A Knowledge-based Preprocessor for Approximate Joins in Improperly Designed Transaction Databases

by

Lloyd Brodsky
B.A., Economics, Vassar College
(1976)

Submitted to the Sloan School of Management
in Partial Fulfillment of
the Requirements of the Degree of
Doctor of Philosophy in Management

at the

Massachusetts Institute of Technology

September 1990

Signature of Author _____
Sloan School of Management
June 19, 1990

Certified by _____
Stuart Madnick
Professor of Management
Thesis Supervisor

Accepted by _____
James Orlin
Chairman, Phd Committee

Abstract

Retrieving data from multiple databases requires keys; identically defined attributes which enable connecting the databases. Databases developed independently of each other do not necessarily contain the needed keys.

The contents of these multiple databases frequently reflect different aspects of a common entity. For example, the contents of a hospital bill database and a physician bill database both are related to particular episodes of illness of particular patients. In the absence of a key, knowledge of the implicit connection may be used to form an approximate compound key from existing attributes. For example, a surgical operation must be done during a corresponding hospital stay. A series of such comparisons taken together approximate the effect of an exact key.

I propose that such approximate keys can be constructed by finding pairs of comparable attributes and comparing them. Such pairs can be found by dividing the existing databases into sets of application-defined universal attribute types and taking a cross-product of the database dictionary information. Pairs of attributes of the same attribute type but in different databases can potentially be compared. I define tests of consistency when the attributes to be compared are not directly comparable.

The relevant comparisons are defined by end-users defining the attribute types which are relevant to identifying the missing entity.

To test this theory, I built software which solicits data analysts for universal attribute type membership for attributes and which solicits end users for relevant attribute type. The system then:

- produces comparison pairs by cross-tabbing the data dictionary (done once per database pair)
- selects relevant comparison pairs (done once per query class)
- infers how to compare the attributes in each of the comparison pairs selected and instantiates an implementing SQL clause (done once per relevant comparison pair)
- assembles the subtemplates into a template reusable for a class of similar queries and updates a menu of available query classes
- transforms a query class template into an executable SQL query by soliciting needed attribute values

Three test queries are generated using the software I wrote and the results are analyzed.

Thesis Supervisor: Stuart Madnick

Title: John Norris Maguire Professor of Management Science

Table of Contents

Chapter 1: Introduction 4

Chapter 2: Literature Review 13
2.1 Overview 13
2.2 Schema Integration 18
2.3 Business Planning 22
2.4 Semantic Databases 22
2.5 Object-oriented databases 24
2.6 Logical Databases 26
2.7 Distributed databases 27
2.8 Federated databases 28
2.9 Incomplete Information 29
2.10 Conceptual Cluster Analysis 31
2.11 Causal models 32
2.12 Intersection of Databases and AI 34
2.13 Natural language interfaces 36
2.14 Fuzzy set theory 37
2.15 Summary 37

Chapter 3: A Theory of Inferring Incomplete Joins . . . 38
3.1 Introduction 38
3.2 Discovering Keys Under Complete Information . . 48
3.3 Testing for Consistency under Incomplete Information 57
3.4 Discovering Pairs of Attributes Worth Comparing 70
3.5 Logical Confidence Intervals Under Incomplete Information 84
3.6 Selecting a Comparison Method When Attributes Are Defined Differently 92
3.7 Probability of Accurate Retrieval 95
3.8 Storing and Reusing Retrofit Knowledge 99

Chapter 4: Implementation 105
4.1 Overview 105
4.2 Database Knowledge Acquisition 112
4.3 Subject Knowledge Acquisition 121
4.4 Query Class Template Creation 129
4.5 Transforming Query Class Templates into Executable Queries 146

Chapter 5: Sample Retrievals 154
5.1 Overview 154
5.2 MIT Blue Cross Database Description 155
5.3 Example I: Hysterectomy 156
5.4 Example II: Osteoarthritis 172
5.5 Example III: Hurricanes 182

Chapter 6: Conclusions 197
6.1 Summary 197
6.2 Practical results 198
6.3 Research Contributions 200
6.4 Future Research 203

Citations 205

Chapter 1: Introduction

Retrieving data from more than one database requires keys which connect pairs of databases. Keys are commonly defined attributes in both databases which permit identifying the records of interest. Unfortunately, a common problem is that databases developed independently of each other do not necessarily contain the needed keys.

Frequently, these multiple databases reflect different aspects of a common entity. For example, the contents of a hospital bill database and a physician bill database both reflect the diseases of patients. To query both databases one would like to have a key identifying particular episodes of medical care for particular patients.

In the absence of a key, knowledge of the implicit connection may be used to form an approximate compound key from the existing attributes. For example, if an operation is done during a hospital stay, then the policy number of the person being operated on will be the same as the policy number of the person admitted to the hospital. A series of such comparisons taken together approximates the effect of a key.

This approach requires analyzing two problems. First, the system needs to know which attributes from each database to compare. Second, the system needs to know how to compare those attributes.

To solve the first problem one needs to view databases as being built to serve the data needs of particular applications,

not for the mere joy of storing data. The type of application partially defines what must be in the database. For example, one would expect all accounting systems to have attributes that correspond to what was sold, who bought it, who sold it, when it was sold, what the price was, and possibly where it was sold. We may not know how those attributes are represented or named, but we know that attributes representing those types of data have to be present. I call these types of necessary data "universal attribute types".

Given any two databases of the same application type one can find pairs of potentially comparable attributes of the same universal attribute type, but from different databases. These "comparison pairs" are the basis for reconstructing a key. Knowledge of which universal attribute types should be compared (who bought may be relevant; who sold may not be) is also needed to approximate a key.

The second problem is determining how to compare the two attributes in each pair. If both attributes were defined the same way one could simply check for equality, such as testing for equality of social security numbers in different databases. However, if the attributes to be compared are defined differently the best we can do is a test of what I'll call "consistency".

For two attributes that are not defined the same way but are of the same universal attribute type a knowledgeable individual (most likely the end user) may be able to specify consistent

sets of attribute values. For example, an attribute which records physician procedure cannot be directly compared to an attribute which records diagnosis. However, for many physician actions, only a subset of all possible diagnoses is consistent. For example, a hysterectomy is an inappropriate treatment for a broken leg, but is an appropriate treatment for uterine cancer.

This approach is practical when comparing attributes whose meanings are "common knowledge" to the end-users making the query because the end-users can tell the system what those consistent values <u>are</u>. Although passing these tests of consistency does not guarantee correct retrieval, the conditional probability that a given record will be correctly retrieved goes up as more of these tests of consistency are passed.

I also define additional tests of consistency that operate on "event instances", groups of records which belong to the same episode. Here I compare a list of what must be present in each instance of an event with what was actually retrieved. For example, one might check that there was a surgeon's bill, an anesthesiologist's bill, and a hospital bill for each group of retrieved records corresponding to an instance of surgery-requiring-hospitalization. I also define a order-consistency test comparing the contents of retrieved records with their required sequence -- for example, a check that the diagnosis (if there were one) happened before the surgery.

A series of such tests of consistency forms an approximation to a compound key. If the approximated key is not exact, we can form a "logical confidence interval" using incomplete information principles. I compute both an upper bound (might-be) answer using only absolutely accurate tests of consistency and a lower bound (must-be) answer adding additional tests on potentially inaccurate data.

Developing a procedure for discovering those tests and then processing those tests into a query executable in the local query language is the goal of this dissertation. I assume the attributes the user wants retrieved are present in the underlying databases and the "only" thing preventing access is the lack of keys linking the databases. My hypothesis is that we can retrofit an approximate retrieval from databases lacking common keys because:

- The key is really an inference on the raw data which define membership in an occurrence of an event. This inference could be done anytime if the appropriate data and knowledge were available.
- Attributes in each database serving a particular type of application can be divided into application-required "universal attribute types". I assume data analysts can identify those types and assign attributes to them.
- Different databases may be implicitly linked by having attributes of the same types. Potentially useful pairs of comparable attributes can be discovered by taking a

cross-product of the database-dictionary information and retaining those attribute pairs which are in different databases but of the same attribute type.

- I assume end-users are able to identify those types which are relevant to the query and to supply sets of attribute values needed for consistency tests if asked (such as identifying which diagnoses are consistent with a particular operation).
- The system can then select comparison methods and process those tests into an executable query in the target query language.

This dissertation contains the following chapters:

- Chapter 2: Literature review. After an overview (2.1) I will discuss the relevant work in schema integration (2.2), business planning (2.3), semantic databases (2.4), object-oriented databases (2.5), logical databases (2.6), distributed databases (2.7), federated databases (2.8), incomplete information (2.9), conceptual cluster analysis (2.10), causal models (2.11), the intersection of databases and artificial intelligence (2.12), natural language interfaces (2.13), fuzzy set theory (2.14), and a summary (2.15).
- Chapter 3: Develops a theory of discovering approximate compound keys. This will be broken down into the following sections:

- 3.1 Overview: which discusses alternative ways of looking at data that leads to designs which leave out the keys we need and introduces the rest of the chapter.
- 3.2 Discovering Keys Under Complete Information: the concept of "discovering" a compound key connecting two existing databases by discovering "comparison pairs" which would be a compound key if a way could be found to compare the attributes in the pairs.
- Section 3.3 Testing for Consistency under Incomplete Information: the means of testing particular pairs of records and instances of events
- 3.4 Discovering Pairs of Attributes Worth Comparing: the means of actually discovering what attributes ought to be compared.
- 3.5 Logical Confidence Intervals Under Incomplete Information moves on to discuss how to cope when some: but not all of the attributes needed to define a compound key are present. I define three comparison methods.
- 3.6 Selecting a Comparison Method When Attributes Are Defined Differently: I then discuss means of selecting comparison methods and carrying out those comparisons.

- 3.7 Probability of Accurate Retrieval: a probabilistic discussion of why the above theory is likely to produce acceptable results.
- 3.8 Storing and Reusing Retrofit Knowledge: discusses retaining inputs and intermediate work product to be reused in future queries.

o Chapter 4: Implementation
  - 4.1 Overview: A brief discussion of the system I developed to test this theory.
  - 4.2 Database Knowledge Acquisition: How analysts add database knowledge to the system.
  - 4.3 Subject Knowledge Acquisition: How end-users add subject knowledge to the system.
  - 4.4 Query Class Template Creation: Discussion of the creation and use of the query class templates created in the process of creating a query.
  - 4.5 Transforming Query Class Templates into Executable Queries: How the templates are used to produce executable queries.

o Chapter 5: Sample Retrievals

  This chapter contains:
  - 5.1 Overview: Information about the three example questions which will be analyzed in this chapter.
  - 5.2 MIT Blue Cross Database Description: This

section also discusses the contents of the MIT Blue Cross transactions log, a real database which is the database used by the first two sample questions.

- 5.3 Example I: Hysterectomy: A sample query where an insurance company sums doctor and hospital hysterectomy costs for 1985.
- 5.3 Example II: Osteoarthritis: A sample query where an insurance company sums doctor and hospital osteoarthritis costs for 1985.
- 5.4 Example III: Hurricanes: A sample query where a parent company consolidates insurance receipts by its Mexican and US subsidiaries for an August, 1985 hurricane.

o Chapter 6: Conclusions

This chapter contains:

- 6.1 Summary: a summary of the research actually done
- 6.2 Practical results: a discussion of the practical implications of the current research
- 6.3 Research contributions: a discussion of the connections between the current research and other research
- 6.4 Future research: A discussion of things which could be done to extend this line of research

Chapter 2: Literature Review

2.1 Overview

In sections 2.2 (Schema Integration) and 2.3 (Business Planning) I discuss the literature on the specific knowledge required to retrieve data.

Schema integration is the discipline of integrating local views of the data into a single global database schema -- a task necessary when the organization of the firm makes it difficult to naturally arrive at a unified schema or when the project is too big for any single designer. Schema integration may be seen as a subset of the business planning methodology which includes consulting styles and tools for analyzing data needs in the context of business requirements and developing system designs to meet those needs.

I'll then move on to semantic databases (2.4), object-oriented databases (2.5), and logical databases (2.6). These either introduce data models that show more complicated (and hopefully more meaningful) relationships among entities in the database and/or add inference mechanisms to the query language.

In section 2.7 I will discuss such distributed database techniques as Multibase and McLeod's federated databases (2.8).

An unavoidable practical issue arising in retrofitting access to existing databases is incomplete information (2.9), a branch of database theory dealing with the problem of retrievals when the value of attributes is not known precisely. I discuss this and the corresponding Dempster-Shafer theory of

evidence.

It is also useful to discuss the artificial intelligence literature on conceptual cluster analysis (2.10) and causal models (2.11). The former addresses the problem of defining clusters which summarize a database. The latter is important here because of the role causality plays in identifying relevant attributes.

I will also discuss briefly the relationship between relations, objects, and rules in a section on the intersection of databases and artificial intelligence (2.12).

In 2.13 I will discuss natural language interfaces and in 2.14 fuzzy set theory, two areas that might appear to be relevant, but which have not turned out to be very helpful. Finally, I summarize this literature review in 2.15.

These research areas are listed in Diagram I, Desired Features. In this diagram, I have labelled the X-axis with capabilities I have found to be important in retrofit and the Y-axis with the names of the various areas of research.

In Diagram I, I have attempted to break down the various disciplines into their ability to carry out the five main tasks my system will need to accomplish. These are:

- Find, the ability to discover links databases (e.g, discovering that doctor bills and hospital bills have dates in common).
- Store, storing those links in machine-readable form so that they can be used in later queries.

- Execute, using those links in retrieval.
- Assemble, assembling those links into a query executable in the local query language(s).
- Incomplete, coping with incompleteness (missing and inaccurate data) in the underlying databases.

Schema integration and business planning are used to discover entity links, but little machine support for these techniques has been developed. Since they have the goal of discovering what was needed, they get a yes in the discovery knowledge column. Since nothing machine-readable is produced, they get a no in the other columns.

Semantic, object-oriented, and logical databases have the ability to store complex relationships (yielding a yes under Store), but not to discover those relationships (no on Find). Semantic databases do not even have the inference engine needed to execute that knowledge -- it is up to the person writing the query to read and understand the schema (No on Execute for semantic databases). Semantic databases are designed to enable the enforcing of semantic integrity constraints at data entry, an ability not useful for my problem. Many object-oriented and all logical databases have an inference engine which could execute discovery knowledge, once that knowledge were discovered and entered; these get a yes under Execute.

Diagram I
Desired Features

	Find	Store	Execute	Assemble	Incomplete
Schema Integ & BPlan	yes	no	no	no	no
Semantic DBs	no	yes	no	no	no
Object-oriented DBs	no	yes	yes	no	no
Logical DBs	no	yes	yes	yes	no
Distributed DBs	no	yes	no	no	no
Federated DBs	no	yes	no	no	no
Incomplete Information	no	no	no	no	yes
Conceptual Clusters	yes	yes	yes	no	yes
Causal Models	no	yes	yes	no	no
DBs and AI	yes	yes	no	no	no
Natural language	no	no	no	no	no
Fuzzy Sets	no	no	no	no	no

In logical databases a means of assembling queries could conceivably be written in its language (usually PROLOG), but this is not a built-in function nor the focus of current research in logical databases. Giving the benefit of the doubt, logical DBs get a yes under Assemble.

Distributed databases and federated databases have no knowledge of how to discover links but have an ability to store the links in their schemas once they are discovered. Unfortunately, as in semantic databases, it is left to the end-user to understand their schemas. These methods get a yes on storing knowledge and a no on executing it (although federated databases do have a facility for permitting users to write pieces of code called retrieval scripts. Thesew scripts can be stored). Neither have any assembly or incompleteness facilities. Also, the semantic complexity of what can be stored in a schema is much more limited than the semantic, object-oriented, and logical databases.

Incomplete information databases have one relevant property -- the ability to return a "best results" answer when missing information prevents giving an absolutely accurate answer to the question. This line of research gets a yes in the Incomplete column and a no in the others.

Conceptual cluster analysis has the knowledge to discover clusters. The knowledge acquired in the clustering process is storable and executable. Further, conceptual cluster analysis can cope with incompleteness, at least in the sense that all

the information needed to make a perfect partition is not present.

Causal models do not have any discovery knowledge. It would be surprising if they did, given the different research goals. However, the mechanisms and concepts for storage and execution of knowledge are relevant to my problem.

Finally, fuzzy set theory and natural language theory do not have any direct relevance to my problem.

2.2 Schema Integration

My desire for a retrofit technology may be viewed as a desire for a method of automatic after-the-fact schema integration. However, my goal is to form an "approximate" integrated schema where data conflicts are handled by incomplete information techniques on a case-by-case basis. In traditional schema integration the goal is to form an "accurate" integrated schema with data conflicts blocked at entry time by integrity constraints.

In the schema integration literature, there has been a considerable amount of research on methodologies and support tools as discussed in a recent review article by Batini, Lenzerin, and Navathe [Batini86]. Twelve methods were identified in [Al-Fedaghi81], [Batini84], [Casanova83], [Dayal84], [ElMasi87], [Kahn79], [Mannino84], [Motro81], [Navathe82], [Teorey82], [Wiederhold79], and [Yao82].

All twelve approaches share some version of the following

actions:

Preintegration: collecting correspondences between component schemas in the form of constraints and assertions.

Comparison: identifying conflicts between the several schemas. Such conflicts include such problems as attributes with the same name that do not mean the same thing or with different names that do mean the same thing. Other problems include structural conflicts, different attributes used as keys, different patterns of functional dependency, and different insertion and deletion policies.

Conformance: resolving the conflicts. This generally involves the recognition of interschema properties, recognizing that the several schemas were incomplete representations of something. Discovering interschema properties requires sufficiently deep knowledge of the real world to recognize that something important is missing from the schema. Nobody has yet shown that their schema integration method is complete or determined how conflicting opinions should be resolved.

Merging and restructuring, actually creating the new integrated schema. Batini discusses the general goals of:

- Completeness, seeing that the additional constructs added are sufficient to identify the

interschema properties detected during the conforming step.

- Minimality, the elimination of redundancy in the schema.
- Understandability, the property that the method should pick the most easily understandable of the set of semantically equivalent schemas.

Of course, choosing the alternative schema which best satisfies these goals is something of a value judgement.

What does this research tell us that will be useful in retrofitting? Writing a schema requires detailed knowledge of the application. The application is, in turn, based on knowledge of the real-world situation the application serves. All implemented databases are incomplete in two important respects:

- Not all the attributes in the real world are included, just those in the model.
- Some of the attributes in the database are not truly atomic facts, but record judgments made in the real world (such as a diagnosis).

One might then view the contents of the schema as constrained by the application. This functional dependency on the semantics of the application is rarely recorded in the actual database schema.

The preferred output of schema integration is a machine-readable integrated schema. However, because researchers in

this area have not found an algorithm that will produce a provably complete integration or been able to fully specify the knowledge required to perform such an integration, much of schema integration requires the judgement of the database designer to resolve conflicts. Further, since all this work takes place before any software is written or data collected, answering queries is an abstract (albeit important) concern when doing this work.

In retrofit the goal is different. We still have the schemas of the individual databases. However, this time we also have actual data already entered into the databases those schemas support. The goal is to enable end users to answer queries not supported by those schemas by inferring which attributes in the existing schemas correspond to the missing interschema properties. For example, a query may require identifying specific individuals; in the absence of a known unique identifier the system might be able to substitute NAME and ADDRESS. Further, the database should produce an incomplete answer rather than refusing to answer at all.

The two processes also have different results. In schema integration, the result is an integrated schema with integrity constraints and a non-machine-readable list of conflicts. This result is created <u>before</u> system construction. In retrofit, we seek an executable query that will produce an approximate answer from <u>existing</u> databases.

Unfortunately, schema integration and business planning

methods do not make this "connectivity knowledge" available to the database management system. Rather, the database designer is supposed to build an integrated schema which permits the database to function appropriately once that schema is implemented.

2.3 Business Planning

The business planning literature recommends careful thinking about the firm's data resources as an integral part of its strategic planning and then rebuilding the firm's data systems in accordance with the plan, which may be seen as a superset of schema integration. Such research has not led to software which transforms business goals into provably correct software. Rather this line of research has resulted in process consulting methods which get people within organizations to talk about their data needs and document the results, which might be thought of as adding business strategic considerations to the process of schema integration. Two well-known methodologies are IBM's Business Systems Planning [IBM, 1981] and James Martin's strategic data planning method [Martin, 1982]. There are many other methodologies as well.

2.4 Semantic Databases

The issue of recording semantic connections is the area of semantic databases which Date [Date86] describes as making provision for the encoding of:

- Semantic concepts which appear useful.

- Symbolic objects to represent those concepts.
- Integrity rules related to those objects.
- Operators for manipulating the above.

In evaluating such extensions to currently available databases, one must adopt a particular context. The problem I address is retrieving groups of data objects from separate databases where the objects share a common cause.

The semantic data models are not designed to meet this goal. Rather, the underlying purpose is to include additional integrity constraints not possible without semantic extensions.

However, some of the results may be useful. For example it is conceivable that some level of support for universal attribute types could be had through use of the facilities for sub and supertype definitions of the kind supported in RM/T, Codd's proposed extension to the relation model for the purpose of adding semantic content to relational databases.

There are however, a number of reasons why adding semantic integrity constraints would not be the only or the best solution to the missing key problem. Although such models as RM/T [Codd79] and entity relationship [Chen76] support links that correspond to the unrecorded interschema properties, it may not be possible to determine those links at data entry.

2.5 Object-oriented databases

Most interesting and relevant to this project are the object-oriented databases. Researchers in object-oriented databases have tried to add facilities needed to address the support of complex data objects such as those needed for computer-aided design (CAD).

Of the object-oriented systems, the POSTGRES system under development by Stonebraker and Rowe [Stonebraker87] and the PROBE system of Dayal and the Computer Corporation of America [Dayal86] have the specific features that would be helpful to me. To support inferring relationships between the databases to be linked, one needs the ability to:

- store the universal attribute type memberships and other metadata.
- carry out a relational cross product to produce comparison pairs.
- infer comparison methods.
- write the query.

Existing databases can only handle these operations at a basic level. For example, in chapter 3 I discuss the concept that attributes can be divided into universal attribute types. If there were one set of mutually exclusive types, this could be represented easily enough in a relation. However, having an arbitrarily complex system of types and subtypes may be useful in some circumstances and would be quite difficult to implement

in a mainstream database. It would be easy to implement using PROBE because it supports arbitrarily complex types and inheritance.

If conditional type memberships were needed implementation would be straightforward with the alerter and trigger mechanisms in POSTGRES and traumatic in a mainstream database. While these object-oriented databases were not designed for retrofit, their facilities would make them useful in building a system for retrofit.

Finally, my implementation is limited by its inability to reconstruct multistage events -- it is limited to matching two relations. Reconstructing multistage events requires recursion because each stage depends on the products of the preceding stage. PROBE supports recursion.

Other non-CAD object-oriented approaches include Su's system [Su83], which uses an object-oriented approach to managing aggregated business data and the Wang & Madnick [Wang88a, Wang88b] work on building mechanisms for retrofit. Unfortunately, Wang & Madnick have not yet released a theory paper on their interesting work and it would be unfair to comment on a preliminary working paper which merely defines the problem.

2.6 Logical Databases

PROLOG's usefulness as an artificial intelligence (AI) development platform and the observation that the mathematical concept of a relation was central to both predicate and relational calculus led to the concept of logical databases. A logical database would have the inference facilities I need. This research typically involves enhancing PROLOG with various database facilities. Articles have been written which couple an existing database system with a logic programming system [Chang85], [Jarke85], or adding database facilities to a logic programming system [Chomicki83], [Lloyd83], [Naish83], [Sciore85]. Minker & Walker discuss an intensional database project. [Minker80].

Unfortunately, as Stonebraker has pointed out [Stonebraker87], most logic systems implicitly assume that database extracts will be sufficiently small to be loadable into RAM for inference. This assumption is inappropriate. In my context the extract remaining after tests have been passed is likely to be small. However, explicit control of the order in which tests were executed would be required to reduce the chance of exceeding the memory limit. Such explicit control is considered a kludge by the artificial intelligence community.

Finally, the partition data model is an intriguing alternative which is built upon the mathematics of lattices [Spyratos87]. It is concerned with databases that partition data into sets homogeneous along some semantic axis or axes.

For example, the set of highway vehicles might be partitioned into cars, buses, and trucks. Products and sums of partitions are defined in this model. It is possible to produce differing levels of abstraction through appropriate combinations of products and sums. The great appeal of this model is its relative simplicity and its compatibility with the much-revered relational model of data. Unfortunately, it has never been implemented as a functioning database system.

2.7 Distributed databases

The only technical research which creates the ability to answer queries from already-created multiple databases has been the research in such distributed database systems as the MULTIBASE project [Smith81]. MULTIBASE, however, has a very limited ability to cope with schema incompatibility between constituent databases, no ability to infer a connection, and no ability to handle incomplete data.

For example, MULTIBASE could handle unit conversions between databases (such as monthly salary vs. weekly salary) but not my consistency checks. The central contribution was to answer traditional complete database queries against a distributed environment with multiple DBMSs (ie, IMS, DB2 etc) where incomplete schemas and incomplete data were not a problem.

In contrast, I am not dealing with the "physical" problem of distributed data or differing data models; rather I am concerned with a group of schemas which record what we want

retrieved but lack properties needed to process the retrieval. MULTIBASE and other comparable systems typically assume one can produce an integrated and centrally maintained schema ahead of time.

2.8 Federated databases

McLeod's federated database system contemplates separate design of each database with export and import of data possible through "public" schemas available to all databases in the federation [McLeod85]. The approach is distinct from the mainstream MULTIBASE style of having a single central schema with all multi-database queries using the central schema.

In federated databases public data are stored in accordance with rules governing locally-maintained import and export schemas. Both data and metadata could be shared between these semi-autonomous databases by "negotiating" with each other. Towards that end McLeod defined a fairly large set of primitives which programmers could use for writing negotiation scripts.

While this research is certainly heading in the right direction, the current versions require an effort equivalent to writing a program for a "negotiation" to succeed. McLeod's conception of the complexity of negotiation led him to speak of negotiation scripts that would contemplate all possible negotiations and defined the semantics of a node in a negotiation graph as "a string representing a host-language

procedure to be executed". That quote suggests that end-users will not be able to write scripts unless they possess considerable programming skill.

Further, negotiation does not remove the need for each of the databases to be designed in accordance with the import and export rules, which may not be much easier than designing traditional databases.

2.9 Incomplete Information

Another issue in retrofitting access is the high likelihood of not being able to define a complete compound key using the attributes in the underlying databases, because some of the needed attributes are not in one or both of the databases or the attributes' definitions are incompatible. If corresponding attributes in different databases are not defined the same way, the best we may be able to determine is that their values are consistent. For example, if the value of LOCATION in a given record is "New England", is "Maine" in another, and is "Utah" in a third, we can at best confirm that the first two records might be from the same place and the third could not be -- given knowledge that LOCATION represents a geographic location (the database does not know what LOCATION means) and that "New England" is a set of six states which includes Maine and does not include Utah.

There is a formal theory of incomplete information which addresses the problem of retrieving information from databases

when the values of attributes are not uniquely defined. This work may be divided into two areas. Logically incomplete databases are those where the incompleteness is based on omitted variables. The key work in this area was done by Lipski. [Lipski79], [Lipski81], [Imielinski84]. In this line of research, the system determines what set of data objects must satisfy the query and what set might satisfy the query. For example, if we were sure that A = {1 2 3} contained everything that must be relevant and that B = {1 2 3 4 5} contained everything that might be relevant then A is the lower bound and B is the upper bound. If it made sense to add up the set members then the true value is 6 <= sum <= 15 although we cannot say anything about where in the interval the true value of the sum might lie.

Existing database theory may be seen as the special case where the lower and upper bounds are the same. This concept of logically incomplete information is heavily used in this dissertation.

Lipski's definition of upper and lower bounds is virtually the same as the analogous Dempster-Shafer theory of evidence [Zadeh86], although the former is a theory of how to generate database queries and the latter is a statistician's theory of evidence. (see [Prade84]) Related to Lipski's work is Codd's maybe calculus in which attributes take on one of the three values of relevant, irrelevant, or maybe instead of the countably infinite incompleteness of Lipski. A formal theory of

the maybe calculus was developed by Biskup [Biskup83].

Lipski's work is a major theoretical contribution, but computationally burdensome in practice. Fortunately most of the complexity in his work comes from avoiding double-counting in non-disjoint sets, a problem that does not occur in the transactions databases I am concerned with.

The other branch is the probabilistically incomplete information databases where a particular value is uncertain, but has a probability distribution. The usual assumption is that the values are unbiased estimators of a normal distribution and that they are normally distributed and independent. ([Wong82] and [Mendelsohn80])

The probabilistic literature is not relevant to the problem of inferring joins. My problem is missing keys, not uncertain values of existing keys. A probabilistic problem worth solving is when the value of an attribute has a probability π of being correct and a complementary probability of $1-\pi$ of null. Solving such discrete choice problems is a hot area in econometrics. However, the solution techniques are extremely complicated and are beyond the scope of this dissertation. (see Pindyck76).

2.10 Conceptual Cluster Analysis

Conceptual cluster analysis (CCA) is a branch of machine learning whose goal is to discover "concepts" which partition datasets into clusters of similar data objects by processing symbolic attributes as opposed to the statistical approach of

minimizing intra-cluster deviation around a numerical measure. For example, the concept:

([Color,{Blue, Red}],[Size,{large}],[Shape,{sphere,block}])

is a "generalization" of any set of objects that are blue or red in color, and are large in size, and have a block or sphere shape. Creating such a generalization requires much background knowledge to hypothesize candidate clusters and then search for a fit.

My problem might be viewed as conceptual cluster analysis in reverse. Rather than the cluster analysts' goal of discovering concepts which partition a dataset into semantically meaningful clusters my goal is to find a way to retrieve all data objects which are consistent with a user-supplied concept. Although a simpler problem, it remains non-trivial.

2.11 Causal models

Another relevant artificial intelligence area is the area of causal models. Particularly interesting in this context is the relationship of Davis' concept of adjacency to the concept of interschema properties.

Most of the causal models research has been in the area of simple physics and the diagnosis of faults in digital circuits. In the latter context, the goal is to take a description of the outputs of a device known not to be working and explain why the

device is not working.

These models have generally worked by describing a device or function through a set of constraints, where constraint in the AI context simply means a relationship between two things. Given that some faults were more likely than others, Davis [Davis84] simplified the process of finding errors by first checking "adjacent" connections, where adjacency means something comparable to "functionally dependent" in database jargon. In Davis' circuit example two devices might be physically adjacent (thermally connected), electrically adjacent (wired together), electromagnetically adjacent (unshielded), etc. Since different representations make different kinds of adjacency obvious, the task then becomes one of developing multiple representations in which relevant adjacencies are explicit. Davis developed two representations of the circuits -- a physical representation and an electronic representation. It was found to be easier to develop multiple representations from differing points of view rather than one all-encompassing representation. Further, the cause of the error was more likely to be found in the intersection of the two representations.

The process of discovering interschema properties has much in common with the process of developing one all-encompassing representation. In both circumstances one seeks to make explicit approximate connections in a complex environment where the relationships are not easy to articulate. Unfortunately,

since there is to be one (and only one) representation, all kinds of adjacencies must be loaded onto that representation. One might expect that the complexity of the resulting schema would grow exponentially as one overlays the representation with additional kinds of functional dependencies.

The advantages of the adjacency and multiple representation concepts that Davis applied to fault diagnosis are relevant here. In that original context, the goal is to take a device known to be broken and diagnose the cause. However, an important difference is that I am trying to solve a simpler problem. If one continues the metaphor of fault diagnosis, one could say that my problem is like trying to identify functioning devices from a storage closet where some work, some do not, and there is no simple test to determine which is which. This is a simpler problem.

In my context, one could store the adjacencies themselves, rather than a representation of the system which makes it easy to find the adjacencies. Further, although paths of causal interaction are necessary to finding the adjacencies, that knowledge of causal interaction is only required as an intermediary. Since the end users are assumed to understand the interactions inherent in their own operating environment, the system simply needs to ask about adjacencies, not infer their existence or their cause.

2.12 Intersection of Databases and AI

It is useful here to briefly discuss the intersection between database theory and artificial intelligence. A considerable amount has already been written about complementary differences between databases and artificial intelligence, an issue discussed in [Smith86]. There has been less discussion about shared concepts. For example, a production rule in an expert system looks much like an update in a database. An expert system rule has the appearance of:

```
IF <condition-list>
THEN < [attribute-list] is assigned  [value-list]>
```

which is functionally equivalent to:

```
UPDATE <relation-name>
SET <attribute-value pairs>
WHERE <condition-list>
```

The knowledge content is the same. Indeed, search might be said to be the fundamental process underlying both expert systems and database management systems. However, the search in query evaluation is usually much simpler than the search in inferencing. This translates into a significant difference in performance. Further, since databases frequently contain extremely large amounts of data, performance optimization in databases has received relatively more academic attention than in artificial intelligence.

The performance improvement comes from a loss in flexibility. In deductive inferencing any new knowledge can be added without affecting old knowledge. In databases new records must be consistent with the existing schema and its integrity constraints. One might reach the conclusion that knowledge

which will not change during the lifetime of the application ought to be represented in a database, with specific instantiations stored as records.

In addition, certain operations are easier to accomplish in one or the other model. For example, it is easier to take a cross product in databases and, as we will see, doing so is quite important in this work.

2.13 Natural language interfaces

I should have a word about techniques which one might think were relevant but are not. Natural language interfaces aim at translating database queries expressed in "natural language" (loosely defined as any question about the database in normal conversational English) into something the database can process. This line of research aims at reducing the amount of knowledge of the data query language required of users, since users already know their own language, but do not necessarily know a database query language. Such an effort is unquestionably useful, especially when the interface can cope with such higher level concepts as "the address of the highest-scoring student". However a natural language interface will not help if the underlying cause of the user's distress is missing keys or incomplete information.

2.14 Fuzzy set theory

Fuzzy set theory is a formal theory for computing the believability of conclusions [Negoita85]. It is another theory not relevant to this work. The goal is to develop formal methods of drawing fuzzy conclusions and getting away from the spurious accuracy of computers. For example, in applying the rule that IF height >= 6' THEN person is tall, somebody who is 6' is tall while someone who is 5' 11.999" is not-tall. There is no difficulty in retrieving the record with the man's height, nor any question as to the value of the height attribute in the record. The fuzziness is in the meaning of that value. Fuzzy set theory is an important line of research but is not relevant to incomplete information, where we lack information on what to retrieve and are not unable to decide on the meaning of data that was found with certainty.

2.15 Summary

In sum, all pieces of the problem have been addressed by one or more of the research areas, but the pieces have not been assembled to form a complete solution to the retrofit problem. The schema integrators have studied what needs to be known, but have no machine support for acquiring, storing, or executing the knowledge acquired; the various semantic databases and the artificial intelligence community are in the reverse position.

Chapter 3: A Theory of Inferring Incomplete Joins

3.1 Introduction

All database designs are models of the world they attempt to serve. The question is, why do so many of these models lack the keys needed to what frequently appear to be utterly reasonable questions. In order to discuss the cause of the missing key problem, it is helpful to discuss the different points of view that need to be compromised in order to define what is actually in the underlying databases.

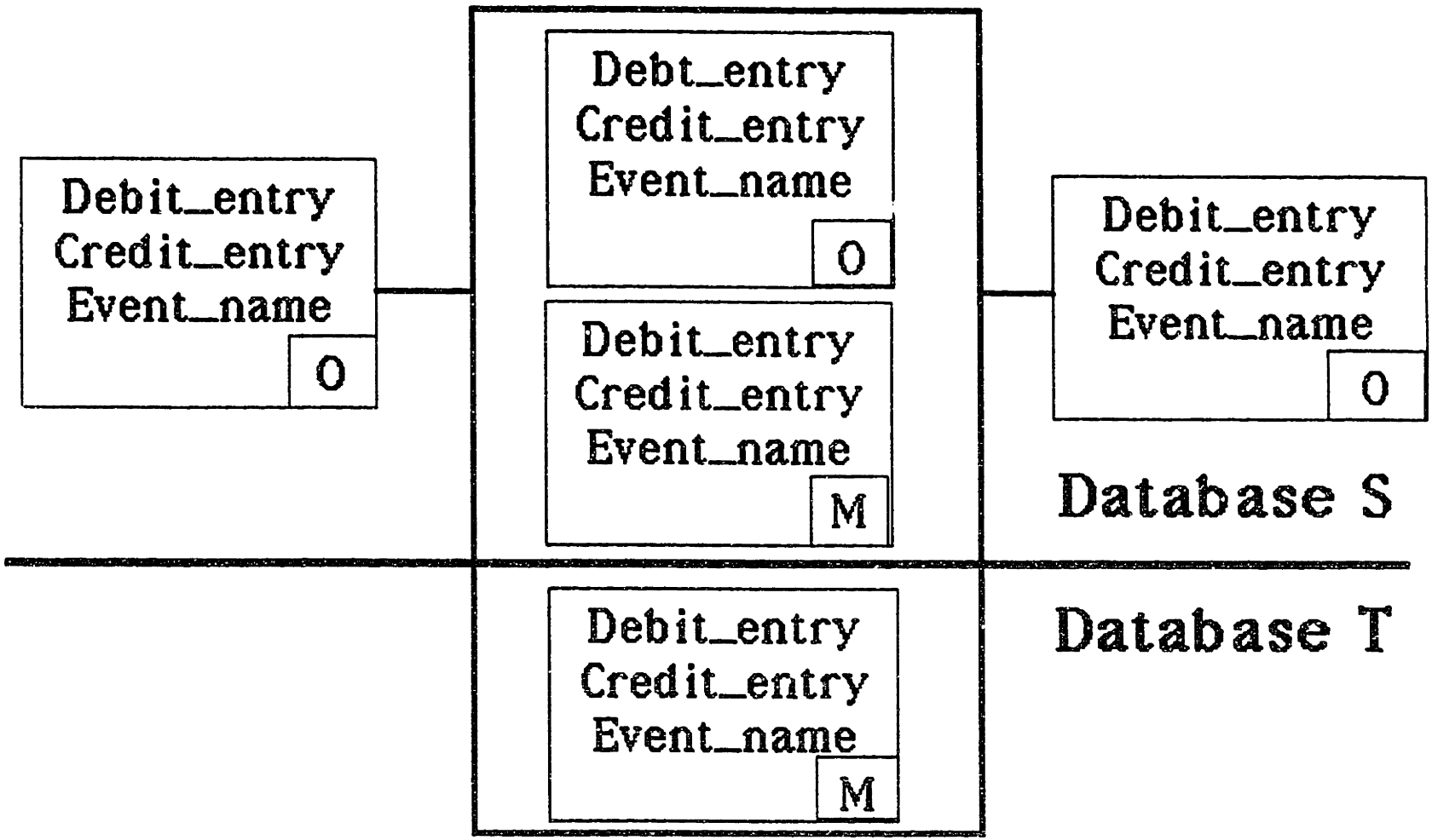
I define the following views:

- Transactions view: A worm's eye high volume high accuracy view of information about individual actions. By worm's eye, I mean an interest in individual transactions only with no interest in the "big picture". There is one record for each action (a "closed world"). If there is no record in the database, then we may conclude that the action did not happen. The large number of actions makes for high volume. The fact that an action cannot be completed if transaction-level data is incorrect places a premium on accuracy in this situation.
- Event view: A superset of the transactions view which adds information about the causality which relates individual transactions. For example, the event of an surgical operation relates several doctor and hospital transactions. This view might be modelled as a series of PERT or value chain charts. Support for an event view is well beyond the capabilities of existing databases.

- o Summary view: An aggregate view of transaction information by cause. Individual transactions views may record information from many different causes; to understand causality there is a need to selectively retrieve and summarize.

I propose that departmental myopia in the usefulness of their data or problems in enforcing integrity constraints leads to the adoption of the norrower transaction view too much of the time. I will go into this in more detail. The event view makes explicit causal relationships for particular products or services. Consider Diagram II, General Event View, where I have chosen the format of an annotated PERT chart to relate causality to the accounting information commonly recorded about actions in a business. Each node in Diagram II corresponds to an action of a department or a supplier which generates an accounting transaction recorded in one of several databases. Diagram II shows several kinds of information which are not recorded in the transactions view.

Diagram II
General Event View



Information other than the attributes needed for the transaction record[1] likely to be omitted from the transactions database include:

- o A description of the node which explains its purpose relative to the entire event, such as identifying the first action as a sales call. I refer to this as an event context descriptor.
- o Whether the presence of a particular action must be present for that event to belong to a particular event class (a mandatory transaction, marked "M" in the diagram) or may be present (an optional transaction, marked "O").
- o Pointers to immediately preceding and following transactions.
- o Pointers to transactions necessarily occurring at the same time.

Diagram III is a specific example of an event view for the transactions in a hypothetical occurrence of elective surgery. In the real world, insurance companies record their reimbursement of physician bills and hospital bills in different databases, even though both hospital and physician bills have a common source -- specific patients with particular maladies.

---

[1]which would be a journal entry if it were an accounting system

In diagram III, Specific Example of an Event View, we see a transaction marked diagnosis which is connected to a box containing four transactions which must occur at the same time. Each transaction contains a journal entry which consists of:

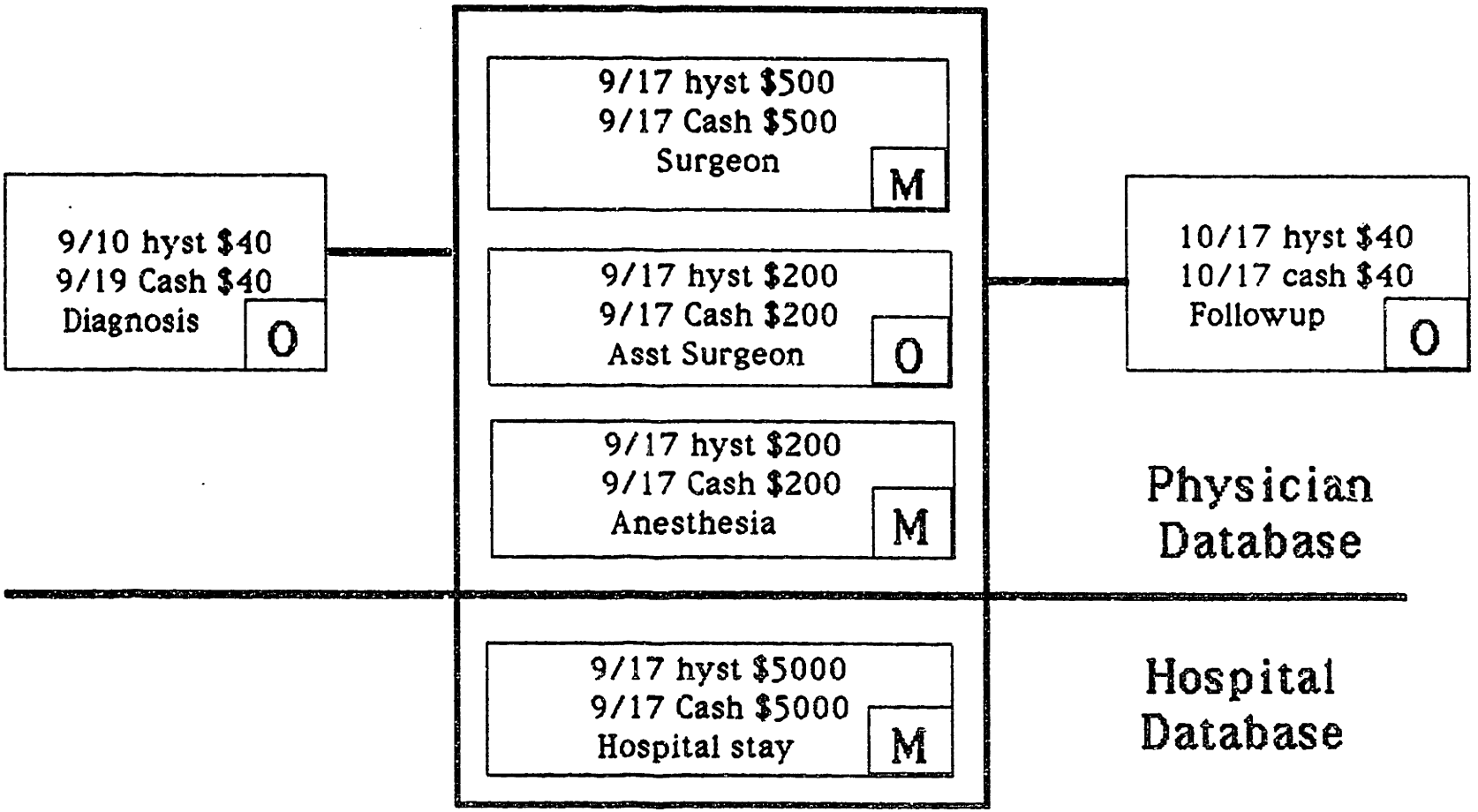
- a debit and a credit[2], each containing a date, an account name, and an amount.
- a event context descriptor (which is not needed for accounting).
- and a mandatory/optional indicator.

On the right is the name of the database, either physician or hospital, to which that accounting transaction is written. (The existence of two databases with no keys is, of course, my problem.) The two marked "surgeon" and "anesthesia" are mandatory -- surgery is impossible without both -- and are written to the physician database. The one marked "assisting surgeon" is optional -- it is possible to have surgery without an assisting surgeon -- and is also written to the physician database. These three physician bills and the hospital bill take place during the same period of time. The transaction marked "hospital" is also mandatory but is written to the hospital database instead. Finally, there is an optional follow-up visit to a physician.

---

[2]In double-entry accounting, each transaction is represented by two journal entries. Virtually all accounting is double-entry.

Diagram III
Specfic Example of An Event View
Elective Surgery



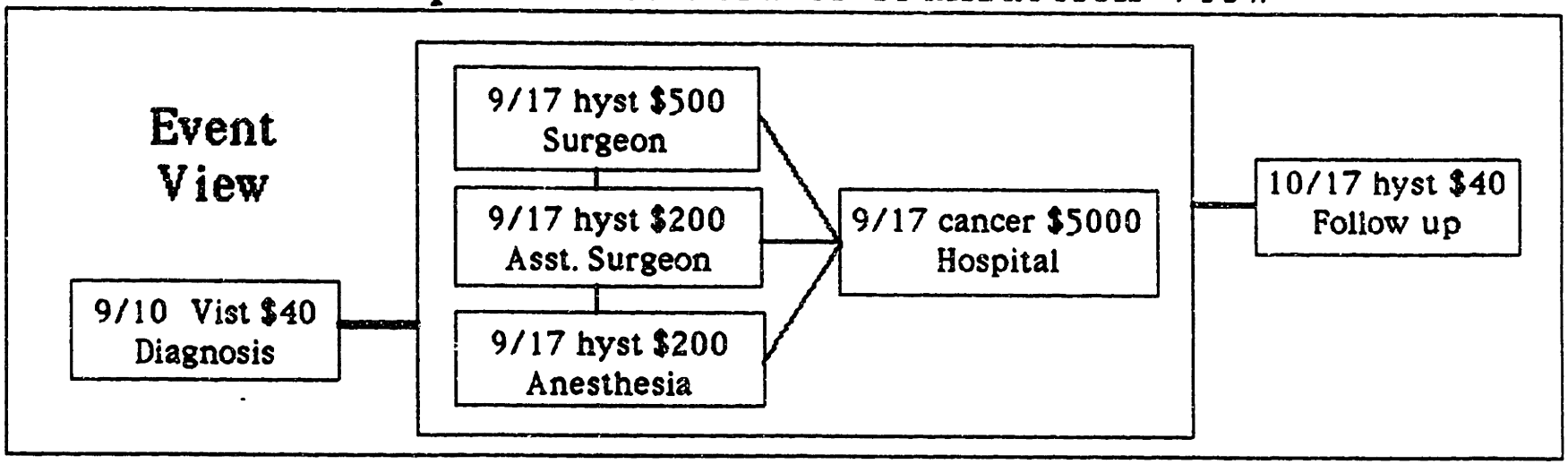
The transaction processing view is what is left of the event view information when order, simultaneity, and event context descriptors are stripped away. What remains is sufficient to meet the applications' demands for processing individual transactions. For example, a billing system does not need to know that a particular visit was a diagnosis (as opposed to a follow-up) in order to prepare a correct bill.

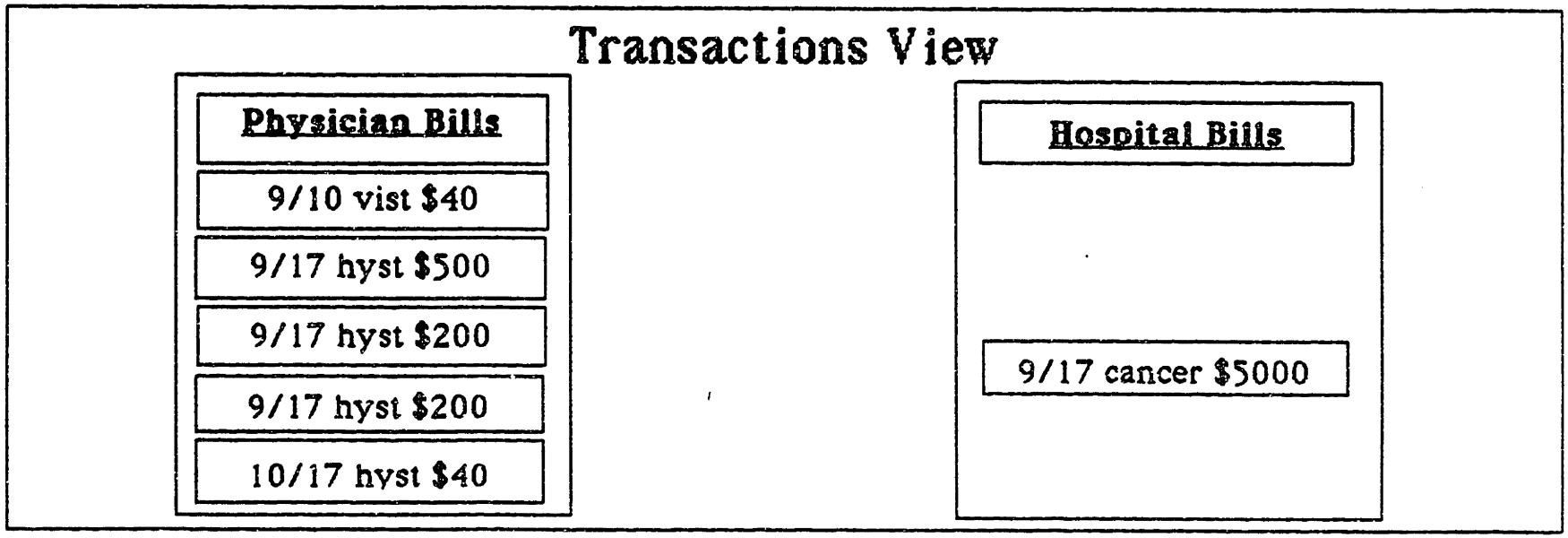
Diagram IV, Relationship of Event View to Transaction View, shows the relationship between the transactions view and the event view. In this diagram, we see nodes in the event view that are members of the same event as shown by their presence in the event view box and by the arcs connecting all the records. Those nodes are recorded as records in the two databases in the transactions view as shown by the transaction view box. Although the transactions information in all these nodes is recorded, the arcs between nodes are not. It follows that event membership is not recorded. More formally, we may speak of the transaction view as the projection of transaction information from the event view into the several databases, which loses non-transaction attributes and the connecting arcs.

What may be complete information from the viewpoint of accounting for a department may not be complete in the summary view. In the classical control theory paradigm, for example, a manager has a goal and, from time to time, determines whether

Diagram IV

Relationship of Event View to Transaction View





results meet the goal. If what actually happened is not the same as what the manager hoped would happen, the manager responds to correct the discrepancy. This implies the manager is able to retrieve and sum those transactions processing records relevant to that goal in order to determine what actually happened.

This database problem may be solved simply if there is no need to disaggregate the total. Since transactions databases are closed worlds in order to find, say, total expense one need only add up the expense attribute in all records in all databases without regard to relationships between individual records in the two databases.

The simplification goes away when there is a need to disaggregate any event class in order to "explain" the total. Disaggregating the total requires the ability to relate members of the same class across the databases -- in effect, establishing the need for a key because of the need to join two databases instead of appending them.

The need to disaggregate into meaningful classes defines the business problem. For decision-making purposes, we would find and perhaps summarize the records in meaningful event classes. In order to do that when multiple databases are involved, we either need all relevant relations to possess a "query attribute[3]" which permits finding relevant records or a key to join those relations with a query attribute to those without.

---

[3]Defined formally in section 3.2

In this chapter, I will develop the concepts needed to build a generalizable system. This chapter will discuss:

- Section 3.2, Discovering Keys Under Complete Information: The concept of "discovering" a compound key connecting two existing databases by discovering "comparison pairs" which would be a compound key if a means could be found to compare the attributes in the pairs.
- Section 3.3, Testing for Consistency under Incomplete Information: the means of testing particular pairs of records and instances of events.
- Section 3.4, Discovering Pairs of Attributes Worth Comparing: the means of actually discovering what attributes ought to be compared.
- Section 3.5, Logical Confidence Intervals Under Incomplete Information discusses how to cope when some, but not all, of the attributes needed to define a compound key are present. I define three comparison methods.
- Section 3.6, Selecting a Comparison Method When Attributes Are Defined Differently, I then discuss means of carrying out those tests. This section does not describe an implementation; rather it is a functional specification of what those tests must accomplish. This work might be viewed as applying incomplete information concepts to a join.

- Section 3.7, Probability of Accurate Retrieval, a probabilistic discussion of why the above theory is likely to produce acceptable results.
- Section 3.8, Storing and Reusing Retrofit Knowledge, discusses the means of retaining intermediate work product in the form of templates which can be used for similar queries.

3.2 Discovering Keys Under Complete Information

Let us say there exists a universal set of application types $\{A_1, A_2, A_i, \ldots, A_n\}$. These application types are classes of applications such as accounting or word processing.

A firm or department selects some subset of {A} as the set of applications to implement. Many applications involve the processing of data. For each application type requiring data which a firm wishes to implement the firm must design a series of data views $V_{i,1}, V_{i,2}, \ldots, V_{i,n}$. These V_i's show each end-user an end-user relevant subset of the data associated with an application type. Each V_i contains a set of attributes $a_{i,j,1}$, $a_{i,j,2}$, $a_{i,j,k}$, $\ldots$ $a_{i,n,m}$ where k indexes the attribute.

It is important to note that I am dealing with the views presented to the end-user, which are not necessarily equivalent to the base storage structures. This follows from the use of properties inherited from the application design, which are independent of normalization considerations. However, if a view exists in an implemented system, the application designer must

have been able to successfully implement each V_i as a combination of base relations $R_1, R_2, \ldots, R_n$. These relations are the storage structures containing the actual data needed to service an application, which is, in turn, an instance of an application type.

Before discussing the joining of two relations which lack common keys, let me introduce some terminology which applies to retrieving tuples from a single relation. Let E_a be the domain of attribute a. A given attribute has the property of being a "query attribute" if there exists a subset of E_a, E_q*, such that t_i, a retrieved tuple, is in R (the relation we are creating through the retrieval) if a_q is in E_q* and t_i is not in R if a_q is not in E_q*.

For example, suppose a particular view contains:

STATE-NAME, STATE-POPULATION

If we wanted to know the population of New England, STATE-NAME is a query attribute and E_q* = (ME,CT,MA,NH,VT,RI). If we wanted to know the population of the Boston metropolitan area there is no query attribute because the definition does not correspond to a state. The question cannot be answered with the information available.

In order to find the relevant tuples when two relations are involved, one of three (and only three) conditions must be true:

- o $a_{q.S}$ in V_S and $a_{q.T}$ in V_T: a query attribute is present in the schema for both relations[4] or,
- o $a_{q.S}$ in V_S and a_k in $\{V_S, V_T\}$: a query attribute is in one of the relations and there is an attribute (an "event key") in $\{V_S, V_T\}$ whose value is the same for all tuples which belong to the same event. In this situation, I will refer to S as the source view and T as the target view.
- o The query addresses all records in both views, which eliminates the need to join.

If one of these three conditions were true, resolving the query implicit in the summary view can be done straightforwardly (although perhaps not simply) in a relational database. In the following examples I am using the SUM function, although it should be noted that any aggregation function meets my definition of the summary view on p. 39.

In the first case, in which both relations have one or more query attributes computing a sum from both views would require adding the result of the following pair of queries:

```
SELECT SUM(S.a_c)
FROM V_S
WHERE S.a_q in ( {E_q,s*} )

SELECT SUM(T.a_c)
FROM V_T
WHERE T.a_q in ( {E_q,t*} )
```

[4] $a_{q.S}$ and $a_{q.T}$ may be defined the same way. However, they might not be. For example, if the query had to do with adding up the population of Massachusetts, $a_{q.S}$ might be zip codes and $a_{q.T}$ might be counties.

where a_c is the attribute we are trying to aggregate[2] and the results of these two queries are followed by summing the results of the two queries to give the grand total we seek (an operation not directly supported by SQL).

One should note that I am <u>not</u> assuming that the query attributes in the two views are defined the same way. For example, if S and T were two accounts receivable views, S might include:

CITY-NAME, CITY-POPULATION

and T might have:

COUNTY-NAME, COUNTY-POPULATION

We can still identify relevant tuples for particular standard metropolitan statistical areas (SMSAs) because SMSAs are always defined as a set of counties and because individual towns and cities are in one and only one county. We do, however, have to know the names of appropriate cities (i.e., $E_q^*.S$) and appropriate counties (i.e., $E_q^*.T$).

In the second case, one of the relations has a query attribute and there is a key linking the first relation and the second relation. This situation requires adding the result of

[2]For simplicity's sake, I am assuming that a_c is defined the same way in both database views.

the following pair of queries:

```
SELECT SUM(S.a_c)
FROM V_s
WHERE S.a_q in E_q*

SELECT SUM(T.a_c)
FROM V_s,V_T
WHERE S.a_q in E_q*
AND (T.a_k = S.a_k)[6]
```

where a_k is the key.

The above commands project the relevant tuples from S, aggregate those tuples, then use the relevant tuples retrieved from S to join with T.

In the third case one need only sum all records in both views:

```
SELECT SUM(S.a_c)
FROM V_s

SELECT SUM(T.a_c)
FROM V_T
```

Here, if we want to aggregate a_c from all the tuples (all tuples are defined as relevant) in both views, there is no need to know if a key links particular source tuples with particular target tuples. The presence or absence of query attributes is irrelevant because no subset is to be selected. One can merely

---

[6]Note that at a theoretical level one could write:

```
SELECT SUM(S.a_c) + SUM(T.a_c)
FROM V_s, Vt
WHERE S.a_q in E_q,s*
AND (T.a_k = S.a_k)
```

However the above query could not be evaluated in thirty hours of operation in my version of Ingres, while these two summation queries operated in fifteen minutes.

append the two views together and then aggregate.

The retrieval problem becomes more complicated if there is neither a connecting key nor a_q's presence in both relations. In such cases, we might be able to retrieve those records if we were able to construct a compound key. This requires knowledge of what attributes compose the key.

Let a_c, a compound key be defined as:

$$a_k \equiv a_1 \times a_2 \times \ldots \times a_n$$

where $a_1, a_2, \ldots, a_n$ are the attributes used to construct the key. A join on each of those attributes is equivalent to a join on the key as in the following SQL query:

```
SELECT function(dependent-attribute)
FROM V_S, V_T
WHERE a_q in E_q,s*
AND   S.a_1 = T.a_1
AND   S.a_2 = T.a_2
AND       .          .       .
AND       .          .       .
AND   S.a_n = T.a_n
```

where:

- a_i is one of the specific attributes named above
- V_S and V_T are the two views needing to be joined

Note that for a compound key to be usable we need to know which attributes make up the compound key and all those attributes have to be present in the database views.

The problem then becomes discovering the attributes which make up a key. To do this, I first need to discuss the relationship between application design and database contents. Suppose each of the application types defined at the beginning

of this section is partially defined by a number of universal attribute types (M_a) such that for each A_i there exists a set of universal attribute types ($M_{i,1}$, $M_{i,2}$, $M_{i,j}$, ..., $M_{i,n}$) where i indexes the application type and j indexes the universal attribute type. I make no claim as to how many of these universal attribute types exist for any given application. The more structured an application is, the more of these universal attribute types can be defined. For example, because accounting rules impose more structure on accounting information systems than document preparation traditions impose on word processing systems, more universal attribute types can be defined for accounting systems.

The schemas of the databases serving the applications implemented by the firm each must contain at least one (but possibly more) attributes $a_{A,M,i}$, instances of a universal attribute type, where A indexes the application, M indexes the universal attribute type, and i selects a particular attribute serving that universal attribute type. For example, all accounts payable systems may be said to be instances of a universal accounts payable type and any database serving an accounts payable applications must have an attribute which is an instance of the universal attribute type of MONEY.

I propose we can construct an approximate join by defining tests of consistency between attributes of the same universal attribute type stored in different databases. If V_i and V_j are both views serving different instances of A_i (the same

application type), then V_i and V_j both have the same M_i. The a_A,M,i in V_i and in V_j can be matched up to form sets of attributes which are instances of the same universal attribute type. In partition notation, we can compute $M \times (V_i + V_j)$. Further, if we can define a comparison function $f(M \times (V_i + V_j))$ which produces executable tests of consistency between attributes of the same universal attribute type stored in different databases we can execute those tests in the local query language against actual data.

The accuracy of the approximate join will depend on:

- The analyst's ability to define the application and the application's universal attribute types at the level of generality required by the end-user and comprehensible to the end-user.
- The explanatory power of the universal attribute types and their usefulness in distinguishing relevant from irrelevant tuples.
- The accuracy and comparability of the attributes implementing those classes in actual databases.
- Whether sufficient tuples will be retrieved to permit the law of large numbers to damp out errors.

This issue of accuracy is discussed at length in section 3.7.

In the medical example I have been using, accounts payable is an application, A_A. Both the physician and hospital payment logs are instances of an accounts payable application, $V_{S,doc}$ and $V_{S,hosp}$. Accounts payable may be divided into universal attribute classes of {BUYER, SELLER, WHAT, WHEN, MONEY}. This set of universal attribute types defines M_A. Suppose $V_{S,doc}$ contains {policyID, groupID, procedure, date, MDID, payment} and $V_{S,hosp}$ contains {policyID, diagnosis, adm-date, HospID, payment}.

M_A x $V_{S,doc}$ = {
{BUYER [PolicyID]
{BUYER [groupID] ,
{SELLER [MDID]},
{WHAT [procedure]},
{WHEN [date]},
{MONEY [payment]}).

M_A x $V_{S,HOSP}$ = {
{BUYER [PolicyID]},
{SELLER [HospID]},
{WHAT [diagnosis]},
{WHEN [adm-date]},
{MONEY [payment]}).

Multiplying together M_A x $V_{S,doc}$ and MA x $V_{S,HCSP}$ gives us:
{{BUYER [doc.PolicyID, Hosp.PolicyID],
{BUYER [doc.groupID, Hosp.PolicyID]},
{SELLER [doc.MDID, Hosp.HospID]},
{WHAT [doc.procedure, Hosp.diagnosis]},
{WHEN [doc.date, Hosp.adm-date]},
{MONEY [doc.payment, Hosp.payment]}}

The above cross-product is used to establish tests of record-level consistency, discussed at length in section 3.3.1. The next section defines various tests of consistency when there is insufficient information to establish or prove correct retrieval. In other words, if we cannot establish the correctness of the retrieval we try to establish that it is not

provably incorrect. If absolutely accurate retrievals are not required accuracy may be sufficiently high after applying several tests of sufficient discriminatory power. Each test that eliminates an impossible retrieval increases the conditional probability that the remaining tuples are correctly retrieved.

In addition, if we know how many retrievals there should be, we can be sure we have found the right retrievals once we have pared down the number of retrievals to that number, assuming that we only eliminate impossible retrievals and not just unlikely ones. For example, if there is a one-to-one relationship between surgical operations and hospital stays and if the number of surgeon's tuples retrieved matches the number of hospital tuples retrieved then we know that we have retrieved the right ones.

3.3 Testing for Consistency under Incomplete Information

I define three kinds of consistency:

- Record, consistency within a single pair of records. For example, a hospital record would be record-level inconsistent with a surgical record if the surgery did not take place during the hospital stay[7]. (Later, in section 3.5, I extend the definition of record

[7]Neither group contents nor order consistency are implemented or designed; the scope of this work is limited to a functional specification.

consistency to handle incomplete information -- that is, the presence of attributes whose values are not known for certain. This extension does not affect how tests of group and order consistency function, since those tests operate only on tuples that have passed the record-level consistency tests)

- Group contents, consistency with what is known to be present in the set of records that define an occurrence of an event. For example, if there were no surgeon's bill in a set of retrieved records record-level consistent with an occurrence of elective surgery one would conclude that it was group contents inconsistent.
- Order, consistency with the order of particular records that define an event. For example, if the date of a record whose event context is "diagnosis" were after the date of a record whose event context is "follow-up" in a set of retrieved records for an occurrence of elective surgery, one would conclude that it was order inconsistent.

3.3.1 Record Consistency

Diagram V, Key, Compound, and Incomplete Joins, gives an example of record consistency and how it compares to single attribute and compound keys. In this diagram, I show two relations, each containing two records. In the first (source) relation, there is one record for Mary Barnes and one for David

Barnes. In the second (target) relation, there is one record for Fran Barnes and one for David Barnes. Retrieved records are in bold; unretrieved records are in normal. Noting that only the David Barnes record in the source relation has a corresponding entry in the target relation, I look at how the presence or absence of attributes and keys in these relations affect the ability to accurately retrieve the records of David Barnes' prostatectomy on 9/17:

- Case I: Single attribute key, where there is a single attribute defined in both databases such that two records are definitionally related when the values match. The diagram shows that it is possible to retrieve the right records (only David Barnes retrieved) by comparing the keys in both relations.
- Case II: Multiple attribute complete (compound) key, where a key is defined using more than one attribute. For example, if a key were defined on first name, last name, and date, then it is possible to retrieve the right records (only David Barnes retrieved) by comparing the values of those three attributes in both relations and retrieving records when all of the values compared match.

Diagram V
Key, Compound and Incomplete Joins

(Retrieved) (Not Retrieved)

Doctor Bills					Hospital Bills				
Lname	***Fname***	***Date***	***Proc***	***Key***	***Key***	***Lname***	***Fname***	***Date***	***Diagnosis***

Case I Key join: We found David Barnes

Barnes	**David**	**9/17**	**prost**	**KK7**	**KK7**	**Barnes**	**David**	**9/17**	**Pcancer**
Barnes	Mary	9/17	hyst	JT6	ZL4	Barnes	Fran	9/17	Ucancer

Case II Compound Key: We found David Barnes

Barnes	**David**	**9/17**	**prost**		**Barnes**	**David**	**9/17**	**Pcancer**
Barnes	Mary	9/17	hyst		Barnes	Fran	9/17	Ucancer

Case III Incomplete Key: We found David Barnes... plus somebody else

Barnes	**9/17**	**prost**	**Barnes**	**9/17**	**Pcancer**
Barnes	**9/17**	**hyst**	**Barnes**	**9/17**	**Ucancer**

- Case III: Incomplete compound key, where not all of the attributes which define a key are present in both databases. If one of the key-defining attributes were missing (such as the first name) the correct record would still be retrieved -- along with any record that matched on the 2 attributes which are present. (In the diagram, this is why Fran Barnes' hospital record is retrieved as well as David Barnes.)

Unfortunately the solution of creating a compound key based on available attributes works perfectly if and only if we know that attributes which would function as a compound key exist and we know what those attributes are. This is not a trivial problem and is the subject of the rest of this dissertation.

3.3.2 Group contents consistency

I now turn to the second kind of consistency; group contents consistency. We can use knowledge of which kinds of tuples must be present to create an additional test of consistency with the event definition. The method I have specified so far checks for consistency with the event definition by verifying that the values of the key-defining attributes were consistent across the databases.

Suppose we also possessed knowledge that certain tuples were required to be present by the definition of the event. For example:

- For each surgery done, there must be one corresponding hospital record.
- For each surgery done, there must be one corresponding anesthesiologist's bill.

Let us say that we have completed a retrieval checked for record level consistency. Let us group the set of retrieved records into instances of events by sorting on the key-defining attributes. Then we can apply the following rules against each of the n event instance groups:

```
IF number of mandatory records actually found  <
required number of mandatory records
THEN event_retrieval_status := anomaly

IF number of mandatory records actually found  =
required number of mandatory records
THEN event_retrieval_status := certain

IF number of mandatory records actually found  >
required number of mandatory records
THEN event_retrieval_status := ambiguity
```

The appropriate interpretation of each case is:

- o Anomaly: We have a list of transactions that must have taken place for the event to have taken place, but cannot find all the required pieces. For example, we have found one surgeon's bill for a hysterectomy, but cannot find any hospital records. We know something must be wrong, but do not have the data to conclude whether a hospital record is missing or a physician bill was improperly coded. If one wishes to avoid an underestimate, one adds the cost recorded in the retrieved record to the average cost of missing records.
- o Certainty: We have found one of each of the expected transactions.
- o Ambiguity: We have found more than one of each of the transactions we expected to find. We do not have the information needed to identify the right one. In the accounting context of this example, the conservative action when evaluating expenses is to use the most expensive one.

Diagram VI, Group Level Consistency, shows examples of the three possible results. In the anomalous case, two records were retrieved. It is inconceivable that anyone would be anesthetized without a surgeon's bill[8]. Either the anesthesia bill was erroneously recorded or the surgeon's bill was

[8]It is conceivable that the surgeon decided not to proceed with the operation after anesthesia, but he would still bill for something.

erroneously not recorded or not retrieved[9]. We do not know which occurred, and so the amount spent is an open question. To compute the marginal effect of this event occurrence on the maximum and minimum total costs of all occurrences of this event, one could add the known costs to the estimated value of the missing costs for the maximum estimate (a procedure similar to handling missing values in statistics) and exclude everything for the minimum estimate.

In the certain case, one instance of each activity must have taken place with no extras. The sum of the activity costs is the only possible answer. There is no incomplete information problem.

In the ambiguous case, more records were retrieved than should exist, according to the event definition. We have no information to determine which of these hospital records should have been retrieved. This ambiguity occurred because another person covered under the same policy was hospitalized at the same time. Because our goal is to determine the amount spent, the answer is certainly either $5,700 or $6,700. The upper bound is $6,700 and the lower bound is $5,700.

---

[9]In order to do this, it was first necessary to infer an event context descriptor for the retrieved records. For example, in the database I'm using, there is no attribute which could identify which physician bill is the surgeon's. However, one could infer that with the heuristic that whoever got paid the most was the surgeon. This is outside the scope of this work.

Diagram VI Group Level Consistency

Retrieved records

ID #	What	Date	Money	
18201	Anesthesia	9/17	200	Anomaly (no surgeon bill) (anesthesia shouldn't be there or surgeon's bill should also be included)
18201	Hospital Stay	9/17-9/19	5000	
			?	
48702	Anesthesia	9/17	200	Certain (everything is there)
48702	Hospital Stay	9/17-9/19	5000	
48702	Surgeon	9/17	500	
			5700	
89904	Anesthesia	9/17	200	Ambiguous (too many records) (which is the right one?)
89904	Hospital Stay	9/17-9/19	6000	
89904	Surgeon	9/17	500	
89904	Hospital Stay	9/17-9/19	5000	
			5700 or 6700	

Mandatory records: Anesthesia, Surgeon, Hospital Stay

Group contents consistency, which is not being implemented in this prototype, might be implemented by viewing each group separately and testing for the existence of the required number of each mandatory record by counting the number of retrievals in each category. In order to do this test of group consistency you need:

- o A mandatory tuple information table storing:
  - Query name: the name of the query for which the tuple's presence is mandatory. In my hysterectomy example, hysterectomy is the name of the query.
  - Event context descriptor: a field which identifies the meaning of the record in the context of the entire event. For example, a hysterectomy requires a surgeon's bill, an anesthesiologist's bill and a hospital stay.
  - Required quantity: a field recording the number of that tuple required for consistency with the event definition. For example, we anticipate one surgeon's bill.

Software to ask for the above information would be have to be added to the subject knowledge acquisition process. (see section 4.3, Subject Knowledge Acquisition)

- o Records grouped by event instance: The record-level consistency query does not sort retrieved records by event instance. This sort is needed to check for group

contents consistency and can be done by sorting on the test attributes, such as specific policyID and time values (in the medical example).

- o The retrieved record's event context descriptor: which will either be an existing attribute in the database or something inferred for each record from existing attributes. For example, a bill may have an attribute identifying it as a "diagnosis". If not, an expert subsystem is required to infer that the bill is a diagnosis because it is the earliest.
- o An enhancement of the query assembly software which writes additional queries to check whether each group contains the number of records listed as mandatory in the Mandatory Tuple table. The software described in sections 4.4 and 4.5 needs to tally the number of occurrences of each event context descriptor and then compare the tallies of each event context descriptor in each group with the number that should be there, as stored in the Mandatory Tuple Table.

3.3.3 Order consistency

Finally, I define a test of order consistency based on the order of occurrence of actions within each group. Suppose we possess knowledge of the sequence in which actions must or may occur. For instance, we know for my medical example:

- o The mandatory hospital stay, the mandatory surgeon's

service, the optional assisting surgeon's service, and the mandatory anesthesiologist's service all must occur at the same time.

- o The optional diagnosis service must precede the surgeon's service.

Note that the first of these is a stiffer version of the corresponding group contents test -- not only must mandatory actions be present but they must be present at the same time. Note also that adding an order test enables some verification of optional actions, such as the presence of a diagnostic visit or an assisting surgeon. Optional actions could not be verified with the group contents because no error could be inferred from their absence.

One may establish the general principle:

```
IF A is present in the group
AND B is present in the group
AND A is supposed to happen before B
AND B happens before A
THEN anomaly

IF A is present in the group
AND B is present in the group
AND A & B are supposed to be at the same time
AND A & B are not at the same time
THEN anomaly
```

The interpretation of anomaly here is the same as the interpretation of anomaly in the group contents test -- one of two records is improperly retrieved, but we cannot determine which one.

Implementation would have much in common with implementing the group contents consistency test. One needs to:

- Group retrieved records by event instance
- Divide retrieved records into same-time subevents
- Infer or have each retrieved record's event context descriptor (as defined on the preceding page)
- Enhance the query assembler to test for the aforementioned anomalies

This test of order consistency could be implemented by defining a Same-Time Subevent Table which would define sets of actions that needed to take place at the same time and a Mandatory Order Table which records precedence and simultaneity relationships between pairs of transactions. The query would test for consistency with those relationships. The storage mechanism would be similar to those of project management systems which store the arcs/node-pairs needed to reconstruct PERT and other kinds of activity graphs. For example:

SAME-TIME SUBEVENT TABLE

Subevent	Action	Man/Op
Diagnosis	Diagnosis	O
Surg-stay	Surgeon	M
Surg-stay	Asst. Surgeon	O
Surg-stay	Hosp-stay	M
Surg-stay	Anesthesia	M
Follow-up	Follow-up	O

MANDATORY ORDER TABLE

Query	Action 1	Action 2	Relationship
hyst	surg-stay	follow-up	precede
hyst	diagnosis	surg-stay	precede

The storage task is simplified by the transitivity of temporal precedence -- that is, if A happens before B and B happens before C, then A happens before C[10].

3.4 Discovering Pairs of Attributes Worth Comparing

I now turn to the problem of determining which attributes to use in testing for consistency. There are six basic steps:

1: Define a set of universal attribute types for the application type whose instances the end-user is trying to link.

2: Assess, using the methods in section 3.7, whether those types are sufficiently accurate and general to be useful (If the assessment finds the types are insufficiently accurate and general, this theory will not be helpful.)

3: Assign each attribute in the schemas into one of the universal attribute types defined in step 1.

4: Form potential comparison pairs by taking the cross-product of that augmented schema and using the subset of the result when the attributes are in different databases but are of the same attribute type.

5: Ask the end-user to specify which universal attribute

[10]I am not considering conditional precedents -- where C may happen if A OR B happened. This adds problems beyond the scope of this work.

types are relevant to the event.

6: Select the subset of comparison pairs which belong to the universal attribute types selected as relevant.

A universal attribute type is a kind of datum that must be present in system output in order for the system to be an application of a particular type. For example, if a database has no attribute representing MONEY one may infer that the system is not an accounting system. A particular type of application can be partially characterized by its universal attribute types.

However, knowing that any accounting system must produce reports about money does not tell us how "money" is stored or what the name(s) of the attribute(s) representing money is/are. The representation may be:

- straightforward, as when a single attribute represents the universal attribute type or,
- complex, as when there are several attributes in the universal attribute type.

Suppose each of the several databases can be classified with respect to application type. For example, in a particular firm with six transaction database views, each might be partitioned with respect to application type as follows[11]:

$$\text{ACCOUNTING} = \{D_1, D_3\}$$
$$\text{SCHEDULING} = \{D_2\}$$
$$\text{WORD PROCESSING} = \{D_4, D_5\}$$
$$\text{MEDICAL IMAGING} = \{D_6\}$$

Suppose we define ACCOUNTING as consisting of (BUYER, SELLER, WHAT, WHEN, MONEY) and FORM-LETTER as consisting of (RECIPIENT, CONTENTS). At least one attribute (and possibly more) must be defined in each type in each database view. RECIPIENT, for example, could be represented by NAME, ADDRESS, CITY, STATE, and ZIP in an actual database.

This is suggested in Diagram VII, Departments and Their Generic Applications. This diagram shows two hypothetical firms or departments, each with several databases partitioned into universal application types. Department one has an accounting system, a scheduling system, and a word processing system. Department two also has accounting and word processing but not scheduling. We can see that applications of the same type have the same kind of data, although we do not know how those universal attribute types are implemented in the actual databases.

Individual attributes in the actual databases are instances

---

[11]Recall that I am talking about a V_i, a database view. I am unconcerned with how the DBMS software relates base structures to the view.

of these universal attribute types. In my health example the BUYER type is represented by (policyID, groupID, AGE, SEX).

Consider diagram VIII, Inheritance of Universal Attribute Types, which shows the construction of this relationship between type and underlying databases for a pair of hypothetical payment logs for physicians and hospitals. In this diagram, I show the attributes in both of the accounting databases are instances of the same generic attribute types, even though the two databases have different attribute names and possibly different internal representations. I assume the type inheritance is unique -- each attribute inherits membership in one (and only one) of the universal attribute types. For example, Date and LOS (length of stay) are both in the WHEN type.

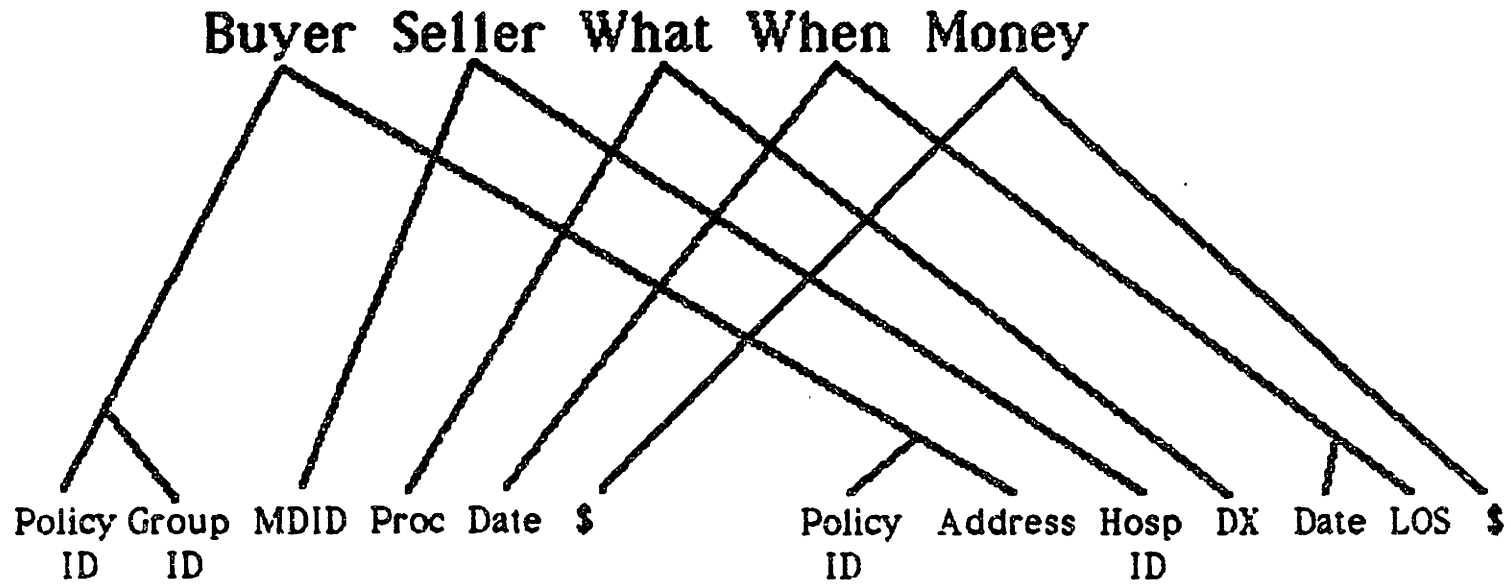
If the attributes inherit the universal attribute type, the universal attribute type inherits the application type, and if there are two databases of the same application type we can form possible comparison pairs by taking the cross-product of the data dictionary. (We can find comparisons with two databases not of the same application type only to the extent that attribute types are common to both applications).

Diagram VII
Departments and Their Generic Applications

Department One	Department Two
Accounting Buyer Seller What When Money	**Accounting** Buyer Seller What When Money
Scheduling From To Needed_by Length Name	**Medical Records** Who Vitals ClinLab Xray Drugs Diet
Word Processing File description Filename	**Word Processing** File description Filename

Diagram VIII
Inheritance of Universal Attribute Types

Accounting



Physician Bill Database **Hospital Bill Database**

Let datadic be a relation containing data dictionary information. For example, in the widely used database query language SQL:

```
SELECT S.type, S.DB, S.att, S.domain, T.DB,
    T.att,T.domain
FROM datadic S, datadic T
WHERE not S.DB = T.DB
AND S.type = T.type
AND S.DB = sourceDB
```

where:

- S is the source database and T is the target database
- type is the name of the attribute's type
- DB is the database storing that attribute
- att is the name of that attribute
- domain is the name of that attribute's domain
- sourceDB is the name of the source database

This slightly obtuse SQL query creates the aliases S and T for datadic (the data dictionary relation) and then joins the two "copies". The table inferred with the above query gives possible comparison pairs -- pairs of attributes, one in each of the two databases to be joined which are of the same type along with information as to the databases those attributes came from and how they are defined.

For example, suppose we had a data dictionary containing the following four attributes from two different databases:

DB	Attname	Type	Domain
S	Name	BUYER	discrete
S	MDID	SELLER	discrete
T	Name	BUYER	discrete
T	HospID	SELLER	discrete

Then the following result would be produced:

S.DB	S.att	T.DB	T.att	Type	S.dom	T.dom
T	Name	T	Name	BUYER	disc	disc
T	MDID	T	HospID	SELLER	disc	disc

Where:

- o S.DB is the name of the source database
- o S.att is the name of an attribute in the source database
- o T.DB is the name of the target database
- o T.att is the name of an attribute in the target database
- o Type is the name of the attribute type to which both attributes belong[12]
- o S.dom is the domain of the source database attribute
- o T.dom is the domain of the target database attribute

If some subset of these comparison pairs defined a compound key and both members of all pairs were defined the same way then we could process those pairs into an accurate (not approximate) query.

To identify that subset we need to know which universal attribute types were relevant. By "relevance" I mean those attribute types which the end-user requires to be consistent across databases. The end-user requires these to be consistent in order to implicitly identify a data concept which is missing from the databases. Suppose one wished to infer the patient connection between a physician payments and a hospital payments log. The concept which is missing from the underlying databases is the episode of illness, which I will define as a specific episode of illness and its associated package of treatments. (There is an n:1 relationship between episodes of illness and

---

[12]constrained to be the same fcr both attributes, so it does not matter whether S.class or T.class is specified in the query producing the table

patients). One cannot retrieve what was not recorded. However, since the concept of an episode of illness is an inescapable part of reality the database in any application operating in the real world will have the possible values of its attributes constrained by those episodes of illness.

In this medical/accounting example recall that the five universal attribute types in accounting are (BUYER, SELLER, WHAT, WHEN, MONEY) and that both of the databases we are trying to join are accounting databases. The relevant universal attribute types are:

- BUYER, since the individual operated upon should be the same individual as the one admitted to the hospital.
- WHEN, since the surgery should take place while the patient is in the hospital.
- WHAT, since the surgical procedure should be logically consistent with the hospital services.

Irrelevant types would be:

- SELLER, since there is no reason why the identity of the doctor should be the same as the identity of the hospital.
- MONEY, since there is no reason why the physician's bill should be for the same amount of money as the hospital bill.

The issue has been raised as to whether the relevance of the aforementioned classes is independent or dependent of the

query. To answer this requires asking three distinct questions:

1. Whether a particular universal attribute type is necessary to a particular application type. For example, I hold the concept of identity or "whoness" to be inherent in an accounts payable systems. In the context of the current example, answering the questions involves determining whether who was treated is sufficiently important to store. (One could build a system which stored information of what treatment was provided, but not record to whom that treatment was provided)
2. Whether a particular universal attribute type must be held consistent between databases to identify the missing concept. In the current context, I hold that the attribute type WHO was relevant because the individual operated upon should be the same individual as the one admitted to the hospital.
3. Whether the degree of precision in implementing each of these universal attribute types in the actual databases is sufficiently "accurate" for my purposes. This is an empirical question, not a theoretical one. Although different application types may happen to have common universal attribute types there is no particular reason why they should be at the same precision level either. WHO is implemented at the level of "family" by Blue Cross. We are interested in WHO at the level of

"individual" when analyzing pattern of medical resource utilization. If we were analyzing automobile or magazine utilization, we would probably not care if WHO were implemented at the level of family because automobiles and magazines can be used by more than one family member.

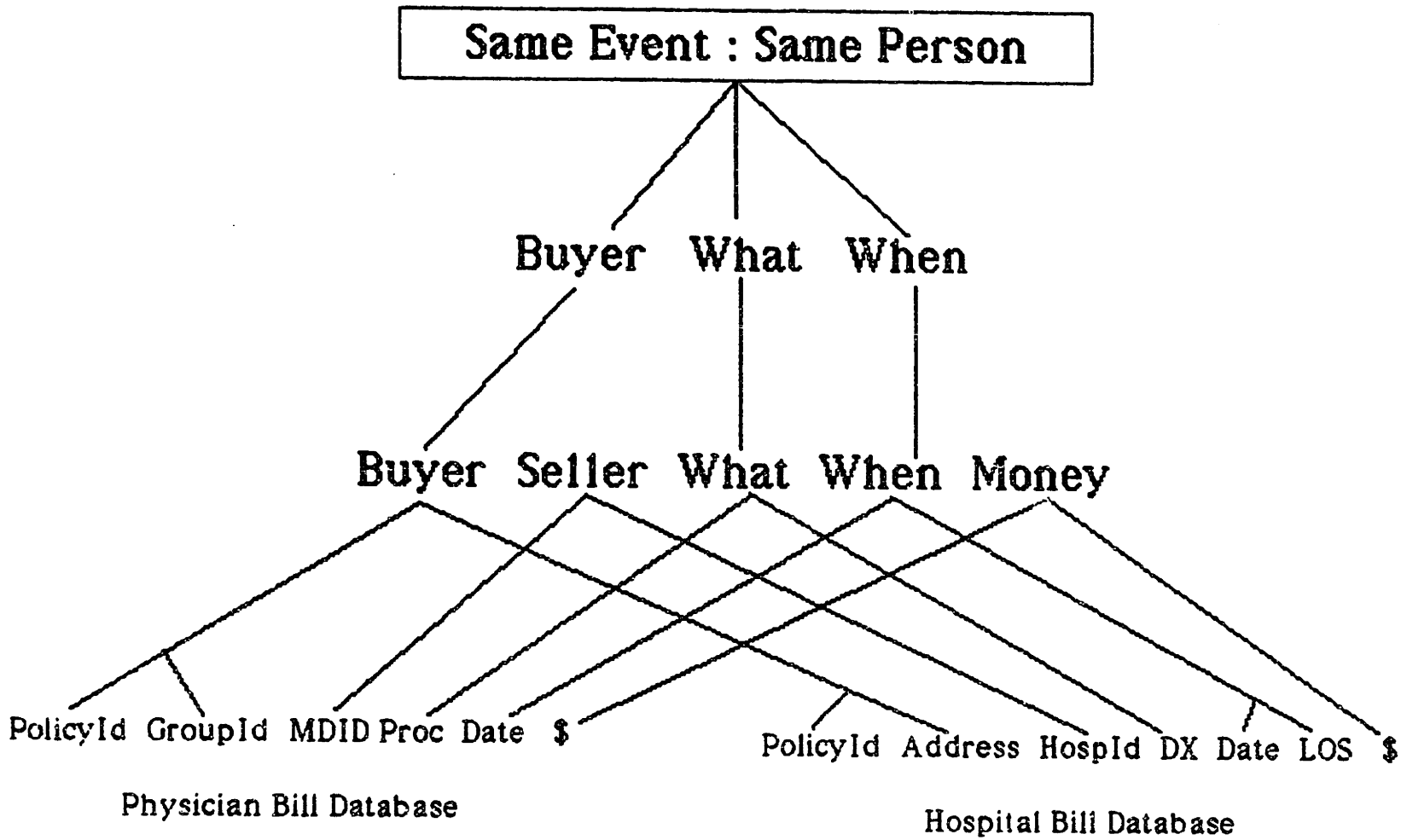
The issue of universality revolves around whether the answers to the above questions are dependent solely on end-user opinion or are constrained by an external reality which forces end-users to hold that opinion. I hold the latter position. Reality is a powerful constraint on the potential beliefs of knowledgeable end-users. It is unlikely that anybody in any position of authority in the health industry (for example), whether administrative or medical, would be able to retain their positions if they held the belief that it was unimportant who in a family was treated upon the illness of a family member. Reality forces universality.

With regard to the implementation of the universal attribute type in the databases to be joined there is no particular reason why any given implementation of an application type should contain information at the precision level needed to serve a purpose for which the application instance was not designed. It is purely a matter of luck -- the subject of section 3.7. The relevant question is how to take advantage of your good fortune if you get lucky and how to best cope with your misfortune if you are not lucky.

This relationship between relevant attribute types and missing data concepts is illustrated in diagram IX, Relevant Universal Attribute Types where I show that the subject of an event -- an episode of medical care requiring hospitalization in this example -- is identified by three of the potential comparison pairs in accounting databases. To be the same event happening to the same person, the values of the attributes implementing the BUYER, WHAT, and WHEN categories have to remain consistent across relations.

Fortunately, it is not necessary for the end user to enter knowledge of the causal model which would be required to infer which types were the relevant types, nor does the system need to have structures to store and use that causal knowledge. It would be difficult to overemphasize the unimportance of causal knowledge for this system. Having the system (as opposed to the end-user) decide which attribute types were relevant would require a large quantity of domain-specific knowledge in addition to the very general search knowledge. Then the value of the system having a causal model of the subject domain would be quite low because the end-user would be extremely likely to know that model himself.

Diagram IX
Relevant Universal Attribute Types



For example, it would not be news to health benefits managers to learn that doctors admit sick people to hospitals instead of the other way around. Indeed, it would be a considerable technical accomplishment to get a computer to infer from a knowledge base of general medical principles that the identity of an individual operated on should be the same as the identity of an individual admitted to a hospital. (Successful expert systems usually provide advice on something the user does not already know, such as how to configure a VAX. The value of something one needs and does not have is generally higher than the value of something one already has.)

This notion of "knowledge separability" is important. Creation of database queries require four distinct kinds of knowledge:

- database knowledge: the meaning of attributes in the specific databases being queried.
- search knowledge: the use of joins and/or other methods of implementing retrieval in the model(s) of the local databases.
- language knowledge: expression of the above in the language(s) of the local databases.
- subject knowledge: the ability of the end-user to come up with a meaningful question based on underlying structures.

I built the necessary language and search knowledge into the system at construction. The user interface permits database

knowledge and subject knowledge to be entered independently.

The end-user need not be trained in the specifics of database content, search procedure, or local query language to the extent needed in traditional systems because that knowledge is either part of the system (search and language) or was entered previously by a database administrator. Similarly, the database administrator does not need to know as much about the subject the application serves as do end-users in order to be able to enable retrieval by the people he is servicing.

3.5 Logical Confidence Intervals Under Incomplete Information

There is a practical problem when some of the attributes needed to form a compound key are missing. I will now turn to additional steps to deal with this problem. There are three distinct problems with incompleteness:

- Selecting comparison pairs when there is more than one comparison pair within a particular universal attribute type.
- Determining how to resolve the comparisons when the two attributes in a comparison pair are not directly comparable.
- Evaluation of the effect of the missing attributes on the accuracy of the result.

This section will deal with the first two problems. Accuracy will be dealt with in section 3.7.

The step of taking a cross-product of the data dictionary and retaining pairs in different databases of the same application type yields a list of possible comparisons. However, three potential problems arise:

- Attributes in the same attribute type do not necessarily have comparable meanings. For example, comparing AGE to SEX and SEX to AGE makes no sense, even though both AGE and SEX are in the BUYER type -- they both describe the entity buying services. Since it does make sense to compare DIAGNOSIS to PROCEDURE simply testing for equality of domains does not solve the problem. Unfortunately, the database does not "know" anything about the meaning of AGE and SEX which would allow it to make that judgement. Enabling the system to decide whether two different domains might be compared via a table look-up is beyond the scope of this work.
- Some of the comparisons which do make sense are redundant. For example, comparing groupID to groupID is redundant because no policyID is in multiple groupIDs. A groupID comparison adds no more information to a policyID comparison.
- Some attributes have unreliable values; comparisons between those attributes could result in improper non-retrieval. As one adds additional tests based on attributes with potentially incorrect values the joint probability of improper rejection goes up quickly. For

example, if the test used ten attributes that were each 80% accurate (i.e, 80% of the time the attribute value is correct; 20% of the time it is wrong), then the joint probability of a retrieval is only 10% (.8^10).

The first problem is simply due to the fact that there is more to know about an attribute's meaning than its application type. A reasonable heuristic for pruning away these incomparable comparison pairs is to note that similar attributes may be present in both databases and have the same naming tradition. The heuristic:

```
IF names are unique
AND there is a pair where both attributes have the same name
THEN eliminate other pairs where an attribute has that name
```

would reach the desirable result of (in the above example) finding the AGE:AGE and SEX:SEX pairs. This heuristic only works if the same attribute name implies the same domain and usage -- a questionable assumption in the context of retrofit, although I believe it would be a good idea to have the machine present a list of such possible correspondences and then ask for user confirmation or rejection. This heuristic is not implemented in the prototype system and the correct result is done by hand.

The second problem, redundancy, is "only" a performance issue when attributes are recorded accurately. The ability of some attributes to discriminate is dominated by other

attributes whose domain has a higher cardinality. For example, in the databases used for the prototype (policyID, groupID, AGE, SEX) are in the BUYER type. Both policyID and groupID are audited. No information is added by checking groupID as well since all employees are contained within one and only one group (assuming accurate attribute values). I refer to this as a "containment test".

However, other contained attributes in the same type should not be eliminated. For example, if there were attributes corresponding to NAME and ADDRESS in the database to be joined by the system information is gained by using both because neither contains the other -- which is precisely what the system I have implemented for this dissertation would do.

The final problem specified above is the rejection of relevant records because of inaccuracy in recording the values of attributes. Mainstream database theory assumes that there is never an error in recording attribute values. In reality, there is a certain probability, π, that the value of an attribute is what the database says it is and a complementary probability, $1-\pi$, that the value is wrong. The theoretically "right" way to account for inaccuracy is to determine the probability distribution of error on each attribute and then carry out the appropriate discrete choice analysis. (see Pindyck76) This is a very complex computation, even if the probability

distribution underlying the attribute were known[13].

An alternative way to handle differences in attribute reliability is to turn to the Dempster-Shafer theory of evidence, whose database analogue is Lipski's theory of incomplete databases [Lipski85]. (See section 2.9 in the literature review) In this approach, the answer is given as an upper bound (everything that might satisfy the query) and a lower bound (everything that must satisfy the query).

For example, we should certainly have more faith that a record was correctly retrieved if policyID matched and AGE and SEX matched, rather than policyID alone. Similarly, we might say that uterine cancer is record consistent with hysterectomy. Ovarian cancer is not, strictly speaking, record consistent with hysterectomy because removing a uterus does not do anything for cancer in an ovary. However, existing medical coding policy requires picking one and one only primary diagnosis -- and it is the primary diagnosis that appears on the bill. Since it is possible that the ovarian cancer travelled down the reproductive system and caused cancer in the uterus, an end-user would not be surprised to see ovarian cancer as a cause for a hysterectomy.

The system creates two queries -- one upper bound with tests on the reliable data and a second lower bound query adding

---

[13]Conceivably, one could use a certainty factor calculus, but there is little understanding of what such figures mean when applied to atomic assertion, let alone entire relations. Analyzing this issue is beyond the scope of this dissertation.

constraints which test potentially inaccurate data. This pair of queries will generate a "logical confidence interval".

An example is given in Diagram X, Incomplete Joins -- Upper and Lower Bounds, in which I have taken the Barnes example of Diagram V in section 3.3.1 and have added information about sex and age. In Diagram X, we see my system's approach to dealing with the ambiguity when information is incomplete. Recall the David Barnes example of Diagram V where the system was unable to distinguish between a David Barnes record and a Fran Barnes record using only last name and date.

In this diagram, I show an upper bound based on tests of audited data and a lower bound based on tests of both audited and unaudited data. In the upper bound (Case IIIa), we see that the same improper retrieval persists -- given the audited information we have, we see that both records might be relevant. In the lower bound (Case IIIb), the unaudited information about sex and age is added. If all criteria match, we conclude that the record must be relevant. Here, the Fran Barnes record is rejected on those unaudited criteria and only the record that should have been retrieved is retrieved. Note, however, that if AGE or SEX in either record had been improperly recorded we would have arrived at the wrong answer.

Diagram X

Incomplete Joins -- Upper and Lower bounds

Doctor Bills	Hospital Bills
Lname Fname Date Proc Sex Age	*Sex Age Lname Fname Date Diagnosis*

Case III (from Diagram V)

Incomplete Key: We found David Barnes...plus somebody else

Barnes	9/17	prost		Barnes	9/17 Pcancer
Barnes	9/17	hyst		Barnes	9/17 Ucancer

Case IIIa -- Upper Bound

We accept David Barnes--and the other record we cannot reject

Barnes	9/17	prost		Barnes	9/17 Pcancer
Barnes	9/17	hyst		Barnes	9/17 Ucancer

Case IIIb -- Lower Bound

We accept David Barnes -- and reject the other on age & sex, unaudited criteria

Barnes	9/17	prost	M 60	M 60 Barnes		9/17	Pcancer
Barnes	9/17	hyst	F 57	F 57 Barnes		9/17	Ucancer

Because the audited criteria are incomplete the upper bound criteria may accept records that should not be retrieved. These improper retrievals would cause totals based on summing a particular numeric field in the retrieved records to be higher than the true total. Because the lower bound unaudited criteria may be inaccurate, one would expect some false rejections, which would cause totals based on summing a particular numeric field in the retrieved records to be lower than the true total.

The question then arises "For the set of possible comparison pairs, how can we divide these comparisons into upper bound and lower bound tests"?

In deciding which attributes to rely on in a retrieval, it would be appropriate to use knowledge of which attributes were more likely to be accurate. The need for accuracy in attribute values in transactions databases differs from attribute to attribute and is primarily driven by the real-world consequences of error. An error in the attribute corresponding to amount paid is more serious than an error in the SEX or AGE attributes in a billing system. One would expect to find more verification of amount paid than sex.

My criterion is to prefer pairs of AUDITED attributes, where I define an "audited attribute" as an attribute whose value is believed to be reliable because of the validation procedures in place to assure that accuracy. This audited attribute test is implemented by including audit status for each attribute in the data dictionary and passing that information along in the

comparison pairs. The containment test described above is implemented in the same way. The lower bound query is generated by retaining the tests on those unaudited attribute pairs.

3.6 Selecting a Comparison Method When Attributes Are Defined Differently

Another important issue is determining how to compare members of each comparison pair. This section discusses the rules for selecting a comparison method. The implementation of these comparison methods is discussed in section 4.4, Query Class Template Creation. This requires search knowledge, which is built into the implemented system. One of the following four rules will apply to each comparison pair:

Case 1: A rule producing an equijoin. This is the special case where both attribute domains are directly comparable so that one requires no knowledge of how to compare specific values.

```
IF domain A = domain B
THEN equijoin
```

Case 2: A rule producing a range check, where consistency is tested by an attribute's value or range of values lying inside the range or value of its comparison attribute. Checking for consistency requires knowledge of which attributes represent the beginning and duration of an interval and how to implement the comparison. This comparison method will most commonly be used for time comparisons, but could be used for any range.

```
If domain A = range
OR domain B = range
THEN range check
```

Case 3: A rule producing a table check, where consistency is tested by comparing the source attribute to a table of valid target attribute values. This requires identifying relevant values of the target attribute. The implemented system permits the user to specify both an upper bound and a lower bound list of valid target attribute values.

```
IF domain A not = domain B
AND domain A is discrete
AND domain B is discrete
THEN upper is upper bound table check
AND lower is lower bound table check
```

Case 4: A rule preventing a comparison when the two domains are incomparable. (This is an error trap)

```
IF domain A not = domain B
AND (domain A is continuous
AND domain B is discrete)
OR (domain A is discrete
AND domain B is continuous)
Then implement no test
```

Let me turn again to my surgery-requiring-hospitalization example. The attribute policyID is defined the same way in both databases. The system will apply the first rule and produce an equijoin. The same equijoin test will be included in both the upper and lower bound queries since the test is the same in both queries.

Surgery takes place on a specific day; hospital stays are a range of days. This comparison cannot be done with an equijoin. The implemented system will apply the second rule. The system

tests that the interval of the source record is on or before the end of the target interval and at or after the beginning of the target interval.

The implemented system applies the third rule when comparing PROC to DX (a WHAT test). Physician bills contain PROC (the service rendered) but not an attribute for diagnosis; the hospital bills have the reverse problem. This cannot be resolved with an equijoin either. However, we can check that the diagnosis is consistent with the surgical procedure if we have a table that records that relationship. Such a table can be created if both attributes are coded in a scheme known to the end-user and the list of relevant values is short enough for an end-user to enter the list when prompted (which happens only once, the first time the query is made. At other times, the values stored at initial entry are retrieved automatically. (see section 4.5, Transforming Query Class Templates into Executable Queries).

If one is doing the retrieval to sum an attribute the upper and lower bound limit on the sum is defined by those records which have passed all the tests. The lower bound will have a smaller total than the upper bound because the stricter tests lead to the exclusion of more records. The true value lies somewhere between the upper and lower bound totals. No statement can be made as to where on the interval the true value is; just that the true value is somewhere in the interval.

3.7 Probability of Accurate Retrieval

Let me now turn to a comparison between the ex ante and the retrofit approaches. In the traditional ex ante approach, the retrieval error rate is definitionally zero given an assumption of accurate attributes (that is, there are no data entry errors) and the presence of keys:

$$Pr(error | a_K \text{ in } R_A = a_k \text{ in } R_B) \equiv 0$$

where:
- R_A is relation A
- R_B is relation B
- a_K is the key needed to join relations A and B

We can think of a_K as being the output of a function which operates on the set of key-defining attributes needed to identify the object of the query. If the query were about individuals and an individual's complete name defined uniqueness, a trivial example would be the concatenation of FIRST_NAME and LAST_NAME. The Cartesian product of FIRST_NAME and LAST_NAME defines a compound key. If the query were about individuals' specific episodes of care, then uniqueness would be defined on a complete name and diagnosis, where diagnosis is in turn an inference on a large number of difficult to specify observations -- a difficult task.

We can operate directly on the attributes used by the key-defining function. Suppose $a_1 \ldots a_n$ defines U_{AB}, the set of attributes needed to produce a compound key. The probability of error if the membership of that set is known and those attributes are present in the views to be joined is definitionally zero.

More formally:

$$\pi_i Pr(error | a_i \text{ in } R_A = a_i \text{ in } R_B) = 0$$

If a member of U_{AB} were missing, we have an incomplete information problem with respect to the attributes needed to effect a join.

The probability of error is the joint probability of improperly retrieving a record if all the comparisons on available defining attributes were satisfied; if the values of those defining attributes were statistically uncorrelated we could write:

$$\pi_j Pr(error | a_{j,A} = a_{j,B}) = k$$

Where:
a_j is a member of U_{AB}

For example, suppose two attributes, a_1 and a_2, were defined as the compound key. If a_1, a_2 are both present in both databases and the values of both pairs match then Pr error = 0. If only a_1 is present in both databases and that pair matches, then the probability of error (false retrieval) is the probability that the unrecorded values of a_2 do not match. (If the assumption of no data entry errors holds, then the reverse problem of failing to retrieve a record when a_1 does match does not exist) More formally:

$$Pr(\{error | a_1=a_1^*\} >= Pr\{error | a_1=a_1^* \ \& \ a_2=a_2^*\}) = 0$$

where a_k* is the correct element of the domain of a_k. If our goal ends at retrieving the records, I can say nothing more about the effect of the error on the whole. If the goal were to add up the a_m's, the numeric measurement attributes in each of the records, we are interested in the relative effect the errors will have on the total. Since a retrieval will retrieve all the correct instances of a_m (where $a_1 = a_1^*$ and $a_2=a_2^*$) and will also perform an improper retrieval where $a_1 = a_1^*$ and a_2 not $= a_2^*$ there is a positive bias of:

$$Pr(error)*mean(a_m)$$

that is, the expected upwards bias is the expected value of the measurement attribute times the expected number of improperly retrieved records. What this bias actually is depends on the distribution of a_2 and the distribution of a_m, so we cannot conclude its size; only that it must be positive.

We can do better in two respects. First, we may be able to substitute a'_2 for a_2 where:

$$\text{for all } a_2 \text{ there exists an } a'_2 \;\&\; ||a_2|| >= ||a'_2||$$

where $||a_2||$ is the cardinality of the domain of a_2. Suppose we want to retrieve individual persons, but no individual ID is available. Suppose there were a family ID. Every individual is contained in a family, albeit with a family size of one if living alone. Using family reduces Pr(error) to:

$$Pr((error|a_1=a_1^* \;\&\; a'_2=a'_2)$$

Note that:

$$Pr(\{error|a_1=a_1^*\} >= Pr(\{error|a_1=a_1^* \ \& \ a'_2=a'_2)$$

and

$$Pr(\{error|a_1=a_1^* \ \& \ a'_2=a'_2\} >= Pr(error|a_1=a_1^* \ \& \ a_2=a_2^*)) = 0$$

In other words, the probability of false retrieval given a_1 matches and a_2' matches is the average relative cardinality of a_2' and a_2. The actual amount of this reduction is dependent on the probability distribution of A'_2 and $||a_2||/||a'_2||$[14].

Suppose, for example, that:

EPISODE = INDIVIDUAL x TIME x DIAGNOSIS

that is, EPISODE is a compound key based upon INDIVIDUAL, TIME, and DIAGNOSIS. Now suppose that an attribute corresponding to individual were not in the database but a family identifier was. Since each individual belongs to only one family, and there is a certain (smaller) probability of error if the names match. Suppose the database also contained an address. Address is not a perfect substitute for an individual identifier, since several people can live at the same address. The probability of error is then the probability that two individuals with the same name were living at the same address -- not impossible, but much less likely than if the only test were that two individuals had different names[15]. The system will infer this

---

[14]the cardinality of A_2 divided by the cardinality of A'_2. In this example this means the number of individuals divided by the number of families.

[15]There is, of course, the possibility that somebody would move during the event or that address were erroneously or ambiguously recorded.

properly because the only comparison pairs eliminated are the ones dominated out by the heuristics discussed above.

3.8 Storing and Reusing Retrofit Knowledge

The process described in this chapter need not be repeated for every query. For example, for retrieval purposes there is a lot in common between a hysterectomy and a prostatectomy -- or any other kind of physician procedure requiring hospitalization. Indeed, for all queries about physician procedures requiring hospitalization:

- the source and target databases would remain the same.
- the universal attribute types that need to be held consistent across the databases would remain the same.
- the relevant comparison pairs and their instantiation method would remain the same.

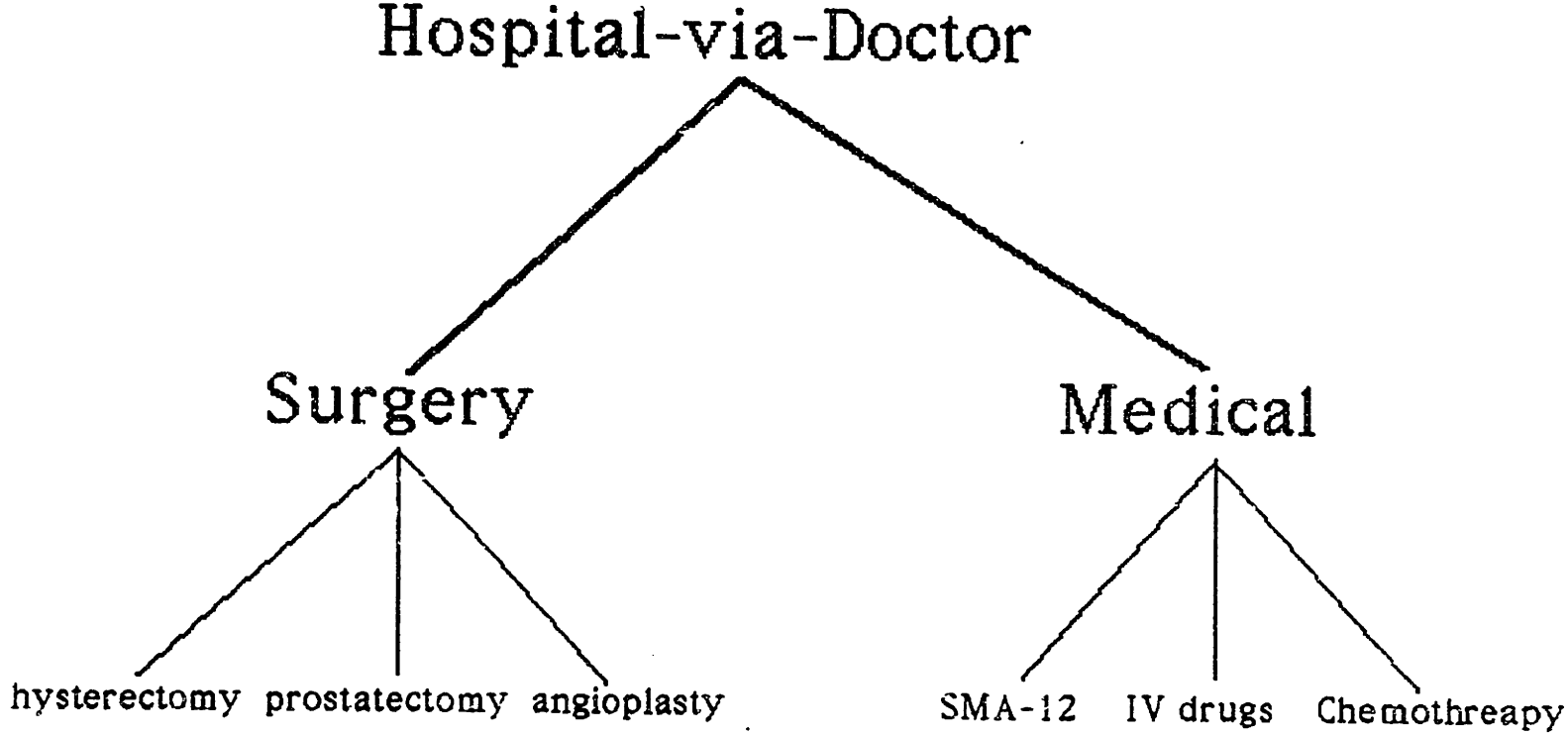
All queries generated to retrieve physician procedure requiring hospitalization and their associated hospital records will always be resolved by:

- physician as source database
- hospital as target database
- person/episode by the three universal attribute types of {BUYER, WHAT, WHEN} comparing:
  - policyID compared as an equijoin
  - DATE compared as a range
  - PROC compared to DX as a table look-up

The only thing that distinguishes queries in the same query class are the values used in table look-ups. I define a "template" as an executable query with a token for each set of needed table values. I define a query class as the queries which can be produced by swapping in the appropriate table values.

Diagram XI, Query Classes and Queries, makes this relationship between query classes and queries explicit. Here, we see that a number of different queries sharing the characteristic of physician records as source and hospital records as target are members of the hospital-via-doctor class, at the top of the diagram.

Diagram XI
Query Classes and Queries



The hospital-via-doctor class is the most general class possible via that particular access path -- the doctor database is the source; the hospital database is the target. If one could find the relevant source records and if the universal attribute types and their comparison pairs were sufficiently discriminating then one could find anything in the target database.

These conditions do not necessarily hold for the most general class of queries but may be true for a semantically-cohesive subclass. Such is suggested in the diagram by dividing of hospital-via-doc into highly-discernable and less-discernable subclasses, medical and surgical. There are relatively few reasons why anybody would have a given surgical operation. There are many reasons for having, say, a chest x-ray. It makes sense to explicitly create only the subclass that works well (surgery).

The question then arises as to why one would establish this intermediate step of creating a class rather then creating each query from scratch. The primary reason is that starting from scratch is a waste of both computer and human resources. Consider the frequency with which various kinds of knowledge need to be loaded into the system:

Search knowledge: The heuristics for selecting comparison methods, for requesting query attributes need only be entered once. (In my implementation these are hardwired into the software.)

Language knowledge: The metatemplates (also defined and discussed at length in Chapter 4) which implement comparison method choices into an executable language need only be entered once per language. As my implementation produces queries only in SQL, these are also hardwired. It would not be difficult to create several sets, one for each language with a menu choice providing for output in different languages.

Database knowledge (contents of specific schemas): The required data dictionary and universal attribute type information need be entered only once per database pair. The universal attribute types need be defined only once per application type. The cross-tabs (preferred comparison pairs) also need only be done once per database pair and can be stored for future reference (and are stored in my implementation).

Query class level subject knowledge: Knowledge about what attribute types are relevant, what databases contain relevant information, and what the goal of the query is need be entered only once per class of semantically homogeneous queries. Once this knowledge is entered and the above knowledge is also present, I can produce a template which is executable but for the absence of sets of values for specific queries needed for source and table checks. In my implementation, such a template is prepared and a menu is automatically updated to tell the user of its existence. Doing so not only makes it more convenient when asking new queries -- a user can use an existing template if it fits the question -- but it also

reduces the response time since there is no need to redo done work.

Query level subject knowledge: The table values required to convert a template into an executable query are query-specific. However, they can be stored and reused as well.

The construction and use of templates is discussed in section 4.4, Query Class Template Creation, of the Implementation chapter.

Chapter 4: Implementation

4.1 Overview

I tested this theory by building a system which will produce SQL queries whose execution answers summary questions about the databases (sum, avg, etc). In the next chapter, I run three such queries against a real database (the MIT Blue Cross transactions log) and a hypothetical database.

This prototype is limited to implementing record-level consistency tests[16] and is not dealing with any distributed database issues. The "databases" are relations within the same INGRES database on the same computer.

The system has several distinct tasks to carry out, each discussed in a different section:

- in Section 4.2, Database Knowledge Acquisition explains how the database analyst enters the needed additional information about the database into a data dictionary and how the system then infers potential comparisons from that augmented data dictionary.
- in Section 4.3 Subject knowledge acquisition shows how the end-user enters needed subject knowledge into the system.
- in Section 4.4 Query Class Template Creation demonstrates how the system uses the above to create a template for the query class.

---

[16]I am not implementing any group contents or order tests of consistency.

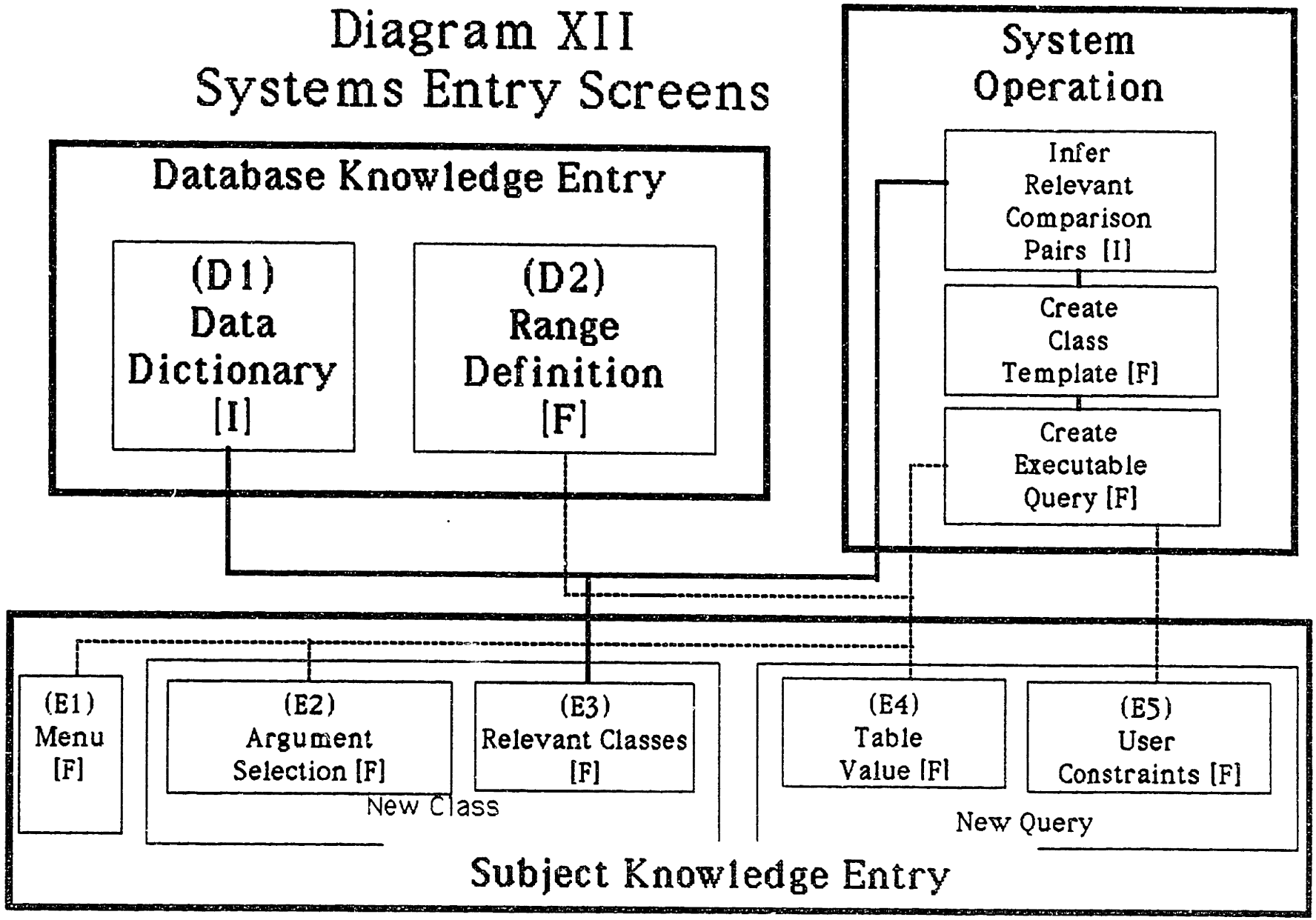
- in Section 4.5 Transforming Query Class Templates into Executable Queries discusses acquisition of query-specific table values for the query class template.

Let me now turn to an overview of the implemented system. I used two different development environments, one on the database side and one on the subject knowledge side. On the database side, the actual database and the data dictionary was built in Ingres-PC, the personal computer commercial version of an influential academic relational database system. Ingres is used to manage the large actual database, to store the data dictionary, and to produce preferred comparison pairs from the data dictionary.

On the subject knowledge side I selected Framework, an environment with interface development facilities superior to those found in Ingres. The user sees a menu which is modified as additional query classes and queries are created.

Diagram XII, System Entry Screens, shows the entry screens in the system, grouped by type of knowledge. Database administrators are expected to know about database contents and use the Data Dictionary Entry Screen [D1] to enter information into the data dictionary and the Range Definition Entry Screen [D2] to enter the names of atomic attributes which define range attributes (defined in section 4.2, Data Knowledge Acquisition).

Diagram XII
Systems Entry Screens



End-users select desired actions from the Menu [E1]. When creating new query classes, end-users select query goals on the Argument Selection [E2] screen and list the universal attribute types with the Relevant Types [E3] screen. End-users also need to enter the permitted values for those comparison pairs resolved by a table look-up, which is done with the Table Value Entry Screen [E4]. Finally, end-users may wish to enter some constraints which are unrelated to the join (such as choosing a particular period of time. This is done by the User Constraints Entry Screen [E5]. Examples of all the above screens are in the sections discussing their use in depth.

The system knows the local query language and how to use the database and subject knowledge entered above to produce templates and executable queries. The system knowledge was entered into the system at system development time by the author and is not modified through the use of the system.

To produce a class template the system needs to know:

- how to best carry out the comparisons which requires the search knowledge built into the system
- how to find the relevant records in the source database records which requires information collected via the Argument Selection Entry Screen [E2] and the Table Value Entry Screen [E4]
- User constraints collected from the User Constraints Entry Screen [E5]

- The names of the databases containing the data and how the retrieved data is to be summarized. This requires information collected from the Argument Selection Entry Screen [E2]

Creating a query from a class template requires table values specific to the query. These values are collected by the Table Value Entry Screen [E4].

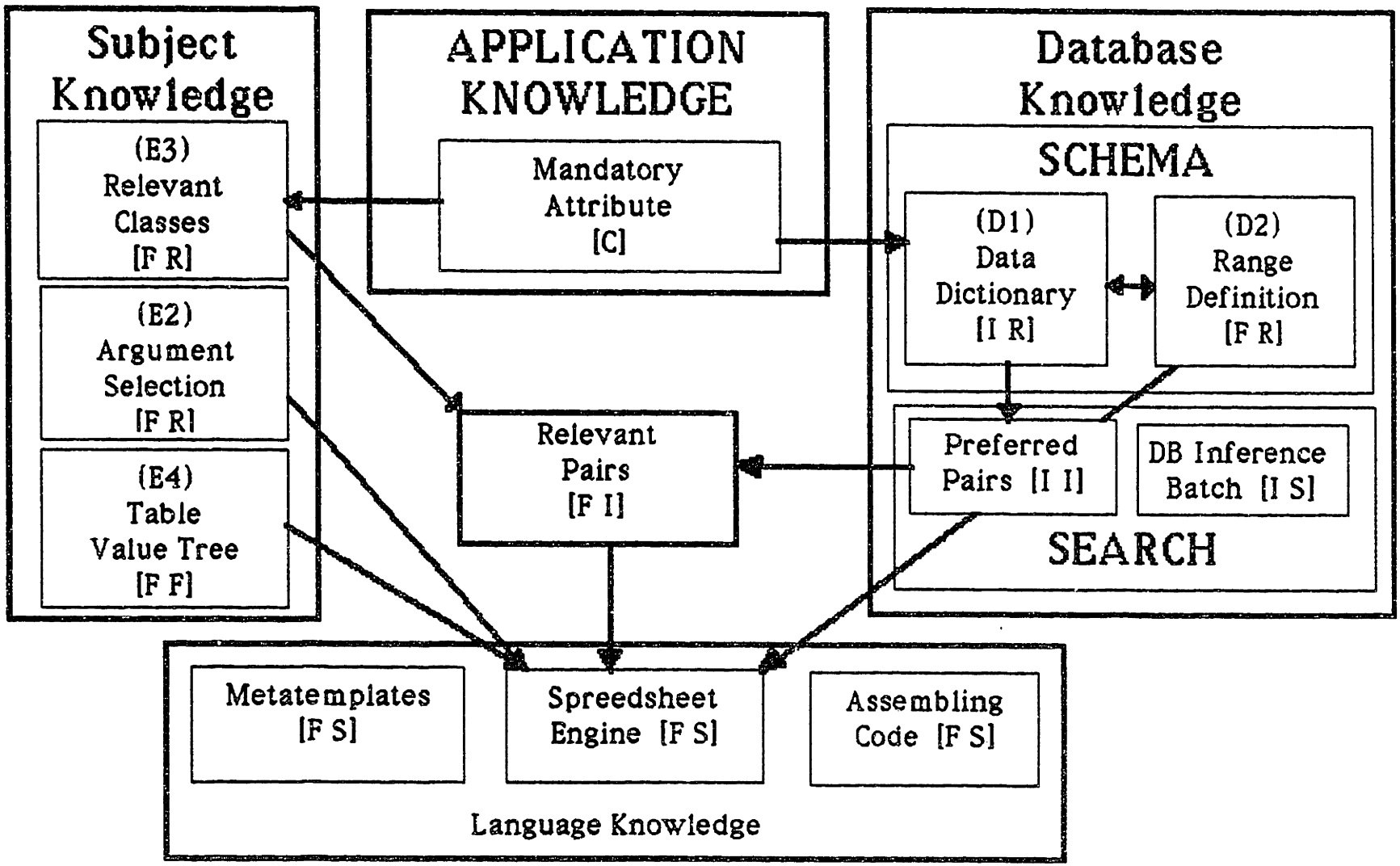
Diagram XIII, System Storage Structures shows system organization with both actual and "conceptual" storage mechanisms. I will start in the middle, where agreement must be reached on what the universal attribute types are. This is marked C for conceptual as it is up to the system user to define and consistently use universal attribute types. The system has no facilities for enforcing universal attribute class integrity.

On the database side the Data Dictionary is a relation in Ingres (marked I R for Ingres Relation). Preferred Pairs (I I for Ingres Inferred) are produced by executing a batch file of database queries (DB inference batch, marked I S for Ingres System). These queries take a cross-product of the data dictionary, filter out redundancies and make sure that attribute pairs are in the same class and in different databases. This happens once for each pair of databases.

Relevant pairs (F I, Framework Inferred) are the subset of those preferred comparison pairs whose class is in the Relevant Class table (F R for Framework Relation). These relevant pairs

are then processed into a query template. This routine collects needed information from the Argument Selection relation (F R) and creates the root of what will become a tree of frames to record table values (marked F F).

Diagram XIII
System Storage Structures



4.2 Database Knowledge Acquisition

The database-specific steps are:

1. Record universal attribute class and audit status for all attributes in all databases
2. Determine whether any pairs of actual attributes define intervals and record those pairs as "range" attributes

The first step is to enter the following for each data attribute via the Data Dictionary Entry Screen [D1]:

- Universal attribute class, defined above.
- Database membership, the name of the attribute's database.
- Attribute type, the name of the attribute's domain.
- Audited status, indicating whether the attribute has been audited.
- Domain cardinality, the number of actual (not theoretical) values. (For example, the theoretical number of account numbers may be infinity, the possible number of account numbers may be 2^{32} in a 4-byte account number attribute, but in practice there may be only 5,000)
- Contained status, indicating whether the attribute is an atom or can be built up from other attributes of the same class (such as families being constructed from individuals)

The Data Dictionary Contents Report reports the complete set of data attribute information. For my example medical databases,

this yields exhibit [D3]. The second step is to identify pairs of attributes which define continuous intervals. For example, consider a hospital stay. Database systems generally do not support the concept of an interval. A hospital stay is represented in the hospital database as two attributes recording the start and duration of the interval -- HOSP.DATE and HOSP.LOS (Length of Stay). The analyst defines a compound attribute I've chosen to call STAY. He then puts the names of the start and duration attributes into the Range Definition Entry Screen [D2].

The system then infers the pairs of potentially comparable attributes by executing a query which joins the data dictionary to itself. This results in a cross-product containing n^2 pairs of attributes.

These n^2 pairs are then trimmed down by:

- eliminating pairs in the same database,
- eliminating pairs of different attribute types, and
- eliminating pairs which are identical, save for attribute order (a cross product will create a duplicate for each combination)

Additional redundancy can be eliminated by adding criteria for:

- preference for audited constraints.
- preference for contained attributes.

D1 DATA DICTIONARY ENTRY SCREEN

Query Target Name is datadic

TABLE IS datadic

class: seller dbname: doc attname: MDID

domain: discrete audited: y contained: n

cardinality: 5000

D3 Database Dictionary Contents

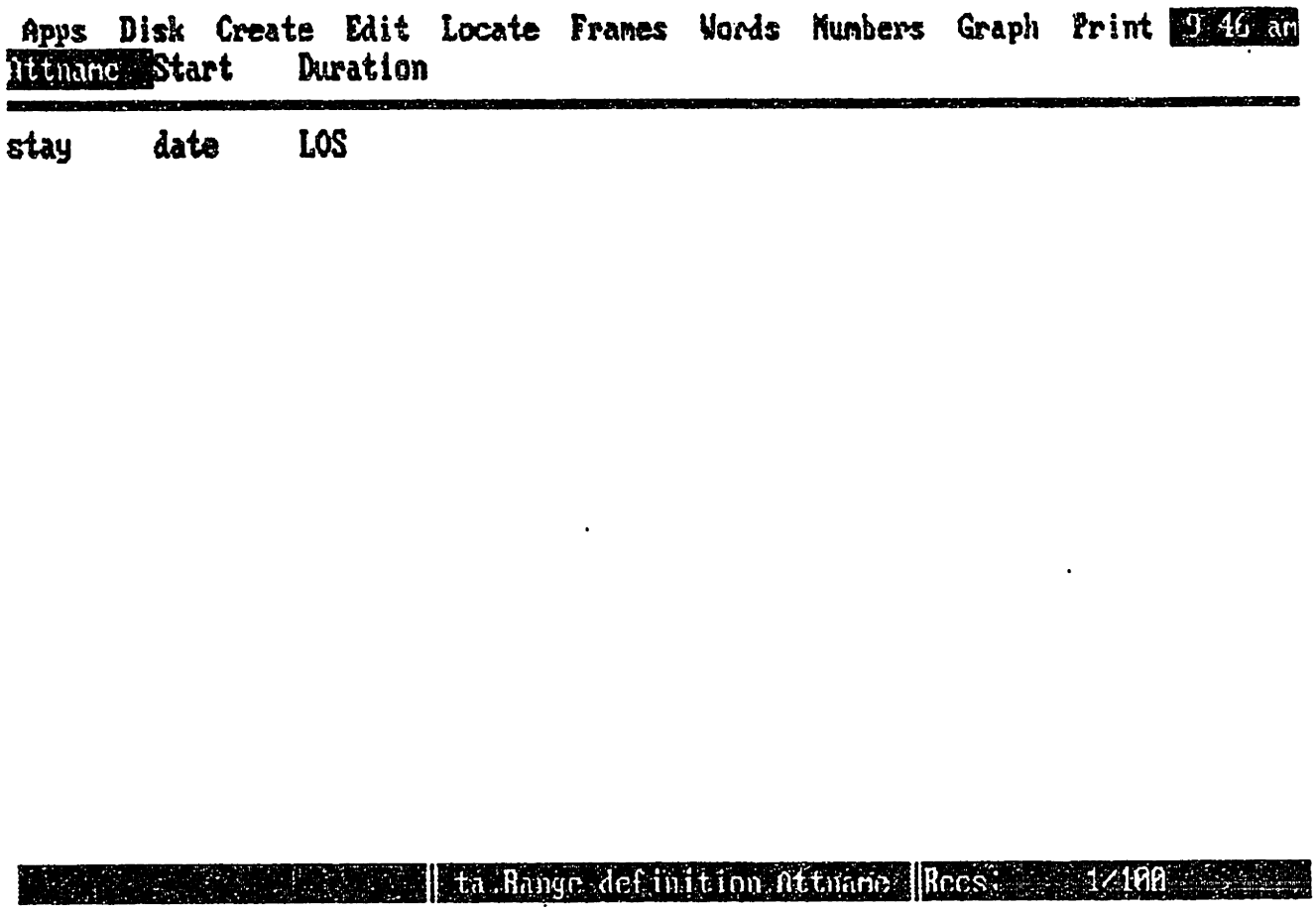
___ Output _____

```
>select distinct class, dbname, attname, domain, audited,
>contained, cardinality
>from datadic2 order by class,attname;
```

class	dbname	attname	domain	audite	contai	cardinality
						0
$	hosp	ancbill	numeric	y	n	99999
$	doc	charges	numeric	y	n	99999
$	dbname	payment	numeric	y	n	99999
$	hosp	payment	numeric	y	n	99999
$	hosp	roombill	numeric	y	n	99999
buyer	doc	age	numeric	n	n	100
buyer	hosp	age	numeric	n	n	100
buyer	hosp	birthdt	numeric	n	n	36500
buyer	doc	cert	discrete	y	n	5000
buyer	hosp	cert	discrete	y	n	5000
buyer	hosp	contype	discrete	n	y	9
buyer	doc	groupid	discrete	y	y	4
buyer	hosp	groupid	discrete	y	y	4
buyer	doc	relship	discrete	n	n	6
buyer	doc	sex	discrete	n	n	2
buyer	hosp	sex	discrete	n	n	2
buyer	hosp	sexrel	discrete	n	n	6
seller	doc	mdid	discrete	y	n	5000
seller	doc	mdspec	discrete	n	n	50
seller	hosp	provcode	discrete	y	n	5000
what	hosp	admtype	discrete	n	n	9
what	doc	contype	discrete	n	n	20
what	hosp	dx	discrete	y	n	3000
what	hosp	iocode	discrete	y	n	2
what	doc	misc	discrete	n	n	90
what	doc	proc	discrete	y	n	8000
what	doc	procct	numeric	y	n	10
what	doc	sercode	numeric	n	n	50
what	doc	sertype	discrete	n	n	50
when	hosp	admday	discrete	n	n	7
when	doc	date	numeric	y	n	999999
when	hosp	date	range	y	n	365
when	hosp	LOS	numeric	n	n	9999

End of Request - 34 Rows

D2 Range Definition Screen



The above is produced by the following SQL statement

```
CREATE TABLE pcp AS
SELECT S.type,S.dbname, S.attribute-name, S.audited,
S.contained,T.dbname,T.attribute-name,
T.audited,T.contained
FROM datadic S, datadic T
WHERE S.type = T.type
AND S.dbname not = T.dbname
AND S.dbname = sourceDB
AND S.contained = "n"
AND T.contained = "n"
AND S.audited = "y"
AND T.audited = "y"
```

where "pcp" represents preferred comparison pairs.

The results of this query against the data dictionary containing physician and hospital data is attached. Two versions -- one without the audit criteria (Possible Comparisons Table [D4]) and one with (Preferred Comparisons table [D5]). Here, we see that:

- PolicyID comparison is used in both tests because both sides are audited.
- The AGE and SEX comparisons are in the lower bound test only because those attributes are not audited (and are, therefore, not in the preferred list).
- The groupID comparison does not appear because groupID contains policyID and contributes no additional information.
- Incomparables such as AGE:SEX and SEX:AGE comparisons were eliminated by hand[17].

[17]Note the suggestion in section 3.5 for a heuristic that would reduce the number of incomparables if the names were compatible.

D4 Possible Comparison Pairs

```
___ Output _____________________________________________
>select
>s.class, s.dbname, s.attname, s.audited, s.contained,
>         t.dbname, t.attname, t.audited, t.contained
>from datadic2 s, datadic2 t
>where s.class = t.class
>and not s.dbname = t.dbname
>and s.dbname = "doc";
```

class	dbname	attname	audite	contai	dbname	attname	audite	contai
buyer	doc	cert	y	n	hosp	cert	y	n
buyer	doc	cert	y	n	hosp	sex	n	n
buyer	doc	cert	y	n	hosp	birthdt	n	n
buyer	doc	cert	y	n	hosp	age	n	n
buyer	doc	cert	y	n	hosp	sexrel	n	n
buyer	doc	cert	y	n	hosp	groupid	y	y
buyer	doc	cert	y	n	hosp	contype	n	y
buyer	doc	age	n	n	hosp	cert	y	n
buyer	doc	age	n	n	hosp	sex	n	n
buyer	doc	age	n	n	hosp	birthdt	n	n
buyer	doc	age	n	n	hosp	age	n	n
buyer	doc	age	n	n	hosp	sexrel	n	n
buyer	doc	age	n	n	hosp	groupid	y	y
buyer	doc	age	n	n	hosp	contype	n	y
buyer	doc	sex	n	n	hosp	cert	y	n
buyer	doc	sex	n	n	hosp	sex	n	n
buyer	doc	sex	n	n	hosp	birthdt	n	n
buyer	doc	sex	n	n	hosp	age	n	n
buyer	doc	sex	n	n	hosp	sexrel	n	n
buyer	doc	sex	n	n	hosp	groupid	y	y
buyer	doc	sex	n	n	hosp	contype	n	y
when	doc	date	y	n	hosp	date	y	n
when	doc	date	y	n	hosp	admday	n	n
when	doc	date	y	n	hosp	outdate	n	n
what	doc	proc	y	n	hosp	dx	y	n
what	doc	proc	y	n	hosp	iocode	y	n
what	doc	proc	y	n	hosp	admtype	n	n
seller	doc	mdid	y	n	hosp	provcode	y	n
seller	doc	mdspec	n	n	hosp	provcode	y	n
what	doc	contype	n	n	hosp	dx	y	n
what	doc	contype	n	n	hosp	iocode	y	n
what	doc	contype	n	n	hosp	admtype	n	n
what	doc	sertype	n	n	hosp	dx	y	n
what	doc	sertype	n	n	hosp	iocode	y	n
what	doc	sertype	n	n	hosp	admtype	n	n
what	doc	procct	y	n	hosp	dx	y	n
what	doc	procct	y	n	hosp	iocode	y	n
what	doc	procct	y	n	hosp	admtype	n	n
$	doc	charges	y	n	hosp	roombill	y	n
$	doc	charges	y	n	hosp	ancbill	y	n
$	doc	charges	y	n	hosp	payment	y	n
$	doc	charges	y	n	dbname	payment	y	n
what	doc	misc	n	n	hosp	dx	y	n
what	doc	misc	n	n	hosp	iocode	y	n
what	doc	misc	n	n	hosp	admtype	n	n
buyer	doc	relship	n	n	hosp	cert	y	n
buyer	doc	relship	n	n	hosp	sex	n	n
buyer	doc	relship	n	n	hosp	birthdt	n	n
buyer	doc	relship	n	n	hosp	age	n	n
buyer	doc	relship	n	n	hosp	sexrel	n	n
buyer	doc	relship	n	n	hosp	groupid	y	y

D4 Possible Comparison Pairs

buyer	doc	relship	n	n	hosp	contype	n	y
buyer	doc	groupid	y	y	hosp	cert	y	n
buyer	doc	groupid	y	y	hosp	sex	n	n
buyer	doc	groupid	y	y	hosp	birthdt	n	n
buyer	doc	groupid	y	y	hosp	age	n	n
buyer	doc	groupid	y	y	hosp	sexrel	n	n
buyer	doc	groupid	y	y	hosp	groupid	y	y
buyer	doc	groupid	y	y	hosp	contype	n	y
what	doc	sercode	n	n	hosp	dx	y	n
what	doc	sercode	n	n	hosp	iocode	y	n
what	doc	sercode	n	n	hosp	admtype	n	n

End of Request - 62 Rows

D5 Preferred Comparison Pairs

```
___ Output _____________________________________________
>select
>s.class, s.dbname, s.attname, s.audited, s.contained,
>         t.dbname, t.attname, t.audited, t.contained
>from datadic2 s, datadic2 t
>where s.class = t.class
>and not s.dbname = t.dbname
>and s.dbname = "doc"
>and s.contained = "n"
>and t.contained = "n"
>and s.audited = "y"
>and t.audited = "y";
```

class	dbname	attname	audite	contai	dbname	attname	audite	contai
buyer	doc	cert	y	n	hosp	cert	y	n
when	doc	date	y	n	hosp	date	y	n
what	doc	proc	y	n	hosp	dx	y	n
what	doc	proc	y	n	hosp	iocode	y	n
seller	doc	mdid	y	n	hosp	provcode	y	n
what	doc	procct	y	n	hosp	dx	y	n
what	doc	procct	y	n	hosp	iocode	y	n
$	doc	charges	y	n	hosp	roombill	y	n
$	doc	charges	y	n	hosp	ancbill	y	n
$	doc	charges	y	n	hosp	payment	y	n
$	doc	charges	y	n	dbname	payment	y	n

```
End of Request - 11 Rows
_____________________________________________________
```

To sum up, in this section I have discussed the database knowledge needed by the system to find preferred comparison pairs and how that knowledge gets into the system. Further, this section has walked through the process of inferring the preferred comparison pairs from that database knowledge. I turn to the acquisition of subject knowledge in the next section.

4.3 Subject Knowledge Acquisition

If a particular query class has not already been created, an end-user needs to define:

- What to retrieve, information corresponding to the SELECT clause of an SQL query.
- The names of the source and target databases, information corresponding to the FROM clause.
- The attribute identifying relevant records in the source database.
- The relevant universal attribute types.

The end-user will see the following screens in new query class creation:

- Menu (screen E1), which displays any previously defined query classes, queries within those classes, and options for creating new query classes and new queries. This sample screen shows the menu in its initial state.
- Argument Selection Entry Screen (screen E2), which asks for the name of the query class being created, the

summarization function (SUM), the attribute to be summarized (PAYMENT), the names of the databases involved ("doc" and "hosp" in this example), and the name of the attribute in the source database which identifies relevant source records (PROC).

- Relevant Types Entry Screen (screen E3), which asks the names of the universal attribute types relevant to the query class.

In this example, let us suppose that the user wants the answer to the question "How much was spent on hysterectomies in 1985".

Since no such class/query combination is displayed on the menu, the user starts by selecting "New Class" from the menu, screen [E1]. The system will first ask for the name of the new class by displaying a prompt at the bottom of the screen. After the user supplies a name followed by pressing the return key the system creates:

- A new menu entry for the class.
- Empty frames to contain the class's templates and tables.

The system then displays the Argument Selection Entry Screen [E2], which requests the user to specify:

- query class, the name of the class being created
- Summary function (SUM)
- Name of the summary function's argument (PAYMENT)

- The source database (the database that contains a_q, which is doc in this example)
- The target database, i.e. the database without the course attribute (Hosp)
- a_q, the attribute in the source database which identifies relevant source records (PROC in this example)

The above information is needed to produce the SELECT and FROM clauses in an SQL query (or their equivalent in other database languages).

The system then displays the Relevant Types Entry Screen [E3]. The user enters the names of the universal attribute types which must be consistent across databases. I have referred to these as the relevant universal attribute types. Suppose one wished to infer the patient connection between a physician payments and a hospital payments log. Recall that the five universal attribute types in accounting are {BUYER, SELLER, WHAT, WHEN, MONEY} and that both of the databases we are trying to join are accounting databases.

E1 Menu

7:47 pm

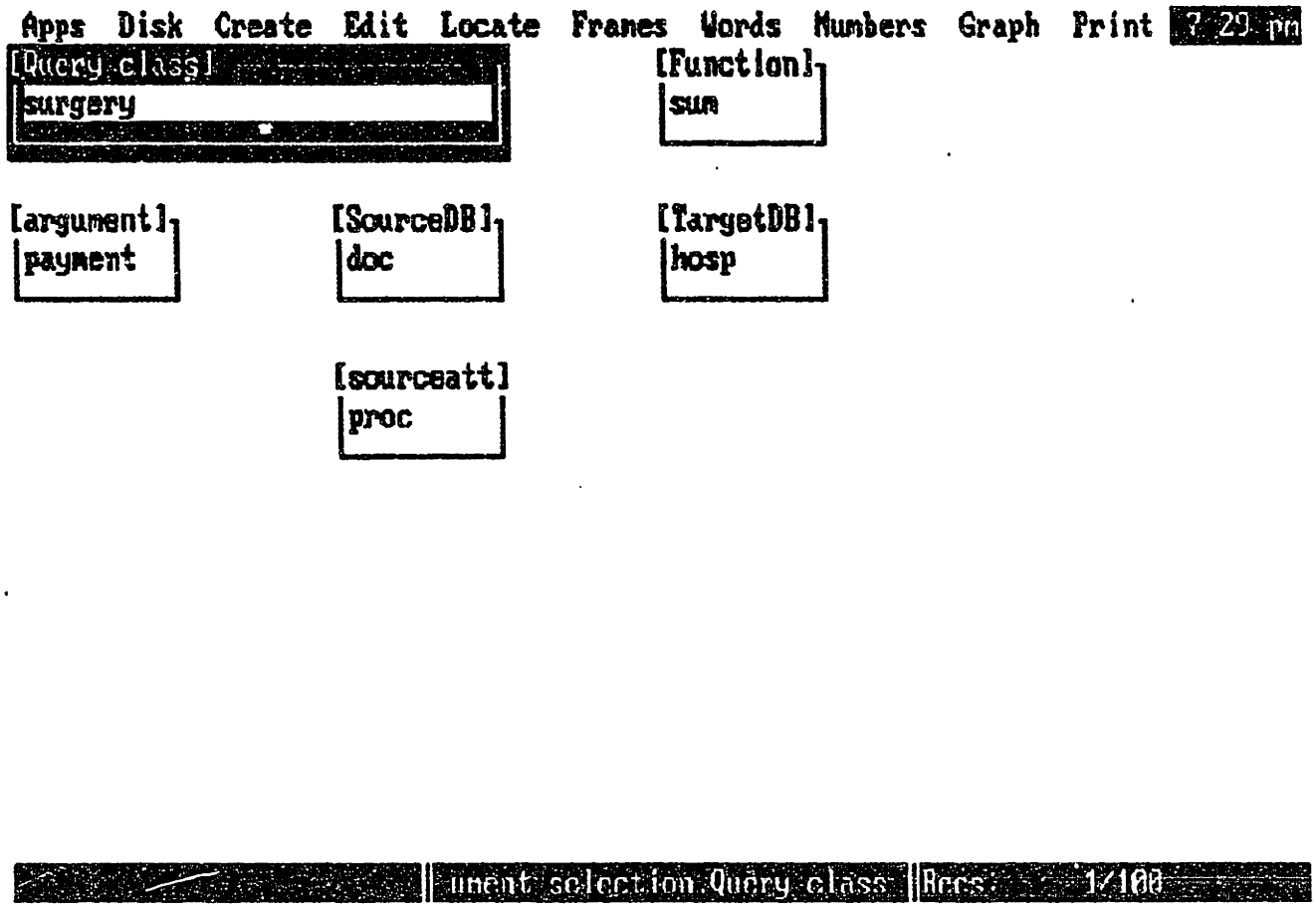
Menu:

New Class

Existing Class

Quit

E2 Argument Selection Screen



The relevant universal attribute types are:

- o BUYER, since the individual operated upon should be the same individual admitted to the hospital
- o WHEN, since the surgery should take place while the patient is in the hospital
- o WHAT, since the surgical procedure should be logically consistent with the hospital services.

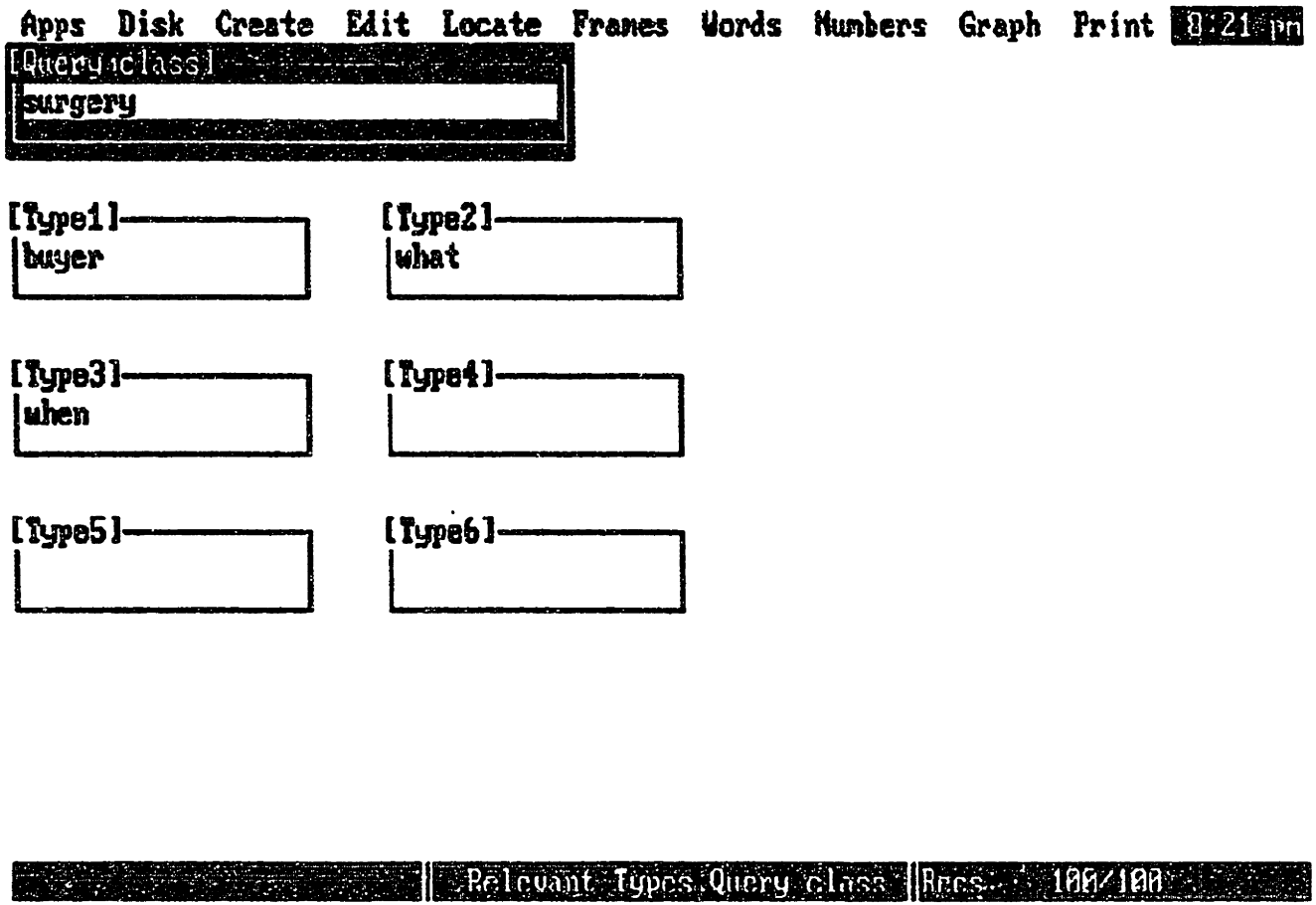
Irrelevant types would be:

- o SELLER, since there is no reason why the identity of the doctor should be the same as the identity of the hospital.
- o MONEY, since there is no reason why the physician's bill should be for the same amount of money as the hospital bill.

The above is sufficient information to prepare a template for the class. However, if any of the tests were implemented by table lookup, the end user would need to enter the appropriate table values. The system asks the end-user if he would like to proceed directly to creating a query for the newly-created class immediately after the system has prepared a class template. An end-user is likely to wish to do so, as a template requires at least one query to become useful. In addition, an end-user may select "New Query" from the system menu, which would be used when an end-user wants to define a new query on an existing class. End-users are prompted for needed table values via the Table Value screen [E4]. The Table Value screen

will be presented twice for each attribute which requires a table lookup, once for upper bound values and once for lower bound values with the message at the bottom of the screen changing appropriately. (An example of the Table Value Entry Screen is in section 4.5.)

E3 Relevant Types Entry Screen



4.4 Query Class Template Creation

4.4.1 Query Class Template Creation Overview

This section discusses the mechanics of creating a class template. The system assembles a query one clause at a time. The general form of an SQL query in Backus-Naur form is:

```
query ::= SELECT function ( argument )
          FROM   [db1,db2...]
          WHERE predicate

predicate ::=
      condition
    | condition AND predicate
    | condition OR predicate
    | NOT predicate
```

In the specific context of this dissertation, there are five specific kinds of conditions:

```
condition ::=
      equijoin
    | range
    | table.upper
    | table.lower
    | user
```

This might be seen as a specific form of the questions:

```
SELECT What you are looking for
FROM where the information is stored
WHERE these conditions are met
```

The system generates four SQL queries. Two will retrieve the upper and lower bound relevant records from the source database. The other two join the target database with the relevant source records.

This follows the general form:

```
query-pair ::=        source-query
                      target-query

target-query ::=      target-header
                      target-predicate
```

I have written what I will call "metatemplates". These are a series of templates which resolve to a clause in a query once the underlying attribute names are known. The following metatemplates have been written for the system:

- source-query, which produces the first query in the pair, the one that retrieves records from the source database.
- target-header, which produces the SELECT & FROM clauses in the target database query, as well as the subquery identifying relevant source records[18].
- equijoin, an additional WHERE clause to test equality of the attributes' values.
- range, an additional WHERE clause testing inclusion in a continuous interval.
- table, an additional WHERE clause membership in a set of permissible values.

---

[18]The source query is needed twice, once to add and once as a subquery in the target query. There is an obvious opportunity for performance optimization here.

After the Argument Selection screen is filled in, the source-select and target header metatemplates are resolved. These metatemplates are defined as follows:

source-query ::=
SELECT function (argument)
FROM sourceDB
WHERE a_q in (a_q-list)

target-header ::=
SELECT function (targetDB . argument)
FROM sourceDB , targetDB
WHERE sourceDB . a_q in (a_q-list)

where:
- function is the summary function to be applied to argument
- argument is the attribute in the database whose summarization is desired
- sourceDB is the database containing a_q
- a_q is the attribute in the source database whose domains enable identification of relevant records
- a_q-list is the name of the frame containing E_q*, the list of relevant values of a_q
- targetDB is the database without the key we are trying to retrieve relevant records from

function ::=
COUNT ¦ SUM ¦ AVG ¦ MAX ¦ MIN

The example question, retrieving hysterectomy-associated costs, can be seen as a special case of retrieving surgery-related costs. I will now move on to adding comparison metatemplates to complete the process of template creation.

4.4.2 Metatemplate Selection Rules

The system will have previously loaded the list of Preferred Comparison Pairs inferred by Ingres. The system now retrieves the subset of comparison pairs which have the same type as those listed as relevant (WHAT, WHEN and BUYER in this example). For this database this yields:

Type	Aud	S.DB	S.att	S.dom	T.DB	T.att	T.dom
BUYER	n	DOC	Age	num	Hosp	Age	num
BUYER	n	DOC	Sex	disc	Hosp	Sex	disc
WHEN	y	DOC	Date	num	Hosp	Stay	range
WHAT	y	DOC	Proc	disc	Hosp	DX	disc
BUYER	y	DOC	policyID	disc	Hosp	policyID	disc
SELLER	y	DOC	MD_ID	disc	Hosp	Hosp_ID	disc
MONEY	y	DOC	Payment	num	Hosp	Payment	num

Where:

- S.DB is the name of the source database
- S.att is the name of an attribute in the source database
- T.DB is the name of the target database
- T.att is the name of an attribute in the target database
- Type is the name of the attribute type to which both attributes belong[19]
- S.dom is the domain of the source database attribute
- T.dom is the domain of the target database attribute
- num is an abbreviation of numeric
- disc is an abbreviation of discrete

For each of these comparison pairs the system now needs to:

- select a comparison method
- instantiate the metatemplate associated with that method and append the result to the class template under construction
- add storage frames for table values, if needed

---

[19]The attribute type is constrained to be the same for both attributes, so it does not matter whether S.class or T.class is specified in the query producing the table.

I will now discuss the rules for selecting each of these comparison methods.

The rule for selecting an equijoin is:

```
IF       both attributes in the pair have the same name
AND both attributes in the pair have the same domains
AND neither attribute in the pair is a range attribute
THEN     implement the comparison as an equijoin
```

The rule for selecting a range comparison is:

```
IF       one or both of the attributes is defined as range
THEN     implement the comparison as a range
```

The rule for selecting a table look-up is:

```
IF       the attributes have different domains
AND the attributes are discrete-valued
THEN     implement comparison as a table lookup
```

4.4.3 Formal Definitions of Comparison Metatemplates

This section provides a formal definition of the metatemplates which implement comparison clauses. First, I simplify notation by defining:

- sourcename ::= sourceDB . sourceatt
- targetname ::= targetDB . targetatt

```
condition ::=
        equijoin
    |   range
    |   table.upper
    |   table.lower
    |   user
```

Then:

```
equijoin ::=     AND sourcename   =   targetname
```

```
range ::= AND  sourcestart <= targetfinish
               AND  sourcestart >= targetstart

table.upper ::=  AND   targetname     IN (
                 TEMPLATES.ACTIVE.UPPER.attname    )

table.lower ::=  AND   targetname     IN (
                 TEMPLATES.ACTIVE.LOWER.attname    )
```

User constraints are described in section 4.5, Transforming Query Class Templates into Executable Queries.

In this medical example, policyID is defined the same way in both databases and has the same name. This becomes:

```
AND doc.policyID = hosp.policyID
```

Since policyID is audited in both databases this test is the same in both bounds. The system appends the result to both the upper and lower bound templates.

The date:stay pair is done as a range because stay is defined as a range. For each member of a comparison pair whose domain is a range the system retrieves the names of the atomic attributes defining the range attribute(s) from the Range Definition table (in Framework). This becomes:

```
AND doc.date <= hosp.adm-date + LOS
AND doc.date >= hosp.adm-date
```

The same form is used for both the lower and upper bound tests.

The third comparison pair, doc.PROC and hosp.DX are both defined as discrete, but have different names. This is resolved as a table condition. This becomes:

```
AND   hosp.DX  IN (  TEMPLATES.ACTIVE.UPPER.DX )
   (applies to the upper bound template)

AND   hosp.DX  IN ( TEMPLATES.ACTIVE.LOWER.DX )
   (applies to the lower bound template)
```

It is also necessary to add new frames to contain lower and upper bound permissible values, which will be filled in later.

4.4.4 Query Class Template assembly

The metatemplates are processed by straightforward string operations. In this application, what we want is one frame whose formula, when recalculated, creates a result which can be exported as an ASCII file and then executed by Ingres.

When a particular metatemplate is needed, the corresponding attribute names are substituted in. The result is appended to the upper and/or lower bound frames which will come to contain the complete template pair. In this way, the upper and lower bound templates are constructed one clause at a time. When it comes time to use the templates, appropriate table values are retrieved and substituted in.

The following pseudocode will assemble an appropriate template from metatemplates:

```
Ingres
   start Ingres
   do data dictionary cross-product
   export Preferred Comparison Pairs view as ASCII
Framework
   start Framework
   load retrieval system frame
   activate menu
   user selects New Class from menu
   system asks name of class
   cause control to transfer to next module if F3 pressed
   In menu
       create entry with name of new class
            create child named New Query
            make copy of new query code with hardwired
            classname
            instantiate New Query code with name of new class
            move copy to formula area of New Query just
            created
            delete copy
```

```
In templates
    create entry with name of new query class
         create child named "upper" for template
         create child named "lower" for template
         create child named "tables for table value names
              create child named "upper"
              create child named "lower"
point at empty row in Argument Selection, prompt and zoom
after user presses F3
    instantiate source-query metatemplate
    instantiate target-header metatemplate
    create child with the name of the query attribute for
    both "upper" and "lower" in the table tree
    append source and header to upper and lower templates
point at empty row in Relevant Type, prompt and zoom after
user presses F3
transfer values in current Relevant Type row to global
storage
recalculate Preferred Comparison Pairs table, selecting
subset of rows with needed type
For each record in the uploaded relevant comparisons table
where both attributes are audited
Case of
    equijoin
         recalculate equijoin metatemplate.
         append result to temporary upper and lower bound
         templates
    range
         recalculate range metatemplate
         append result to temporary upper and lower bound
         templates
    recalculate upper and lower bound metatemplates
         append upper bound result to temp upper bound
append lower bound result to temp lower bound
         attach new upper and lower table frames to table
         tree for query class
Endcase
```

```
For each record in the uploaded relevant comparisons table
where one or both attributes are not audited
Case of
    equijoin
         recalculate equijoin metatemplate.
         append result to temporary lower bound template
    range
         recalculate range metatemplate
         append result to temp lower bound template
    table
         recalculate lower bound metatemplate
         append to temp lower bound template
         attach new lower table frame to table tree for
         class
Endcase
Edit templates to put in executable form
Move contents of temporary templates to formula area of
    permanent templates
Save system
Cause control to transfer to new query if F3 pressed
Cause desktop to be cleared if F4 pressed
Display message to that effect
```

I will now walk through this pseudocode, focusing on internal system function (as opposed to user interface). Let me begin with a technical note about Framework's organization. As the name suggests, the system is frame-oriented. Each frame contains a label area holding text displayed in the "outline view", a formula area which may contain code or formulas executed when the frame is recalculated, a contents area which holds the result of the recalculated formula or data written into the frame from another source, and a system area which stores information about how the frame is to be displayed on screen. Everything in this system is a tree of frames:

I. Startup
II. Menu
   A. New Class
   B. Existing Class
      1. hurricane
         a. New query
         b. aug15
   C. Quit
III. Invocation
   A. newqueryedit
   B. queryinvocation
IV. Code
   A. do_class_constraint
   B. do_next_table
   C. finished
   D. Equijoin
   E. Range
      1. upper
      2. lower
   F. Table
      1. upper
      2. lower
   G. queryname
   H. debug
   I. debugstart
   J. terminate
   K. queryfinished
   L. varnameextract
   M. boundextract
   N. starttempkeyfilter
   O. quitkeyfilter
   P. nqkeyfilter
   Q. NQformula
   R. edit_template
V. Templates
   A. Classdummy
   B. hurricane
      1. upper
      2. lower
      3. tables
         a. upper
            i. culpa
            ii. date
            iii. damage
         b. lower
            i. culpa
            ii. date
            iii. damage

4. aug15
   a. upper
      i. culpa
      ii. date
      iii. damage
   b. lower
      i. culpa
      ii. date
      iii. damage

VI. Parms
   A. Classname
   B. first
   C. function
   D. Argument
   E. Aq
   F. AqDB
   G. attname
   H. upsource
   I. upsourceselect
   J. dnsource
   K. dnsourceselect
   L. upheader
   M. dnheader
   N. uppertemplate
   O. lowertemplate
   P. Queryname
   Q. attname
   R. sourceDB
   S. sourceatt
   T. sourcedomain
   U. targetDB
   V. tgtatt
   W. tgtdomain
   X. here
   Y. Class1
   Z. Class2
   AA. Class3
   AB. Class4
   AC. Class5
   AD. Class6
   AE. audited

VII. Metadata
   A. Argument selection
   B. Preferred Comparison Pairs
   C. Class constraints
   D. Query screen
   E. Range definition

We begin with the user selecting "New Class" from the menu [E1]. The New Class function is implemented as a subtree in the

system tree which is displayed as a point-and-shoot menu, something easy to implement using a built-in Framework function.

The system responds by asking for the name of the new class. In this example, I respond with "surgery". The system then creates a tree of empty frames which will contain the templates and creates a root node of the tree for table value frames. This tree looks like:

```
D. surgery
   1. upper
   2. lower
   3. tables
      a. upper
      b. lower
```

The menu tree is modified to:

```
II.  Menu
   A. New Class
   B. Existing Class
      1. hurricane
         a. New query
         b. aug15
      2. surgery
         a. New query
   C. Quit
```

The next time the menu is executed, the created query class will appear as a menu option. That way, future users will know that all the information we are about to enter has been stored and is available for re-use. The system then displays the "Argument Selection Entry Screen" [E2], for the user to fill in.

The following metatemplates can be completed with Argument Selection information:

source-select ::=
SELECT function (argument)
FROM sourceDB
WHERE a_q in (a_q-list)

header ::=
SELECT function (targetDB . argument)
FROM sourceDB , targetDB
WHERE sourceDB . a_q in (a_q-list)

Both metatemplates are resolved and appended to the currently-empty template frames.

For the hysterectomy example, this resolves to:

```
"SELECT sum(payment)
FROM  doc
WHERE  proc in (" templates.surgery.active.upper.proc   ");
SELECT sum(hosp.payment)
FROM  doc, hosp
WHERE  doc.proc in (" templates.surgery.active.upper.proc
")
```

and the table subtree is modified to:

D. surgery
  1. upper
  2. lower
  3. tables
    a. upper
      i. proc
    b. lower
      i. proc

The system then displays the "Relevant Types Entry Screen" [E3], for the user to fill in. The relevant types for the hysterectomy query are {BUYER, WHAT, WHEN} (see section 4.3, Subject Knowledge Acquisition).

With the relevant class information the system can retrieve the comparison pairs for the above types. There are three relevant pairs for the upper bound query (retrieved from the preferred comparison pairs listed in section 4.4.2):

```
doc.policyID:hosp.policyID
doc.date:hosp.stay
doc.proc:Hosp.DX
```

Each is processed against the following rules:

```
IF       both attributes in the pair have the same name
AND both attributes in the pair have the same domains
AND neither attribute in the pair is a range attribute
THEN     implement the comparison as an equijoin

IF       one or both of the attributes is defined as range
THEN     implement the comparison as a range

IF       the attributes have different domains
AND the attributes are discrete-valued
THEN     implement comparison as a table lookup
```

To simplify both my notation and my code the system computes:

```
sourcename ::= sourceDB   .   sourceatt
targetname ::= targetDB   .   targetatt
```

This resolves to:

```
sourcename = "doc.policyID"
targetname = "hosp.policyID"
```

for the current pair.

The comparison pair of doc.PolicyID and hosp.policyID meets the conditions of the equijoin rule. The equijoin metatemplate is:

```
equijoin ::=     AND sourcename   =   targetname
```

This resolves to:

```
AND doc.policyID = hosp.policyID
```

When appended to the current state of the upper and lower bound templates the template looks like (for both):

```
"SELECT sum(payment)
FROM  doc
WHERE  proc in ("   templates.surgery.active.upper.proc   ")
SELECT sum(hosp.payment)
FROM  doc, hosp
WHERE  doc.proc in ("   templates.surgery.active.upper.proc
")
AND doc.policyID = hosp.policyID
```

hosp.Stay and doc.date meet the range test. The range metatemplate is:

```
range := "AND " sourcestart <= targetfinish
                "AND " sourcestart >= targetstart
```

This metatemplate resolves to:

```
AND doc.date <= hosp.date + hosp.LOS[20]
AND doc.date >= hosp.date
```

When appended to the current upper and lower bound templates the result is:

```
"SELECT sum(payment)
FROM  doc
WHERE  proc in ("   templates.surgery.active.upper.proc   ");
SELECT sum(hosp.payment)
FROM  doc, hosp
WHERE  doc.proc in ("   templates.surgery.active.upper.proc
")
AND doc.policyID = hosp.policyID
AND doc.date <= hosp.date + hosp.LOS
AND doc.date >= hosp.date
```

The comparison pair of doc.proc and hosp.DX meet the table rule. The table metatemplates are:

```
table.upper ::=  AND   targetname    IN (
                 templates.active.upper.attname   )

table.lower ::=  AND   targetname    IN (
                 templates.active.lower.attname   )
```

---

[20]For this formulation to work reliably, the attributes which record dates need to recorded in the database as DATE types. Otherwise, there is a potential problem with date arithmetic.

This metatemplate resolves to:

AND hosp.DX IN TEMPLATES.ACTIVE.UPPER.attname

and

AND hosp.DX IN TEMPLATES.ACTIVE.LOWER.attname

In addition the table tree is expanded to add a frame to contain the table information needed for the test:

D. surgery
   1. upper
   2. lower
   3. tables
      a. upper
         i. proc
         ii. DX
      b. lower
         i. proc
         ii. DX

The upper bound query then becomes:

```
  "SELECT sum(payment)
  FROM  doc
  WHERE  proc in (" &  templates.surgery.active.upper.proc   &
");
  SELECT sum(hosp.payment)
  FROM  doc, hosp
  WHERE  doc.proc in (" & templates.surgery.active.upper.proc
 & ")
  AND doc.policyID = hosp.policyID
  AND doc.date <= hosp.date + hosp.LOS
  AND doc.date >= hosp.date
  AND hosp.DX in templates.active.upper.attname
```

with the lower bound version being identical except for changing the word "upper" in the last line to "lower"

and adding two additional equijoin tests. The additional equijoin tests are produced and appended in the same way as the previous equijoin example. The current state of the template then looks like:

```
  "SELECT sum(payment)
  FROM  doc
  WHERE  proc in (" &  templates.surgery.active.upper.proc  &
");
  SELECT sum(hosp.payment)
  FROM  doc, hosp
  WHERE  doc.proc in (" & templates.surgery.active.upper.proc
& ")
  AND doc.policyID = hosp.policyID
  AND doc.date <= hosp.date + hosp.LOS
  AND doc.date >= hosp.date
  AND hosp.DX in (" & templates.active.upper.attname & ")
&"AND doc.age = hosp.age
  AND doc.sex = hosp.sex"
```

4.5 Transforming Query Class Templates into Executable Queries

Templates require table values for those comparison pairs resolved as table look-ups. The needed table values are query-specific, not case-specific. Also, users may have query-specific constraints to add. For example, a user may want records from 1985 only and not relevant records from all time periods stored in the database. I will now discuss how the system asks for the table values needed for the templates.

After the user selects "New Query" the system asks for the name of the new queries. We respond with "hysterectomy". The user is then asked for the upper and lower bound table values for each of the attributes handled by table look-up (PROC, the query attribute and DX). The user is then asked for any additional tests he might want. Queries are then produced by

swapping the table values into their appropriate places in the query class template and appending the user constraints.

The following is the pseudocode for the query assembly step:

```
Query step
   User presses F4 at end of class run or selects new query
   from menu
   system asks user for queryname
   remap F3 to act as signal to process next frame
   system asks for the user constraint(s)
   copy class' table tree and renames as queryname
   while not end of tree
       case of leaf node status
            if not leaf node then go to next frame
            if leaf node then
                 extract bound (ie upper|lower) and attname
                 form and display prompt
                 zoom frame to display entry area
   endwhile
   update menu for new query
   add query invocation code to new query entry in menu
   remap F3 to close system
   remap F4 to proceed to query invocation
```

The user constraint is one or more valid SQL conditions which are appended to the query. I define this as:

```
user-condition ::=
       compare-condition
   |   between-condition
   |   like-condition
   |   in-condition
   |   exists-condition
   | ( predicate )

compare-condition ::=
       scalar-expression
   |   scalar-expression compare-operator
            ( column-select-expression )
   |   scalar-expression IS [ NOT ] NULL


compare-operator ::=
   = | NOT = | < | NOT < | <= | > | NOT > | >=

column-select-expression ::=
       column-select-clause
```

```
    |   from-clause
    |   [ where-clause ]
    |   [ grouping-clause [ having-clause ] ]

column-select-clause ::=
        SELECT [ DISTINCT ] scalar-expression

between-condition ::=
        column-name [ NOT ] BETWEEN scalar-expression
             AND scalar-expression

like-condition ::=
        column-name [ NOT ] LIKE scalar-expression

in-condition ::=
        scalar-expression [ NOT ] IN ( set-of-scalars )

set-of-scalars ::=
        constant-commalist | column-select-expression

exists-condition ::=
        EXISTS ( select-expression )

scalar-expression ::=
        scalar-term [ arithmetic-operator scalar-expression ]

scalar-term ::=
        [ + | - ] scalar-value

scalar-value ::=
        column-name
    |   function-reference
    |   constant
    |   user
    |   ( scalar-expression )
```

Let us assume the user intends to go straight through to producing a query. Upon the user pressing F4, the system begins soliciting the information needed to prepare a query.

The system begins by asking for a queryname. In this example, I respond with "hysterectomy". The system then makes a copy of the table subtree, inserts the subtree at the same

level of hierarchy as the original table subtree, and gives the name of the query to the new table subtree:

```
   D. surgery
      1. upper
      2. lower
      3. tables
         a. upper
            i.   proc
            ii.  DX
         b. lower
            i.   proc
            ii.  DX
```

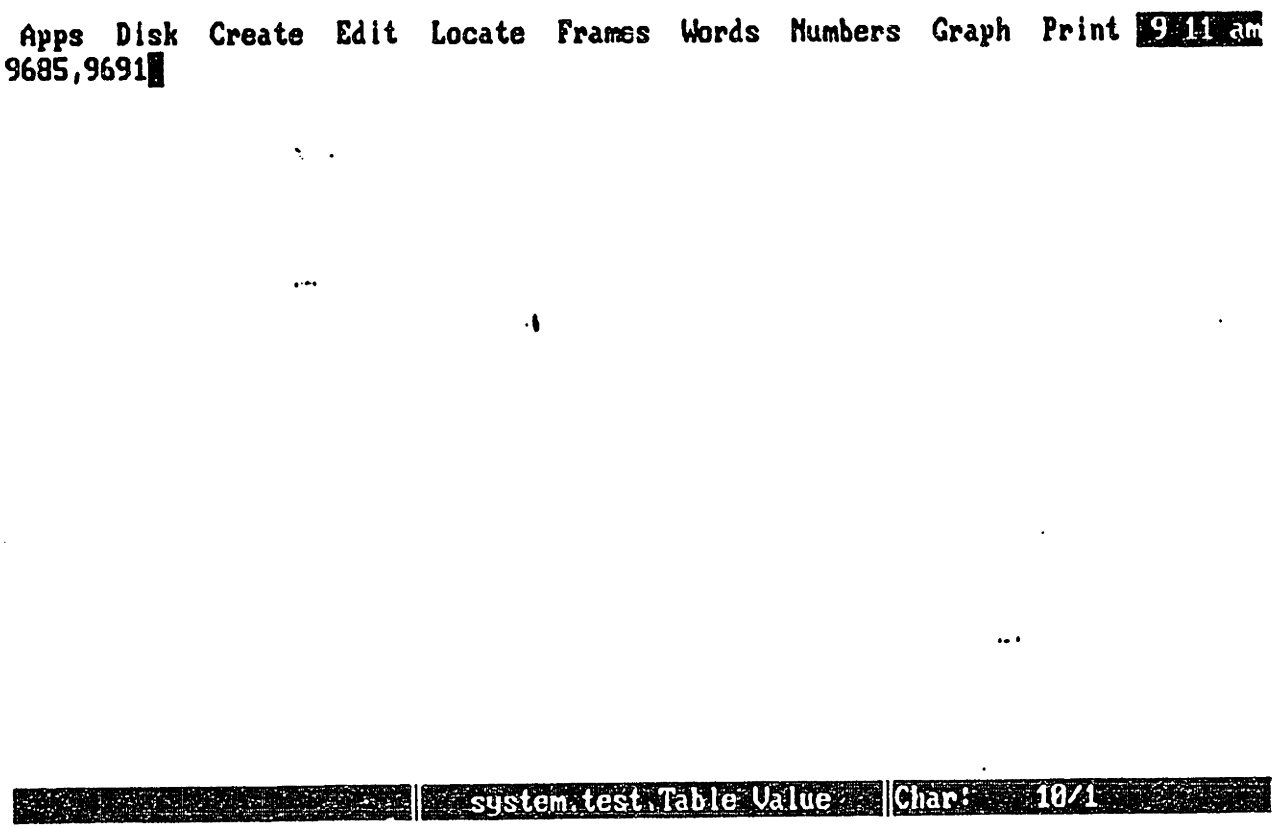
becomes

```
   D. surgery
      1. upper
      2. lower
      3. tables
         a. upper
            i.   proc
            ii.  DX
         b. lower
            i.   proc
            ii.  DX
      4. hysterectomy
         a. upper
            i.   proc
            ii.  DX
         b. lower
            i.   proc
            ii.  DX
```

The system then reads through the table tree and prompts the user to enter values to be stored in each frame. In this example, the user will be prompted for upper and lower bound values for PROC and DX on the "Table Value Entry Screen" [E4]. The system then updates the menu, which changes from:

```
II.  Menu
   A. New Class
   B. Existing Class
      1. hurricane
         a. New query
         b. aug15
      2. surgery
         a. New query
```

E4 TABLE VALUE ENTRY SCREEN



to:

II. Menu
   A. New Class
   B. Existing Class
      1. hurricane
         a. New query
         b. aug15
      2. surgery
         a. New query
         b. hysterectomy

and adds the needed invocation code to the frame of the new menu option.

The system then makes a copy of the query's table subtree and changes the part of the pathname which is the name of the query to "active". This action is taken to cause the template to reference the desired stored references. The system makes a copy of the query class templates and recalculates the template copies which results in the a template with the table values swapped in.

The system then presents the "User Constraints Entry Screen" [E5] which asks the user whether he wants another WHERE clause. For example, the user might want only records on a certain date, a restriction that has nothing to do with the relationships analyzed by the system. In this example, the user constraint needed to restrict records to 1985 is:

```
AND doc.date between 850101,851231
```

When done entering however many (or few) user constraints desired the user presses F3. Whatever was entered is appended

to both the source and target queries of both the upper and lower bound query pairs -- a total of four appends. User constraint(s) are not stored anywhere in the system[21]. This user constraint facility does not do any checking that the constraint is a valid clause.

The upper and lower bound queries are then written to files and the duplicate tree and templates in the system are deleted.

This concludes Chapter 4 and this walkthrough of the pseudocode of my system implementation.

[21]Because the user constraints are not stored, there is no storage facility for user constraints listed on Diagram XIII.

E5 USER CONSTRAINTS ENTRY SCREEN

Enter user constraints

Chapter 5: Sample Retrievals

5.1 Overview

To demonstrate how the concepts in Chapter 3 and the implementation in Chapter 4 relate to the system actually built, I will describe the contents of the MIT Blue Cross database and then work through three examples. These examples are:

- "How much was spent on hysterectomies in 1985?". Hysterectomy is a surgical operation which requires hospitalization. The relevant problem here is finding the hospital records which correspond to relevant physician records in the MIT database.
- "How much was spent on osteoarthritis hospitalizations during 1985?". Osteoarthritis is a diagnosis (as distinct from a surgical procedure). The relevant problem is finding the physician records which correspond to relevant hospital records in the MIT database.
- "How much was received in compensation for damage caused by the hurricane on 8/15/88". A hypothetical company has operations in both Mexico and in the United States with separate general ledgers and separate insurance policies. A hurricane occurring on both sides of the border causes insured damage and the problem is finding American records which correspond to Mexican records in a hypothetical database.

5.2 MIT Blue Cross Database Description

I begin with a description of the MIT Blue Cross transaction logs, the databases for the first two sample queries. The business reason for MIT Health's gracious cooperation in this research is MIT's desire to determine the value of a second opinion program to reduce unnecessary surgery and to assess the value of a so-called "preferred provider organization". The latter provides a financial incentive for plan members to use low-cost physicians, but permits the normal insurance benefits if other physicians are chosen.

MIT's primary data resource for its health care expenditures is a tape from Blue Cross/Blue Shield which contains two files, one of physician bills for Blue Cross subscribers and one of hospital bills for Blue Cross subscribers. The physician file contains one record for each hospital-related claim submitted by an MIT employee who has elected Blue Cross coverage[22]. A different record is generated for each physician contact. For example, if menstrual distress results in a hysterectomy the physician file is likely to contain records for a pre-operation office visit, a surgical fee, an assisting surgical fee, an anesthesia fee, one or more follow-up visits. These records

---

[22]Hospital bills which are not submitted (not uncommon for inexpensive outpatient diagnostic tests) are invisible to the system. Physician bills which are not related to a hospital stay are stored elsewhere. Blue Shield distinguishes between "major medical" (hospital-related) and "extended benefits" (not hospital-related). The tape provided by Blue Cross also contains the hospital records of MIT health plan members (MIT's captive health maintenance organization), although I removed these records in preparing an extract from the tape.

will be tagged with the procedure code for hysterectomy. There are about 32,000 physician records each year.

The hospital file contains a record for each contact with a hospital, both outpatient and inpatient. There are about 1,200 hospitalizations per year and about 12,000 outpatient visits. This tape is the sole source of information about health expenditures for MIT. Most other employers using Blue Cross do not have better information on health expenditures. There is no episode of care key linking the physician and hospital files nor are there any patient-specific identifiers[23].

5.3 Example I: Hysterectomy

In the first example, I work though the process of answering the question:

How much was spent on hysterectomies in 1985?

If the system menu [E1] does not display a query class which would handle hysterectomies -- a "surgery" class -- the absence of that class means there is no template which could be used to answer this question and that nobody has previously entered into the system the information needed to create that template. The steps necessary for the system to produce the necessary template and executable query are:

---

[23]It would seem reasonable that PATIENT_NAME should be on the tape. It is not on the tape and Blue Cross refuses to put it there or provide an authoritative explanation of why.

- The database analyst enters universal attribute type, audit status, and similar information about each attribute in the databases into the system's data dictionary using the "Data Dictionary Entry Screen" [D1] and the "Range Definition Entry Screen" [D2].
- The system produces comparison pairs from the data attribute information entered above. The resulting set of possible comparison pairs can be edited by the database analyst and then uploaded into the Framework part of the system. These comparison pairs are stored for future use by any query referencing the same pair of databases.
- The end user selects "New Class" from the system menu (Menu, [E1]) and names the new query class by responding to the "New Class name" input prompt. I named the new query class name "surgery" to represent the same relationships that exist for all instances of surgery-requiring-hospitalization.) The user then fills in the "Argument Selection Entry Screen" [E2] and identifies the relevant universal attribute types in the "Relevant Type Entry Screen" [E3].
- The system selects a method for comparing the attributes for all of the relevant comparison pair and prepares the corresponding metatemplate(s) to implement the selected comparison method(s). The system then assembles all of the instantiated metatemplates into a template for the

query class and updates the system menu with the name of the new query class "surgery" so that succeeding users will know that the template has been prepared.

- The end user selects New Query (either from the menu or in response to a query when New Class has finished), gives a name to the query (here, "Hyst"), provides the table values for any comparison implemented as a table look-up (Table Value Entry Screen [E4]) and adds any user constraints (such as the "in 1985" restriction) as are desired (User Constraint prompt [E5]). I should note that we know that New Query had to be used, since if there were no previously defined class that would handle our query, there could not possibly have been a previously defined query. If there were, we'd use it. The system responds by writing the queries into a file, storing the table values, and creating an entry for "Hyst" under the query class of "Surgery" in the system menu.

The details of the above were discussed in greater detail in the preceding chapter, Implementation (Chapter 4).

In this hysterectomy example, the software executed in the New Class function determines that the following five comparison pairs are in the preferred comparison pairs import and are also in the relevant types specified by the end-user:

Buyer - policyID:policyID done as an equijoin (tested for both the upper & lower queries)

When - Date:Stay done as a range comparison (upper & lower)

What - Procedure:Diagnosis done as a table lookup (upper & lower)

Buyer - Age:Age done as an equijoin (lower only)

Buyer - Sex:Sex done as an equijoin (lower only)

The resulting template requires a set of permissible values for PROC in order to locate the relevant source database records and a set of permissible values for DX in order to run the PROC:DX table check test of consistency. Solicitation for these two sets, storage of the response, and updating the system menu takes place automatically when the end-user selects the New Query function. (The end-user is also solicited for a user constraint, which is not stored)

The system's output after completing New Query is two pairs of SQL queries (the ones at the end of the preceding chapter). The first member of the upper bound pair does the upper bound summation of the source (MD) database. The second member of the upper bound pair joins the records found in the source database query with the target database and sums the target database

records retrieved. The second pair functions in the same way using the lower bound tests.

When the query doing the source extract is executed the result contains 59 records, labelled Hysterectomy Physician Records [P1]. Of these 59 records, there are 17 distinct policy identification numbers, implying that 17 hysterectomies were done. In turn, the existence of 17 hysterectomies implies at least 17 hospital stays. (There are potentially more stays, because patients are occasionally transferred from one hospital to another in the course of treatment). Worth noting on this extract is the following:

- The first record has a payment of 0 because reimbursement was refused.
- The low value payments ($22 and $23) are for office visits.
- Several payments are for $822 and $921. Different maximum payments are computed for <u>each</u> procedure for <u>each</u> doctor; clever doctors find out what their maximum is and make certain to bill that amount.
- The field "link" cross-references to anomalous hospital records discussed later in this section.

For explanatory purposes, I have retrieved the hospital records based solely on the most discriminating criterion (policyID) into a spreadsheet. The retrieval is done by using the system to generate an SQL query, which I manually edit to remove the unwanted criteria and to place its result in a file.

I then load the resulting file into a spreadsheet with formulas which annotate each record with the remaining tests selected by the system. This process permits identifying the effect of each individual test -- something which would be masked by simply showing the records which passed all tests. Although this process is currently partly manual, it could be completely automated.

The upper bound tests are:

- Date: checks for the existence of a hospital record in the interval defined by the hospital stay.
- UpDX: checks for a diagnosis that might be consistent with hysterectomy. All medical problems that affect the female reproductive system are acceptable.

For the lower bound:

- LowDX: checks for a diagnosis which definitely is consistent with hysterectomy. For example, the diagnosis of breast cancer is rejected in the lower bound and accepted in the upper bound because hysterectomy is not a treatment for breast cancer but is a treatment for diseases of related organs.
- SexT: checks that the sex of the patient on the hospital bill is same as the sex of the patient on the physician bill. This is not an upper bound test, because sex is not an audited field.
- AgeT: checks that the ages are the same. Age is also unaudited.

I have encoded these tests as formulas in a spreadsheet, placing "1" in the cell if the test fails and "OK" otherwise. To compute upper and lower bound totals, I have two columns (U and L, respectively) whose value is "OK" if all the respective tests are passed and "1" otherwise. I then sum the PAYMENT field in the rows where the record is "OK" for that bound.

The query retrieves 139 hospital records of which 27 are for hospital stays. The others are for outpatient bills. Attached are two annotated printouts. The shorter one labelled "[P2] Hysterectomy query: annotated inpatient hospital records" is inpatient only and excludes tests and other work done to outpatients[24]. The longer one is labelled "[P3] Hysterectomy query:annotated hospital records" and contains all the retrieved hospital records.

There are 10 more hospital stays than we expected to find. While some of these may have been transfers, it is more likely that these additional records are services provided to other members of the family during 1985.

There were $89,343 in upper bound hospital bills and $57,999 in lower bound bills. Adding in the $19,978 in physician bills (shown on the list of physician records retrieved) I conclude that $77,977 <= true cost <= $109,321.

---

[24]The database contains a inpatient/outpatient flag which can be referred to explicitly in a user constraint or implicitly in a database view. I did not exclude earlier because my policy in this section is to exclude by most discriminating criterion only and then annotate.

P1 Hysterectomy Physician Records

Itemnum	PolicyID	Age	Sex	Date	Procedure	Payment
1	911274580	23	F	851007	4685	0
2	911274580	45	F	851007	4685	161
3	911274580	45	F	851007	4685	760
4	911274580	45	F	851106	4685	23
5	911656282	33	F	850201	4685	349
6	911656282	33	F	850201	4685	882
7	911771392	44	F	850815	4685	161
8	911771392	44	F	850815	4685	323
9	911771392	44	F	850815	4685	921
10	912518363	41	F	850110	4685	154
11	912518363	41	F	850110	4685	882
12	912518363	41	F	850122	4685	22
13	912518363	41	F	850213	4685	22
14	912518363	43	F	850725	4685	921
15	912531863	43	F	850904	4685	23
16	912574861	54	F	850416	4685	882
17	912780559	61	F	850205	4685	0
18	912780559	61	F	850205	4685	154
19	912780559	61	F	850205	4685	312
20	912780559	61	F	850205	4685	882
21	912780559	61	F	850219	4685	22
22	912780559	61	F	850227	4685	22
23	912853634	29	F	850520	4685	882
24	912853634	29	F	850521	4685	154
25	912853634	29	F	850613	4685	22
26	912853634	29	F	850716	4685	23
27	913769311	57	F	850927	4685	161
28	913769311	57	F	850927	4685	921
29	913769311	57	F	851021	4685	23
30	913769311	57	F	851118	4685	23
31	915004728	41	F	850109	4685	154
32	915004728	41	F	850109	4685	349
33	915004728	41	F	850109	4685	882
34	915332236	44	F	850108	4685	22
35	915332236	45	F	850129	4685	22
36	915946797	34	F	850318	4685	312
37	915946797	34	F	850319	4685	154
38	916047555	48	F	850115	4685	312
39	916047555	48	F	850115	4685	882
40	916677374	44	F	850410	4685	143
41	916677374	44	F	850410	4685	257
42	918094874	57	F	851115	4685	307
43	918094874	57	F	851115	4685	921
44	918165837	46	F	850123	4685	882
45	918165837	46	F	850124	4685	312
46	918458775	49	F	850723	4685	161
47	918458775	49	F	850723	4685	287
48	918458775	49	F	850723	4685	921
49	918458775	49	F	850819	4685	23
50	918458775	49	F	850919	4685	23

P1 Hysterectomy Physician Records (cont.)

52	918512502	46	F	850605	4685	338
53	918512502	46	F	850605	4685	882
54	918512502	47	F	850618	4685	33
55	918512502	47	F	850628	4685	33
56	918608757	47	F	850215	4685	312
57	919128412	61	F	850812	4685	921
58	919128412	61	F	850904	4685	23
59	919128412	61	F	850920	4685	23

The annotations give insight into why certain records would have been automatically excluded if the SQL queries had been executed directly without this explanatory annotation step. The field "link" identifies these anomalous records and cross-references them to the Hysterectomy Physician Records printout [P1]. Starting from the top of the inpatient printout P2 we see:

- 911274580 (link A) was diagnosed as having ovarian cancer. Ovarian cancer need not be consistent with hysterectomy so the lower bound test failed. The record was accepted as upper bound because ovarian cancer is related to uterine cancer and might be relevant.
- 912574861 (link B) had one admission for a fractured humerus during February and another admission for misc cervix in April. The February admission was rejected because a diagnosis of broken bone is inconsistent with a hysterectomy at both the upper and lower levels. Further, a hysterectomy done in April could not be related to a hospital stay in February. The April admission does pass the tests.
- 91285364 (link C) The hospital stay with the tonsillectomy diagnosis is rejected for both upper and lower bound purposes. The hospital stay with the uterine fibroma diagnosis is accepted for both bounds.
- 915332236 (link D) is a 44 year old man with rectal cancer. This record was rejected on diagnosis and date.

Although it is not something the system will recognize, a knowledgeable end-user could notice that there is no hysterectomy record. Looking back through the physician records he would see that there were two physician bills. However, as both were for $22 neither could possibly have been a surgeon's fee because the fee is much too low. He would also know that a hysterectomy could not have been performed on a male. There is insufficient information for anyone to figure out what actually happened.

- 915946797 (link E) finds a 34 year old woman with cancer and a 3 year old with stomach problems. The 3 year old's records do not match on date, diagnosis, or age and are rejected.

Note that additional questions in the same class of "surgery-requiring-hospitalization" can be asked by selecting "New Query" and entering the requested table values for PROC for the new kind of surgery and for DX for a list of consistent diagnosis codes. The template created in "New Class" will be re-used. For example, if one wanted to ask about appendectomies, the system would ask for the procedure code(s) which define appendectomy and the diagnosis codes which are consistent with appendectomy. This reusability of stored knowledge is discussed at great length in sections 3.8 (Storing and Reusing Retrofit Knowledge), 4.4 (Query Class Template

Creation), and 4.5 (Transforming Query Class Templates into Executable Queries).

P2 Hysterectomy Query: Annotated Inpatient Hospital Records

PolicyID	Sex	Date	In?	Age	DocAge	LOS	DX	Payment	Up	Low	DocDate	Date	UpDX	LowDX	Sex	Age
911656282	F	850131	1	33	33	8	Misc uterus	7631	OK	OK	850201	1	OK	OK	OK	OK
912518363	F	850109	1	41	41	5	Uterine fibroma	3294	OK	OK	850110	OK	OK	OK	OK	OK
912531863	F	850725	1	43	41	7	Uterine fibroma	6423	OK	OK	850725	OK	OK	OK	OK	1
912853634	F	850519	1	48	29	7	Uterine fibroma	7405	OK	1	850520	OK	OK	OK	OK	1
913769311	F	850927	1	57	57	6	uterus cancer	4625	OK	OK	850927	OK	OK	OK	OK	OK
915004728	F	850108	1	41	41	8	Uterine fibroma	6405	OK	OK	850109	OK	OK	OK	OK	OK
916047555	F	850114	1	48	48	7	Vagina prolaps	4072	OK	OK	850115	OK	OK	OK	OK	OK
918094874	F	851114	1	57	57	7	Uterine fibroma	6714	OK	OK	851115	1	OK	OK	OK	OK
918608757	F	850214	1	47	47	7	Uterine fibroma	6502	OK	OK	850215	OK	OK	OK	OK	OK
919128412	F	850811	1	61	61	6	Uterine fibroma	4928	OK	OK	850812	OK	OK	OK	OK	OK
911274580	F	851006	1	45	45	7	Ovary cancer	3808	OK	1	851007	OK	OK	1	OK	OK
911771392	F	850814	1	44	44	4	Misc cancer	2425	OK	1	850815	OK	OK	1	OK	OK
912574861	F	850415	1	54	54	5	Misc cervix	3193	OK	1		OK	OK	1	OK	OK
912780559	F	850204	1	61	61	10	Misc uterus	5024	OK	1	850205	OK	OK	OK	OK	OK
915946797	F	850318	1	34	34	10	Misc cancer	5221	OK	1	850318	OK	OK	1	OK	OK
916677374	F	850409	1	44	44	6	uterus cancer	2377	OK	1	850410	OK	OK	OK	OK	OK
918458775	F	850722	1	49	49	6	Uterine fibroma	3787	OK	1	850723	OK	OK	OK	OK	OK
918512502	F	850604	1	46	46	7	Uterine fibroma	5245	OK	1	850605	OK	OK	OK	OK	OK
912574861	F	850217	1	54	54	1	Humerus fractur	452	1	1	850416	1	1	1	OK	OK
912853634	F	850626	1	18	29	1	Tonsils	1766	1	1		OK	1	1	OK	1
913769311	F	850501	1	57	57	5	breast cancer	2142	1	1		1	OK	1	OK	OK
913769311	F	850520	1	57	57	4	breast cancer	2705	1	1		1	OK	1	OK	OK
913769311	F	850811	1	57	57	3	uterus cancer	2102	1	1		1	OK	OK	OK	OK
915332236	M	850726	1	47	44	4	rectal cancer	2897	1	1		1	1	1	1	1
915946797	M	851201	1	3	34	3	Esophagus	1735	1	1		1	1	1	1	1
915946797	M	851025	1	3	34	1	Misc congenital	1466	1	1		1	1	1	1	1
915946797	M	851230	1	3	34	2	Misc stomach	400	1	1		1	1	1	1	1

P3 Hysterectomy Query: Annotated Hospital Records

PolicyID	Sex	Date	In?	Age	DocAge	LOS	DX	Payment	Up	Low	DocDate	Date	UpDX	LowDX	Sex	Age
911656282	F	850131	1	33	33	8	Misc uterus	7631	OK	OK	850201	1	OK	OK	OK	OK
912518363	F	850109	1	41	41	5	Uterine fibroma	3294	OK	OK	850110	OK	OK	OK	OK	OK
912531863	F	850725	1	43	41	7	Uterine fibroma	6423	OK	OK	850725	OK	OK	OK	OK	1
912853634	F	850519	1	48	29	7	Uterine fibroma	7405	OK	OK	850520	OK	OK	OK	OK	1
913769311	F	850927	1	57	57	6	uterus cancer	4625	OK	OK	850927	OK	OK	OK	OK	OK
915304728	F	850108	1	41	41	8	Uterine fibroma	6405	OK	OK	850109	OK	OK	OK	OK	OK
916047555	F	850114	1	48	48	7	Vagina prolaps	4072	OK	OK	850115	OK	OK	OK	OK	OK
918094874	F	851114	1	57	57	7	Uterine fibroma	6714	OK	OK	851115	1	OK	OK	OK	OK
918608757	F	850214	1	47	47	7	Uterine fibroma	6502	OK	OK	850215	OK	OK	OK	OK	OK
919128412	F	850811	1	61	61	6	Uterine fibroma	4928	OK	OK	850812	OK	OK	OK	OK	OK
911274580	F	851006	1	45	45	7	Ovary cancer	3808	OK	1	851007	OK	OK	1	OK	OK
911771392	F	850814	1	44	44	4	Misc cancer	2425	OK	1	850815	OK	OK	1	OK	OK
912531863	M	850604	0	49	41	0	Misc cancer	226	OK	1		OK	OK	1	1	1
912574861	F	850415	1	54	54	5	Misc cervix	3193	OK	1		OK	OK	1	OK	OK
912780559	F	850204	1	61	61	10	Misc uterus	5024	OK	1	850205	OK	OK	OK	OK	OK
915332236	F	850129	0	45	44	0	YOO	38	OK	1		OK			OK	OK
915946797	F	850318	1	34	34	10	Misc cancer	5221	OK	1	850318	OK	OK	1	OK	OK
916677374	F	850409	1	44	44	6	uterus cancer	2377	OK	1	850410	OK	OK	OK	OK	OK
918458775	F	850722	1	49	49	6	Uterine fibroma	3787	OK	1	850723	OK	OK	OK	OK	OK
918512502	F	850604	1	46	46	7	Uterine fibroma	5245	OK	1	850605	OK	OK	OK	OK	OK
911274580	M	850327	0	16	45	0	Cardio symptoms	62	1	1		1	1	1	1	1
911274580	F	851029	0	18	45	0	Cardio symptoms	26	1	1		1	1	1	OK	1
911274580	F	850731	0	18	45	0	Misc heart	750	1	1		1	1	1	OK	1
911274580	F	851124	0	18	45	0	Misc muscle	45	1	1		1	1	1	OK	1
911274580	F	850111	0	18	45	0	Misc sprain	92	1	1		1	1	1	OK	1
911274580	F	850821	0	45	45	0	Misc U genital	133	1	1		1	OK	OK	OK	OK
911274580	F	850217	0	18	45	0	Misc wound	88	1	1		1	1	1	OK	1
911274580	F	850621	0	45	45	0	Uterus infect	815	1	1		1	OK	OK	OK	OK
911274580	M	851008	0	17	45	0	YOO	31	1	1		1			1	1
911274580	F	850514	0	18	45	0	YOO	42	1	1		1			OK	1
911274580	F	850719	0	18	45	0	YOO	18	1	1		1			OK	1
911274580	M	850719	0	20	45	0	YOO	44	1	1		1			1	1
911656282	F	850813	0	33	33	0	endocrine dysf	166	1	1		1	1	1	OK	OK
911656282	F	850409	0	33	33	0	Misc metabolic	11	1	1		1	1	1	OK	OK
911656282	F	850321	0	33	33	0	YOO	44	1	1		1			OK	OK
911656282	F	850403	0	33	33	0	YOO	102	1	1		1			OK	OK
911656282	F	850516	0	33	33	0	YOO	116	1	1		1			OK	OK
911771392	F	850624	0	44	44	0	Cervix infect	90	1	1		1	OK	1	OK	OK
911771392	M	850509	0	20	44	0	Misc cancer	28	1	1		1	OK	1	1	1
911771392	F	850703	0	41	44	0	Misc ovary	86	1	1		1	OK	1	OK	1
911771392	F	850702	0	44	44	0	Misc ovary	130	1	1		1	OK	1	OK	OK
912518363	F	850526	0	41	41	0	Cardio symptoms	34	1	1		OK	1	1	OK	OK
912518363	F	851127	0	42	41	0	Misc ovary	9	1	1		1	OK	1	OK	OK
912518363	F	850626	0	41	41	0	Misc stomach	158	1	1		OK	1	1	OK	OK
912518363	F	850813	0	41	41	0	Respire infect	147	1	1		OK	1	1	OK	OK
912518363	F	850301	0	41	41	0	YOO	8	1	1		OK			OK	OK
912531863	M	850108	0	48	41	0	Cardio symptoms	21	1	1		1	1	1	1	1
912574861	F	850217	1	54	54	1	Humerus fractur	452	1	1	850416	1	1	1	OK	OK
912574861	F	850131	0	54	54	0	infection	54	1	1		1	1	1	OK	OK
912574861	F	850201	0	54	54	0	Misc muscle	665	1	1		1	1	1	OK	OK
912574861	F	850724	0	54	54	0	Misc ovary	38	1	1		1	OK	1	OK	OK

P3 Hysterectomy Query: Annotated Hospital Records (cont.)

PolicyID	Sex	Date	In?	Age	BocAge	LOS	DX	Payment	Up	Low	BocDate	Date	UpDX	LowDX	Sex	Age
912574861	F	850103	0	54	54	0	Skin glands	38	1	1		1	1	1	OK	OK
912780559	F	850218	0	61	61	0	995	15	1	1		OK	1	1	OK	OK
912853634	F	850626	1	18	29	1	Tonsils	1766	1	1		OK	1	1	OK	1
912853634	F	850425	0	22	29	0	Misc dermatitis	54	1	1		1	1	1	OK	1
912853634	F	850429	0	48	29	0	Misc intestine	133	1	1		1	1	1	OK	1
912853634	F	850820	0	48	29	0	Misc ovary	69	1	1		1	OK	1	OK	1
912853634	F	850506	0	48	29	0	Misc uterus	133	1	1		1	OK	OK	OK	1
912853634	F	850322	0	48	29	0	Skin glands	38	1	1		1	1	1	OK	1
912853634	F	850809	0	18	29	0	V00	51	1	1		1			OK	1
912853634	F	850926	0	48	29	0	V00	61	1	1		1			OK	1
913769311	F	850501	1	57	57	5	breast cancer	2142	1	1		1	OK	1	OK	OK
913769311	F	850520	1	57	57	4	breast cancer	2705	1	1		1	OK	1	OK	OK
913769311	F	850811	1	57	57	3	uterus cancer	2102	1	1		1	OK	OK	OK	OK
913769311	F	850412	0	56	57	0	breast cancer	49	1	1		1	OK	1	OK	OK
913769311	F	850426	0	57	57	0	breast cancer	95	1	1		1	OK	1	OK	OK
913769311	F	850618	0	57	57	0	breast cancer	436	1	1		1	OK	1	OK	OK
913769311	F	850624	0	57	57	0	breast cancer	1557	1	1		1	OK	1	OK	OK
913769311	F	850529	0	57	57	0	Cardio symptoms	575	1	1		1	1	1	OK	OK
913769311	M	850124	0	60	57	0	endocrine dysf	7	1	1		1	1	1	1	1
913769311	F	850501	0	57	57	0	lymph cancer	201	1	1		1	OK	1	OK	OK
913769311	F	850409	0	56	57	0	Misc ovary	65	1	1		1	OK	1	OK	OK
913769311	M	850426	0	18	57	0	Misc sprain	87	1	1		1	1	1	1	1
913769311	M	850801	0	60	57	0	Sacroiliac	44	1	1		1	1	1	1	1
913769311	F	851217	0	57	57	0	Thyroid disease	23	1	1		1	1	1	OK	OK
915004728	F	850304	0	41	41	0	Misc ovary	69	1	1		1	OK	1	OK	OK
915004728	F	851217	0	42	41	0	Retinitis	45	1	1		1	1	1	OK	OK
915004728	F	850108	0	41	41	0	V00	15	1	1		1			OK	OK
915332236	M	850726	1	47	44	4	rectal cancer	2897	1	1		1	1	1	1	1
915332236	M	850720	0	47	44	0	Skin infect	46	1	1		1	1	1	1	1
915332236	M	850723	0	47	44	0	V00	22	1	1		1			1	1
915946797	M	851201	1	3	34	3	Esophagus	1735	1	1		1	1	1	1	1
915946797	M	851025	1	3	34	1	Misc congenital	1466	1	1		1	1	1	1	1
915946797	M	851230	1	3	34	2	Misc stomach	400	1	1		1	1	1	1	1
915946797	M	850210	0	37	34	0	Dental	27	1	1		1	1	1	1	1
915946797	M	850524	0	2	34	0	Misc congenital	52	1	1		1	1	1	1	1
915946797	M	850926	0	3	34	0	Misc congenital	55	1	1		1	1	1	1	1
915946797	F	850227	0	34	34	0	Misc kidney	26	1	1		1	1	1	OK	OK
915946797	F	850220	0	34	34	0	Misc metabolic	49	1	1		1	1	1	OK	OK
915946797	F	850803	0	34	34	0	Misc metabolic	6	1	1		1	1	1	OK	OK
915946797	F	850412	0	34	34	0	Misc ovary	272	1	1		1	OK	1	OK	OK
915946797	M	851115	0	3	34	0	Misc stomach	98	1	1		1	1	1	1	1
915946797	M	851230	0	3	34	0	Misc stomach	125	1	1		1	1	1	1	1
915946797	M	850225	0	2	34	0	Skull fracture	128	1	1		1	1	1	1	1
915946797	F	850402	0	34	34	0	V00	30	1	1		1			OK	OK
916047555	M	850611	0	49	48	0	hypertension	24	1	1		1	1	1	1	OK
916047555	F	851106	0	49	48	0	Joint derange	62	1	1		1	1	1	OK	OK
916047555	F	851113	0	49	48	0	misc cancer	216	1	1		1	OK	1	OK	OK
916047555	F	851129	0	49	48	0	Misc circ	298	1	1		1	1	1	OK	OK
916047555	F	850427	0	48	48	0	Misc heart	111	1	1		1	1	1	OK	OK
916047555	F	850301	0	48	48	0	Misc metabolic	9	1	1		1	1	1	OK	OK
916047555	F	850111	0	48	48	0	Skin infect	33	1	1		1	1	1	OK	OK

P3 Hysterectomy Query: Annotated Hospital Records (cont.)

PolicyID	Sex	Date	In?	Age	DocAge	LOS	DX	Payment	Up	Low	DocDate	Date	UpDX	LowDX	Sex	Age
916047555	F	851015	0	49	48	0	Thyroid disease	131	1	1		1	1	1	OK	OK
916677374	M	850728	0	37	44	0	For body in eye	49	1	1		1	1	1	1	1
916677374	M	850410	0	13	44	0	limb fracture	109	1	1		OK	1	1	1	1
916677374	M	850224	0	13	44	0	limb fracture	109	1	1		1	1	1	1	1
916677374	F	850925	0	44	44	0	Misc dermatitis	46	1	1		1	1	1	OK	OK
916677374	F	850528	0	37	44	0	Misc ovary	20	1	1		1	OK	1	OK	1
916677374	M	850319	0	13	44	0	Misc sprain	102	1	1		1	1	1	1	1
916677374	F	850408	0	44	44	0	pericardium	255	1	1		1	1	1	OK	OK
916677374	F	850408	0	44	44	0	pericardium	255	1	1		1	1	1	OK	OK
916677374	F	850312	0	44	44	0	YOO	299	1	1		1			OK	OK
916677374	F	850419	0	44	44	0	YOO	27	1	1		1			OK	OK
918094874	F	851104	0	57	57	0	Cardio symptoms	200	1	1		1	1	1	OK	OK
918094874	F	851205	0	57	57	0	Misc cancer	23	1	1		1	OK	1	OK	OK
918094874	F	850619	0	56	57	0	Misc muscle	85	1	1		1	1	1	OK	OK
918094874	F	850718	0	56	57	0	Other eye prob	48	1	1		1	1	1	OK	OK
918094874	F	851107	0	57	57	0	Ovary cyst	55	1	1		1	1	1	OK	OK
918165837	F	850618	0	35	46	0	hypertension	78	1	1		1	1	1	OK	1
918165837	F	850917	0	47	46	0	hypertension	48	1	1		1	1	1	OK	OK
918165837	F	851126	0	47	46	0	YOO	9	1	1		1			OK	OK
918458775	F	850709	0	49	49	0	Misc cancer	95	1	1		1	OK	1	OK	OK
918458775	F	850626	0	49	49	0	Misc ovary	33	1	1		1	OK	1	OK	OK
918458775	F	850625	0	49	49	0	YOO	182	1	1		1			OK	OK
918512502	F	850326	0	46	46	0	Cardio symptoms	9	1	1		1	1	1	OK	OK
918512502	F	850501	0	46	46	0	Cardio symptoms	82	1	1		1	1	1	OK	OK
918512502	F	850315	0	10	46	0	Influenza	13	1	1		1	1	1	OK	1
918512502	M	850102	0	43	46	0	Misc cancer	50	1	1		1	OK	1	1	1
918512502	M	850515	0	44	46	0	Misc cancer	18	1	1		1	OK	1	1	1
918512502	M	851030	0	44	46	0	Misc cancer	18	1	1		1	OK	1	1	1
918512502	M	851127	0	44	46	0	Misc cancer	27	1	1		1	OK	1	1	1
918512502	M	850703	0	44	46	0	Misc circ	82	1	1		1	1	1	1	1
918512502	M	850718	0	44	46	0	Misc circ	43	1	1		1	1	1	1	1
918512502	F	850430	0	46	46	0	Misc dermatitis	163	1	1		1	1	1	OK	OK
918512502	M	850719	0	44	46	0	Misc heart	195	1	1		1	1	1	1	1
918512502	M	850617	0	44	46	0	Misc metabolic	21	1	1		OK	1	1	1	1
918512502	M	850627	0	44	46	0	Misc stomach	161	1	1		OK	1	1	1	1
918512502	M	850206	0	44	46	0	skin cancer	422	1	1		1	1	1	1	1
919128412	F	850701	0	61	61	0	Misc cancer	1911	1	1		1	OK	1	OK	OK

5.4 Example II: Osteoarthritis

For the second example, I will ask the question:

How much was spent on osteoarthritis in 1985?

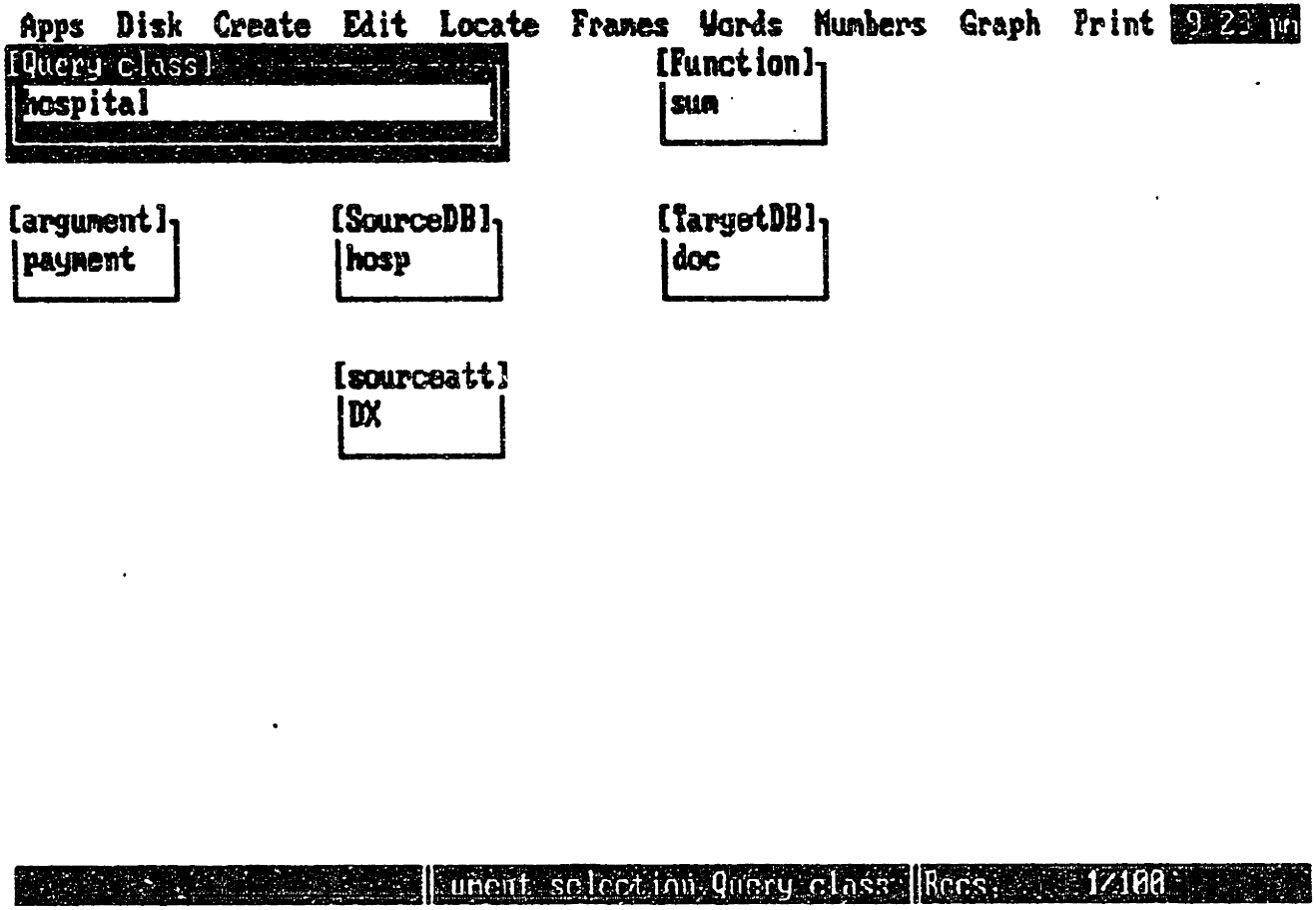
The preceding question involved an attempt to find records in the hospital database using physician records. Here, I want to find physician records given hospital records, as osteoarthritis is a diagnosis (found in the hospital database) and not a procedure (found in the physician database). Osteoarthritis, like other kinds of arthritis, is a disease which attacks the joints. Although usually treated on an outpatient basis, if arthritis becomes severe enough surgical intervention to replace a destroyed joint may be tried. Patients may also be admitted to a hospital for a battery of diagnostic tests.

Given a system which already contains the information about the same databases needed to answer the hysterectomy question, the system:

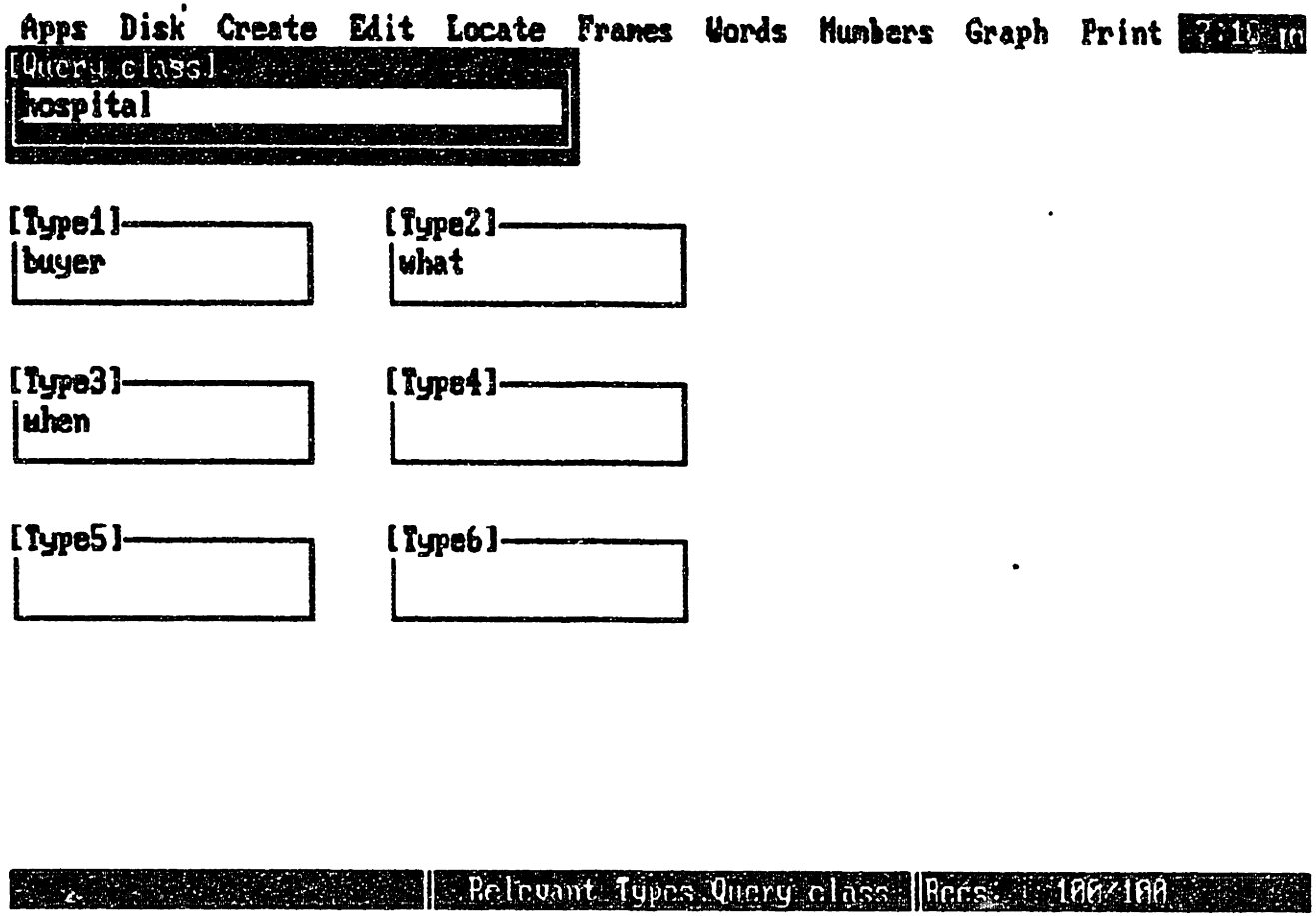
- Does not require the data analyst to enter any new information about data attributes because the same hospital and physician databases used in the preceding example are used. No data dictionary cross-tab is needed because one has already been done and the results stored. The existing possible comparison pairs are used.

- does require the end user to select "New Class" from the system menu (Menu, [E1]) and name the new query class by responding to the New Class name input prompt. The identities of the source and target databases have reversed, so a new query class and template are required. I named the new query class "Hospital" to represent the same relationships which exist for all instances of hospitalization-requiring-surgery. The user then fills in the "Argument Selection Entry Screen" [E20] and identifies the relevant universal attribute types in the "Relevant Type Entry Screen" [E30]. I attach here new versions of these screens showing these screens with values for this particular query class.
- The system selects a method for comparing the attributes in each relevant comparison pair and prepares a metatemplate to implement the chosen comparison method. The system then assembles the instantiated metatemplates into a template for the query class and updates the system menu so that succeeding users will know that the template has been prepared.

E2O Argument Selection (with Hospital values)



E3O Relevant Types Entry Screen (with Hospital values)



- The end user selects New Query (either from the menu or in response to a query when New Class has finished). The software associated with this function will prompt for a name of the query (here, "Osteo"), will display the Table Value Entry Screen [E4] for needed table values for the query attribute and for each consistency test implemented as a table lookup, and will display the User Constraints Entry Screen [E5] for any user constraints. The system responds by generating an upper and lower bound pair of SQL queries, writing the queries into a file, storing the table values, and creating an entry for "Osteo" under the query class of "Hospital" in the system menu.

As in the hysterectomy example, the following five members of the Preferred Comparison Pairs are in the end-user specified types:

Buyer - policyID:policyID done as an equijoin (upper & lower)

When - Date:Stay done as a range comparison (upper & lower)

What - Procedure:Diagnosis done as a table lookup (upper & lower)

Buyer - Age:Age done as an equijoin (lower only)

Buyer - Sex:Sex done as an equijoin (lower only)

In the New Query step, the software displays the Table Value Entry Screen [E4] with an appropriate prompt for the values of

DX (the query attribute) which identify osteoarthritis and also for the values of PROC which are consistent with diagnosis. The system then displays the User Constraint Entry Screen [E5] for any user constraints (such as the "in 1985" restriction) as are desired.

The 8 records retrieved from the source (hospital) database are attached, labelled as Hospital Osteoarthritis records [P4]. Note that this extract contains 7 distinct policyIDs. This implies there must be at least 7 distinct physician bills because no physician would admit a patient without doing anything.

When we retrieve the physician records based solely on the most discriminating criterion (policyID) we retrieve 99 records, 92 more physician bills than the seven suggested from the seven distinct policyIDs found on the hospital side. This printout is labelled Osteoarthritis Physician records [P5]. We know that we have retrieved all the correct records plus all the physician bills for both the individuals hospitalized and their family members.

The retrieved records are loaded into a spreadsheet with the other criteria executed by formulas in the same manner as the hysterectomy example. Summing the payments of those physician records tagged as upper yields $13,897. Summing the payments of those records tagged as lower yields $9,571. Adding in the $78,019 of hospital bills (which are the source records in this example) we have a range of $87,590 <= sum <= $91,916.

In this section's example, all the records should have been included in the upper bound. There were no records I could identify as improperly retrieved. All but one of the retrieved hospital records had a joint replacement operation. The one that did not had a number of diagnostic procedures done while in the hospital which were consistent with a diagnosis of osteoarthritis. These retrievals are consistent with what one might expect in the treatment of osteoarthritis.

P4 Hospital Osteoarthritis Records

Itemnum	PolicyID	Sex	Date	In?	Age	LOS	DX	Payment
1	912006782	M	850819	1	63	11	713	13735
2	913296967	F	850721	1	63	13	713	13122
3	913971302	M	850625	1	57	4	713	1754
4	916813432	M	850618	1	58	11	713	10198
5	918587255	M	850725	1	43	8	713	11574
6	919248300	M	850305	1	61	16	713	10163
7	919248300	M	850321	1	61	21	713	4435
8	919653860	M	850203	1	55	13	713	13038

P5 Osteoarthritis Physician Records

Item	PolicyID	Age	Sex	Date	Proc	Procname	Payment	U	L	UProc	LProc	Udate	Ldate	AgeT	SexT
1	912006782	63	M	850820	1330	replace hi	499	OK	OK	OK	OK	OK	OK	OK	OK
2	912006782	63	M	850923	1330	replace hi	23			OK	OK	1	1	OK	OK
3	912006782	63	M	850820	8938	muscle bio	34	OK		OK	1	OK	OK	OK	OK
4	912006782	63	M	850819	9153	exam	47	OK		OK	1	OK	OK	OK	OK
5	912006782	63	M	850819	9372	exam	14	OK		OK	1	OK	OK	OK	OK
6	913296967	57	F	850225	7222	X-ray	0			OK	1	1	1	1	OK
7	913296967	57	F	850225	7222	X-ray	11			OK	1	1	1	1	OK
8	913296967	57	F	850225	7308	X-ray	10			OK	1	1	1	1	OK
9	913296967	57	F	850225	7567	X-ray	14			OK	1	1	1	1	OK
10	913296967	57	F	850721	7102	X-ray	15	OK		OK	1	OK	OK	1	OK
11	913296967	57	F	850722	9153	exam	97	OK		OK	1	OK	OK	1	OK
12	913296967	57	F	850723	9153	exam	92	OK		OK	1	OK	OK	1	OK
13	913296967	57	F	850724	1330	replace hi	0	OK		OK	OK	OK	OK	1	OK
14	913296967	57	F	850724	1330	replace hi	765	OK		OK	OK	OK	OK	1	OK
15	913296967	57	F	850724	1330	replace hi	1824	OK		OK	OK	OK	OK	1	OK
16	913296967	57	F	850724	7220	X-ray	13	OK		OK	1	OK	OK	1	OK
17	913296967	57	F	850725	9074	exam	52	OK		OK	1	OK	OK	1	OK
18	913296967	57	F	850730	9074	exam	26	OK		OK	1	OK	OK	1	OK
19	913296967	58	F	850905	1330	replace hi	23			OK	OK	1	1	1	OK
20	913296967	58	F	850905	7222	X-ray	11			OK	1	1	1	1	OK
21	913296967	58	F	851107	7222	X-ray	11			OK	1	1	1	1	OK
22	913296967	58	F	851107	9025	exam	0			OK	1	1	1	1	OK
23	913971302	57	M	850531	7200	X-ray	75			OK	1	1	1	OK	OK
24	913971302	57	M	850531	7301	X-ray	32			OK	1	1	1	OK	OK
25	913971302	57	M	850610	7591	X-ray	80			OK	1	1	1	OK	OK
26	913971302	57	M	850625	9076	exam	73	OK		OK	1	OK	OK	OK	OK
27	913971302	57	M	850626	5193	myelograph	198	OK		OK	1	OK	OK	OK	OK
28	913971302	57	M	850626	9075	exam	29	OK		OK	1	OK	OK	OK	OK
29	913971302	57	M	850627	9074	exam	66	OK		OK	1	OK	OK	OK	OK
30	916813432	54	F	850411	9024	exam	28			OK	1	1	1	1	1
31	916813432	58	M	850528	7310	X-ray	8			OK	1	1	1	OK	OK
32	916813432	58	M	850528	7323	X-ray	10			OK	1	1	1	OK	OK
33	916813432	58	M	850528	9024	exam	0			OK	1	1	1	OK	OK
34	916813432	58	M	850528	9026	exam	0			OK	1	1	1	OK	OK
35	916813432	54	F	850618	7102	X-ray	17			OK	1	1	1	1	1
36	916813432	58	M	850618	9449	exam	0	OK		OK	1	OK	OK	OK	OK
37	916813432	58	M	850619	1335	replace kn	349	OK	OK	OK	OK	OK	OK	OK	OK
38	916813432	58	M	850619	1335	replace kn	2206	OK	OK	OK	OK	OK	OK	OK	OK
39	916813432	58	M	850619	6903	exam	33	OK		OK	1	OK	OK	OK	OK
40	916813432	58	M	850619	7312	X-ray	10	OK		OK	1	OK	OK	OK	OK
41	916813432	58	M	850621	9449	exam	0	OK		OK	1	OK	OK	OK	OK
42	916813432	58	M	850622	7323	X-ray	10	OK		OK	1	OK	OK	OK	OK
43	916813432	58	M	850625	9449	exam	0	OK		OK	1	OK	OK	OK	OK
44	916813432	58	M	850626	9153	exam	73	OK		OK	1	OK	OK	OK	OK
45	916813432	58	M	850627	7312	X-ray	10	OK		OK	1	OK	OK	OK	OK
46	916813432	58	M	850628	9449	exam	0	OK		OK	1	OK	OK	OK	OK
47	916813432	58	M	850726	7310	X-ray	9			OK	1	1	1	OK	OK
48	916813432	58	M	850726	7323	X-ray	10			OK	1	1	1	OK	OK
49	916813432	58	M	850726	9024	exam	0			OK	1	1	1	OK	OK
50	918587255	43	M	850515	9026	exam	0			OK	1	1	1	OK	OK
52	918587255	43	F	850725	7102	X-ray	15	OK		OK	1	OK	OK	1	OK

P5 Osteoarthritis Physician Records (cont.)

53	918587255	43	M	850725	9372	exam	13	OK	OK	1	OK	OK	OK	OK
54	918587255	43	M	850726	1330	replace hi	383	OK	OK	OK	OK	OK	OK	OK
55	918587255	43	M	850726	1330	replace hi	2610	OK	OK	OK	OK	OK	OK	OK
56	918587255	43	F	850726	7220	X-ray	14	OK	OK	1	OK	OK	1	OK
57	918587255	43	F	850726	7301	X-ray	17	OK	OK	1	OK	OK	1	OK
58	918587255	43	F	850801	7301	X-ray	17	OK	OK	1	OK	OK	1	OK
59	918587255	43	M	850807	195	ren suture	23		OK	1	1	1	OK	OK
60	918587255	43	F	850917	7301	X-ray	17		OK	1	1	1	1	OK
61	918587255	43	F	850917	7322	X-ray	21		OK	1	1	1	1	OK
62	918587255	43	M	850918	9026	exam	0		OK	1	1	1	OK	OK
63	918587255	43	F	851030	7301	X-ray	17		OK	1	1	1	1	OK
64	918587255	43	M	851030	9026	exam	32		OK	1	1	1	OK	OK
65	919248300	61	M	850206	1330	replace hi	2500		OK	OK	1	1	OK	OK
66	919248300	61	M	850211	8000	muscle exa	7	OK	OK	1	OK	OK	OK	OK
67	919248300	61	M	850306	1330	replace hi	470	OK	OK	OK	OK	OK	OK	OK
68	919248300	61	M	850306	9153	exam	73	OK	OK	1	OK	OK	OK	OK
69	919248300	61	M	850321	9153	exam	73	OK	OK	1	OK	OK	OK	OK
70	919248300	59	M	850322	9090	exam	63	OK	OK	1	OK	OK	1	OK
71	919248300	61	M	850326	9153	exam	73	OK	OK	1	OK	OK	OK	OK
72	919248300	61	M	850326	9428	exam	0	OK	OK	1	OK	OK	OK	OK
73	919248300	61	M	850326	9443	exam	0	OK	OK	1	OK	OK	OK	OK
74	919248300	59	M	850327	9093	exam	30	OK	OK	1	OK	OK	1	OK
75	919248300	61	M	850329	9173	exam	73	OK	OK	1	OK	OK	OK	OK
76	919248300	59	M	850401	9093	exam	30	OK	OK	1	OK	OK	1	OK
77	919248300	61	M	850402	9093	exam	38	OK	OK	1	OK	OK	OK	OK
78	919248300	61	M	850402	9172	exam	71	OK	OK	1	OK	OK	OK	OK
79	919248300	59	M	850405	9093	exam	30	OK	OK	1	OK	OK	1	OK
80	919248300	61	M	850409	9093	exam	38	OK	OK	1	OK	OK	OK	OK
81	919248300	61	M	850603	7304	X-ray	29		OK	1	1	1	OK	OK
82	919248300	61	M	850603	9025	exam	0		OK	1	1	1	OK	OK
83	919248300	61	M	850512	7880	X-ray	51		OK	1	1	1	OK	OK
84	919248300	61	M	850709	3560	liver biop	115		OK	1	1	1	OK	OK
85	919248300	61	M	850710	6793	exam	15		OK	1	1	1	OK	OK
86	919248300	61	M	850710	6903	exam	52		OK	1	1	1	OK	OK
87	919248300	19	F	851013	9123	exam	0		OK	1	1	1	1	1
88	919653860	55	M	850122	7304	X-ray	14		OK	1	1	1	OK	OK
89	919653860	55	M	850203	7102	X-ray	14	OK	OK	1	OK	OK	OK	OK
89	919653860	55	M	850122	7555	X-ray	17		OK	1	1	1	OK	OK
90	919653860	55	M	850203	7308	X-ray	10	OK	OK	1	OK	OK	OK	OK
91	919653860	55	M	850204	1330	replace hi	564	OK	OK	OK	OK	OK	OK	OK
92	919653860	55	M	850204	1330	replace hi	2490	OK	OK	OK	OK	OK	OK	OK
93	919653860	55	M	850204	7220	X-ray	11	OK	OK	1	OK	OK	OK	OK
94	919653860	55	M	850204	7308	X-ray	10	OK	OK	1	OK	OK	OK	OK
95	919653860	55	M	850219	2560	venogram	89	OK	OK	1	OK	OK	OK	OK
96	919653860	55	M	850219	7484	X-ray	36	OK	OK	1	OK	OK	OK	OK
97	919653860	55	M	850329	7304	X-ray	14		OK	1	1	1	OK	OK
98	919653860	55	M	850510	7222	X-ray	11		OK	1	1	1	OK	OK
99	919653860	55	M	850802	7222	X-ray	11		OK	1	1	1	OK	OK

5.5 Example III: Hurricanes

For the third example, I ask the question:

How much was received from our insurers in compensation for damage in the August 15th, 1985 hurricane on the Texas-Mexico border?

I will begin by discussing the hypothetical situation giving rise to these databases. Here, let us suppose that the problem is the parent company consolidating (adding up) insurance receipts from multiple subsidiaries, with each subsidiary having its own insurance company and general ledger (accounting) system. In this example, a firm has its headquarters in New York, a factory in Brownsville, Texas, an office in Galveston, Texas (350 miles away), and a maquiladora (duty-free manufacturing facility) just across the border in Matamoros, Tamulpais, Mexico. Operations in the United States have a different insurance company from the facility in Matamoros. Consolidating receipts for this hurricane is complicated because the Mexican subsidiary's database explicitly identifies cause and the US subsidiary's database does not.

Since the Mexican database contains a_q, "causa", the query attribute which identifies relevant records, it is the source database. The American database is the target database. The Mexican database has the following contents:

- causa (cause, WHAT type), the event causing the damage (ie, a_q),
- fecha (date, WHEN type), the day the damage occurred,
- dano (damage, WHAT type), the kind of damage,
- pago (payment, MONEY type), the amount of money paid in US dollars[25],
- ciuadad (city, WHERE type), city in which the payout was made, and
- estado (state, WHERE type), state in which the payout was made.

All records in the Mexican and American databases are implicitly payments to the parent of the subsidiary companies. The American database is the same as the Mexican database, except for the absence of the cause field:

- date, WHEN type, the day the damage occurred
- damage, WHAT type, the kind of damage
- payment, MONEY type, the amount of money paid
- city, WHERE type, city in which the payout was made
- state, WHERE type, state in which the payout was made

This is shown in [D3M], data dictionary contents for the two databases in this example.

---

[25]It is quite common for such contracts to provide for payments in a well-known stable currency when dealing with the capital assets of multinationals. This most commonly happens when the host country is relatively tiny (such as Israel or the Netherlands) or when the host country's currency is unstable (such as Mexico or Argentina).

D3M DATABASE DICITONARY CONTENTS (HURRICANE EXAMPLE)

___ 10-MAR-1989 14:23:29 _____

```
>select distinct *
>from datadich order by class, attname;
```

class	dbname	attname	domain	audited	contained	cardinality
$	US	payment	cont	y	n	1000000
$	mex	payment	cont	y	n	1000000
what	mex	causa	discrete	y	n	1000
what	US	damage	discrete	y	n	1000
what	mex	dano	discrete	y	n	1000
when	US	date	cont	y	n	1000000
when	mex	fecha	cont	y	n	1000000
where	US	city	discrete	y	n	1000
where	mex	ciuadad	discrete	y	n	100
where	mex	estado	discrete	y	y	20
where	US	state	discrete	y	y	50

End of Request - 11 Rows

In this section's example:

- The database analyst enters universal attribute type, audit status, and similar information about each attribute in the databases into the system's data dictionary using the Data Dictionary Entry Screen [D1M] and the Range Definition Entry Screen [D2M]. This information is required because the system does not yet have any information about these two databases. Here, rather than having a physician transaction log and a hospital transactions log, the parent company has two similar (but not identical) transactions logs from independent insurance companies. The database analyst must partition the logs' schemas into universal attribute types. Since this example also involves an instance of an accounting application, I use the same attribute types as in the hysterectomy question {BUYER, SELLER, WHAT, WHEN, MONEY}, and add WHERE, as property insurance is more geographically sensitive than medical insurance.
- Ingres produces comparison pairs from the data attribute information entered above. The resulting set of possible comparison pairs can be edited by the database analyst and then uploaded into the Framework part of the system. These comparison pairs are stored for future use by any query referencing this pair of databases. The comparison pairs are listed in [D4M] and [D5M], "Possible

Comparison Pairs for Mexico" and "Preferred Comparison Pairs for Mexico".

- The end user selects "New Class" from the system menu (Menu, [E1]) and names the new query class by responding to the New Class name input prompt. This step is necessary to create a template. I named the new query class "Mextex". The user then fills in the "Argument Selection Entry Screen" [E2M] and identifies the relevant universal attribute types in the "Relevant Type Entry Screen" [E3M]. I attach here copies of the above two screens showing values for this particular query class.
- The software associated with the New Class/template creation function selects a method for comparing the attributes in each relevant comparison pair and prepares a metatemplate to implement the chosen comparison method. The system then assembles the instantiated metatemplates into a template for the query class and updates the system menu so that succeeding users will know that the template has been prepared.
- The end user selects "New Query" (either from the menu or in response to a query when New Class has finished), gives a name to the query (here, "Bordercane"). The system then displays the Table Value Entry Screen [E5] for values for CAUSA, the query attribute, and also for CITY and DAMAGE, two attributes in comparison pairs

implememented as table lookups. After table values are entered, the system displays the User Constraints Entry Screen [E5] to permit the user to enter such constraints as are desired. In this case, he will enter an SQL clause which constrains retrieval to records on the dates the hurricane hit. The system responds by creating the SQL queries, writing them into a file, storing the table values for future use, and creating an entry for queryname under the query class of "Mextex" in the system menu.

The user names the relevant types as WHAT, WHEN, and WHERE, based on knowledge that the effects of a hurricane will show up as particular kinds of damage (eg wind, not earthquake), during a particular interval of time, in a particular geographic area. In this example, it is important to understand that the U.S. state of Texas and the Mexican stats of Tamulpais share a border and a hurricane landing at the border will cause damage in both states.

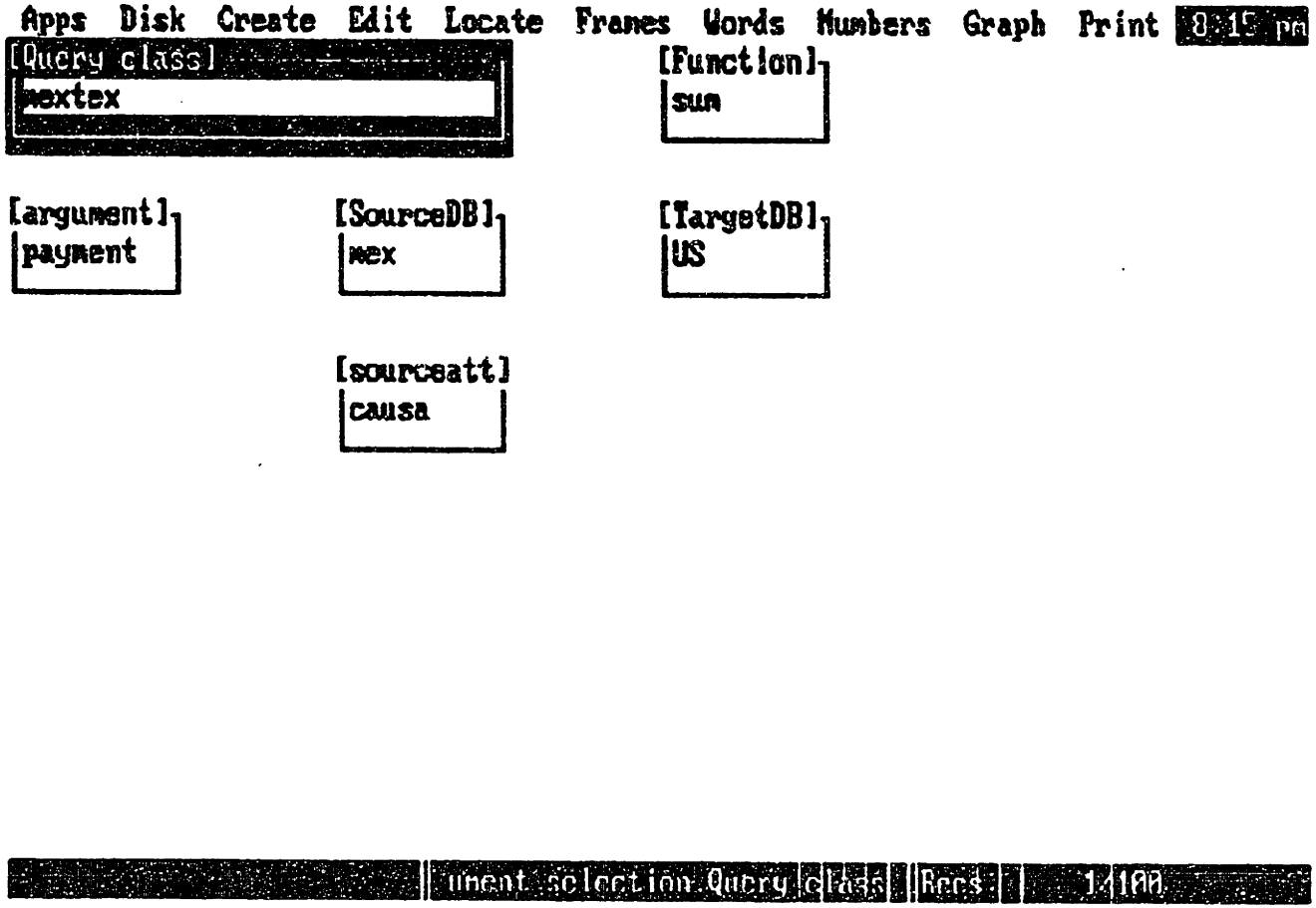
Three comparisons in Preferred Comparison pairs were in the types specified in the Relevant Type table:

Where - ciudad:city as table look-up (note that the estado:state comparison was eliminated in the database step as redundant)

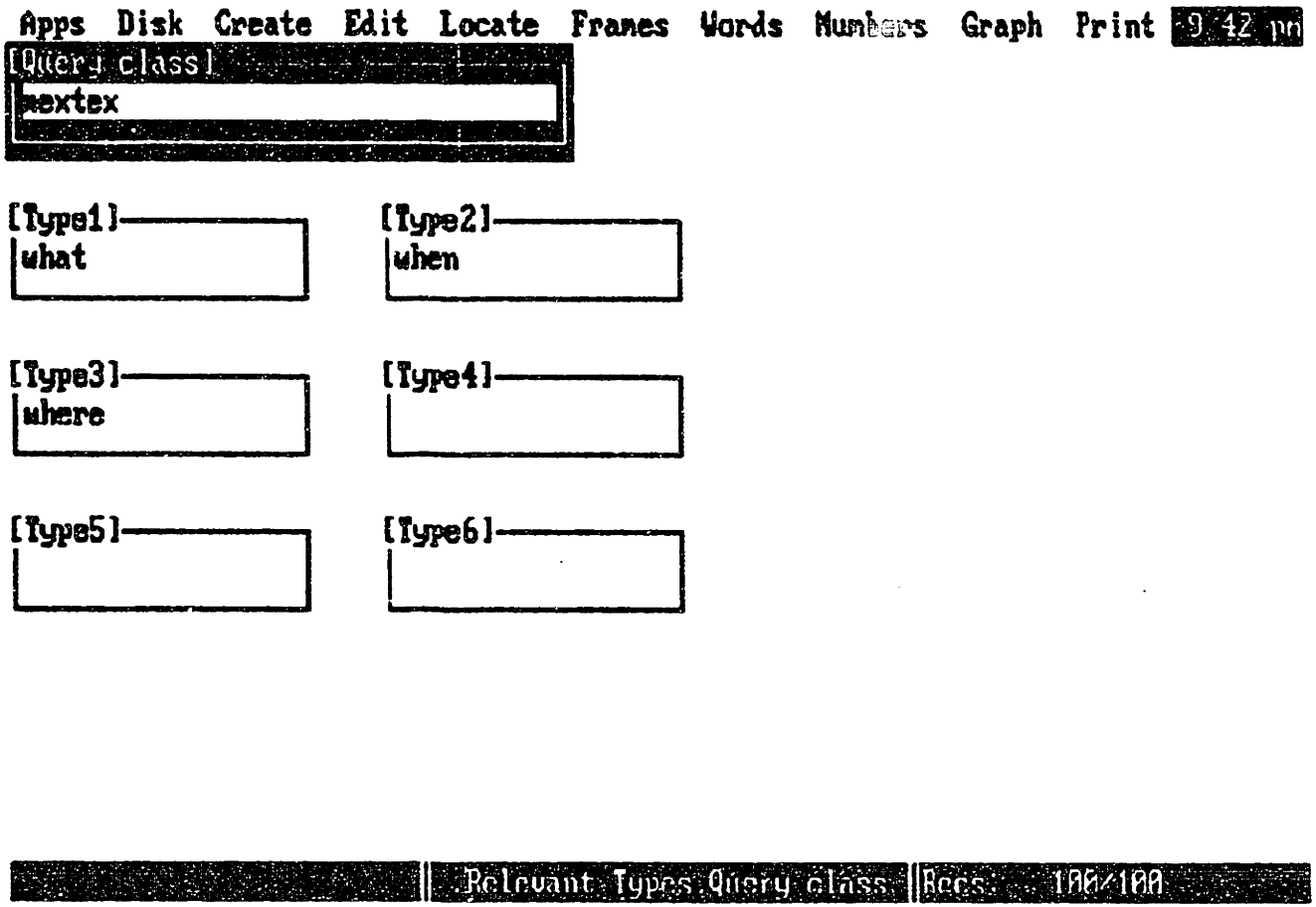
What - dano:damage as a table lookup (upper & lower) (incomparable domains)

When - fecha:date as an equijoin

E2M Argument Selection (with Mextex values)



E3M Relevant Types Entry Screen (with Mextex values)



D4M POSSIBLE COMPARISONS TABLE (HURRICANE EXAMPLE)

```
___ 10-MAR-1989 14:24:54 ___________________________________________________
>select
>a.class, a.dbname, a.attname, a.audited, a.contained,
>         b.dbname, b.attname, b.audited, b.contained
>from datadich a, datadich b
>where a.class = b.class
>and not a.dbname = b.dbname
>and a.dbname = "mex";
```

class	dbname	attname	audited	contained	dbname	attname	audited	contained
$	mex	payment	y	n	US	payment	y	n
what	mex	causa	y	n	US	damage	y	n
when	mex	fecha	y	n	US	date	y	n
what	mex	dano	y	n	US	damage	y	n
where	mex	ciuadad	y	n	US	city	y	n
where	mex	ciuadad	y	n	US	state	y	y
where	mex	estado	y	y	US	city	y	n
where	mex	estado	y	y	US	state	y	y

End of Request - 8 Rows

D5M PREFFERED COMPARISONS TABLE (HURRICANE EXAMPLE)

```
>select
>a.class, a.dbname, a.attname, a.audited, a.contained,
>         b.dbname, b.attname, b.audited, b.contained
>from datadich a, datadich b
>where a.class = b.class
>and not a.dbname = b.dbname
>and a.dbname = "mex"
>and a.contained = "n"
>and b.contained = "n"
>and a.audited = "y"
>and b.audited = "y";
```

class	dbname	attname	audited	contained	dbname	attname	audited	contained
$	mex	payment	y	n	US	payment	y	n
what	mex	causa	y	n	US	damage	y	n
when	mex	fecha	y	n	US	date	y	n
what	mex	daño	y	n	US	damage	y	n
where	mex	ciuadad	y	n	US	city	y	n

End of Request - 5 Rows

After the user has selected "New Query" and named the query being created, the system determines that it needs a set of values for "CAUSA" in order to locate relevant source records, values for "CITY" which are consistent with the storm track, and values for "DAMAGE" which are consistent with the likely hurricane damage. The system also prompts for user constraints and we enter a requirement testing for dates when the hurricane was present. (Otherwise it would find all hurricanes that had ever hit the area and were in these databases.)

Executing the source query retrieves three records for Matamoros, one for wind damage, one for water damage, and one for glass breakage. These are in the printout labeled [P6] Mexican Insurance Records (Source Database). These total to $18,000.

P6 Mexican Insurance Records (Source Database)

causa	fecha	dano	pago	ciudad	estado
huracan	850815	viento	5000	Matamoros	Tam
huracan	850815	inudacion	10000	Matamoros	Tam
huracan	850515	vidrio	3000	Matamoros	Tam

In the target database retrieval I have listed all the records in the entire hypothetical database and applied the above three tests of consistency. This is on the printout labelled [P7] US Insurance Records (Target Database). In this hypothetical database:

- Record one is excluded because it is in New York, which is outside of the storm track.
- Record two is excluded because there was no hurricane on that day.
- Record three is excluded because the date is wrong[26].
- Record four is excluded from the lower bound because Galveston is not in the storm track. It is in the upper bound since it is completely plausible that a hurricane which hit Brownsville caused sufficiently heavy rain in Galveston to cause flooding.
- Records five and six meet all upper and lower bound tests.
- Record seven is excluded from the lower bound because it is not certain the hurricane caused the fire. It is, however, certainly possible a power line downed by the storm did cause the fire.

---

[26]I note that there is unlikely to be a problem with two-day hurricanes, since damage will most likely be caused on both sides of the border on both days, leaving a source record which will link up with target records. The Select Distinct function in SQL prevents double-counting.

P7 US Insurance Records (Target Database)

date	damage	payment	city	St	DateT	UDamageT	LDamageT	UCityT	LCityT
850815	glass	1000	New York	NY	OK	OK	OK	1	1
850218	water	3000	Brownsville	TX	1	OK	OK	OK	OK
850319	car crash	10000	Brownsville	TX	1	1	1	OK	OK
850815	water	6000	Galveston	TX	OK	OK	Ok	OK	1
850815	glass	2000	Brownsville	TX	OK	OK	OK	OK	OK
850815	water	5000	Brownsville	TX	OK	OK	OK	OK	OK
850515	fire	500	Brownsville	TX	OK	OK	1	OK	OK

The upper bound target database total is $13,500 and the lower total is $7,000. Adding in the source database retrievals we conclude $25,000 <= x <= $31,500.

The goal of this chapter was to work through three example retrievals. Now that this task is concluded, I turn to general conclusions in the next chapter.

Chapter 6: Conclusions

6.1 Summary

In this work, I have developed a theory of approximating keys to connect multiple databases whose data have a common cause. If we could discover which attributes reflected that common cause and how to compare those attributes, we would have a series of tests which, taken together, would approximate the effect of a key.

This discovery can be done by classifying the attributes in each database serving a particular type of application into application-required universal types. I assume data analysts can do this classification.

Different databases may be implicitly linked by having attributes in the same types. Potentially useful tests of consistency can be discovered by taking a cross-product of the data dictionary (a collection of information about the databases' attributes) to create a series of potentially comparable pairs of attributes. End-users then select the universal attribute types relevant to the query.

The system then selects comparison methods using built-in heuristic rules and processes those comparison methods into an executable query in the local database's query language. I dealt with the problem of non-comparable attributes by extending the mainstream concept of equality comparison (equijoin) to include incomplete methods of range comparison and table comparison. I dealt with the problem of potentially

inaccurate attribute values by using incomplete information principles to produce both a must-be and might-be query which bounds the true result.

6.2 Practical results

There are a number of practical reasons why this research is significant. First, this method is a true retrofit which does not affect the underlying databases. True retrofits are useful for the following reasons:

- Unanticipated query Retrofit is the only option for answering queries that had not been anticipated or could not have been anticipated at design time.
- Present value of delay All other things being equal, delay in paying construction costs saves money if interest rates and construction costs are positive. If the inaccuracy in the methods described here are acceptable, retrofit would be preferred due to present value considerations.
- Uncertainty of future query The value of the ability to retrofit goes up as the future need to ask the query becomes more uncertain.

Second, I have made a contribution to user interface design because the present system reduces what an end-user must know to query a database. Unless the database designer has anticipated all future questions and written a set of easy-to-

find canned queries, traditional databases require an end-user to possess the following types of knowledge:

- Schema: Which attributes correspond to the information sought and which attributes are needed to identify the records to be retrieved.
- Search: Enough database theory to enable using the above knowledge to find and retrieve the desired records. For example, understanding joins is important when more than one relation is to be accessed in a relational database.
- Language: Must know the local query language well enough to write the query.

It is useful to discuss what things the users do not have to know in my system. They do not need to know:

- the names of data items, other than the type name of the one they are trying to aggregate.
- anything about joins, the creation of comparison pairs or the methods of comparing members of comparison pairs.
- the grammar or vocabulary of the underlying query language.

Third, this method enables the system to service legitimately inconsistent views defined on the same data. Because this is a true retrofit, multiple interpretations can be stored. There may be legitimate business reasons for having multiple definitions of "the same thing" due to differences in the use of the data. For example, tax accountants want "income" to be

high. Both are defined on the same set of actual events. Using my approach, one can create a different query class for each of the non-agreeing interest groups. Each of these query classes would have associated with it a number of queries with table values meaningful to that particular interest group. When retrieving, one would simply use the query classes associated with one's own interest group.

6.3 Research Contributions

The above are the principal practical reasons for this research; I will now turn to the research contributions. I can divide the discussion into the two areas of database metaknowledge and the ways that knowledge would be represented, stored, and executed.

Application definitions force attributes with particular meanings to be present in the schema of a supporting database. The restrictions inherited from the application type are stable across multiple databases of the same application type. The actual amount of knowledge required per query is not large because the process does not require the system to have an understanding of causality.

The second area is how the required knowledge was to be stored, represented, and executed. I believe much of this work to be consistent with ongoing research in databases and in artificial intelligence. Federated databases refers to McLeod's research in enabling connectivity without a centrally

maintained schema. A federated architecture works by having each database maintain an import schema and an export schema, with the query processor "negotiating" between the databases at query time. Actually writing a negotiation script, however, requires users to do the equivalent of writing code, as the facilities defined to support negotiation are fairly low-level.

My research adds a higher-level representation using knowledge about the application which makes creating the equivalent of a negotiation script much simpler. As discussed in the preceding comments on user interface benefits there is a division of labor between database analysts and end-users, with each contributing knowledge and information each is likely to possess -- as opposed to end-users writing negotiation scripts using relatively low-level constructs.

Further, only the contributions of the database analysts need be centrally stored and maintained. Reducing the amount of central storage and, by extension, the amount of consistency checking and standards enforcement is a key contribution of McLeod's work. In my approach, database analysts need develop a series of universal attribute types, which need be done only once per application type. Database analysts must also classify attributes in existing databases into those types, which need be done only once per database. The resulting comparison pairs need to be stored centrally. Avoiding conflicts in universal attribute type definition may be easier than avoiding conflicts in the design of the central schema -- particularly since the

former may be changed retroactively without affecting the underlying data in the databases. (Predefined queries would need to be rewritten).

Causal models and conceptual cluster analysis: Davis' research in fault detection emphasized the importance of paths of causal interaction and found that adjacencies (points of contact between alternative representations of the same thing) are important and much easier to find by having representations from different viewpoints in which those adjacencies were readily apparent. (See discussion in the literature review)

In Davis' approach to fault detection, multiple complete representations of the underlying system had to be loaded into the system and the goal was to find the adjacencies relevant to solving the problem. Here, we have considerable prior knowledge as to what the adjacencies must be -- the adjacencies must be in pairs of attributes in the same type but of different databases. Further, the person trying to "solve the problem" -- the end-user trying to retrieve relevant information -- is unlikely to have much difficulty stating which types are relevant, so we do not need complete multiple representations of, say, medicine or meteorology to enable the system to attempt to infer which attribute types are relevant. However, the concept of multiple representations and of hits in the intersection remains important here since database knowledge and subject knowledge are separately loaded, and separately

stored, and are brought together by the system to accomplish something not possible with either alone.

Finally, the use of tables to resolve comparisons that can only be resolved by checking for membership in a table of reasonable values is consistent with research on incomplete information databases. The focus of my research is different, however, as that literature does not deal with joins, but with retrieving data from a specific view. In the future, it would be appropriate to expand my system to use incomplete information principles to relax the assumption that the relevant records in the source database can be precisely identified.

6.4 Future Research

Some of the limitations in this work would be ameliorated by judicious use of object-oriented databases. More specifically:

- my use of table look-ups for resolving comparison pairs would be facilitated by the extensible types of PROBE and POSTGRES
- extending this system to recreate event chains would be facilitated by the recursion features of PROBE
- adding contingent knowledge would be facilitated by POSTGRES' alerters and triggers
- implementing more descriptive application descriptions would also be facilitated.

This concludes the dissertation. Some enhancements are appropriate for future research. Certainly implementation of the group consistency and order consistency tests discussed in section 3.3 (Testing for Consistency Under Incomplete Information) would be highly desirable. The present implementation does only a test of record consistency.

Also desirable would be support for hierarchies of type and table value definitions. The current implementation is "flat".

Finally, support for inconsistent views with a mechanism for explaining the inconsistency would be interesting.

Citations

[Al-Fedaghi81]
Al-Fedaghi, S., and Scheurmann, P. Mapping considerations in the design of schemas for the relational model, IEEE Trans. Softw. Eng SE-7, 1 (Jan.)

[Batini84]
Batini, C., Demo B., and DiLeva, A. A methodology for conceptual design of office database systems. Inf. Syst. 9, 3, 251-263

[Batini86]
Batini, C. Lenzerin, M., and Navathe, S.B., "A Comparative Analysis of Methodologies for Database Schema Integration", ACM Computing Surveys, Vol. 18, December, 1986 (pp. 323-364)

[Biskup83]
Biskup, "A Foundation of Codd's Relational Maybe Operations", ACM Transactions on Database Systems, Vol. 4, December, 1983 (pp. 608-636)

[Casanova83]
Casanova, M. and Vidal, M. "Towards a sound view integration methodology", In Proceedings of the 2nd SIGACT/SIGMOD Conference on Principles of Database Systems (Atlanta, Ga. Mar. 21-23). ACM, New York, pp.36-47

[Chang85]
Chang, C.L. and Walker, A., "PROSQL: A Prolog programming interface with SQL/DS", Proceedings of the First International Workshop in Expert Database Systems, Larry Kerschberg, Ed, Benjamin/Cummings, Menlo Park, CA 1986 (p. 233)

[Chen76]
Chen, P. "The Entity-Relationship Model -- Toward A Unified View of Data", ACM Transactions on Database Systems, Vol. 1, March, 1976 (pp. 9-36)

[Chomicki83]
Chomicki, J. and Grundzinski, W., "A Database Support System for PROLOG", Proceedings of the Logic Programming Conference, Portugal, June, 1983

[Codd79]
Codd, E.F., "Extending the Relational Model to Capture More Meaning", ACM Transactions on Database Systems, Vol. 4, December, 1979, (pp. 395-434)

[Date86]
Date, C.J., "An Introduction to Database Systems, Vol. 1", Addison-Wesley, 1981

[Davis84]
Davis, R. "Diagnostic reasoning based on structure and behavior", in Bobrow, Daniel ed. "Qualitative Reasoning about Physical Systems", Cambridge:MIT Press, 1984

[Dayal84]
Dayal, U. and Hwang, H. "View definition and generalization for database integration in multibase: A system for heterogenous distributes databases", IEEE Trans. Softw. Eng. SE-10, 6 (Nov.), 628-644

[Dayal86]
Dayal, U. et al, "PROBE -- A preliminary analysis", Computer Corporation of America, Cambridge, MA

[ElMasri87]
Elmasri, R., Larson, J., and Navathe, S.B., "Integration algorithms for federated databases and logical database design", Tech.Rep., Honeywell Corporate Research Center.

[Gallaire78]
Gallaire, H. and Minker, J. Eds, Logic and Databases, Plenum, New York, 1978

[Gallaire84]
Gallaire, H., Minker, J., and Nicolas, "Logic & Databases: A Deductive Approach, Computing Surveys, Vol. 12, No. 2, June, 1984, (p. 152)

[IBM81]
International Business Machines, Inc., Business Systems Planning, IBM Manual #GE20-0527-3, July, 1981

[Imielinski84]
Imielinski, T. and Lipski, W., "Incomplete information in relational databases, Journal of the ACM, Vol. 31, No. 4, October, 1984, (pp. 761-791)

[Jarke85]
Jarke, M. and Vassiliou, Y., "Coupling Expert Systems with Database Management Systems" in Artificial Intelligence Applications for Business, W. Reitman ed, Ablex, 1984

[Kahn79]
Kahn, B., A structured logical database design methodology, Ph.D. dissertation, Computer Science Dept, Univ. of Michigan, Ann Arbor, Mich.

[Lipski79]
Lipski, W., "On semantic issues connected with incomplete information databases, ACM Transactions on Database Systems, Vol. 4, No. 3, September, 1979, p. 262

[Lipski81]
Lipski, W., "On databases with incomplete information", Journal of the ACM, Vol. 28, No. 1, January, 1981, (pp. 41-70)

[Lloyd83]
Lloyd, J., "An Introduction to Deductive Database Systems", The Australian Computer Journal, Vol. 15, May, 1983 (pp. 52-57)

[McLeod85]
Heimbigner, D. and McLeod, D., "A federated architecture for information management," ACM Transactions on Office Information Systems, Vol. 4 No. 3, (July, 1986), (pp. 271-287)

[Mannino84]
Mannino, M.V., and Effelsberg, W., 1984 :A methodology for global schema design", Computer and Information Sciences Dept., Univ of Florida, Tech. Rep. No. TR-84-1, Sept.

[Martin82]
Martin, J., Strategic Data-planning Methodologies, Prentice-Hall: Englewood Cliffs, NJ, 1982

[Mendelsohn86]
Mendelsohn, H. and Saharia, A., "Incomplete Information Costs & Database Design", ACM Transactions on Database Systems, Vol. 11 No. 2, June, 1986, (p. 159)

[Minker80]
Minker, J., "An Inferential Relational System", ACM Transactions on Database Systems, Vol. 3, No. 1, March, 1978, (pp. 1 - 31)

[Motro81]
Motro, A. and Buneman, P., 1981, "Constructing superviews", in Proceedings of the International Conference on the Management of Data, (Ann Arbor, MI, Apr.29 - May 1), ACM, New York

[Naish83]
Naish, L. and Thom, J.A., "The MU-Prolog Deductive DB", Technical Report 83/10, Department of Computer Science, University of Melbourne, Australia

[Navathe82]
Navathe, S.B., and Gadgil, S.G., "A methodology for view integration in logical database design", in Proceedings of the 8th International Conference on Very Large Databases (Mexico City) VLDB Endowment, Saratoga, CA

[Parker86]
Parker, Carey, Golshani, Jarke, Sciore, and Walker, "Logic Programming & Databases", in Proceedings of the First International Workshop in Expert Database Systems, Larry Kerschberg, Ed, Benjamin/Cummings, Menlo Park, CA 1986

[Prade84]
Prade, H., "Lipski's approach to incomplete information databases restated and generalized in the setting of Zadeh's possibility theory", Information Systems, Vol. 9 No. 1, (pp. 27-42)

[Pindyck76]
Pindyck, R.S. and Rubinfeld, D.L., Econometric Mocdels and Economic Forecasts, New York:McGraw-Hill, 1976

[Reisner75]
Reisner, P., "Human factors studies of database query languages: a survey and assessment", Computing Surveys, Vol. 13, No. 1, March, 1981, (P. 13-31)

[Reiter78]
Reiter, R., "On closed world databases" in [Gallaire78]

[Sciore85]
Sciore, E. and Warren, D., "Towards an integrated databases -- PROLOG system", in Proceedings of the First International Workshop in Expert Database Systems, Larry Kerschberg, Ed, Benjamin/Cummings, Menlo Park, CA 1986, p. 293

[Smith81]
Smith, J. et al, "MULTIBASE -- Integrating heterogenous distributed database systems," National Computer Conference, 1981, (pp. 487-499)

[Smith86]
Smith, J. "Expert database systems: a database perspective", Proceedings of the First International Workshop in Expert Database Systems, Larry Kerschberg, Ed, Benjamin/Cummings, Menlo Park, CA 1986

[Spyratos87]
Spyratos, N., "The partition model: a deductive database model", ACM Transactions on Database Systems, Vol. 12, No. 1, March, 1987, (p. 1)

[Stonebraker84]
Stonebraker, M., "Extending a relational interface for expert systems applications", Dept. EECS Technical report, University of California, Berkeley, 1984

[Stonebraker87]
Stonebraker, M. and Rowe, L., "The Design of POSTGRES" in The POSTGRES PAPERS, University of California, Berkeley ERL Memo M86/85, June 25, 1987

[Su83]
Su, S.Y., "SAM*: A semantic association model for corporate and scientific-statistical databases," Information Sciences, Vol 29, 1983, (pp. 151-199)

[Teorey82]
Teorey, T. and Fry, J., 1982, Design of Database Structures, Prentice-Hall, Englewood Cliffs, NJ

[Ullman85]
Ullman, J., "Implementation of Logical Query Languages for Databases", ACM Transactions on Database Systems, Vol. 10, No. 3, September, 1985, (pp. 289-321)

[Vassiliou79]
Vassiliou, Y., "Null values in database management" A denotational semantics approach", in Proceedings of SIGMOD of the ACM, (Boston, May 30-June 1, 1979) (pp. 162-169)

[Vassiliou80]
Vassiliou, Y. "Functional dependencies and incomplete information" in Proceedings of the 6th Conf. on VLDB, (Montreal, Oct. 1-3, 1980) ACM, NY (pp. 260-269)

[Wang88a]
Wang, Y.R. & Madnick, S., "Connectivity among information systems", Sloan Working paper #2025-88

[Wang88b]
Wang, Y.R., Madnick, S., and Horton, D., "Inter-database instance identification in composite information systems", Sloan Working paper #2029-88

[Wiederhold79]
Wiederhold, G., and ElMasri, R. 1979, "A structural model for database systems", Rep. STAN-CA-79-722, Computer Sciences Dept., Stanford Univ, Stanford, CA

[Wong82]
Wong, E., "A statistical approach to incomplete information in database systems", ACM Transactions on Database Systems, Vol. 3, No. 3, September, 1982, (pp. 470)

[Yao82]
Yao, S.B., Waddle, V. and Housel, B. 1982, "View Modelling and integration using the functional data model" IEEE Trans. Softw. Eng. SE-6, 6, 544-553

[Zadeh86]
Zadeh, L., "A simple view of the Dempster-Shafer theory of evidence and its implication for the rule of combination", AI magazine, Vol. 7, No. 2, Summer, 1986, (pp.85-90)

[Zaniolo86]
Zaniolo, C., "PROLOG: A database query language for all seasons", Proceedings of the First International Workshop in Expert Database Systems, Larry Kerschberg, Ed, Benjamin/Cummings, Menlo Park, CA 1986