alternating Attributes in Storage" page 51

Figure 4.7 "Historian's Record of Storage Retrieval" page 53

Figure 4.8 "Rings" page 58

Figure 4.9 "Protection Scheme" page 59

Figure 4.10 "Entity Level Attribute Protection" page 60

Figure 5.1 "Two Datasets with Var Del Attributes" page 63

Figure 5.2 "Two Datasets with Expanded Var Del Attributes" page 64

Figure 5.3 "Creating a Relation Between Two Expanded Datasets" page 66

Figure 5.4 "Retrieving Drugs to Treat Pneumonia" page 68

Figure 5.5 "Two New Datasets with Var Del Attributes" page 69

Figure 5.6 "Two New Datasets with Expanded Var Del Attributes" page 70

List of Figures Continued

Figure 5.7 "Creating a Relation Between Two New Expanded Datasets" page 72

Figure 5.8a "Finding Drugs to Treat a Cough and Fever" page 73

Figure 5.8b "Finding a Disease Whose Symptoms are a Cough and Fever" page 74

Figure 5.9 "A Synonym Table" page 77

Figure 5.10 "A Codelist Dataset" page 79

Figure 6.1 "Three Interrelated Datasets" page 85

Figure 6.2 "Linking Dataset Three to Dataset One" page 87

Figure 6.2b "Lists Before any Match is Found" page 92

Figure 6.2c "One Match Found for First List" page 93

Figure 6.3 "Four Interrelated Datasets" page 95

Figure 6.4 "Hash Table for Datasets One, Two, Three, & Four" page 96

Figure 6.5 "Linking Dataset One to Dataset Four" page 97

Figure 6.6 "Multiple Analytic List Combinations" page 98

Figure 6.7 "Successful Multi-level Link from One to Four" page 100

Figure 7.1 "A Database Consisting of Four Datasets" page 102

Figure 7.2 "Hash Table for Revised Datasets One, Two, Three, and Four" page 104

Figure 7.3 "Analytic Lists with One Relation Added" page 108

Figure 7.4 "Analytic Lists with One Match" page 109

List of Figures Continued

Figure 7.5 "Analytic Lists with Solution" page 110

Figure 8.1 "Revised Model of Janus" page 113

Figure 8.2 "A Basic Information System" page 116

Figure 8.3 "Primary Model of Janus" page 117

Figure 8.4 "Block Diagram of a Full Generalized Information System" page 118

Introduction

Since the inception of computers as information management instruments, attempts have been made to improve the speed, flexibility, and programmability of information management systems. In spite of these efforts to produce more generalized information management packages, there is still roughly one unique information system for each data management application. This phenomenon is due, partly, to the lack of any generalized model for an information system. In creating any invention, one must have a generalized model which describes possible variations and alternatives which the system may take. A model allows one a framework in which ones particular application may be placed in perspective. To date, we have models for compilers and assemblers. Models for operating systems are in the final stages of development. However, a model for a generalized information storage and retrieval system has not yet been specified from which all systems may be derived.

In an attempt to confront this problem, this paper will present first a brief overview of current information systems noting their basic capabilities and differences. It will then present one prototype information system and a model for it which incorporates many of the more attractive features of present information systems. From there, the paper will offer a sample of capabilities which should be available but are lacking in the prototype system and offer possible implementation schemes. It will finish with a revised model for a generalized information system incorporating the proposed changes to the prototype.

Chapter One

A Who's Who of Information Systems

In light of the wealth of information systems generated to date, the decision as to which systems to examine could be somewhat difficult. However, the work performed by the CODASYL systems committee (3, 4) appears to be the most accessible, comprehensive, and thorough examination of existing information systems currently published. For this reason, this paper will use their work to examine four of the more powerful systems which appear to have features which one would expect in more advanced and sophisticated information systems. First, this paper will present a brief overview of each system and then go into some of the features of the systems which distinguish them from each other and point up desired characteristics of higher level information storage and retrieval systems.

The first system to be discussed is IBM's GIS implemented in 1969 under OS/MFT on the 360/40G with 92K, and under OS/MVT on the 360/50I with 512K using sequential and indexed sequential storage.(3, 4) The system was developed to interrogate and maintain arbitrary user files responding to "unstructured" and unanticipated user requests. The files were defined using a special data description language and retrieval and updates were performed using a "high-level" procedural language (a compiler language) which was designed for use by non-programmers. The system could be operated in batch or interactive modes and the internal files were compatible with standard OS/360 sequential and indexed sequential storage structures permitting regular OS programs to create and access GIS files. Provision was made to invoke user-written assembly language programs explicitly from procedural task specifications or implicitly from input/output validation and conversion processes to perform operations on the data. Tasks were divided into data description tasks which defined the data structure and the input files, and procedural tasks for interrogation, update, and creation. Data definition included defining aspects of storage structure, file and item access locks, input and output data validation and transformation and event recording. A limited redefinition capability was provided to redefine certain fields of data, if unused; interspersed storage could be used to adjust for the change without touching existing data. Procedural tasks included queries in which up to 16 files could be reported and stored in temporary files. An elaborate report generator was probably the most attractive feature of the system.

The second system to be examined is System Development

Corporation's TDMS, implemented under the ADEPT 50 operating system on the 360/50H in 256K also brought up in 1969.(3,4) The file structure was a completely inverted file with cross-indexed tree structures implemented on disk and drum. The language, designed for non-programmers, could handle ad-hoc inquiries with rapid response into a database with hierarchical tree structures. The language was supplied with subsetting, sorting, and merging facilities and a modest report-generating capability. Inputted data could be monitored during input allowing correction of raw values if necessary. Special translation programs existed to cope with data in foreign formats coming from other machines or systems. The system was provided with an on-line "help" facility to assist the user in solving a particular problem. The user had the capability to modify, combine, or rearrange groups of his data.

A third system to be examined is IDS, implemented by Honeywell on H6000 hardware in 512K (IBM bytes) using disk storage.(3,4) Developed in 1963, this system allowed the user to specify a database specifically tailored to the requirements of the given application. The system was heavily dependent on the COBOL compiler being used in the operating system, and the system used the COBOL compiler to perform many of its functions. The IDS functions described storage, retrieval, and update tasks while standard COBOL handled all other data manipulation, validations, and reporting functions. Chain pointers were used to define structural relationships between groups. A basic structural element was a "chain" consisting of a "master" group and any number of "detail" groups. Chains had pointers going both directions in the sequence. The system had three group classes for storage and retrieval: calculated groups accessed on the value of the item within the group, primary groups accessed using user-furnished pointers, and secondary groups accessed through their relationship to a specified master group.

The final system to be examined is IBM's IMS implemented in 1969 under OS/MFT on a 360/50 in 256K and under OS/MVT on a 360/50 in 512K using disk storage.(3) In IMS the user was required to write application programs to access the database and control transmission of messages to and from the terminal. The data definition facility was provided to define the structure and attributes of a file in the user's database with the capability to re-define database files as "logical" files which could be accessed with application programs. Database services included fetching of data groups through specified identifiers, fetching of dependents of previous groups, and replacing, deleting, and storing new groups. The system had a checkpoint-restart facility to provide recovery from host system crashes. The system also provided a facility for message switching and editing.

With the introduction of IDS in 1963, the capability for network-structured files was initiated, dependent, of course , on the capabilities of the host language (COBOL) with which it was implemented. Although IMS provided some network-structuring of relations, this had to be specified before-hand and was not allowed as an after-thought. With TDMS's completely inverted tree-structured file, an attribute could be used to subset the database quickly and easily, or relate two entity groups together (An entity may be considered the specific unit of analysis: questionnaire, personnel record of one person, project record, etc.). However, unlike GIS, IMS, and IDS, the user had little control over how the information was structured internally in TDMS. Therefore, if a request referenced data located in many widely scattered areas of the system, the user had no capability to restructure the data to make the operation less expensive.

GIS appeared to be a large database management system for handling mass quantities of information. Relations had to be programmed into the database managment system through the initial structuring of the information. Otherwise, lengthly, expensive searches might be needed later to establish subsets and relations between parts of the database.

IMS represented a trade-off between the large-scale GIS-type information manager and the highly flexible, but somewhat expensive TDMS system. Relations could be created and information could be stored semi-conveniently if the user knew ahead of time just exactly what types of relations and subsets would be needed. The information could be stored in tree-structured format and relations could be built-in across the structure. Unfortunately, revising the trees to any great extent became a cumbersome process which could result in having to re-create the entire storage structure.

Another characteristic of these systems was the small number of data types they were capable of handling. IMS allowed any data type which the user wished to declare. However, the system never used this information to handle the data since all information was simply byte strings to IMS. One could compute the mean of a number of text strings as easily as computing the mean of some numbers. All of the systems allowed numeric (integer or floating-point data) and text strings consisting of standard alphabetic and special alphanumeric characters. However, beyond these data types, only TDMS provided another item type which it called "date". Dates could be stored in GIS by creating three consecutive integers or a text string to handle the value, but no descriptive information told the system that the value was a date which could be added, subtracted, and manipulated as a single value. TDMS supplied the capability for Nominal data (male/female, Yes/No, etc.), but none of the other systems had this. Other data types not supplied in any of the systems were Nominal Mutiple (languages spoken: French and/or English and/or German etc.), and raw bit strings to handle other data types not anticipated by the system.

Regarding security, GIS had a rather sophisticated system in which each item, or an entire file could have query and update access codes (passwords). Thus, two types of specified read and write access existed at two levels in the file structure hierarchy. Security on IDS was implemented through COBOL at the 01 level definition of a group. The authority "lock", supplied when the group was defined, had to match the "key" supplied by the user's OPEN statement to acquire write access to the data. This did not, however, provide any protection against other users reading the data. TDMS did not supply any security to its own database. However, since multiple users required multiple copies of the same database, security could be provided through the host operating system by giving each user only the information from the master database which he needed to work. Updating the database might become rather problematical in this mode. By using multiple copies of the database, the host system prevented users from reading or modifying each others' databases. IMS implemented security by specifying which programs could read or write a file. The user then had one password which he used to log into the terminal and others which he issued to invoke certain programs which he was authorized to use. Only certain files could be modified or read in specific ways by a given program to which the user had access.

Logging of transactions on the system can be useful both to remember what types of functions are performed most frequently, and to provide some record or history as a form of crash protection. Neither GIS, TDMS, nor IDS provided this capability. IMS, however, logged both incoming and outgoing messages. Query logging could be suppressed but update logging never was. The log could then be used to duplicate input lost during a crash. Performance and activity statistics were, however, much more difficult to derive and analyze from this log.

In the programming systems, IMS and IDS, there was no descriptive database which the programmer could use to determine at run-time the structure of his database. Therefore, each program had to know at compile time what the structure of the database was. If that structure was changed, then all the programs which referenced altered portions of the database had to be recompiled with the new structure.

Once the database was entered into the system, very little could be done to define and create new databases as functions of the old ones with these systems. GIS could provide a modest capability for accomplishing this, but it was far from ideal. None of the other systems appeared to have this capability.

Chapter Two

Just How Janus Got Started

Like many good ideas, Janus began as a result of dissatisfaction with existing information management systems. During 1968-1969, its present supervisor, Jeff Stamen, and Alan Kessler, formerly of the CIS at MIT, were consulting with research projects and supervising students who were manipulating political science data with a system called ADMINS.

ADMINS was designed to be an interactive data manipulation language to aid political scientists with their work in handling broadly varying databases. It provided the capability for watching data during input and flagging down bad data items before they were internally stored in the database. Thus, rather than having to edit and re-input data after computing a statistic having spent two days discovering that the erroneous results were caused by garbage in the data, the data could be caught as it was entered into the system and modified to prevent wildly varying results.

Another powerful feature of ADMINS was its subset analyzer that allowed extensive and efficient subset specifications which became part of the database. Unions, complements, intersections, and special entity numbers could be stored and used to rapidly access subsetted portions of the database.

Many special relations could be defined between datasets varying from one-to-one relations to one-to-many and many-to-one relations between datasets. Thus, a user could link two datasets by finding an attribute which was common to both and then defining a relation between the two datasets linking through identical occurrences of the attribute in each dataset. The relations were, unfortunately, unidirectional, requiring that the inverse relation be defined to go the other way in the relation. Also these relations could not handle more complicated sociometric relations such as a person having many friends who claimed him as one of their friends. The above would result in many friends being related to many other friends or a "many-many" relation. A modest selection of parametric and non-parametric statistics were included which facilitated analyzing the data from a statistical standpoint.

Since Admins was a CTSS subsystem, it incorporated a number of components of the CTSS time-sharing system into its own system. At any point in the execution the user could be receiving error messages from the compiler, the CTSS file structure system, or elsewhere. The entire system demanded certain core loads, limiting the amount of storage that was available for the user's data and analysis work. All-in-all it tended to be a system for the more sophisticated user and the simple user could become lost in the complexity of where the system had left him at any moment.

Another problem and perhaps the most important one in its obsolescence was its dependence on the CTSS time-sharing system at MIT. CTSS was a development time-sharing system, and once the immediate need for time-shared computing power was met by other, more sophisticated systems, CTSS was phased out and users removed from the system. Facing this situation, Mr. Stamen and Alan Kessler began to rough out a specification for a new data management system. The design, called "Penelope", was to be used to construct a computer model for "a theory of human record handling".(7)

Penelope was specified to handle the following functions:

1. Acquisition of certain structures of information under a categorization scheme as computer records of information.

2. Manipulation of the records with logical and mathematical techniques for information handling and for scientific purposes.

3. Generation of new records for use by other information handlers.

The specification went on to state:

> Penelope will manipulate records about items, i.e. people, things or objects. The records must contain categorized information either about items and their characteristics or about items in dyadic relations with other items. The descriptors of this categorized information with respect to its form, content, and procedure will also be managed as categorized information. Thus information used as data for scientific purposes and the descriptions of this data are managed in the same structures with the same processes.(7)

Penelope, which can be thought of as the first draft for Janus, furnished two major new concepts in data manipulation. First, there would be a large body of descriptive data recording the form, content, and definition of data items. Second, this body of descriptive data would be managed by the same processes with which the user's data was managed. Consequently, the user had both the on-line capability for altering and modifying his own data, but the added capability for modifying the structures describing his database. A wide variety of of manipulations of the data could be performed while recording the actual changes in the descriptive data. With the descriptive data, the problem of older program obsolescence with changing data structures was eliminated, because the descriptive structures which a program used to access the data changed with the data modification. Thus, "application programs" became self-modifying to the extent that the descriptive data they used was self-modifying.

While Penelope was being specified at the CIS at MIT, another statistical, computer effort, the Cambridge Project, was also working on the problem of database management. The project was designed to offer social and behavioral scientists a wide variety of statistical and data manipulative capability in one "consistent" environment which could solve the complex probems in analyzing social science data. A summer study during 1970 at the Cambridge Project established that a large part of the Project should become devoted to the subject of data handling. After a careful investigation of what was currently available in the field of data managment, a conclusion was reached that no single existing system could handle the problem or had adequate capabilities to be revised to solve the Cambridge Project's data management problem.

A Data Handling Committee was established to address the problem and devise a specification for one system which would supply all the needed features. The committee worked for the summer with little result since the diversity of opinions and experience combined with the need for modularity seemed to lead to numerous dead-ends. Finally, Jerry Miller of the Stamford Graduate School of Business, who had developed Datanal at MIT, entered the committee. Although Dr. Miller's expertise had previously been more statistically than data handling oriented, Dr. Miller's presence sparked the group into positive steps towards a procedural specification of a data management system. Together with the help of Dr. Miller and Fred Brookstem, who had worked on Datatext, Mr. Stamen was able to insert a chapter into the summer study of the data management problem which outlined a proposal for Janus.

In November of 1970, Mr. Stamen was moved into the Cambridge Project Central Staff and the specification for Janus continued. By April of 1971, clearance was acquired to begin writing a prototype of the Janus system. The prototype was designed to provide a model of some of the features which should be present in a more advanced information handling system. Its presence proved that the concepts could be implemented, and work was to progress for a full Janus system from there.

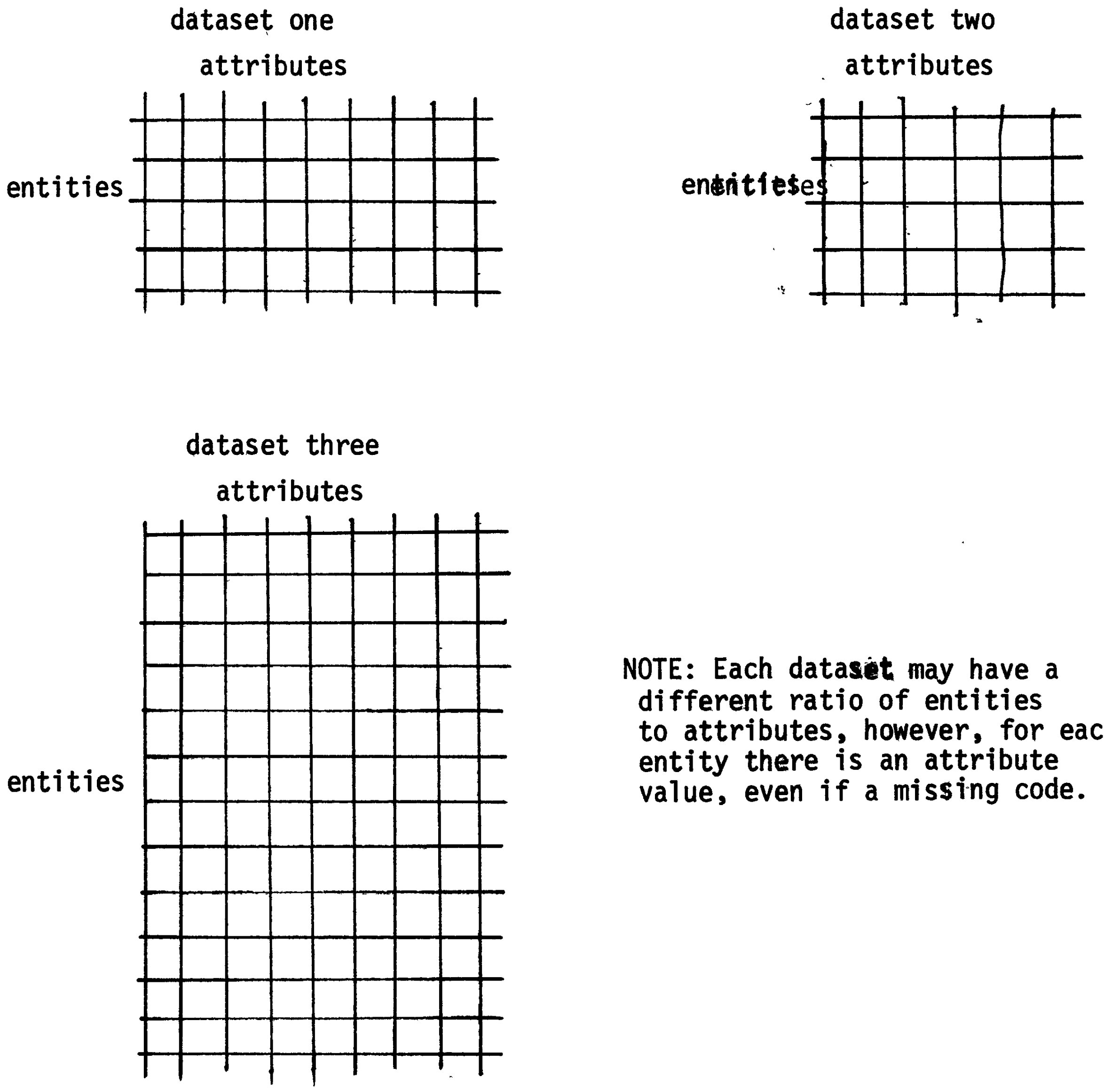
Chapter Three

A Brief Description of Janus

At this point, a description of the Janus system might be helpful in understanding the concepts which will be presented later in this paper. An entity is the specific unit of analysis which may be a questionnaire, personnel record, or some other unit of interest. Attributes represent characteristics of entities such as occupation, salary, due-date, date-ordered, etc.. A Janus database is divided into datasets which have a "population" of entities. For each entity there are attributes common to all entities with missing values indicating that a given entity does not have a value for that attribute. Each database may have a varying number of datasets, and each dataset, its own population of entities and attributes per entity. A schematic of a Janus database appears in figure 3.1, and an example of a single dataset appears in Figure 3.1a.

FIGURE 3.1

A Janus Database



NOTE: Each dataset may have a different ratio of entities to attributes, however, for each entity there is an attribute value, even if a missing code.

Figure 3.1a

A Single Janus Dataset

	Age	School	Grade
person1	21	Harvard	A
person2	19	MIT	A
person3	23	MIT	B
person4	22	B.U.	B
person5	20	N.E.	A

Janus has 12 basic types of internally stored data:

1. nominal
2. nominal multiple
3. floating-point number
4. integer
5. text string
6. date-time
7. bit string
8. attribute definition
9. dataset definition
10. relation definition
11. macro definition
12. attribute

Nominal expresses unique values for non-ordinal categories (Male/Female, Graduate/Undergraduate, etc.). The values indicating the category are stored internally as integers although the system deals with them as nominal, remembering what each integer means.

Integer data is a fixed binary(35) number (as PL/1 is implemented on GE/645 hardware) and floating-point is floating binary(27) (again, as implemented) real floating-point number.

Text is a string of characters of a given (by the user) length which usually represents a name or a description of something. A bit string is used to contain any other type of data which is not specifically implemented on the system.

Nominal multiple may be used to express multiple instances of nominal categories. For instance, languages spoken: French and/or English and/or German and/or Italian etc..

Date-time is used to record a date and the time of day of any given event.

Attribute definitions are stored to remember if a new attribute is created as a function of old attributes. They record what the new attribute means as a function of the old attributes.

Dataset definitions are recorded for much the same reason as attribute definitions only to remember what function has been used to define the new dataset from the old one. Relation definitions specify what two attributes in two given datasets were used to create the relation and whether the relation was many-to-one, one-to-many etc..

Macro definitions record any abbreviations which the user may create to perform a number of Janus commands with only a few characters.

"Attribute" definitions, as distinguished from the "attribute definitions" listed above, are the entire set of form, content, and procedural definitions which are stored with an attribute when it is defined and created.

Finally, bit strings are provided to create any data type which is not already explicitly handled by the system.

Each attribute value for a single entity may have up to three dimensions: the first two are fixed and the third is varying. For instance, the attribute, I.Q. score, may have five rows in one dimension for the number of years over which it was taken, and ten columns in another dimension for the number of times each year the score was taken. I.Q. would then become a two dimensional (5,10) array of 50 scores.

By using the "varying del string" or "var del" attribute, a third dimension could be added. I.Q. could become a var del attribute by adding a dimension representing a varying number of times per week that the I.Q. test was taken if students wanted their scores to be an average of scores taken throughout the week to guard against the "one bad day" effect. Since the number of scores varies by entity and by attribute(i,j), each weekly score consists of a unique number of daily scores.

Using Janus, there is the basic capability to define a raw dataset and define and create attributes. New attributes can be defined as functions or multiple arithmetic expressions of old ones. This capability to define new attributes in terms of old ones appeared to be lacking in the previously examined systems. For instance, from the raw attributes, age, education, and programming experience; a new attribute, promotability, can be defined as 1 for age <= 25 and education <= 3 and experience < 2; 2 for age > 25 and age < 30 and education > 3 & < 5 and programming experience > 2, etc.. A new attribute can be defined as an arithmetic function: promotability = education X experience / age, and so on.

Internal datasets can be defined from raw datasets consisting only of entities for which a given attribute meets some condition or for a random sample of a given dataset. Datasets and individual attributes can also be deleted once they are of no further use.

Individual attribute values can be modified by using the "alter" command and missing data has special codes to indicate that a value is missing from the attribute vector.

Any attribute or combination of attributes can be displayed using the "display" command. Conditions on other or displayed attributes can be specified or entity numbers given to determine for which entities the values are displayed. For instance, a user may ask to have income and age displayed for age greater than 42 and occupation equal to "salesman".

In the prototype a small statistical facility exists to compute means, medians, distributions, correlations, T-tests, crosstabulations, and stem-and-leaf plots for given attributes. A wider range of statistics is available in the "Consistent System" which is the file environment developed by the Cambridge Project which Janus uses for raw, unstructured file space.

The "define_attribute_map" command is provided to transfer attributes from old datasets into new datasets or into new attributes whose value is the mean or some other function of the original attribute values or an attribute vector in an old dataset. An entire attribute array can be turned into a mean value for that array or the count of the number of non-missing attributes in the array.

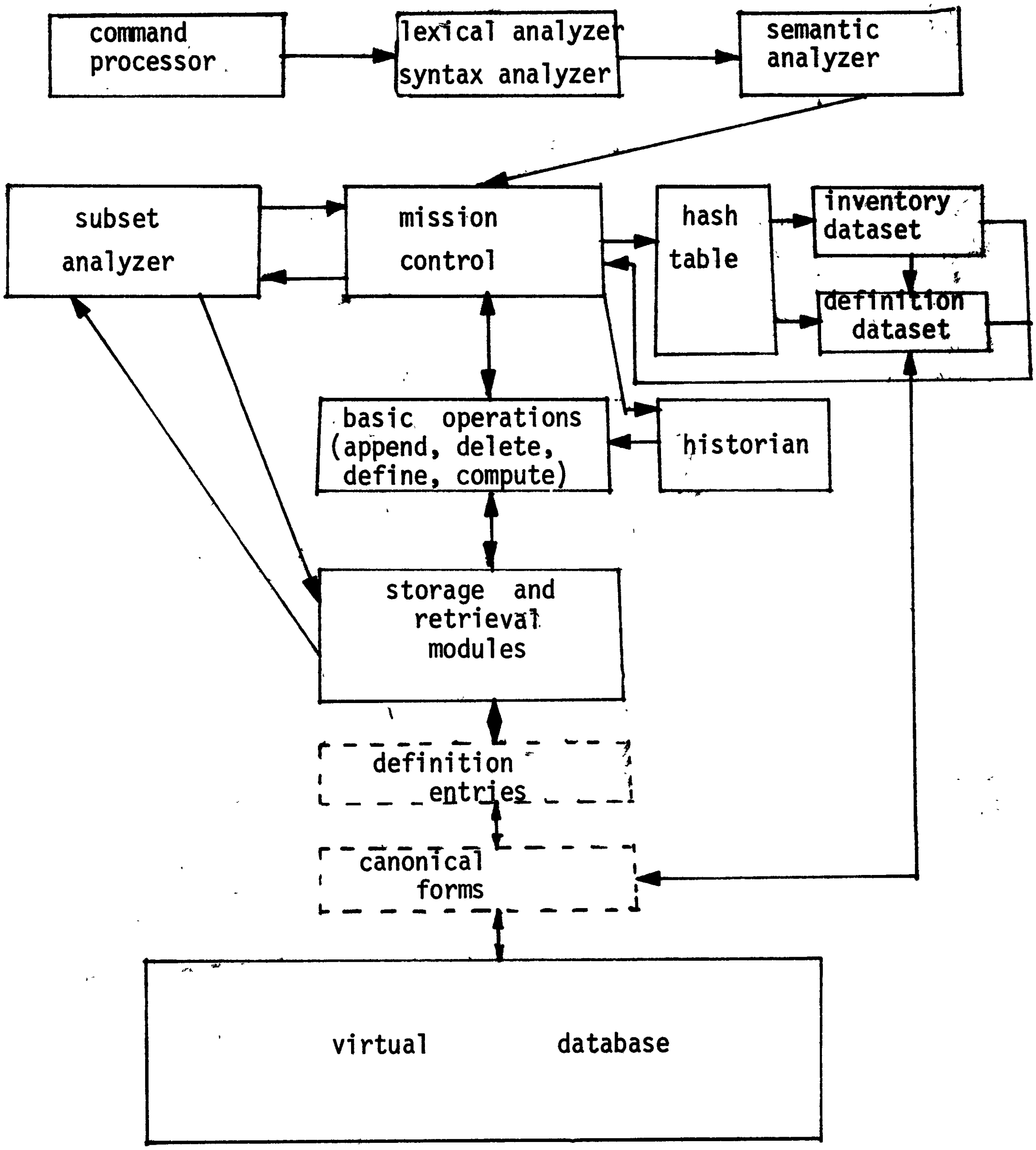
The most important feature of Janus is probably the capability to define relations between datasets. Through an attribute which the user defines to be common to both datasets, a user can define a link between the two datasets based on a common or similar attribute between the datasets. These relations are bi-directional in character allowing the user to reference either from dataset "A" to dataset "B" or back from dataset "B" to dataset "A". Consequently, there is no hierarchy in the relations defining one dataset as superior or parent to another. What results is a matrix of relations in the database with each relation defining a two-way link between two datasets rather than a tree structure. This is an important step towards implementing the relational model of data specified by Codd.(1)

Implementing the capability described in the previous passage is the structure outlined in figure 3.2. The user issues commands at the terminal in a narrative-keyword type of language. These commands go from a command processor to a lexical analyzer which passes the command broken down into tokens to a syntax analyzer. The syntax analyzer ascertains whether or not the command is syntactically correct and passes it to a semantic analyzer which turns attribute names into entries in the definition dataset, dataset names into entries in the inventory dataset, and conditional expressions into a series of entity numbers representing entities whose attribute values satisfy the specified condition, and so on.

The definition dataset stores the information about the attribute's element type (integer, floating, etc.), when it was defined and created, what type of dataset it describes (since the descriptive datasets are also fully described in the descriptive datasets), and other useful information about the attribute. The definition dataset also contains the information necessary to locate and retrieve the item values for an attribute . These include a pointer to the attribute, the length in bits of each element of the

FIGURE 3.2

Model of Janus



attribute, and information necessary to unpack the attribute values if they are stored "packed" in an attribute record. The inventory dataset contains an entry for each dataset in the system including itself and the other system datasets. This entry specifies what type of dataset the entry is, what its specific ID's are, and various information regarding its definition and creation such as time defined, created, last modified and what are its component forms if it was defined as a function of another dataset.

The semantic analyzer performs its task by running the attribute name or dataset name through a hash table to get a key into the proper dataset and the correct location where the entry is stored in the system dataset thus producing an inventory entry for a dataset name and a definition entry for an attribute name. The conditions are sent to a subset analyzer which sends the condition tokens to a precedence analyzer. The precedence analyzer uses the definition entries to retrieve attribute values and interpret the conditions based on the precedence of the user-specified operations or functions to derive a set of entity numbers which meet the specified conditions.

Once a basic operation has output from the semantic analyzer, it simply retrieves the attribute values by passing the definition and inventory entry information and the specified subset of entity numbers to storage and retrieval modules which retrieve the requested information. The routine then performs the requested operation: statistic, display, modification, or deletion that the command requests.

All of the basic operations could be accessed through regular PL/1 subroutine calls, allowing the user to write his own interface to the Janus system. However, Janus was designed to be a non-programming system and accessing desired modules through standard PL/1 could require an experienced programmer. The user would have to input his own pointers to definition and inventory entries and generate his own calls to the subset analyzer. These tasks would require greater knowledge of the internal algorithm of Janus than its designers anticipated for the average user.

The basic operations call various storage and retrieval strategy modules which know from the attribute's definition entry just exactly how the requested information should be retrieved. These modules return the requested attribute values to the calling routine.

For instance, the user types:

"compute distribution of grades"

The command finally reaches the "compute" basic operation which calls a program called, "retrieve", to fetch the values for the attribute, "grade". It passes retrieve a pointer to an entry in the definition dataset which tells retrieve that "grade" is a nominal attribute with five values: 1,2,3,4, and 5, and that each value takes up three bits of packed storage. Retrieve then returns to compute a set of full fixed binary numbers representing the attribute values. Compute then uses the definition entry information to figure out that "1" means "A", "2" means "B" etc. and produces the following output:

```
Grade     Number     %Total

  A          50         25%
  B         100         50%
  C          30         15%
  D          10          5%
  F          10          5%
```

When a relation is requested, the relation definition is used by the basic operation to access definition entries for the related attributes in the two datasets, and the paired entity numbers stored in the relation definition which define the many-to-one or one-to-many pairing of attributes in the two datasets are used to form the cross-transfer between the two datasets through the common attribute in each dataset.

New datasets are added to the database by appending entries to the inventory dataset and new attributes are added to a dataset by appending entries to the definition dataset. Once the attributes are created, the information about the creation such as date-time-created, attribute-record-type, location etc., is stored in the definition dataset.

For a more complete summary of some of the less complicated features of the full Janus system see (6). For a more thorough documentation of Janus's beginning "prototype" system see (5).

Chapter Four

Miscellaneous Modifications to Janus

Janus's solution to the information management problem is by far the most flexible solution of the systems presented in this paper, but certain areas of inefficiency leave room for improvement. First, the user may wish more control over how the data is actually stored. A user has the capability to pack two attribute vectors in a single attribute record, but it may be desirable in some cases to alternate one item value of one attribute followed by another value of another attribute. Second, the user must know exactly what his relations are and how they fit into his question to use the relation capability. Third, specific hardware constraints on the population of a database roll in at about 2000 entities because of the maximum size of a Multics segment and the sorting times in creating relations. Finally, protection is an issue for which Janus has high potential because of the Multics environment but must be investigated more thoroughly.

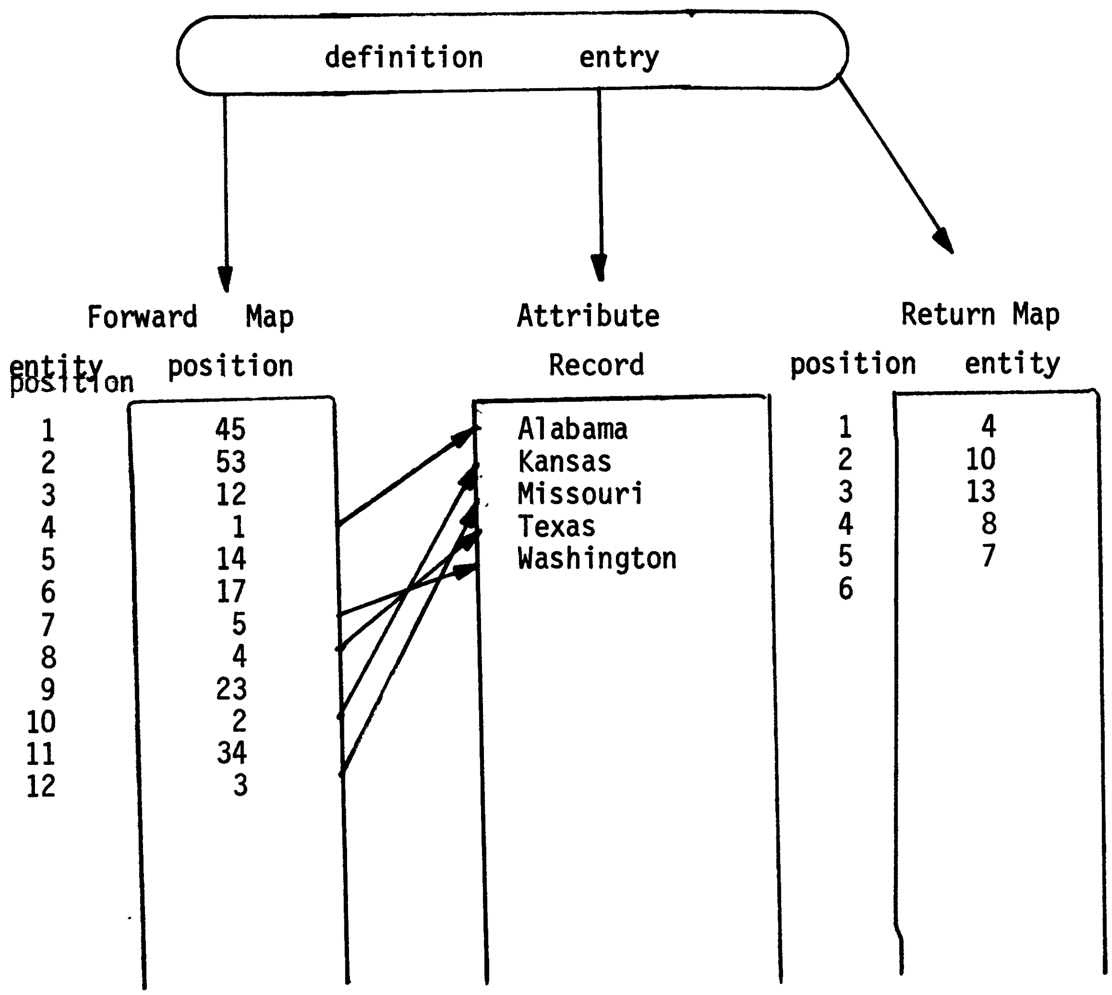
User control over storage structure breaks down into two basic problems. First, there is the problem of linearly searching attribute records to generate a set of entity numbers for the attribute values which satisfy a given condition. This problem could be alleviated by storing the attributes in sorted form. Second is the issue of storing together attributes which will frequently be refernced in the same command.

Conditional Storage Strategies

Storing the attributes in sorted order would be a reasonable solution to the problem of searching entities whose attributes satisfy a specific condition. With this solution, a binary search modified to expand both ways from the matching endpoint to locate multiple occurrences of a searched key could be used to locate the desired subset of entity numbers saisfying the user's requested condition. This, however, would require modifications at both ends of the retrieval strategy scheme shown in figure 4.1. First, since the attributes are no longer stored by implicit entity number order (the first entity in position one, the second in position two, etc.), a pointer in the definition entry must point to a set of offsets describing which index within the attribute record holds the sought-after entity's attribute value, remembering, of course, that our user may still wish to reference each entity's attribute value by entity. This group of entity numbers is the "forward map". Second, since one must have a way to go from the attribute value back to the specified entity number, the entity

FIGURE 4.1

Sorted Attribute Storage



Forward Map

entity position	position
1	45
2	53
3	12
4	1
5	14
6	17
7	5
8	4
9	23
10	2
11	34
12	3

Attribute Record

Alabama
Kansas
Missouri
Texas
Washington

Return Map

position	entity
1	4
2	10
3	13
4	8
5	7
6	

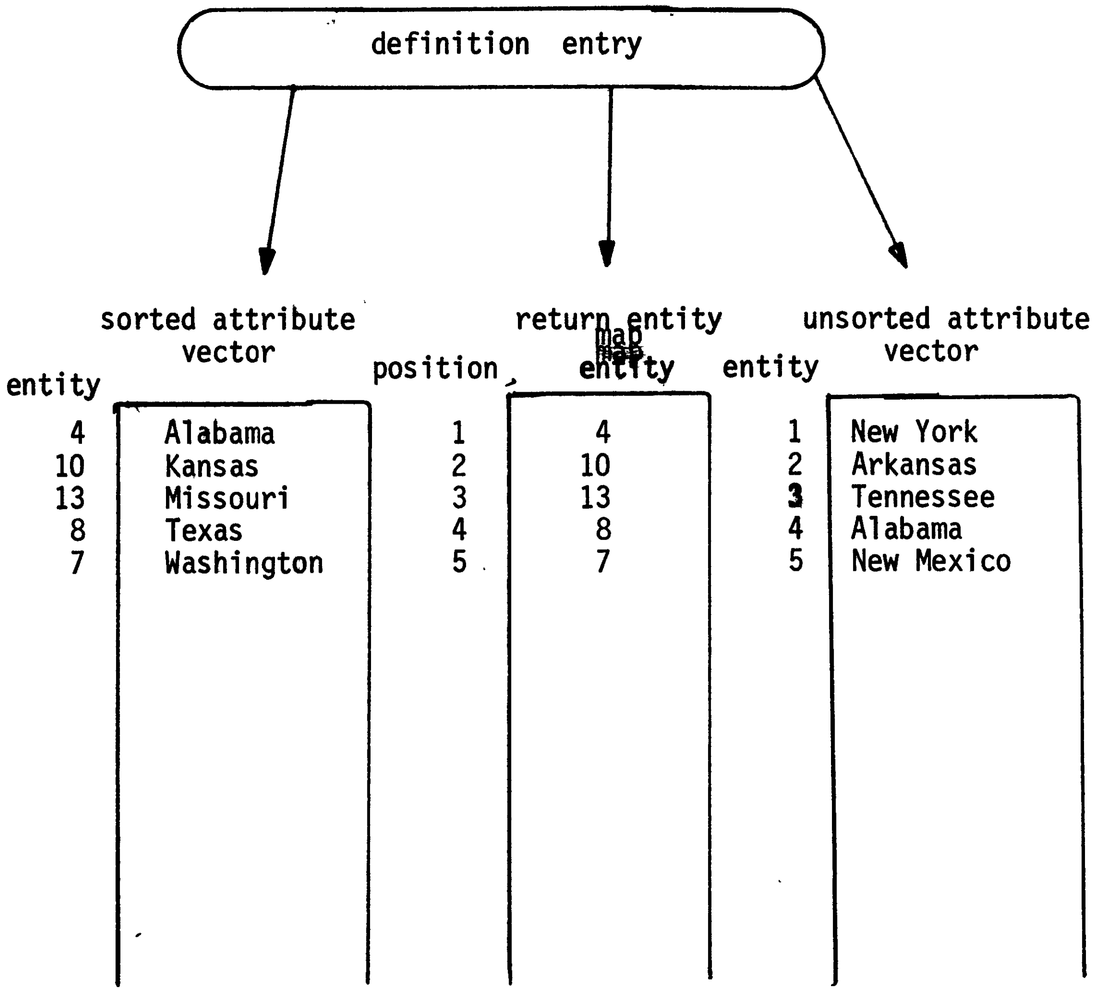
numbers associated with each attribute value would have to be stored with the attribute value or in some other location in the same order as the attribute values to derive the entity numbers once a condition was found. This entity group is the "return map". This list of entity numbers would represent the inverse of the list used by the definition entry to locate the attribute values by entity number for the user. The definition dataset would then maintain, in addition to its pointer to the attribute record, the two pointers to the two entity number records or one pointer to a single record which contained both maps similar to the double set of attribute pairings used in relations.

Unfortunately, instead of each attribute occupying one attribute record, it would occupy one attribute and two entity number records. If the attribute was merely a value between one and five, and there were 400 entities in the dataset, this would represent a serious increase in storage size. Whereas before only three bits of information were necessary to store the attribute value, now 3+9+9 bits are necessary for each entity (2**9 = 512). Since Janus packs its attribute records (using only three bits to store three bits worth of information), this would increase the attribute storage 600% for each entity in this case. This doesn't even count the increased storage for the new pointers and information in each definition entry. A possible variation for cases in which the attribute values took far fewer bits of storage than the entity number record, would be to store the attributes twice, once in sorted, and once in unsorted form. Then, only one "return entity" map would be needed as seen in figure 4.2. This would require 3+3+9 bits per entity amounting to a 400% increase in storage space, still an unattractive overhead, but a lesser burden than a 600% overhead.

For an attribute which has only five unique values (1|2|3|4|5), perhaps each value should be stored only once. With each value could be stored the entity numbers that maintain that value for a given attribute. Thus, the pointer that pointed to an entity number record in the first case could be flagged to point to an attribute value tree as in figure 4.3. At the first node of the tree would be a number telling how many distinct values the attribute took on. This first node could have "n" value pointers to the second level of nodes. Each second level node would tell how many entity numbers existed for that value of the attribute. and list the entity numbers for that specific, unique attribute value. This structure is very similar to what Janus currently uses for command syntax trees. By using this form of storage, in addition to the original attribute record, each of the entity numbers would only have to be stored only once. However, this would entail the burden of

FIGURE 4.2

Sorted Attribute Storage



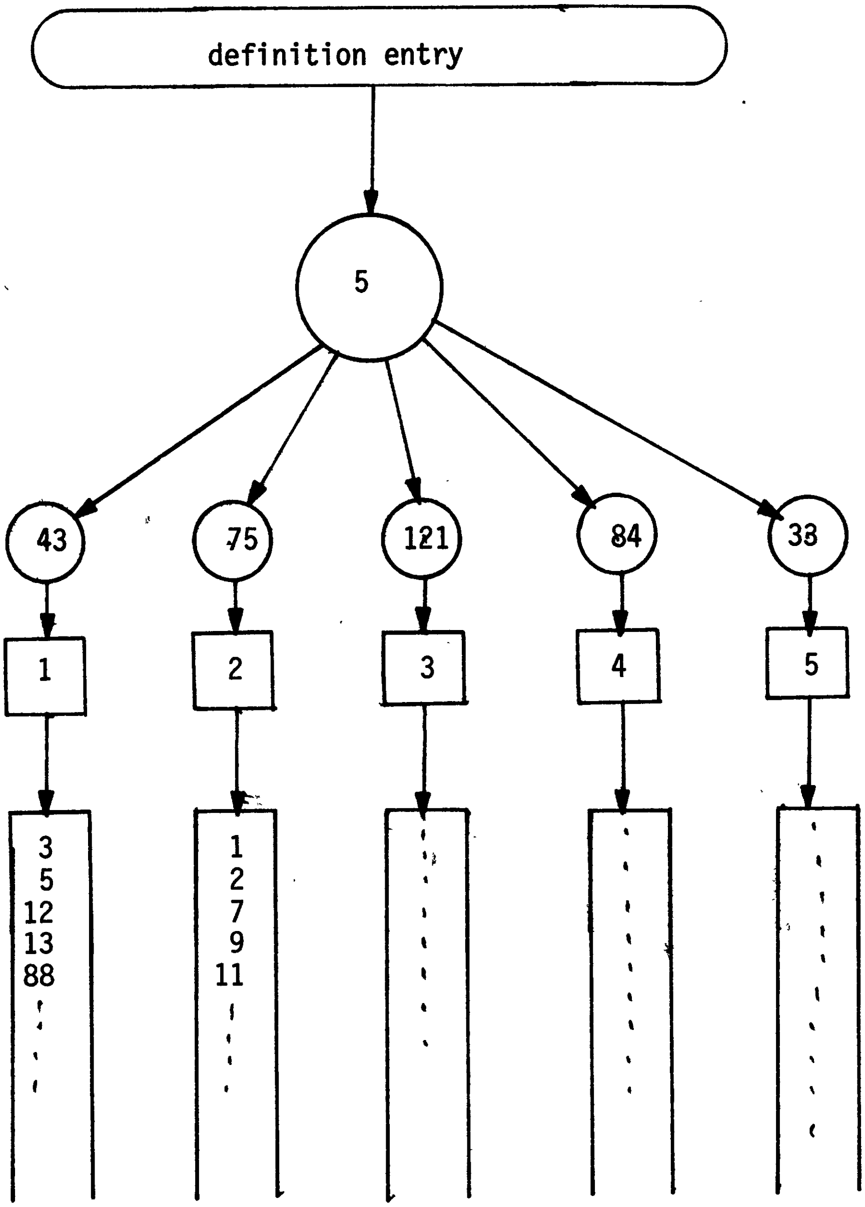
entity	sorted attribute vector
4	Alabama
10	Kansas
13	Missouri
8	Texas
7	Washington

position	return entity map entity
1	4
2	10
3	13
4	8
5	7

entity	unsorted attribute vector
1	New York
2	Arkansas
3	Tennessee
4	Alabama
5	New Mexico

FIGURE 4.3

"Pattern Tree" Attributes



indirect references through the pattern tree to get to the entity-attribute types. This might be considered as a specialized solution for large databases with attributes having few unique values. A second variation is to eliminate the first record of normally stored attribute values keeping only the tree of values. However, this would necessitate searching each entity record to find given attribute values or sets of attribute values when a condition is given for another attribute. This overhead would become prohibitively expensive.

Yet a third possible solution, and the most likely candidate for implementation on the first full version of Janus is the idea of a "set record". This entity number record would contain simply a set of entity numbers generated for one frequently referenced condition of an attribute. This would be implemented by having the subset module first test the attribute value stored at the beginning of all set records for the attribute pointed to by the definition entry. If the specified condition on an attribute matched the attribute value for which the set record of entity numbers was stored, then it would retrieve the entity numbers directly from the set record instead of expensively interpreting the condition on all the attribute values to generate a new set of entity numbers. This would be a limited, singular case of the "pattern tree" storage described above in which only one or a few sets of entity numbers would be maintained or only one or two branches of the "pattern tree" stored.

A fourth possible solution for attributes would be to store and retrieve the values using a hash coding scheme. The values would be hashed into an attribute record. This would require, however, that the attribute record be roughly 1.5 times the size of a regular attribute record and would incur the same overhead as the sorted attribute case. Therefore, it could only be considered feasible for enormously long text or bit strings.

Thus, four possible solutions to more efficient subset reference storage have been presented in this passage. Each might be useful for a specific variation of subset management problems as illustrated in figure 4.4.

Figure 4.4

Summary of Storage Alternatives

Storage Method	Conditions For Invocation
sorted attribute records	large number of unique attribute values each of which is an equally likely candidate for subsetting
"pattern tree" set records	few unique attribute values and a large number of entities
singular set records	one subset of attribute values used frequently to subset the database
hashed attributes	attributes are very long text or bit strings

The first, sorted attribute storage with entity and inverse entity indices, is for cases where there are a number of unique attribute values, each of which is an equally likely candidate for sample subsetting. Second, is the case where there are very few unique attribute values and a very large database. The third is a case of one particular subset being used very frequently and dominating the subsetting capability, a "set record" of entity numbers. And finally, when attribute values are long text or bit strings, a special case of the first solution, hash the attribute values. The user should have the capability to specify which of these alternate storage froms he might wish to use.

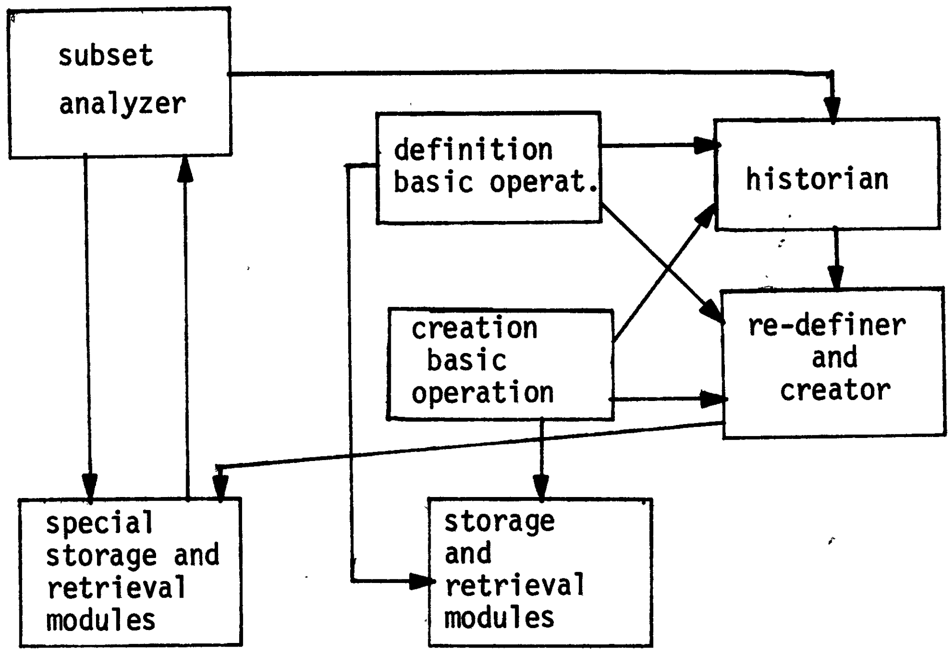
However, a slightly intelligent system might wish to prompt him. If the subset module reported to the historian frequent varying or unique references to a particular attribute, the historian might ask a definition facility to query the user as to whether he wished that attribute to be stored in one of the forms outlined in cases one and three. Secondly, the raw attribute creation facility could note the presence of very few unique attribute values (as would be the case with nominal data that could be caught by the definition facility), note the population, and query the user as to whether "method two" should be added to the storage structures. Third, the data definition facility could catch a long text string attribute as the user defined it and query the user as to whether the hash-coding technique should be used to store the attribute. The user could then be guided by the system as to which storage form might be most beneficial to him. These additions to the Janus model are schematically diagrammed in figure 4.5.

Convenient Storage

The second major storage issue to be addressed is that of keeping attributes together which are referenced frequently in the same command. Two major attributes may often be used together for conditions or displayed together. For this reason, the user may want to store them in the same

FIGURE 4.5

Janus Model Modifications
for New Storage Strategies

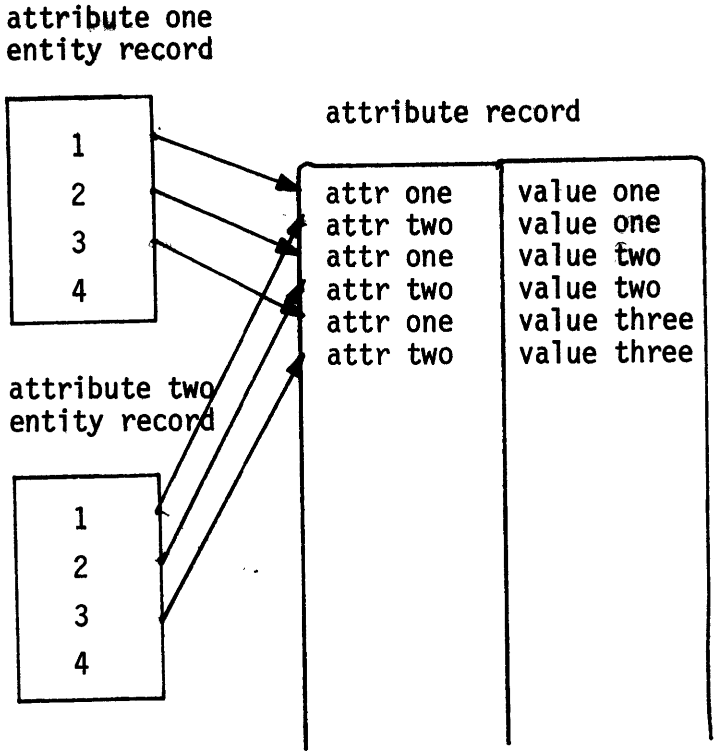


general location so that they can be accessed quickly. In the Multics environment, one has very little control over how data is actually stored on secondary storage. However, as a rule, page faults are one of the more expensive costs of the system and information stored on a single page, stays together. Therefore, if all needed information can be stored on one 1024 word page, costs are minimized. If two attributes were stored alternately (first one followed by the other) as in figure 4.6, then a set of offsets for each entity's value would have to be recorded for the entry. An alternate solution would be to record the function used to compute the attribute's position in the attribute record from the entity number. However, the advantage of using either of the above two methods appear to be far outweighed by disadvantages. The only real advantage would be if the two referenced attribute records could not fit, once packed consecutively, in a single page. By having them located side-by-side, a given condition might be found before having to cross a page boundary. This latter saving also assumes that the attribute values would be rank-ordered, consistent with the case one storage proposal. However, if the attributes are not rank-ordered, even this advantage is lost.

The more practical implementation would be one in which the attributes were stored consecutively in an attribute

FIGURE 4.6

Alternating Attributes in Storage



attr one	value one
attr two	value one
attr one	value two
attr two	value two
attr one	value three
attr two	value three

Alternate Function Solution

attribute one address = 2*entity # - 1

attribute two address = 2 * entity #

record. The user himself could request this scheme, or the historian could observe from the information passed to it by the display operations or conditions that two or more attributes were frequently being referenced together. Detecting this condition, the historian would signal the re-definer module (the re-definer previously presented) to query the user as to whether the attributes should be extracted and stored over again in a single page together. The user can already perform this repacking operation himself in the full Janus system.

The historian as currently implemented in the Janus system records for each call to a program how long that call took in computer time and page waits. It also records various other types of information, such as the maximum stack level that a user reaches, how much temporary storage is used in a command line, etc.. The function of the historian could be extended to include a record of each attribute referenced; how many times it is referenced, with what conditions, and with what other attributes it is referenced as illustrated in figure 4.7. "How many times" would be a single index, and "for what conditions" would be an offset to an area containing a single or a set of distinct attribute values representing referenced conditions. "With what other attributes" would be an offset to a set of attribute hash table indices representing

FIGURE 4.7

Historian's Record of Storage Retrieval

attribute index from hash table	number of times referenced	offset to indices of other attribu referenced	offset to conditional values with which referenced
1	48	075432	063421
2	32	036431	053172
3	1	777777	777777
4	3	777777	777777
5	25	368421	421378
6	7	777777	777777
7	0	777777	777777

entries into the definition dataset (attributes). Periodically, at the end of a session, the historian might process this information to look for certain patterns. These patterns would include a high index for "number of references to an attribute". Multiple occurrences of given conditions on each attribute and multiple references to another attribute with a given attribute would be another pattern to be flagged. (If said pattern had not already been spotted and silenced.) Once a condition was flagged, the user could be queried whether action should be taken and further queries would be suppressed by an action flag. This capability would be similar to but more powerful than IMS's "logging of transactions". Crash protection would not be a primary objective since segments are transferred to secondary storage shortly after they are written. But this would provide a greater analytic capability than IMS's transaction logging.

Database Size Constraints

Database sizes are constrained by a number of problems. Among the more important of these are maximum sizes of Multics segments and sorting times for creating relations in large databases.

The segment size constraint is a present hardware limitation in Multics which restricts the size of any storage segment to 64K words. This restriction may be lifted in the future. However, currently slated for implementation is a facility for specifying files as containing multiple segments. A chaining is performed between the segments of these "multi-segment files" giving the appearance of a much larger segment. This could be used as an interim solution to the database size constraint.

Yet another, perhaps more important constraint is the time it takes to sort an array of over 500 items. Already some of the fastest documented sorts are being used to perform this (algorithms 271, & 347 of the CACM, quickersort and faster quickersort). Even radix-exchange sorts do not exceed the speed of this highly-tuned shell sorts. What would remain is to perform sort-merging on arrays of over 2000 elements. This would entail first assessing the distribution of the array to be sorted and establishing what maximum number of pages which could be referenced by one call to quickersort be used given the current load on the operating system. (Each time more than one page is being sorted, one faces the risk of a very expensive page fault. The only surely safe amount to sort at one time is one page.) Then the sort would have to divide the sample into the correct number of "n"-page segments to be sorted. The value, "n", would be determined such that the chance of a page fault during sorting would be minimized while considering the other parameters such as sorting times for "n"/"m" entities.

Another costly solution to the sorting problem is to hash the attribute values. This, however, must include the costs of grabbing the temporary storage necessary to perform the hashing and retrieval of indices which indicate where the hashed values originate. Unfortunately, the number of page faults necessary to perform this process is significantly prohibitive to discourage hashing. However, the availability of scratch space in a paged environment would certainly make this possibility worth considering should the cost of page faults drop immensely.

Protection

A fourth area which Janus is beginning to address is protection. The Multics environment allows user-project unique access keys on each segment. These specify for each user on the system whether or not he may read, write, execute, or add to a given segment. However, this would not be adequate for protection in Janus storage structures since many attributes can be stored in a single segment or attribute record. Consequently, something more clever is needed. Multics also has a protection concept known as "rings". This employs a simple idea of fences in which anyone in an inner ring can get to anyplace in an outer ring with no difficulty. However, only certain programs can cross the fences to get from an outer ring into an inner ring as illustrated in figure 4.8. Access for a user could initially be set to null on programs which crossed the rings. These programs could be stored under secret names in subdirectories of directories to which the user had only write access in a ring external to the database as in figure 4.9. When a user typed in a certain password, he would be given execute access to the proper programs only long enough to perform the given operation on a database. "Quits" would be caught and clean-up performed before returning control to the user. The user would have a difficult problem trying to stop the process in the middle and figure out what was going on before the historian recorded what he was doing and filed a report to the project supervisor. This "ring" or "privileged program" implementation scheme would be similar to that used by IMS in implementing its protection mechanism.

The way that protection could be implemented at the entity level would be an attribute of passwords which the user would have to match for each entity he wished to reference. The same idea could be applied to attribute-level protection by having passwords stored from definition entries which the user would have to match to reference an attribute as in figure 4.10. These passwords could be placed

FIGURE 4.8

"Rings"

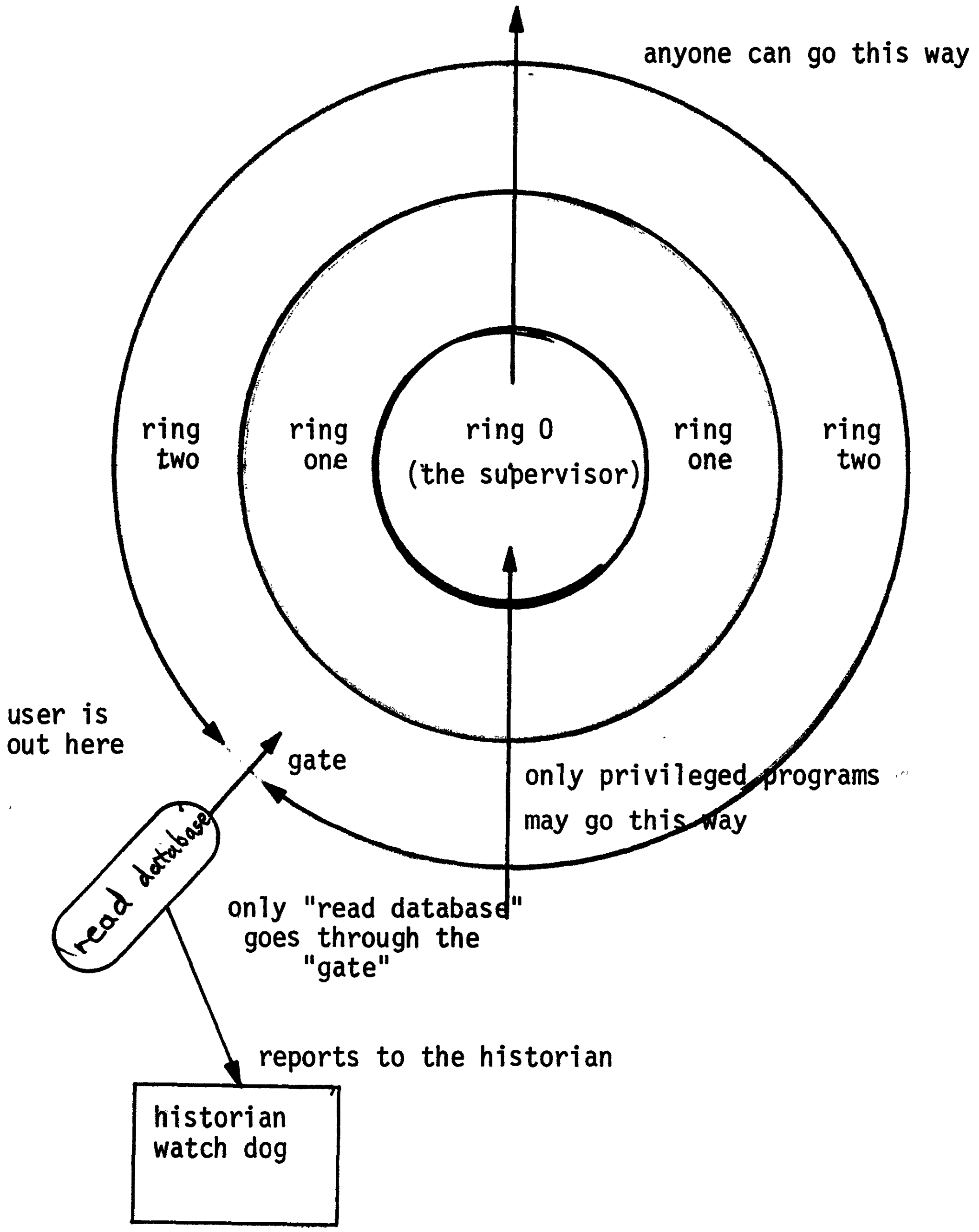
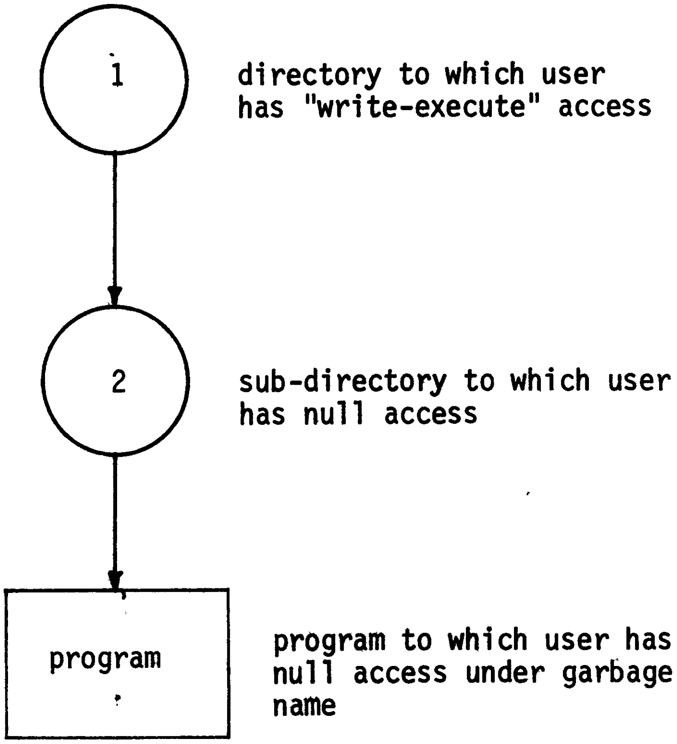


FIGURE 4.9

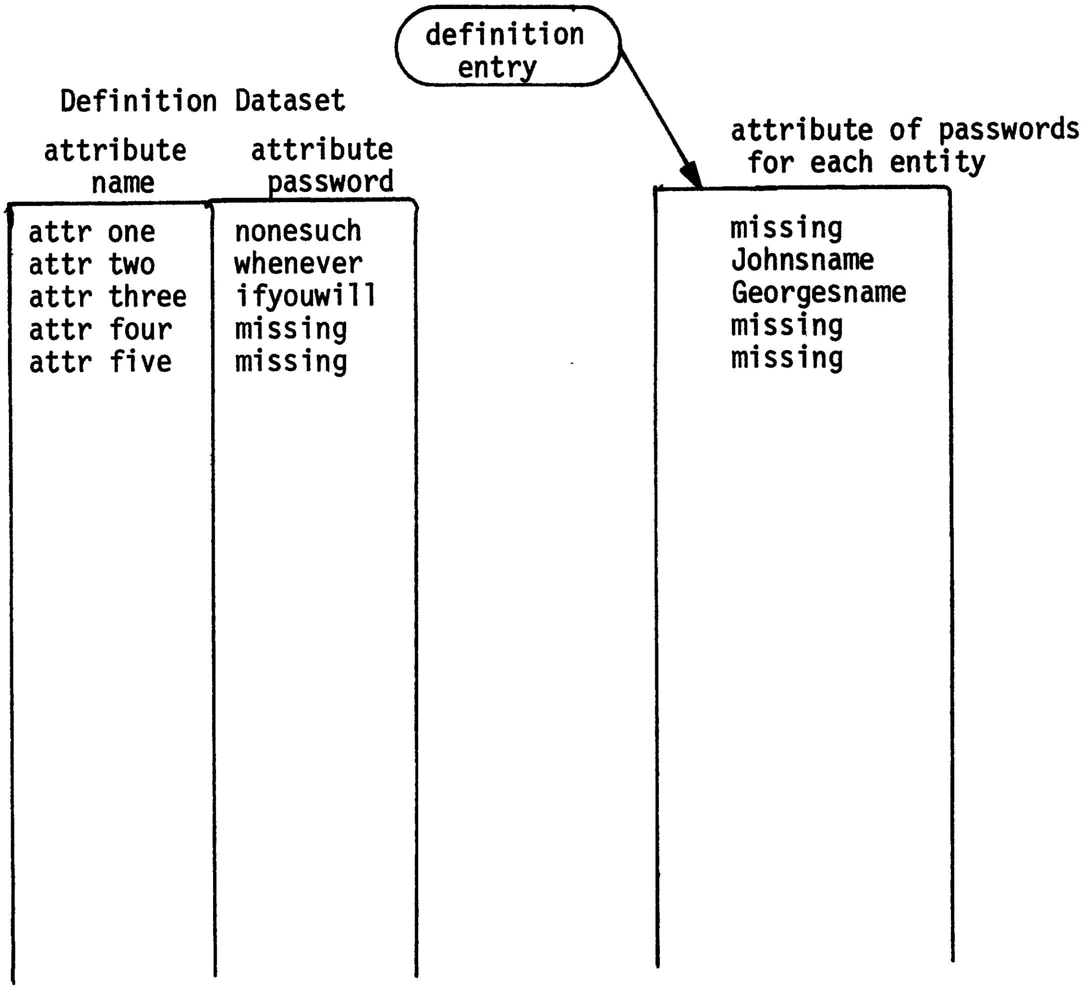
Protection Scheme


1
directory to which user has "write-execute" access
2
sub-directory to which user has null access
program
program to which user has null access under garbage name

Steps:
1. catch all "quits"
2. set access first to "read-execute" on sub-directory
3. set access on program to "read-execute"
4. execute request using the ring-crossing program
5. restøre accesses

FIGURE 4.10

Entity and Attribute Level Protection Schemes



Definition Dataset

attribute name	attribute password
attr one	nonesuch
attr two	whenever
attr three	ifyouwill
attr four	missing
attr five	missing

attribute of passwords for each entity
missing
Johnsname
Georgesname
missing
missing

NOTE: missing values signify either "no access" or "all access" in an inaccessible ring from the user and be referenced by a program that simply matched passwords and did not allow reading the already stored passwords.

Formatted Output

A fifth major area where Janus is weak is in formatted report generation. However, the most difficult aspect of accessing attributes for display is handled by a single PL/1-callable module. A specialized program could be written by a user to handle a specific formatted output problem which called the display module using PL/1. Beyond that capability, writing a more sophisticated display facility is essentially a trivial but tedious process of algorithm design.

Chapter Five

Automatic Relation Defining

The relational capability of Janus is its most powerful analytic feature. Using this, the user can create any relation structure he wishes between datasets including matrix or tree structures. This capability will be demonstrated using a few examples.

Figure 5.1

Two Datasets with Var Del Attributes

Dataset One -- drug_infections

Drug	Infections
penicillin	respiratory, staph, strep
aureomycin	inflammatory, coccol
neomycin	viral respiratory

Dataset Two -- disease_infections

Infections	Diseases
respiratory	pneumonia, tuberculosis, bronchitis
staph	boils, cauliflower, lesions
viral	colds, Hong_kong_flu
inflammatory	strep_throat, acute_congestion

(The author claims to lack even vestigial levels of medical expertise)

In figure 5.1 is a listing of two datasets. The first one is a set of drugs and a variable number of infections which the drugs will cure. The second dataset contains a list of infections and the various diseases which can produce these infections. A doctor comes with the question: "What drug should I use to cure pneumonia?"

To solve this problem one must first expand these datasets to create unique pairs of drugs and the infections which they cure, and infections with the diseases they represent. This creates two expanded datasets whose attributes appear as seen in figure 5.2.

Figure 5.2

Two Datasets with Expanded Var Del Attributes

Dataset Three -- drug_infections_expand

Drug	Infections
1.penicillin	respiratory
2.penicillin	staph
3.penicillin	strep
4.aureomycin	inflammatory
5.aureomycin	coccal
6.neomycin	viral
7.neomycin	respiratory

Dataset four -- disease_infections_expand

Infections	Diseases
a.respiratory	pneumonia
b.respiratory	tuberculosis
c.respiratory	bronchitis
d.staph	boils
e.staph	cauliflower
f.staph	lesions
g.viral	colds
h.viral	Hong_Cong_flu
i.inflammatory	strep_throat
j.inflammatory	acute_congestion

Each infection has listed for it one drug which is used to treat the infection. Each disease has one infection which represents the symptoms of the disease.

A relation is defined between the two datasets based on all infections which match between the two datasets. This generates a set of entity number pairs going from the expanded drug_infections dataset to the expanded disease_infections dataset and back the other way as seen in figure 5.3.

Figure 5.3
Creating a Relation Between Two Expanded Datasets

```
define_relation drug_diseases many_many from
 infections in drug_infections_expand to
 infections in disease_infections_expand
```

Dataset_three	
1	respiratory
2	staph
3	strep
4	inflammatory
5	coccal
6	viral
7	respiratory

Dataset_four	
a	respiratory
b	respiratory
c	respiratory
d	staph
e	staph
f	staph
g	viral
h	viral
i	inflammatory
j	inflammatory

Relation Pairs

these entity number pairs match entities in dataset three with entities in dataset four using the attribute: "infections"

dset3-index dset4-index

1-a
1-b
1-c
2-d
2-e
2-f
4-i
4-j
6-b
6-h
7-a
7-b
7-c

Inverse relation Pairs

these entity number pairs match entities in dataset four with entities in dataset three using the attribute: "infections"

dset4-index dset3-index

a-1
a-7
b-1
b-7
c-1
c-7
d-2
e-2
g-6
g-6
h-6
i-4
j-4

For each infection in the first dataset there is a matched infection in the second. For instance, the pair, "1-a", in the first column means that entity "1" in dataset three matches entity "a" in dataset four. These match because entity "1" in dataset three and entity "a" in dataset four both contain the attribute value, "respiratory". Going back to the two expanded datasets in figure 5.2, we see that respiratory appears in the first entity for both the third and the fourth datasets. The two attributes, "penicillin" and "pneumonia" also appear for entity one in the two datasets. Therefore, using the "1-a" pair in figure 5.3, one can pair the disease, "pneumonia" with the drug to treat it, "penicillin.

Finally, all that remains is to display for a given disease, all the drugs which will cure it going through the relation between the infections in the two datasets as seen in figure 5.4.

Figure 5.4

Retrieving Drugs to Treat Pneumonia

```
display drugs, in drug_infections_expand
 for image of disease_infections_expand
      for disease = "pneumonia"
```

Output:

```
Drugs

penicillin
neomycin
```

Now, to make problems more difficult, for each infection we have a set of symptoms, and for each disease there is a set of symptoms as in figure 5.5.

Figure 5.5

Two New Datasets with Var Del Attributes

Dataset_five-- infection_symptoms

Infections	Symptoms
respiratory	cough, fever, sore_throat, nasal_drip
staph	fever, swelling
viral	sore_throat, swollen_glands, nasal_drip, aching
inflammatory	fever, aching, swollen_glands

Dataset_six -- Disease_symptoms

Diseases	Symptoms
pneumonia	cough, fever
bronchitis	cough, fever, sore_throat
boils	fever, swelling
colds	cough, core_throat, nasal_drip
Hong_Cong_flu	swollen_glands, aching, sore_thoat

Each of these datasets is expanded into the expanded datasets appearing in figure 5.6.

Figure 5.6

Two New Datasets with Expanded Var Del Attributes

infection_symptoms_expand

Infection	Symptoms
A.respiratory	cough
B.respiratory	fever
C.respiratory	sore_throat
D.respiratory	nasal_drip
E.staph	fever
F.staph	swelling
G.viral	sore_throat
H.viral	swollen_glands
I.viral	nasal_drip
J.viral	aching
K.inflammatory	fever
L.inflammatory	aching
M.inflammatory	swollen_glands

disease_symptoms_expand

Diseases	Symptoms
pneumonia	cough
pneumonia	fever
bronchitis	cough
bronchitis	fever
bronchitis	sore_throat
boils	fever
boils	swelling
colds	cough
colds	sore_throat
colds	nasal_drip
Hong_kong_flu	swollen_glands
Hong_Kong_flu	aching
Hong_kong_flu	sore_throat

A relation is defined between the inflated infection_symptoms dataset and the expanded drug_infections dataset producing the cross-reference shown in figure 5.7.

Figure 5.7

Creating a Relation Between Two New Expanded Datasets

Dataset Three (drug_infections_expand)	infection_symptoms_expand
1 respiratory	A respiratory
2 staph	B respiratory
3 strep	C respiratory
4 inflammatory	D respiratory
5 coccal	E staph
6 viral	F staph
7 respiratory	G viral
	H viral
	I viral
	J viral
	K inflammatory
	L inflammatory
	M inflammatory

Relation pairs	Inverse relation pairs
(see figure 5.3	for explanation)
1-A	A-1
1-B	A-7
1-C	B-1
1-C	B-7
2-E	C-1
2-F	C-7
4-J	D-1
4-L	D-7
4-M	E-2
6-G	F-2
6-H	G-6
6-I	H-6
6-J	I-6
7-A	J-6
7-B	K-4
7-C	L-4
7-D	M-4

Finally, the doctor says, "I have someone with a cough and a fever, what should I give him?" To ask this question, the user would display for symptoms = cough and fever in the expanded infection_symptoms dataset all the drugs that will cure it in the expanded drug_infections dataset going through the relation between the two datasets as seen in figure 5.8a.

Figure 5.8a

Finding Drugs to Treat a Cough and Fever

```
display drug, for image of infection_symptoms
  for symptoms = cough & symptoms = fever
```

Output:

```
Drug

penicillin
penicillin
```

From the prevalence of penicillin as the solution, the doctor might infer that penicillin would be the best choice. From there, the doctor might wish to discover what disease the system assumes he is treating. For this, he would ask for all the diseases and symptoms for symptoms = cough and fever. This would yield the results appearing in figure 5.8b.

Figure 5.8b

Finding a Disease Whose Symptoms are a Cough and Fever

```
display disease in disease_symptoms_expand
for symptoms = cough & symptoms = fever
```

Output:

```
Disease

pneumonia
bronchitis
```

He could conclude from these results that he was treating either pneumonia or bronchitis, with a reasonable question as to which. If he computed a distribution in the first case instead of displaying all of the outputted variables, he could have each solution printed only once.

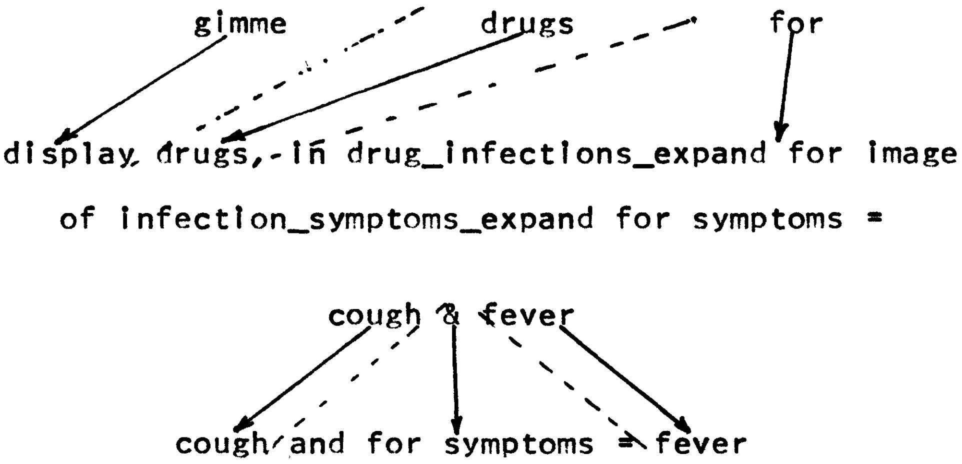
This process of creating maps and cross-references can become somewhat complex. It might be desirable to free the user of some of this burden of mapping, relation-defining, and complicated display commands.

One solution to this problem is to maintain a sophisticated database administrator, as in IMS, whose personal task would be creating all the mappings and relations and maintaining the database. From that point, the more naive user could be given a set of abbreviations which performed the complicated display command more simply.

For instance, the doctor might type:

gimme drugs for cough & fever

The system would then perform the following abbreviation substitutions on the statement:



gimme drugs for

display drugs, in drug_infections_expand for image
of infection_symptoms_expand for symptoms =

cough & fever

cough and for symptoms = fever

This way, the simple statement:

"gimme drugs for cough and fever"
would perform the complicated request for the doctor.

Beyond this capability, defining maps and relations automatically for the user becomes somewhat more complicated. One solution would be for the system to automatically create the map of a defined dataset with var del attributes as soon as the user exits from the system, or upon a command request from the user. After this, the user would simply ask for the display in the normal fashion:

display drugs, for symptoms = cough and fever

The system responds:

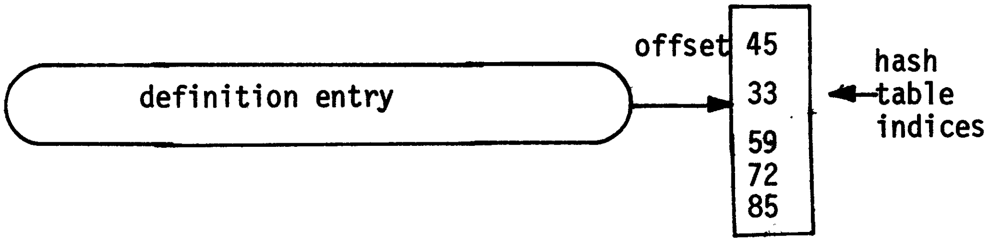
cough and fever are not attributes in your present dataset, would you like to create a relation with the infection_symptoms dataset through the attribute, "infections"?

Before giving this response, the system would first have had to search the definition dataset for the attributes, cough and fever; having ascertained that the attributes did not have the proper dataset ID's (i.e. were not attributes in the current dataset) it would then check both datasets for some attribute with a name common to both datasets. Finding a match, it would query the user whether the matching attribute pair should be used to create the relation. Receiving a "no" reply, the system would continue to search the definition dataset for entries until the list was exhausted as in figure 5.9 At this point, a new mechanism would be needed to create the relation.

A perfectly plausible possibility is that a given attribute in one dataset means the same thing as an attribute by another name in another dataset. To establish the fact of their similarity, a synonym table could be created. This would contain, for each attribute, a list of synonyms by which the attribute was known in other datasets

FIGURE 5.9

A Synonym Table



Hash Table

entity number	attribute name	attribute ID	dataset ID	dataset index
1	population	33	1	33
2	synonym one	777	4	26
3	del_dim_1	24	2	24
4	synonym two	777	4	26
5	real_attribute	24	4	24
6	larengitis	27	4	25
7	cold	32	4	27

actual index in the hash table of the real attribute for which this is a synonym

index of this entry in the given dataset

The ID of the dataset in the inventory

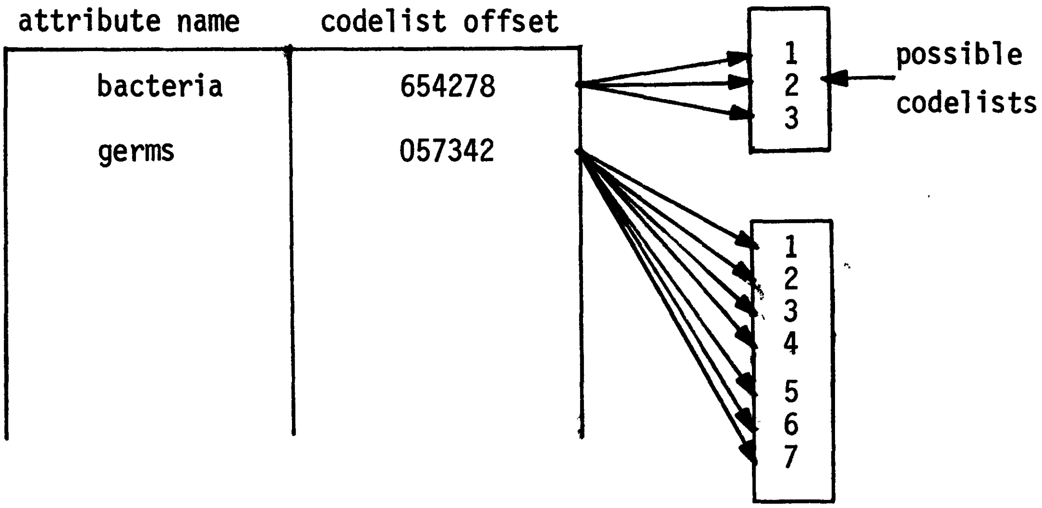
The ID of the attribute in its dataset (777 means it is a synonym) or by other people. This entry in the synonym table could be pointed to by an index in the definition dataset. As the relation-creator searched the hash table, it would reference an attribute's synonyms to check for common names.

Another problem which the system might face would be attributes which mean the same, but have different coding forms in different datasets. For instance, in one dataset there may be an attribute, "marital status", which would, in another dataset, be called "marriage state" as pictured in figure 5.10. Fortunately, such cases as these occur frequently when the attribute is nominal; that is, each attribute takes on only a limited number of possible values. Each of these values has associated with it a specific "codelist" or name for the nominal category. These "codelists" are stored in a "codelist dataset". For two attributes in different datasets which had the same codelists in the codelist dataset, one would need only to check the two attributes' codelists to determine that they represented the same information. Discovering these identical attributes could be a once-per-session task performed at the end of the session. Synonyms would be created in each dataset for the common nominal attributes.

Complicating this issue is the entire subject of attributes or attribute groups which store the same information, but are coded differently. For instance, the

FIGURE 5.10

A Codelist Dataset



	codelist dataset
1	cocci
2	strep
3	staph
4	virus
5	worm
6	word
7	
8	

bacteria are only cocci's, streps, and staphs

but germs include bacteria plus viruses, worms, and words single attr ibute sets, (single/married & separated/divorced) could be re-defined as a new attribute which contained all the information of the original two:

```
     marital status = 1 if attrl = single
         2 if attr 1 = married & attr2 = missing_code
         3 if attr2 = separated
         4 if attr2 = divorced
```

This is a simple exercise in redefining attributes from given specifications. Performing this redefinition automatically, however, presents an extremely complex interpretive problem. The system must first ascertain which attribute in which dataset contains the largest superset of codelists in the other dataset. A table must first be generated containing, for each nominal attribute in each dataset, the number of codelists and what each codelist represents numerically. The system would begin with the attribute in either of the two datasets containing the largest number of codelists and search for all recurrences of those codelists in the other dataset. It would record the number of recurrences and proceed to the next largest attribute recording the number of recurrences. It could be told to dispose of recurrences equalling only one and to create all the other new compound multiples of new attributes. Once the new compound multiples were created, the system would query the user one-by-one starting from the largest consolidated multiple pair (the one which had the greatest number of codelists matching in both the datasets) which the user would wish to use to create the relation.

Unfortunately, the solution could become somewhat expensive as the system searched all possible codelist attributes in both datasets for optimal matches, and the results would not necessarily be meaningful. For instance, the codelist elements, "white" and "black", could be used both as indicators of school colors or some-such, and race. However, a relation between the attributes, "school colors" and "race" would not necessarily be very meaningful. Consequently, the user would have to be asked for each match, whether or not the relation between the two attributes would be desirable.

Therefore, the only practical implementation scheme appears to be through the synonym table. This would use the following algorithm. (Following figure 8.1 through this expanation might prove immensely helpful to readers.):

1. Subset analyzer receives attributes in a condition expression. It cannot locate the given attributes in the default dataset and queries the user whether it should invoke the relation-creator.

2. The relation creator searches the definition dataset for the unidentified attributes. As it searches, each attribute's synonyms, if any, are checked for a match.

Note: although abbreviations could be used to handle all synonyms at the command line, synonyms must be present to make distinctions between synonym names for the system at the dataset level.

3. If the attributes cannot be found in the definition dataset, the system informs the user and returns to command level.

4. If the attributes are found, the specified dataset_id's are noted and a relation is sought between the inital and specified datasets, using the relation table.

5. If either dataset has a map dataset (specified by a flag in a inventory dataset) the map dataset is used instead of the original dataset.

6. If a relation is found, the subset is formed using the image of the specifed attributes.

7. Otherwise, the system attempts to create a relation.

8. The definition dataset is searched using attributes in the initial dataset to search for a match. The attribute name and its synonyms are searched against all attributes and their synonyms in the target dataset.

9. Failing a match, the user is informed and the system returns to command level.

10. Finding a match, the user is queried whether he would like to create the relation using those two attributes. Receving a no, the system returns to step eight.

11. Getting a go-ahead for creation, it creates the relation between the two datasets and goes to step six.

Note: all map datasets must automatically be created on a "leave" from Janus unless the "suppress map" option is turned on in the user's profile. Otherwise, this mapping will have to be performed before the relation may be created.

Using this scheme, automatic relations could be created, somewhat expensively, in Janus. The project administrator would be an excellent insurance against "garbage-in: garbage-out". However, I believe the algorithm presented would be a reasonable substitution for a sophisticated user who understood the system.

Chapter Six

Network Relation Defining

The subset problem is easily solved when the relation needed to produce the subset is dyadic, that is, it connects only two datasets together. But what if the problem becomes more difficult and two datasets must be linked through their relation to yet a third dataset? For instance, suppose we have the three datasets shown in figure 6.1.

Figure 6.1

Three Interrelated Datasets

dataset one (residences)

attributes:

name	soc. sec. #	street	home city
Ackerman	315488577	629 Lincoln	Kokomo
Binder	483126834	372 Shawnee	Marion
Oberst	218364312	212 Lefty	Anderson

dataset two (drivers registration)

attributes:

name	soc. sec. #	registration #
Ackerman	315488577	27A513
Oberst	218364312	34C113

dataset three (parking tickets)

attributes:

city	badge #	registration #
Kokomo	G654	27A513
Anderson	G386	27C633

A user wants to know the cities in which a given person has accumulated parking tickets that were not in his home city using a license plate number. To do this, the user issues the command:

```
display city in parking tickets for city ¬= home_city
and registration in parking tickets = "27A513"
```

The subset analyzer receives this and searches dataset three for the attribute, "home_city". Failing to find the attribute, it hands the condition to the relation-definer. The relation-definer first uses the normal dyadic relation creation process to attempt to find the attribute, "home_city" from the attribute, "city".

Failing a satisfactory match, it then invokes the creation algorithm for triadic relations. This algorithm begins with the two attributes which must be matched: "city" and "home_city". These two attributes form the starting point for two analytic lists as shown in figure 6.2.

Figure 6.2

Linking Dataset Three to Dataset One

Hash Table

attribute name	dataset ID
name	1
ss #	1
street	1
home_city	1
name	2
ss #	2
reg #	2
city	3
badge #	3
reg #	3

Analytic Lists

list for dataset three		list for dataset one	
attr name	dset ID	attr name	dset ID
city	3	home_city	1
reg #	2	name	2

Beginning with the attribute in the "initial dataset" (city), it searches the hash table for an attribute whose dataset ID matches that of the initial attribute which is in this case "reg #". It then searches the hash table for another attribute or synonym that matches "reg #" from another dataset, and finding that "reg #" is also present in dataset two, it enters this attribute and its dataset ID into the analytic list as a potential candidate for a relation between the two datasets. If an attribute with a dataset ID of two had already been added to the list and the potential for a relation with dataset two had already been developed, then there would be no need to add another possibility for a relation with dataset two to the list. The relation creator would then proceed to find a match for the next attribute found whose dataset ID is three in the hash table.

Once a match has been found and entered into the first analytic list for the initial attribute, the process is performed to enter an attribute into the second analytic list for the target attribute ("home_city"). In this case, the attribute which appears in the second analytic list in figure 6.2 is "name" which appears both in dataset one and dataset two. After one attribute has been added to the list for the initial and the target attributes, a comparison is performed between the two analytic lists to discover if any dataset ID appearing in the first list also appears in the second list. In this case, dataset two appears in the first list for the attribute, "reg #", and in the second list for the attribute, "name". Through these two attributes in dataset two a path has been found from dataset one to dataset three. The relation-creator then creates two relations: one from the initial dataset to the intermediate dataset using the attribute, "reg #", and one from the intermediate dataset to the target dataset using the attribute, "name". The relation-creator would issue two commands which, if the user typed them himself, would be:

```
create_relation from reg # in parking tickets (dataset
three) to reg # in drivers_registration (dataset two)

create_relation from name in drivers_registration
(dataset two) to name in residences (dataset one)
```

Once this is accomplished, the relation-creator simply retrieves a double image, going through the two relations just defined to retrieve the requested information. This would be equivalent to typing:

```
display city in parking_tickets for image of
reg#_relation for image of name_relation for home_city
¬= city
```

This command would then display all the cities in which a person had accumulated parking tickets that were not in his home city.

This, then, becomes a process for creating relations through two datasets (which I am calling triadic relations) which may be summarized in the following algorithm:

1. The relation creator has followed the "attempt to create a dyadic relation" process and failed.

2. The two attributes being related and their dataset ID's are placed in two analytic lists. The first list is for the "initial attribute" (the attribute in the current default dataset) and the second list is for the "target attribute" (the attribute in the distant dataset).

3. If any relations have been defined between the initial or the target datasets and other datasets, the attributes forming these relations and the dataset ID's of the related datasets are added to the list and used as new initial or target attributes. This step is iterated repeatedly until all existing relations to the relevent datasets are found.

4. Beginning with the initial attribute's dataset ID, the hash table is searched for another attribute with a dataset ID equal to that of the initial attribute. The attribute matching the initial attribute's dataset ID will be called the "intermediate attribute".

5. The hash table is then searched for another attribute or synonym with the same name as the intermediate attribute's. This matched attribute will be called the "matching attribute".

6. If the matching attribute's dataset ID does not appear previously in the first analytic list (i.e. the dataset with the ID of the matching attribute is not represented by another attribute in the first analytic list) then the user is asked whether the matching attribute should be added to the first (initial attribute's) list. If the answer is "yes" then the attribute is added.

7. If the entire table is searched with no match, then another attribute whose dataset ID matches that of the initial attribute is chosen, and another, until a match is found or the hash table is exhausted of attributes with the inital dataset ID. Step three is re-iterated for the match.

8. If no match is found and the first list has no entries other than the initial attribute, the attempt for the triadic case is declared a failure.

9. If no match is found and the first list has at least one additional entry, steps three through six are repeated for all possible matches in the second list and the algorithm proceeds to step eleven.

10. Steps three through seven are performed for the second list (the target attribute) until a match is found or the hash table is exhausted of attributes in the target attribute's dataset.

11. Once at least one match has been found for the initial and the target attributes, or the condition described in step eight has been reached, a comparison is performed.

12. Each dataset ID in the second list is searched for an occurrence of a dataset ID in the first list. Finding a match between two dataset ID's in the two lists, a path has been found between the initial and the target dataset which we will call the "intermediate dataset" and the relation-definer proceeds to step twelve. Otherwise, steps three through six are performed once again for each list and the relation-definer proceeds to step ten. If the algorithm has already reached step eight, declare a failure.

13. Finding a path, two relations are created: one between the initial dataset and the intermediate dataset based on the first path attribute, and one between the intermediate dataset and the target dataset based on the second path attribute.

14. A double image is then performed through these two newly created relations to derive the requested set of entity numbers.

Using this algorithm, only the hash table need be searched to create the needed relations.

This algorithm would be used in the following fashion to solve the problem:

display city in parking tickets for city ¬= home_city
in parking_tickets and registration = "27A523"

1. An attempt to create a dyadic relations results in no attributes matching from parking tickets to residences.

2. The attributes being related, "city" and "home_city" are placed in the two analytic lists as seen in figure 6.2b.

Figure 6.2b

Lists Before any Match is Found

Analytic Lists

List for Dataset Three		list for Dataset One	
attrname	dset ID	attrname	dset ID
city	3	home_city	1

"City" goes into the initial analytic list and "home_city" goes into the target analytic list.

3. No relations exist in the database.

4. The hash table appearing in figure 6.2 is searched for another attribute whose dataset ID is the same as "city". The first attribute encountered other than the initial attribute is "badge #". "Badge #" becomes the intermediate attribute.

5. The hash table is searched for another attribute with the same name as "badge #". This is found nowhere else in the table, so from the rule in step six, another attribute is chosen.

4. The new attribute which becomes the intermediate attribute is "registration #".

5. The hash table is searched for another attribute with the same name as "registration #". This is found for dataset two so "registration #" becomes the matching attribute. (An alternate form to this step is to have the user suggest attributes to become the matching attributes if he wished; thus allowing the user to lead the system if he liked.)

6. A dataset ID of two does not appear previously in the first analytic list; the user is queried if "registration #" should be added to the analytic list. Receiving an affirmative reply, "registration #" is entered into the first analytic list as seen in figure 6.2c.

Figure 6.2c

One Match Found for First List

Analytic Lists

List for Dataset Three		list for Dataset One	
attrname	dset ID	attrname	dset ID
city	3	home_city	1
reg #	2		

10. Steps three through seven are performed for the second analytic list.

4. The hash table in figure 6.2 is searched for an attribute whose dataset ID is the same as "home_city". The first attribute encountered is "name".

6. A dataset ID of two does not appear previously in the second analytic list; the user is queried if "name" should be added to the target analytic list. Receiving an affirmative reply, "name" is entered into the second analytic list as seen in figure 6.2.

11. One match has been found for each the initial and the target attributes. A comparison should be performed.

12. The target list is searched for an occurrence of dataset two appearing in the initial list. Since a dataset ID of two appears in the first list for the attribute, "registration #", and in the second list for the attribute, "name", a path has been found between the initial and target attributes and dataset two becomes the intermediate dataset.

13. Two relations are created: one between dataset three and dataset two based on the attribute, "registration #", and one between dataset two and dataset one based on the attribute, "name". This process was shown on page 63.

14. A double image is then performed through these two newly created relations to derive the requested set of entity numbers as was shown on page 63. The mapping process was illustrated in figure 5.3.

Once triadic relations have been achieved, it seems only one more logical step to get to quadratic relations. For this, the four datasets appearing in figure 6.3 will be used.

Figure 6.3

Four Interrelated Datasets

dataset one (residences)

attributes: name, social security #, street, home_city

dataset two (drivers registration)

attributes: name, social security #, registration #

dataset three (parking tickets)

attributes: city, badge #, registration #

dataset four (policemen)

attributes: badge #, # arrests, officer

These datasets generate the hash table appearing in figure 6.4.

Figure 6.4

Hash Table for Datasets One, Two, Three, and Four

Hash Table

Index	attribute name	dataset ID
1	name	1
2	ss #	1
3	street	1
4	home_city	1
5	name	2
6	ss #	2
7	reg #	2
8	city	3
9	badge #	3
10	reg #	3
11	badge #	4
12	# arrests	4
13	officer	4

Our user wishes to know if officer O'Reilly has been giving parking tickets to the highly prominent citizens living on Beverly Street. To do this, he types:

```
display street in residences for officer in policemen =
"O'Reilly"
```

The subset analyzer quickly announces failure and passes the subset to the relation-creator who follows the rules for dyadic and triadic relation creation and reaches the point displayed in figure 6.5, ready to announce failure.

Figure 6.5

Linking Dataset One to Dataset Four

Analytic Lists

list for dataset one (initial)		list for dataset four (target)	
attr name	dset ID	attr name	dset ID
street	1	officer	4
name	2	badge #	3

No more unique datasets can be added to these lists, and no match between the attributes' dataset ID's in the two lists can be found.

To continue from here, the two analytic lists must be imagined as list stacks. At each level we achieve a failure, we stack another level on top consisting of a number of initial and target attributes, until step seven is achieved for both lists at which point we stack yet another level. Using each possible pair of attributes in the two lists other than the initial pair as new new initial and target attributes, the process continues.

Figure 6.6

Multiple Combinations of Initial-Target Pairs

Analytic Lists

list for dataset one		list for dataset two	
attr name	dset ID	attr name	dset ID
attr 1	1	attr 2	2
attr 3	3	attr 4	4
attr 5	5	attr 6	6

Analytic Lists

list for dataset three		list for dataset four	
attr name	dset ID	attr name	dset ID
attr 3	3	attr 4	4

Analytic Lists

list for dataset five		list for dataset six	
attr 5	5	attr 6	6

For instance, if as in figure 6.6, there are two lists consisting of two elements each; four possible "initial-target" pairs could be used to form new initial and target attributes and the algorithm presented previously for triadic relations would continue at the next level with the new pair of attributes until a failure was reached. At this point, the second combination would be tried until failure, then the third, etc.. Once all the pairs had failed, the first failure would be used to extract new initial and target attributes and another level would begin. This trial and error approach is similar to that used by the PLANNER and CONNIVER projects at MIT; although it is not as sophisticated and, hopefully, not as expensive in computer time as these higher-level "learning" languages.

As higher levels are reached, this method becomes geometrically costly. However, in the normal case, probably only one new pair would ever get beyond step seven, and the process could be quite orderly, with only one pair being eligible for recursively applying the algorithm at the end of each level. Future research might, however, devote itself to a linear algebra type of solution which might be more efficient for the case of many attribute pair candidates at each given level.

However, in this case the solution is quite simple because only one pair exists at the next level and this single pair is the only candidate for forming new "initial" and "target" attributes and restarting the algorithm. At the second level, in figure 6.7, only one, new, unique match is found because there are only four datasets in all.

Figure 6.7

Successful Muti-Level Link from
Dataset One to Dataset Four

Analytic Lists

list for dataset one		list for dataset two	
attr name	dset ID	attr name	dset ID
street	1	officer	4
name	2	badge #	3

Second Level ....

name	2	badge #	3
reg #	3		

This condition would have been considered at the previous level to be the condition in step seven; a failure under the triadic algorithm. However, the previous level assumed that a dyadic relation had already been attempted and would not look for a dyadic relation at the first level. At the second level, this condition represents a possibility for a simple dyadic relation between the two datasets. As it turns out, step eleven reveals a path between the target attribute's dataset and the first match in the initial attribute's list. This path is created by making datasets two and three equivocal after detecting and creating the dyadic relation at the second level. Since datasets two and three are equivocal from the first relation at the second level, the dataset ID for "name" matches the dataset ID for "badge #" at the first level, and a success is achieved. Consequently at the first level, a relation has been created which will define for us on what steets people live to whom officer O'Reilly has been giving parking tickets.

This chapter has shown how simply employing the information in the hash table can be used to solve simple network-type relational problems in this revised version of Janus. This would become too cumbersome in the case of many datasets with over roughly four levels of relational indirection, but in the simple case, it is a rather quick, effective solution to the problem of creating automatic relations for the user.

Chapter Seven

Sample Session Using Revised Janus

The following projected interaction between the user and the revised Janus has been prepared to show the system in action implementing the automatic relation-defining capabilities outlined in chapters five and six. As each capability is used, a note is made of which proposed modification is being used. The datasets appearing in table 7.1 will be used for this session.

Figure 7.1

A Database Consisting of Four Datasets

dataset one (residences)

attributes: name, social security #, address, domicile

dataset two (drivers registration)

attributes: driver, social security #, registration #

dataset three (parking tickets)

attributes: city, badge #, licence plate

dataset four (policemen)

attributes: badge #, # arrests, officer

Since there are certain attributes whose names do not reflect their direct relation to attributes in other datasets, the user first defines a series of synonyms to indicate the equivalence of one attribute to another in another dataset:

```
addname street for address in residences
addname city for domicile in residences
addname name for driver in drivers_registration
addname reg # for lic plate in parking tickets
addname name for officer in policemen
```

These statements add the first attribute name as a synonym to the second in the specified dataset. Consequently, any reference to that synonym would be equivalent to referencing the real attribute. This series of addnames would result in the hash table appearing in figure 7.2.

Figure 7.2

Hash Table for Revised Datasets
One, Two, Three, and Four

index	attr name	dset ID	attr ID	dset index
1	address	1		
2	badge #	3		
3	badge #	4		
4	city	1	777	6
5	city	3		
6	domicile	1		
7	driver	2		
8	lic plate	3		
9	name	1		
10	name	2	777	7
11	name	4	777	12
12	officer	4		
13	reg #	2		
14	reg #	3	777	8
15	ss #	1		
16	ss #	2		
17	street	1	777	1
18	# arrests	4		

note: Unnecessary elements in hash table have been omitted. Attributes with Attribute ID's of "777" are synonyms whose actual attributes' indices are indicated by the dataset index.

Elements unimportant for the purposes of this analysis have been omitted to improve the ease of understanding the table.

Once the synonyms have been added, it is possible for the relation-definer to create all relations necessary for obtaining needed subsets since attributes which have the same types of values in different datasets are cross-referenced by synonyms of the same name.

The first request the user wishes is for the street on which a driver lives whose registration number is "27A513". To do this the user types:

> display street, for reg # in drivers registration = "27A513"

The subset analyzer ascertains that "street" is not in the "drivers registration" dataset. Upon ascertaining that no relation exists between "residences" and "drivers registration" it queries the user whether a relation should be attempted:

> display: no relation exists between residences and drivers registration, would you like to create one (yes or no):

Receiving an affirmative reply, it passes the problem to the relation-definer who attempts a dyadic relation. "Street" quickly is recognized as a synonym for "address" which resides in dataset one. Since "drivers registration" is dataset two and the attribute, "name" is found to exist in both datasets one and two, name is spotted as a relation possibility and the user is queried:

> Would you like to use the attribute, "name", to form a relation between residences and drivers registration.

Thinking a minute, the user decides that a person's name is a non-unique identifier and suspects a better relaion can be found. He therefore responds negatively and another attribute is sought. Before searching long, the relation-definer finds the attribute, "social security #" which is common to both datasets and once again queries the user:

```
Would you like to use the attribute, "ss #" to create a
relation between residences and drivers registration
(yes or no)?
```

The user quickly acknowledges the uniqueness of a person's social security number and sports an affirmative reply. The system then proceeds to create the relation, and using the process described in figure 5.3, derives the set of entity numbers in residences whose social security numbers match the social security numbers in drivers registration for registration = "27A513". The system then types the following output:

```
#       street
2       629 Lincoln
```

The sign, "#", is the entity number in residences of the attribute satisfying the requested condition. This entity number is typed whether or not it is requested. It provides an easy handle by which to reference the entity uniquely later.

The user now wishes a list of the home addresses of all people who have received parking tickets from officer O'Reilly. The user types:

```
display street in residences for name in policemen
 = "O'Reilly"
```

The subset analyzer balks and queries the user:

```
display: no relation exists between residences and
policemen, would you like to create a relation (yes or
no)?
```

The user responds affirmatively and the relation-definer attempts a dyadic relation. The attribute, "name", (a synonym in the dataset, "policemen") is found to be common to both datasets and the user is once again queried:

```
Would you like to use the attribute, "name", to create
a relation between policemen and residences (yes or
no)?
```

The user ponders this a moment and concludes that this would only give him O'Reilly's address which is not what he wants, so he responds negatively. The system then searches for another possible match, and, finding none, declares the relation attempt defunct:

```
No dyadicc relation can be found between policemen and
residences, would you like to attempt a triadic
relation (yes or no)?
```

The user feels the necessity to continue since there must be a relation somewhere to handle the problem, so he responds affirmatively. The system begins two analytic lists immediately adding the relation from dataset one to dataset two to the first analytic list arriving at the two lists in figure 7.3.

Figure 7.3

Analytic Lists with One Relation Added

Analytic Lists			
list for dataset four		list for dataset three	
attr name	dset ID	attr name	dset ID
street	1	name	4
ss #	2		

It then follows algorithm a little way and discovers that the attribute, "name", appers in both dataset two and dataset four so it quesries the user:

> Would you like to use the attribute, "name" to create a relation between policemen and drivers registration (yes or no)?

The user realizes once again that this merely gives him O'Reilly's registration which is not what he wants, so he answers negatively. The algorithm continues for the first list and finds, "registration number", which is common to datasets two and three, so it queries the user:

> Would you like to use the attribute, "registration number" to create a relation between drivers registration and parking tickets (yes or no)?

Realizing that this is the needed link between parking tickets and a person's registration, the user responds affirmatively. This results in the attribute, "reg #" being added to the first analytic list producing the table appearing in figure 7.4.

Figure 7.4

Analytic lists with One Match

Analytic Lists

list for dataset four		list for dataset three	
attr name	dset ID	attr name	dset ID
street	1	name	4
ss #	2		
reg #	3		

The algorithm then proceeds to find a match for the second list and discovers rather quickly the attribute, "badge #", common to datasets four and three, so it queries the user:

```
Would you like to create a relation between policemen
and parking tickets using the attribute, "badge #",(yes
or no)?
```

The user realizes that this forms the needed match between policemen and the parking tickets they have written so he responds affirmatively. This results in the algorithm adding the attribute, "badge #" to the second analytic list producing the table appearing in figure 7.5.

Figure 7.5

Analytic lists with a Solution

Analytic Lists

list for dataset four		list for dataset three	
attr name	dset ID	attr name	dset ID
street	1	name	4
ss #	2		
reg #	3	badge #	3

At this point, the algorithm spots the fact that dataset three appears in both lists, creates the two relations, and performs the triple image to find the streets on which people live to whom officer O'Reilly gives parking tickets. The following output is produced:

```
#       street
1       629 Lincoln
```

The user is relieved to discover that officer O'Reilly has not given tickets to the organization's people on Fairfax street.

In a brief set of two examples, this chapter has shown how the revised system and the algorithms proposed therein may be used to solve a set of simple problems. Although these exercises have not been a comprehensive display of the system's power, it is felt that they have presented an overview of what could be done with an automatic relation searcher and definer.

Chapter Eight

Revised Model for an Information System

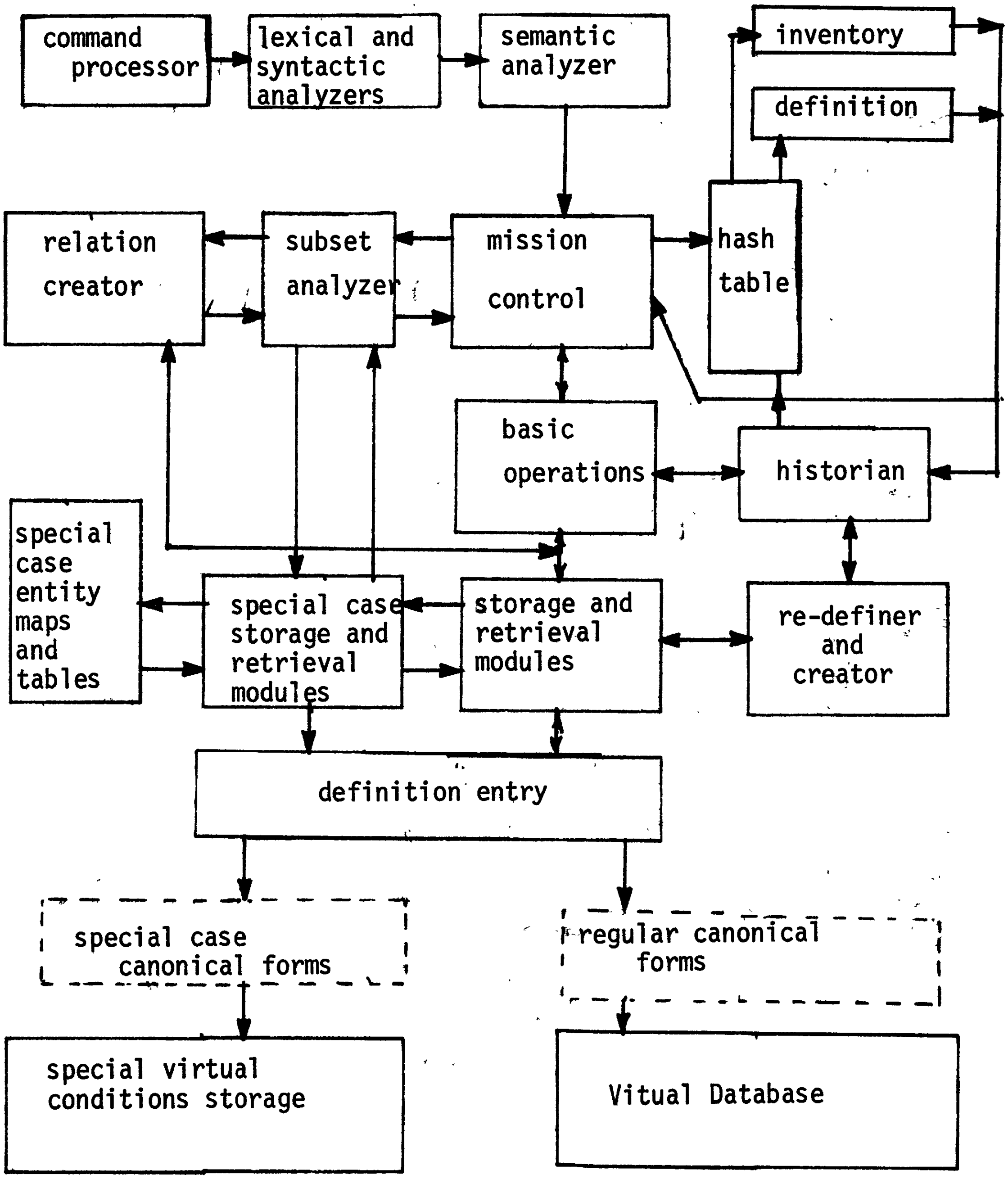
In figure 8.1 is a revised model of Janus which incorporates the ideas presented in chapters four and five. The major changes between this and the primary model of Janus are in the special storage and retrieval modules, the new definition facility, the expanded historian, the relation creator, and a slightly modified hash table which performs the synonym capability suggested in chapter five.

The special storage and retrieval modules handle the set records, sorted and hashed attribute value records and the storage and retrieval forms of these new internal storage strategies. This includes the special entity maps for sorted attributes, the tree structures for full and partial set records, and the supplementary hash tables (instead of entity maps) for the attributes which might be hashed .

The new definition facility provides the capability for the historian's information to be used at the end of a session to re-define and create attributes under one of the special storage forms outlined in chapter four. This "re-definer" would function similarly to the relation-creator attribute mapper at the end of a session,

FIGURE 8.1

Revised Model of Janus



deciding what capability was needed to produce faster, more efficient subset retrieval based on accumulated information from the historian.

The relation-creator is drawn into play at two points in the execution of the Janus system. One occurs at the end of the session when the user exits from the system. Here, the relation creator discovers which datasets have newly created var-del attributes by checking the definition and inventory datasets. Discovering new var-del attributes, it expands the datasets on the var-del attributes creating one new dataset for each var-del attribute. This function could conceivably be invoked just before a relation-creation, thus limiting the number of expanded datasets in the database. The second point at which the relation-creator is invoked is when a condition expression contains attributes which do not belong to the default dataset. The subset-analyser passes the relation-creator the attributes to be used to create new relations.

The final modification in the hash table is simple but effective. Each synonym is stored in the table with the dataset_id, attribute_id, and entry number of its major attribute. In each definition entry is an offset to an area containing the entry numbers of the synonyms in the hash table as shown previously in figure 5.9. Since most references will go from synonym to attribute name, this form of storage will be fast, effective, and efficient.

This, in a nutshell, summarizes the changes to be made to the Janus model to achieve capabilities of a more sophisticated, advanced information system. However, what is also important is what this means conceptually, in terms of a generalized information system model.

In the simplest case, we began with an information system consisting of a user interface, basic operations (alternately accessible from a programming language), storage and retrieval modules, and canonical forms leading to a virtual database as in figure 8.2. This model was expanded to specify components of the user interface: command processor, lexical and syntactic phases, semantic phases, and a mission control module. System descriptive datasets were specified as being used by basic operations and storage and retrieval modules to access the virtual database. A simple historian and the subset analyser were added to arrive at the present model of Janus in figure 8.3. Finally, the special retrieval modules, active historian and two "dummy users" (the relation-creator and new definition facility) were added to create the revised model of Janus appearing in figure 8.1.

In terms of unit modules, figure 8.1 seems to break down into the basic block diagram in figure 8.4. In this structure, there are regular and special storage and

FIGURE 8.2

A Basic Information System

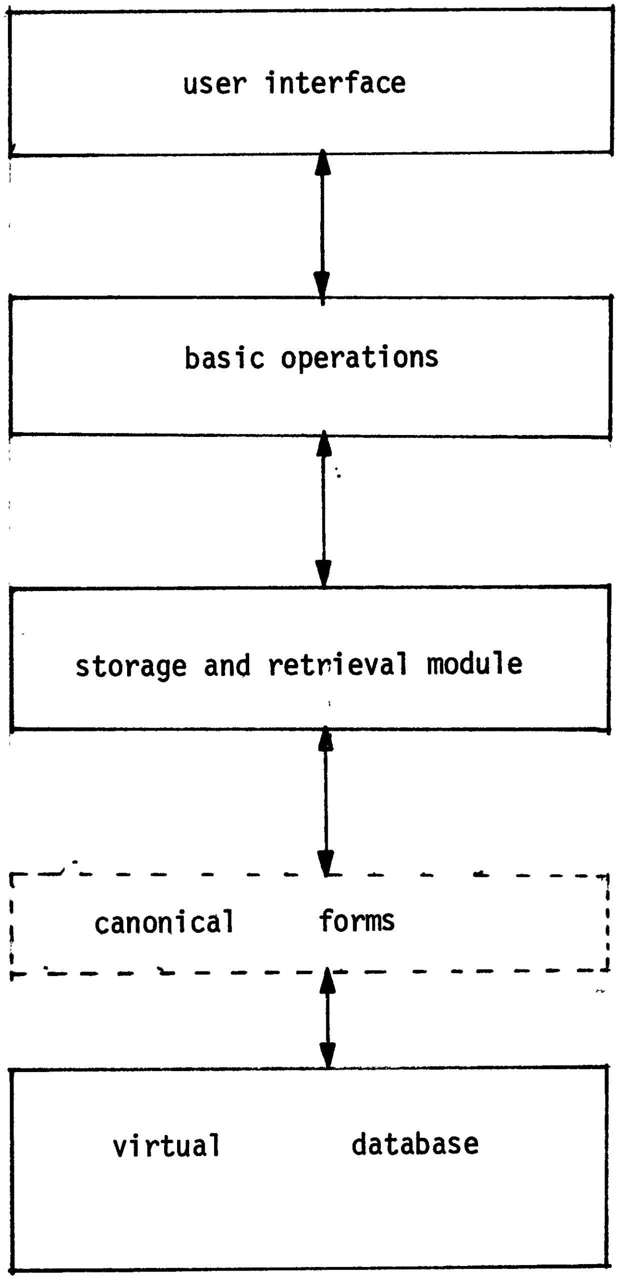


FIGURE 8.3

Model of Janus

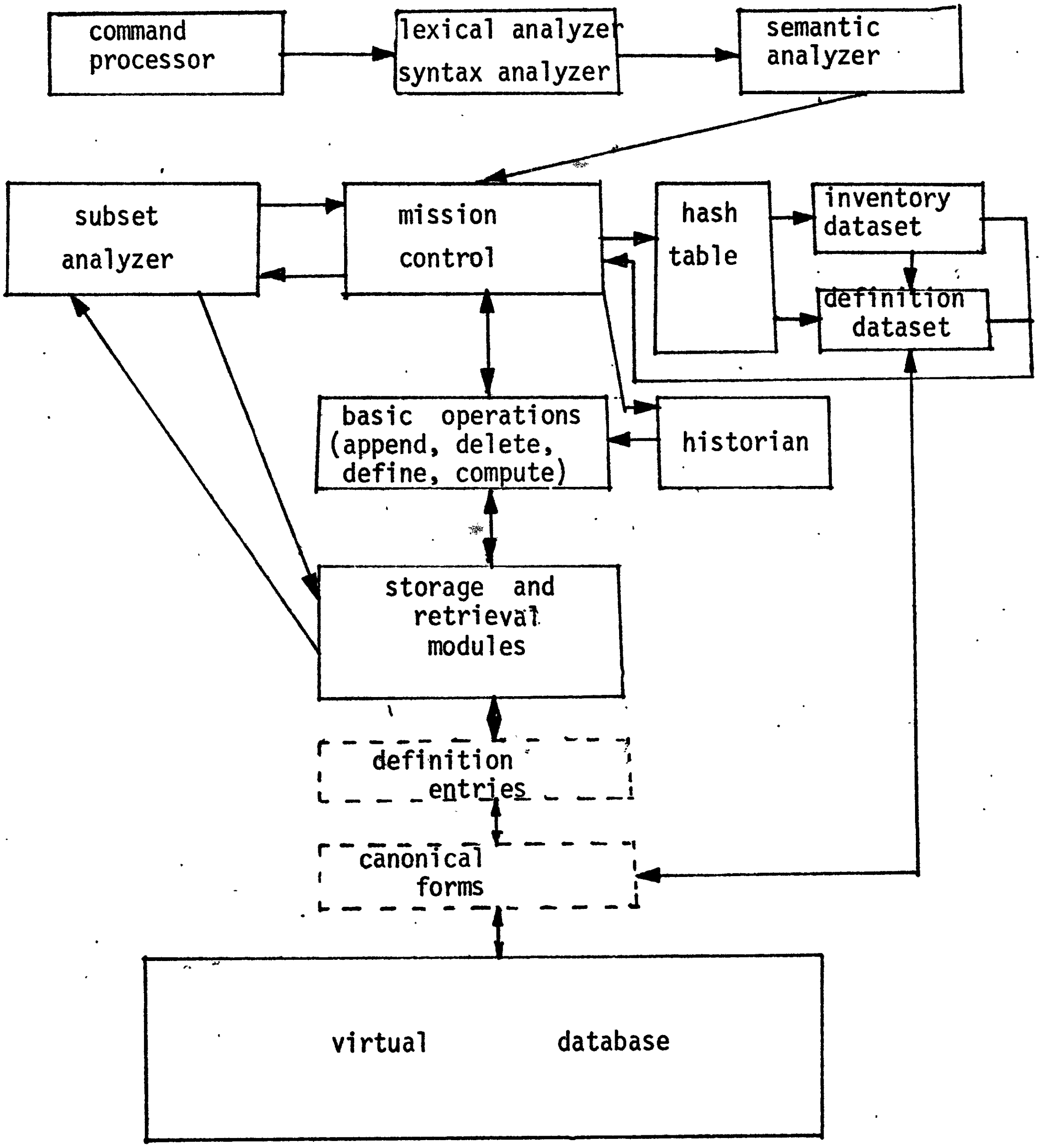
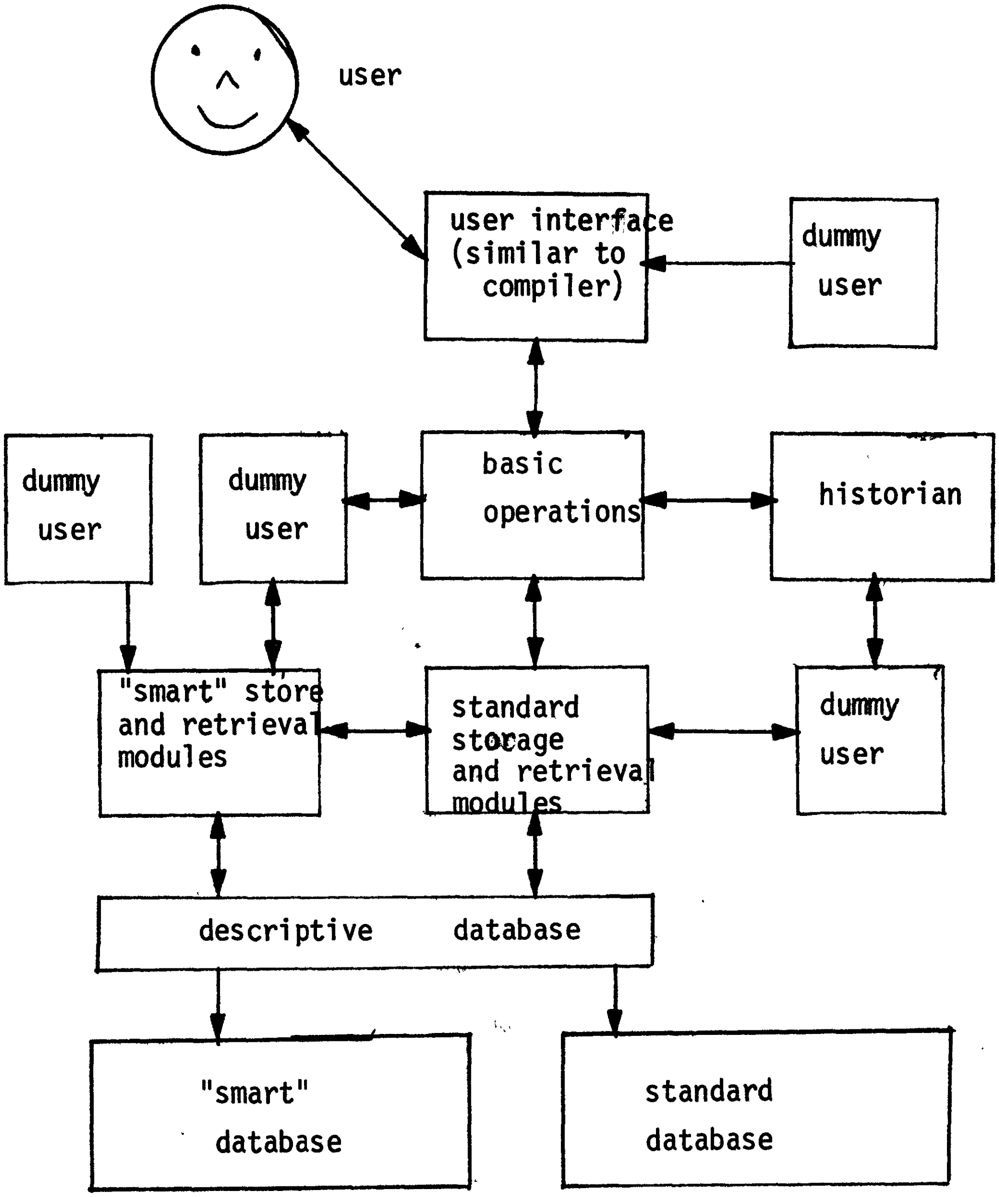


FIGURE 8.4

Block Diagram of a Generalized Information System



retrieval modules and databases and a descriptive dataset. There is a historian and basic operations. There is also a user interface which is rather similar to a compiler in its functons. Surrounding all of this there is finally a number of "dummy users" who perform all types of extra functions for the user when he does not perform them himself. These are types of "helpers" or aids to more efficient work. These "helpers" add characteristics of an intelligent, powerful information system to Janus.

The first "dummy user" could be an automatic "help" facility to assist the user when he encountered trouble in an inputted command line. It would take the flagged output from the syntax analyzer and ascertain whether it meritted instruction on the use of the command to the user, minor patching to create a syntactically correct command, or complete expressed confusion on the part of the command processor.

Dummy user two is the subset module described in the revised Janus model. His function is to determine if a subset exists in the dataset. if it does not, he calls dummy user three, the relation-creator in the revised Janus model whose task is to build any relations which are necessary to deliver the needed subset.

Dummy user four is the redefiner who creates new data storage formats if the historian determines that cheaper storage forms are more appropriate. This dummy user would then create the new storage form as requested by the historian checking the user to make sure it was clear to proceed.

Dummy users two and four were described in chapter four, and dummy user three was described in chapter five.

Conclusion

This paper has presented a number of alternative solutions to the information management challenge, throwing out solutions which were too costly and retaining solutions which solved with some reasonable level of efficiency and effectiveness typical query problems. Any conceptual modification proposed in this paper could become a thesis in itself once assumptions regarding the hardware, the software, the overall environment in which the system resides, and the size and nature of the database being manipulated have been fixed. A few assumptions were made about using a pointer-capable language such as PL/1 and a paged environment, which may be realistic given current computing trends; but the rest of the paper had to deal with database size, data types, and query problems as unknown variables.

I must conclude that any system which offers extensive power and flexibility to handle any type of ad-hoc inquiry will invoke a serious overhead in intelligent but expensive "dummy users" as outlined in this paper. Perhaps for this reason, IBM, with IMS, has demanded that a great deal be known about what types of questions will be asked before the information is stored and structured internally; thereby offering at least semi-optimal effectiveness and efficiency in user requests.

However, the proposals in this paper do offer a great deal of flexibility and power for a minimal cost and will deal with a major subset of information needs. With minor modifications they would also fit neatly into the Janus relational data handling system. Therefore, although I do not contend that these proposals are panaceas to the information problem, I do believe they go a long way towards producing the informational capability demanded of modern computing systems.

Bibliography

1. Codd, E.F., "A Relational Model of Data for Large Shared Data Banks", Communications of the ACM, vol. 13, #6, June, 1970, pp. 377-387.

2. Dodd, C.G., "Elements of Data Management Systems", Computing Surveys, vol. 1, #2, June, 1969, pp.117-132.

3. Feature Analysis of Generalized Database Management Systems, CODASYL Systems Committee Technical Report, ACM, New York, May, 1971.

4. A Survey of Generalized Database Management Systems, CODASYL Systems Committee Technical Report, ACM, New York, April, 1969.

5. User's Manual for the Janus Prototype System, Cambridge Project, Cambridge, Mass., (to be published).

6. Stamen, J.P. and Wallace, R.M., "Janus: A Data Management and Analysis System for the Behavioral Sciences", Cambridge Project, Cambridge, Mass., (to be published).

7. Kessler, A.R. and Stamen, J.P., "Penelope -- Design for an Information Management System for Categorized Dyadic Relations", Center for International Studies, MIT, Cambridge, Mass., May, 1968.