Submitted to ACM Transactions on Database Systems.

Graph Data Models for Genomics

Mark Graves

Department of Cell Biology, Baylor College of Medicine

Correspondence:

Mark Graves
Department of Cell Biology
Baylor College of Medicine
One Baylor Plaza
Houston, TX 77030

mgraves@bcm.tmc.edu (e-mail)
713-798-8271 (voice)
713-798-3759 (fax)

Abstract

Genome databases have specific requirements which limit the usefulness of some database management systems. We have developed a data representation based on graphs which capture the highly interconnected structure of genome data. Graphs form a language which can be tailored for describing genome information, and we develop a data model based on graphs.

Our graph data model defines a graph as a collection of vertices, edges, and edge labels. We extend the basic definition of a graph by adding constructs to capture external identifiers, ordered collections of data, packaged graphs, and graph templates for querying and creating updatable views. We also describe a graph schema language which can be used for conceptual modeling.

1. Introduction

Genome researchers need database systems which will capture information about all the genes in an organism. Each living organism is described by a collection of genes: this collection is called a *genome*. Over the next few years, biologists will succeed in discovering all the genes which make up a person; how each person is different; how humans are biologically different from species such as mice; and what is the genetic bases of disease. Then, there will be many more questions, such as: how are genes regulated?; what does each gene do in isolation?; how do groups of genes interact?; and how can the effects of disease be prevented, avoided, or cured? To answer these questions, biologists are not only going to have to do laboratory experiments, they will also have to retrieve a growing amount of information from databases.

Biological information is heavily interconnected. To study this information in a framework which mimics the processes being studied, the database must support large quantities of heavily interconnected data. Current databases do not support the variety of representations or the range of queries that biologists need. As in any field that depends upon computer science, biologists will continue to move forward despite limitations in computational support. However, the rate at which biologists discover the genetic bases of disease will be affected by the computational support they receive.

Our approach to providing computational support for storing genome information in a database has been to examine the needs of biologists and the requirements from genome data and to develop a data model which captures the structure of genome data. Here, we describe the data model. A Data Note System based on this data model has also been developed.

2. Genome Data

Genome data is difficult to capture using current technology. Although database systems are sufficient to capture the quantity of data being generated, they are inadequate for the complexity of the data. Other areas of computer science have developed means of dealing with these complexities, but no single previously developed system supports the needed functionality. We propose a data model which includes the functionality needed to capture the complexities of genome data.

Genome data is typified by:

1. large, rapidly growing quantities,

2. complex, interconnected structures,

3. rapidly changing types,

4. imprecise definitions,

5. uncertainty, and

6. nonmonotonic relationships.

Genome researchers need database systems to capture genome data. The quantity of data has overwhelmed what can be stored in traditional laboratory notebooks, and the amount of data being generated is growing at an exponential rate. New, automated laboratory techniques are likely to continue increasing the rate of data generation. Database systems are currently capable of storing the data being generated by genome projects and will continue to be pushed by other technologies, such as multi-media systems and astronomy satellites, which generate much larger quantities of data than biology. However, current database systems are not completely adequate.

Genome data is highly-interconnected and has a complex structure which is difficult to capture in current data models. Genetic information consists of large numbers of highly interconnected

constructs: genes, regulatory mechanisms, introns, exons, alternative splicings, binding sites, tissue-specific expression, polymorphisms, homologous regions, etc. Each construct is not primitive, but is defined by its relationships with other constructs, for example a gene is not *a priori* a gene, but is a region of DNA with some sequence which is translated into some protein and expressed in some cell. It is the relationships between the sequence, protein, and expression which make a region of DNA a gene. The intrinsic connectivity of genome data should be captured in a database. Graph database management systems provide a framework for capturing the graph-like interconnection of genome data at a higher level of abstraction than earlier network or semantic database management systems.

Genomics is changing rapidly. *Genomics* is the study of the collection of all genes in an organism: their structure, function, and interaction. Because genome research is advancing at a rapid rate, it is important that genome data be represented in a flexible framework which can be easily extended. The type of data being generated in laboratories changes frequently. Many labs use techniques, such as polymerase chain reaction, which were discovered only a few years ago, and techniques which were a great advance in science a decade ago, such as using restriction fragment length polymorphisms to generate maps, are now obsolete. Databases have never been developed to capture data from such a rapidly changing science, and techniques to speed up the development of databases are still immature. By developing a database management system upon a representation with which the domain experts are familiar, the time spent in designing each database can be reduced [Graves 1995, 1996].

Genome concepts are not logically and precisely defined. Unfortunately, most data representations assume the existence of precise, logical definitions which causes difficulties in using them for biological databases. In some domains, a set of undefined primitives may be defined upon

which all other definitions are built. In an experimental science, such as biology, the concepts are defined over time and the process of describing the concepts, such as genes, promoters, and regulatory mechanisms, embody much of the science. An advantage of using graphs for representing concepts is that the definitions can be slowly refined by adding to the existing structure without making changes which would affect previously developed components or queries. Our graph representation has been influenced by graph languages in artificial intelligence and databases, such as conceptual graphs [Sowa 1984], terminological description languages [Nebel & Smolka 1990], and ψ-types [Ait-Kaci 1984].

Experimental data in inherently uncertain because of the types of observations, experimental errors, and processes being studied. Biologists use indirect methods to study phenomena which cannot be directly observed. A sophisticated collection of protocols have been developed to allow biologists to assay essentially submicroscopic information, such as Southern blots, β-galactosidase colorimetric tests, and hybridization procedures. Many experiments have error rates of 30-70%, and highly-refined processes such as single-pass sequencing rarely have error rates of much less than 1%. Another source of uncertainty is the nature of biological processes themselves where balances, trends, and small changes in minute quantities effect the mechanisms being studied. A data model must support a flexible mechanism to view the data in many ways to both record the uncertainty and allow the user to choose which parts to hide when examining the data.

Uncertainty has an affect on the reasoning that can be done on biological processes. Nonmonotonic reasoning is an area of artificial intelligence which studies reasoning systems where new information can change previously accepted results. Most (if not all) rules in biology have exceptions, and even widely-applicable rules may have strange interactions. For example, one biological experiment, called hybridization, tries to determine if one segment of DNA is essentially

identical to another segment by seeing if samples of the segments will “hybridize”, or connect together, under certain laboratory conditions. In many computational systems, one could express a relationship of “hybridizes-to” and define inference rules for certain kinds of hybridization experiments such as: if A hybridizes-to B and B hybridizes-to C, then A hybridizes-to C. In genome systems rules like this do occur, and inferences should be made that if B hybridizes-to C and C hybridizes-to D, then B hybridizes-to D. Unfortunately, if the “hybridizes-to” relationship represents a hybridization experiment with an error rate of 30%, then while the aforementioned inferences should be made, the inference A hybridizes-to D should not. Our first step in addressing this problem is to provide a query mechanism which simplifies the development of complex queries that will be necessary for developing applications which perform the nonmonotonic reasoning needed in biological problem solving.

3. Graph Representation

In developing any complex, data-intensive system, the most crucial technical decision to make is the choice of data representation. Data representation affects the ease at which algorithms can be developed, the efficiency at which they perform, the extensibility and maintenance of the software, the accuracy of the data to be captured, and the operations which can be performed.

The requirements for the representation of genome data is addressed by several areas of computer science research, such as database systems, software engineering, and artificial intelligence. Database systems provide support for large amounts of data, and software engineering addresses the need to build systems which frequently change [Connell and Shafer 1989; Prieto-Diaz and Arango 1991; Coleman et al. 1994]. Two subareas of artificial intelligence have contributed representation languages which can be used to describe genome data: knowledge representation and computational linguistics. Knowledge representation formalisms can describe complex, highly-in-

terconnected concepts, and computational linguistics has developed ways of capturing the imprecise definitions of nature in a computational language. One useful knowledge representation language for genome data is semantic networks. Semantic networks capture the meaning of concepts by describing how they are related to other concepts. Semantic networks represent concepts and relationships as a graph with labeled nodes and arcs. Each concept is defined by its relationship to other concepts. Semantic network formalisms are used in two genome database projects, LabBase [Goodman et al. 1995] and ACeDB [Durbin and Theirry-Meig 1991]. Computational linguistics provides formal languages to describe vague, ambiguous, and conflicting concepts. Semantic networks are not sufficient to represent concepts which are not precisely and logically defined. One of the most difficult representation problem which has been tackled by computer science is to capture the semantics of natural language. Computational linguists have extended formalisms such as semantic networks to capture the impreciseness of natural language. Luckily, some of these formalisms are also useful for capturing genome data.

A graph representation language could be formed from many possible graph data structures. Graphs are pervasive in computer science and are the basis of several representation languages, including: semantic networks, ER diagrams, functional data models, semantic data models, terminological description languages, attribute value formalisms, finite-state automata, Petri nets, function nets, data flow diagrams, conceptual graphs, ψ-types, feature structures, and feature logics.

The best data representation for capturing genome data is a graph. Graphs capture the intrinsic connectivity of genome relationships, and they are a comfortable communication medium for biologists and computer scientists. However, there are many kinds of graphs and many graph languages available from computer science. Genome data must be examined more closely to

distinguish the kinds of graphs required to capture the structure of the data, and then a graph language should be chosen or developed which can represent the data without unnecessary complexity. After an appropriate graph representation language has been developed, then a data model can be formed to capture genome data in a database system.

3.1 Graphs in Genomics

Graphs provide a framework to flexibly and accurately capture the structure of genome data. Graphs have been shown useful for representing genetics [Mirkin and Rodin 1984], and graph data structures are used in programs of interest to biologists, such as phylogenetics programs, the map drawing program SIGMA [Cinkosky 1992], and sequence assembly software. A representation language designed to capture genome data should:

1. capture highly-interconnected data,
2. be based on binary relationships,
3. emphasize concepts and relationships equally,
4. describe complex, regular structures, and
5. support cyclic graphs.

Genetic data has a graph-like structure. Relationships in genetics are determined by experimental evidence. In molecular biology, most of the science is based on indirect observations, and the results of these experiments are relationships between the reagents used in the experiment and the result found, i.e., hybridization results, order information or function. There are few primitive concepts in genetics, other than the biochemistry, so most of the results are relationships between relationships between relationships, etc. The relationships form a graph-like structure.

A representation language based on binary relationships supports the imprecise definition of genome data. Because most of the aspects of each genome relationship are independent of each other, it is useful to decompose (fully normalize) the relationships into binary relations, i.e., those with an arity of two, which describe one aspect of a concept. Binary relations can be combined with other binary relations to describe concepts which may not have been considered when the database was originally developed. For example, an oligonucleotide may have originally been intended to be a PCR primer, but now may be considered to be a STS or non-polymorphic marker within the database. These binary relationships form the edges of a graph.

A representation language which emphasizes concepts and relationships equally simplifies changing the type of data in the database. Most concepts in genetics are defined in terms of their relationships to other concepts. A representation language which does not create an arbitrary distinction between concepts and relationships allows a concept to grow from a simple, undefined concept to one defined by multiple complex relationships.

A representation language which describes complex, regular structures and supports cyclic graphs captures the complex, interconnected structure of genome data. Genome data has a complex structure which can often be decomposed into regular substructures. In computer science, data structures such as trees are built from simple nodes and leaves. In molecular biology, the three-dimensional structure of a chromosome contains coils of coils. Although these substructures may be described as graphs, more efficient mechanisms may take advantage of the regularities.

Graphs are pervasive within genomics. Cyclic graphs occur in gene regulation [Thieffry and Thomas 1994], metabolic pathways [Ochs 1994; Hofestaedt 1994; Karp and Paley 1994], chemical structure, map order with uncertainty [Graves 1993b], and homology relationships between species. Representing genome information requires more cyclic graphs than many other areas in

which databases are used, and this limits the usefulness of previous systems. Most computational systems are not designed to handle cyclic data structures (recursive definitions) well. Other types of graphs are also common within genomics and can be incorporated within graph-based systems. Trees are useful in representing phylogenetic relations and the organization of DNA coils. Robbins [1995] proposes a database be described as a collection of truncated rational trees, and ACeDB schemas are represented using linked trees which form directed acyclic graphs.

3.2 Graph Data Modeling

A data model may be defined by specifying its data types, operators, and constraints. Data types describe the basic building blocks of data. For example, that the data consists of numbers, strings, and sets. The operators specify how data can be manipulated. For example, it is valid to add an element to a set or add two numbers together to get a third, but there is not usually an operator to delete a digit from the middle of a number. Constraints define the restrictions on which instances of the data types can legally occur in the database. For example, a possible constraint is that a "set" can consist of either numbers or strings but not both. Relational databases were the first to be specified in terms of a data model [Codd 1970, 1980]. The data type for the relational model is the relation. The eight operators are select, project, join, product, union, difference, intersection and division. There are two database-independent integrity constraints: (1) No component of the primary key of a base relation is allowed to accept nulls; (2) The database must not contain any unmatched foreign key values.

Several new data models have been developed within genomics. OP/M [Chen and Markowitz 1994] captures objects and protocols of biological experiments; it has been implemented using a relational database by translating each OP/M schema into a relational one. ACeDB [Durbin and Thierry-Meig 1991] is a stand-alone system developed to provide data persistence for a large

number of genome activities; it uses an internal data structure, called linked trees, which is similar to some semantic data models. Two database systems have been developed using an object-oriented DBMS to investigate new data models. LabBase [Goodman et al. 1995] has been developed to model biological experiments, and Crystal [Lee et al. 1993] is an active DBMS that supports physical mapping.

A limitation of current database technology in developing genome databases is the inability to model the graph structure of genome data. When large quantities of graph-like data are stored into current databases, they must be coerced into a less expressive framework which causes loss of structure and biases the data. Coercing graphs into a linear or tree-like structure will cause some information to be lost and make development of systems on top of that representation more difficult. To store data in existing databases, the graph structure of data must be removed. The removal of structure biases the data and the incompleteness of information can make some tasks difficult to perform. It also becomes difficult to later isolate the structure from the behavior of the application when the changing science necessitates a change in the structure. A useful genome database must be able to represent the complex, graph-like structure of genome data. A graph database management system allows geneticists to develop databases in a framework which flexibly and accurately captures the structure of data while alleviating the loss of structural information.

During the past few years, data models have been developed which take advantage of graph-based representations to model complex structure. All graph data models have as their foundation the mathematical definition of a graph as a collection of vertices and edges. The data models are usually used to support an application, such as G+/GraphLog [Consens and Mendelzon 1990a] for visual querying [Consens and Mendelzon 1990b], GOOD [Gyssens et al. 1990a] for end-user interfaces [Gyssens et al. 1990b], or Hyperlog for hypertext systems [Levene and Poulovassilis

1990]. Hy+, a data visualization system based on G+/Graphlog, has also been used as part of a genome map assembly system [Harley and Bonner 1994]. Unlike the relational data model which has one formal basis, there are several extensions to the definition of graphs which form foundations for different graph data models. For example, graph data models may add labels to the nodes or edges or both, to connect the data to constructs in the real world. Other data models have been developed to better isolate the user from implementation details, such as logic data models, which are defined using mathematical logic and often use a subset of first-order predicate logic. Although graph data models may appear similar to network or semantic data models, they are more closely related to logic data models [Graves 1993a; Consens and Mendelzon 1990a].

Previous data models have incorporated many ideas which originated in artificial intelligence, such as semantic networks, inheritance, and active databases. Other ideas have originated in artificial intelligence and been refined within programming languages before being incorporated in databases such as objects and logic programming. Graph data models are no exception and are based on much of the same foundation within artificial intelligence. Graph data models can also draw upon the body of research in graph theory.

For genomics, graph data models combine advantages of relational and object-oriented models. Relations are pervasive in genomics as is the need for extensibility. Graphs provide more flexibility and extensibility than relational systems. A graph where all edges originate with a single vertex may be considered a relation with variable arity. Graphs also provide more support for relationships and collections than object-oriented systems. Since genome data is graph-like, a major advantage of a graph data model is the ability to store the data in a graph structure without any coercion of the data.

4. Basic Graph Data Model

Data models can be tailored to the structure of a domain which is to be captured. Genome data requires a flexible data model because the schemas will change frequently. Graphs provide a variable arity that makes extensions simple. By storing information on the edges of the graph, the schemas can change while supporting applications which use previous versions of the schema as well as the current version.

4.1 Graph Representation Language

When examining the characteristics of graph languages, we discovered a small, concise set of features which would capture genome data in a formal language understandable by biologists. In our graph data model, graphs are defined, as in graph theory, as a collection of vertices and edges where:

- Vertices are the nodes of the graph. They model the simple concepts and n-ary relations of the genetics.
- Edges connect two vertices.
- Labels on edges define the type of relationship between two vertices.

Vertices refer to entities, relationships or values and are analogous to relations or data values in the relational model and objects in an object-oriented model. Edges are binary relationships which connect vertices in a specific way described by the label. They are directed in the graph to distinguish how they should be interpreted semantically, but mathematically they are a relation. They are similar to the attributes in an object-oriented model, but more properly may be thought of as a pair of attributes: a named attribute and its inverse. A label is similar to the attribute name in an object-oriented model or a column name in a relational model.

The basic graph language describes a graph such as the one shown in Figure 1 below.

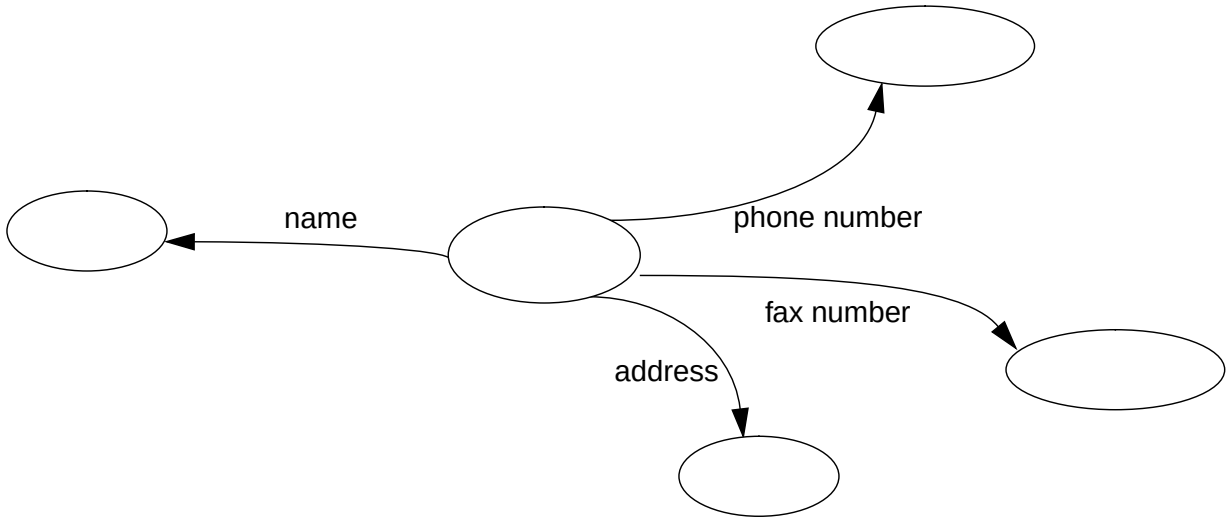

Figure 1: Graph of a person with name and contact information.

All edges are identified by adding labels to them. Node labels are not included in the basic data model, but are included in different ways in the graph schema language (section 5) and in extensions to the basic data model to capture external identifiers (section 6.1).

4.2 Data Model

Our graph representation language describes the concepts of genetics as a graph. To develop a data model, operations must be specified to describe how graphs are built, what operations can be performed on graphs, and any other constraints which describe what form a graph in the database can take.

Our graph data model can be defined in a fashion similar to the relational model. A relation is defined mathematically as a set of n-tuples. Graphs are defined in graph theory as a collection of vertices and edges $G = (V, E)$. The vertices in a graph refer to genome objects or n-ary relations and the edges refer to binary relationships (links) between them.

There are four data types in the basic graph data model: vertices, edges, labels, and a graph database. A graph database is a collection of vertices, edges and labels where:

- Vertices are nodes in the graph database which model the simple concepts and n-ary relations of the domain.
- Edges connect two vertices. There cannot be multiple edges with the same label between two vertices.
- Labels on the edges define the type of relation which holds between the two vertices. Thus, an edge is a relation between two vertices and one label.

Operators are needed for defining data in the database, updating the database, and querying the database. In a graph data model, these operators access and manipulate graphs. The basic model presented here requires only six operators:

1. Create a new, empty graph database.
2. Add a new vertex.
3. Add a new (edge) label with a specific name.
4. Add a new edge with a given label, source node and destination node.
5. Retrieve a label from the graph given its name.
6. Retrieve an edge from the graph given some (or all) of its components -- label, source vertex, edge vertex. Thus, edge retrieval is actually a family of $2^3 = 8$ operators.

There are two database-independent integrity constraints on this graph data model:

1. Labels in a graph are uniquely named.
2. Edges are composed of the labels and vertices of the graph in which the edge occurs, i.e., *G=(V,E,L)* where $E \subseteq L \times V \times V$.

These constraints correspond roughly to the integrity constraints for the relational data model: (1) No component of the primary key of a base relation is allowed to accept nulls; (2) The database must not contain any unmatched foreign key values.

This describes our basic graph data model for genomics. The remainder of the paper describes extensions to the model for conceptual modeling, more expressive data capture, and extensibility.

5. Graph Schema Language

A graph conceptual model is a formal system which represents the concepts and relationships of a domain as a graph. We use a form of the graph data model for capturing the concepts and relationships of genomics. The vertices in a graph refer to genome objects or n-ary relations and the edges refer to binary relationships (links) between them. We have extended the basic graph data model to create a graph schema language which can be used as a conceptual model. More information on how to use the graph schema language to create relational or object-oriented databases can be found in Graves et al. [1996].

An advantage of using a graph data model is that the physical and logical database structures can correspond closely to the conceptual structures with which biologists are comfortable. This simplifies the understanding and use of the database by biologists and simplifies the development and maintenance of it by computer scientists.

The graph model which we use to describe genome concepts consists of two extensions to the basic graph data model. The first extension is the definition of a concept or n-ary relationship as a vertex of a graph. Each vertex is labeled with the name of a concept. The second extension is the addition of cardinality constraints to the edges.

The type of link between two concepts in genomics is important, and the emphasis on edge labels is retained. The edge labels specify the characteristic of one of the objects, a relation between two vertices or the role that one of them has with respect to the other. For example, valid relations between a cosmid and a YAC would include “hybridizesTo”, “generatedFrom”, or “sharesSTS”, or a plasmid might be in relationship to sequence or location objects.

There are four data types in our graph schema language: concepts, edges, edge labels, and cardinalities. A graph is a collection of concepts, edge labels, cardinalities, and edges where:

- Concepts are the nodes of the graph and model the simple concepts and n-ary relations of the domain.
- Edge labels on the edges describe the relation which holds between the two vertices and are uniquely named.
- Cardinalities are either a positive integer or “many”.
- Edges connect two vertices. There can be multiple edges with different edge labels between two vertices. There is a cardinality for each vertex of an edge. Thus, an edge is a relation between two concepts, one link name, and two cardinalities.

In Figure 2 a marker is described which has one name, one type, one location, and many alleles.

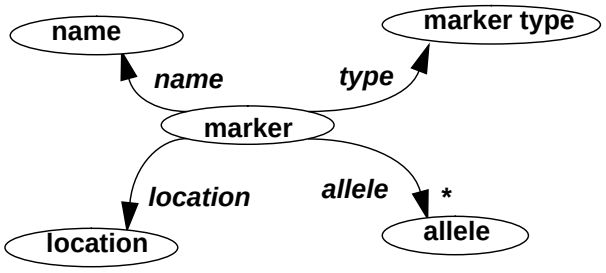

Figure 2: Graph schema of a simple marker

The “many” cardinality is denoted by an “*”, and the “one” cardinalities are omitted.

Peripheral concepts can be omitted from the graph. Sometimes it is clearer to omit some of the concepts from a description which are not needed to communicate the concept being described.

Figure 3 describes a person where some of the peripheral concepts have been omitted from the "leaves" of the graph by drawing edges which appear to not have a destination vertex.

A notation for data types may also be included in the schema. Although the conceptual schema is designed to aid biologists in describing biological concepts with which they are familiar, sometimes common data types are available which can be specified in the design of a system, such as number or string. A notation which is compatible with object-oriented modeling notations is to denote biological concepts which are being designed within a schema as ovals and to denote pre-existing data types as rectangles.

There are an additional four extensions to the graph schema language which are useful in practice, but which we do not describe formally: views, constraints, operations and queries. Views are diagrams which represent the various configurations in which the links in the schema may occur. A conceptual schema is the collection of diagrams which show all the links that a concept may have and the various configurations in which the links may occur. Constraints include cardinality and type. Cardinality constraints are included in our formal model, but other constraints provide additional information. Although some work has been done on graphical representation of constraints, we have found it best to describe the constraints as text. Operations describe how data is stored and accessed and can be described as transactions which are to be performed on the database. Queries are best represented as a variety of English (natural language) queries which are likely to be asked.

6. Extensions

Depending on the type of data to be captured, variations in the data model may be useful. Rather than develop one large, inclusive data model, it is clearer to describe the extensions separately.

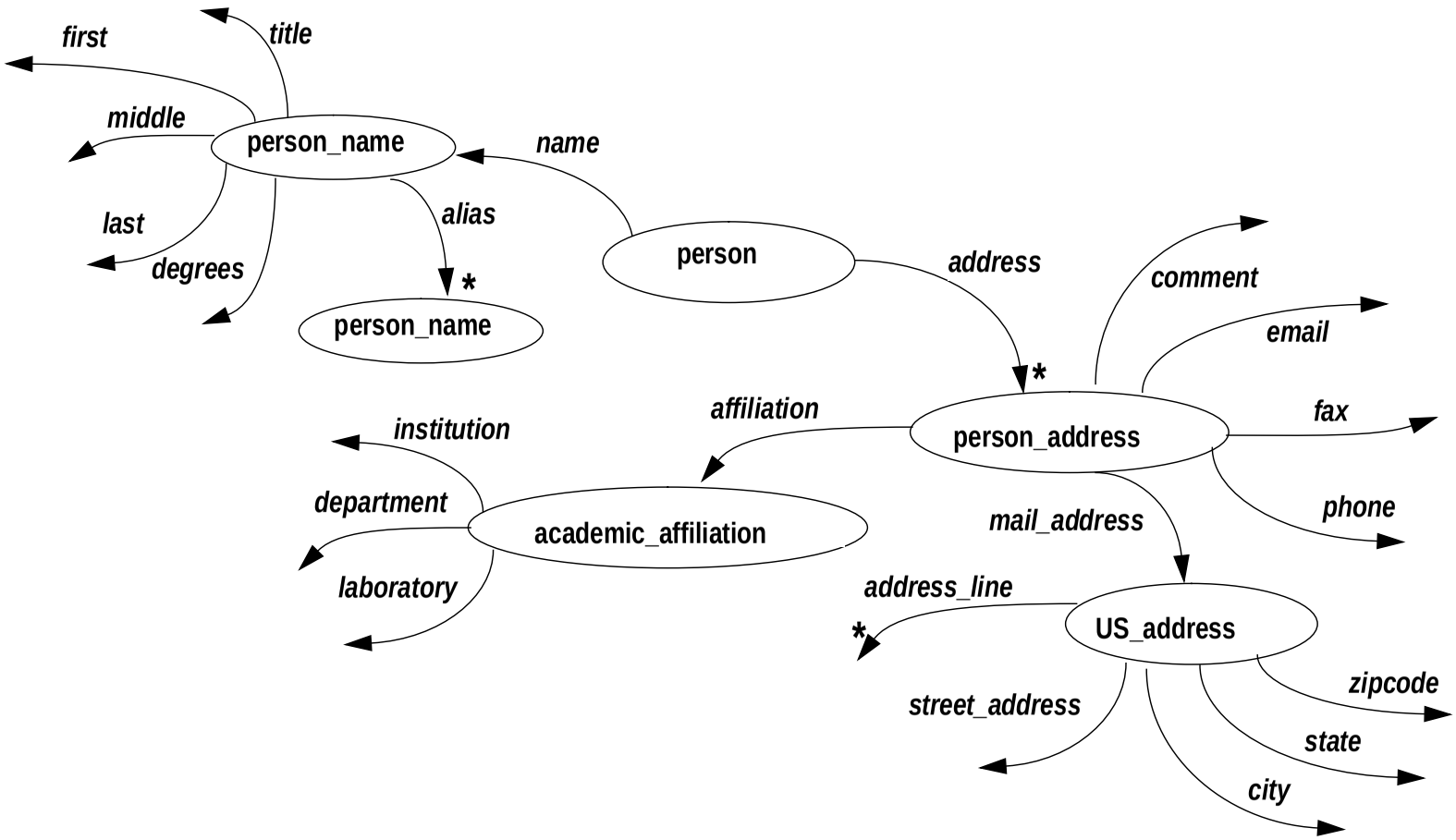

Figure 3: Graph schema of a person

Three extensions are to include mechanisms for capturing: external identifiers, ordered collections of data, and large-scale packages of data.

6.1 Symbols

Symbols are vertices in the graph which identify external objects. A symbol captures the unstructured data in the database, for example the names of people, eartag numbers on mice, email addresses, or experimental test results. This information is often kept in spreadsheets, text documents, or hand-written laboratory notebooks. The graph provides structure to the data by defining relationships which support querying of the data in the database.

Because symbols are references to entities outside the database, it is conceptually simpler to restrict symbols to be only edge destinations. Thus, symbols form the leaves of the graph which makes up the database, as shown in the example in Figure 4 below.

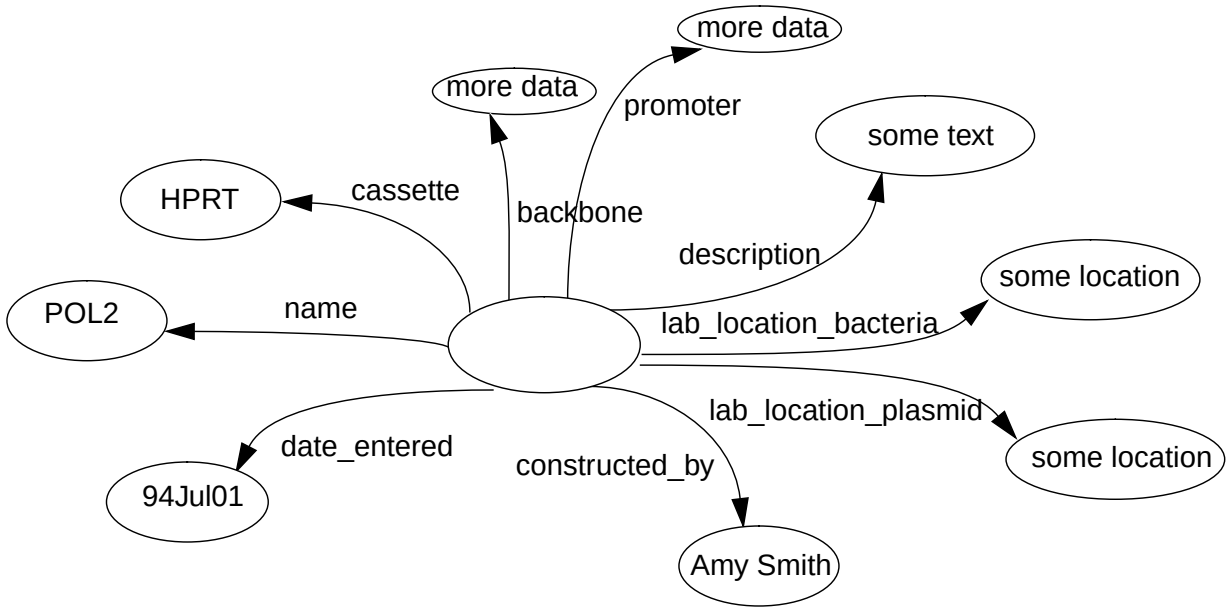

Figure 4: Laboratory Data in a Plasmid Database

Only one operation must be added to the data model to support symbols as the destination of an edge:

1. Add a new symbol to the database with a given name.

In addition, the operation to retrieve an edge from the database must be modified to allow the specification of symbols.

A symbol can be any sequence of characters including types that might be considered separately such as numbers, strings, dates, or file names. Adding symbols to the graph database also allows applications which use the database to either store data of different types directly in the database or by reference to external storage mechanisms. For example, an application in a sequencing lab does not need to store all the output from a sequencing machine in a database. Most of the information is kept in a file format which is specific to the sequencing machine and does not provide information which is useful after the sequence has been determined. A database can store a reference to the file instead. Another example of external identifiers used in genomics are the "accession numbers" which public biomedical databases use to refer to entries in other databases which cover a different scientific area.

Although grouping all external identifiers as symbols is sufficient for many genome databases, the data model can also be extended to refer to other "built-in types", such as positive integer, string, or URL. For many genome databases, these are not necessary, and they are not included in this data model. They may be treated in the same manner as Symbols and restricted to be the "leaf" nodes of a graph.

One extension which we considered and found not useful for genomics was the addition of objects to the data model. The leaves of a graph might be objects in some object-oriented programming language or database. This could form a graph-object hybrid data model. One implementation we considered was to develop graphs as objects in a commercial object-oriented database and allow some of the leaves to be objects implemented in Smalltalk. Although this

might be useful in some domains, we found that it was more useful to capture the structure of genome data in the database and to use objects within the applications. A description of a method for storing the data from objects as a graph can be found in Bergeman et al. [1995].

6.2 Multi-Valued Attributes

Data models usually provide type constructors for creating collections of data or bulk data types. Some semantic data models provide list and set type constructors. Our basic graph data model provides multi-valued attributes to create collections of data.

For genomics, it is useful to consider all relationships as possibly many-to-many. Other data models assume most relationships are 1-to-1. This causes severe problems in maintaining relational and object-oriented databases for genomics. One method for dealing with many-to-many relationships in a relational database is to create linking tables. This is acceptable when there are only a few relationships, however in genomics most (if not all) known naturally occurring relationships are many-to-many. For example, the commonly used genome mapping database GDB 5.0 captured seven primary mapping concepts. When implemented in a relational database with auxiliary tables and linking tables it contained over 250 relations. Designers of smaller laboratory databases often try to avoid the problem by assuming 1-to-1 relations initially. The schema then rapidly becomes unmanageable as linking tables are soon added in an *ad hoc* manner. In developing a database at Baylor College of Medicine for capturing mapping data based on filter hybridization experiments (a commonly used technique), we discovered that only one relationship out of dozens was genuinely 1-to-1. The relationship was that a single biological vector is used to capture data in a man-made library of clones.

Ordered collections are sometimes useful. An extension to our basic data model provides a mechanism to order the edges of a multi-valued attribute by indexing the edge label. This provides

a mechanism for treating the elements collectively or individually. Each edge label is augmented with the ability to create an indexed set of edge labels.

For example, in the laboratory, clones are often kept in 96-well plates. A 96-well plate may be thought of as a fixed array of 96 very small test tubes. Although this can be captured using the constructs of the basic data model, it is may be convenient to represent the link between the plate and each of the 96 individual wells as an indexed relation. An edge label called “well” can be created with 96 indexed relations: well_1, well_2, …, well_{96}. Clone information can be added to the database using the 96 indexed relations. There are two classes of queries which would be useful to ask given a specific plate: what clone is stored at well_n, for any *n*?, and what are the clones stored in all wells?

Each edge label retains its ability to be used in creating edges, regardless of whether it also has indexed edge labels associated with it. In addition, it may occasionally be useful to create relationships in the database using both the non-indexed edge label and its indexed edge labels. Thus, there are four classes of queries which can be asked using indexed relations:

I. All edges created using the non-indexed edge label.

II. All edges created using an indexed edge label.

III. All edges created using any indexed edge label in the collection of indexed edge labels defined by a non-indexed edge label.

IV. All edges created using the non-indexed edge label in addition to all edges created using any of its indexed edge labels.

The class III and IV queries can be created from asking multiple class I and II queries.

For example, to store names of individuals who can be contacted if a question arises about a laboratory reagent, it may be useful to allow either ordered or unordered sets of contact names.

The edge label “contact” could be used either as an indexed or non-indexed edge label. The four classes of queries would have the following interpretation:

I. All edges created using “contact”.

II. All edges created using contact_i, for a specific *i*.

III. All edges created using any of contact_1, contact_2, … .

IV. All edges created using “contact” in addition to all edges created using any of contact_1, contact_2, … .

6.3 Packaging Graphs

Large graphs can become unwieldy. Rather than store the data as one large graph, it can be clearer to subdivide the data into subgraphs. Each graph can be stored individually and consists of a set of edges. Each graph can be considered as a separate database, or they can be combined for querying.

Different (possibly conflicting) data sets can be stored within each graph. Genome data has many experimental results which may conflict. Inconsistent data sets may be stored in separate graphs, and as they are reconciled, data may be transferred to yet another graph without losing the original data sets.

One extension to the packaged graphs presented here would be to allow graphs to be considered as a special type of vertices and to be interrelated within a “graph of graphs”. This would be useful in database management systems designed for genome mapping because of the complex interrelations both within maps and between maps.

Another extension is to weaken the constraint that an edge can only composed of vertices and labels in the graph in which the edge is defined. This allows some edges to cross over the packaged graphs. This would be useful in a phylogenetic database where each “family” could be a separate graph, while one graph could capture the more general part of the taxonomy.

The packaged graphs can be exploited in segmenting the physical storage of data. Large databases typically have to be broken up into smaller pieces for physical storage. Sometimes the database developer knows which queries are more likely to be asked than others. This information can be captured by packaging graphs, and more efficient querying can be performed.

There are four operations which are added to the data model to create, list, verify existence and access the graphs. Graph access operations retrieve the edges of a graph. The operations are:

create_graph: name -> ()

constraint: graphs are uniquely named.

list_graphs: () -> list of names

graph_exists: name -> True iff it exists in database

graph_edges: name -> list of edges

6.4 Implementation Concerns

Different aspects of the data models described here have been incorporated in several systems implemented which have been implemented in Allegro Common Lisp with CLOS [Graves 1993a], Parc Place Smalltalk, Servio’s Gemstone object-oriented database, Unix shell scripts, C, and Tcl/tk. One of these systems has been in regular use since March 1994. Across the different environments, some of the features described here have remained constant while others have improved with each implementation. The primary difference between implementations has been

physical storage with physical storage capabilities ranging from the commercial object-oriented database Gemstone to flat files.

A variation across implementations has been the amount of indexing each edge receives in the database. One way to make querying more efficient is to completely index the edges of the graphs in the database. Because the graph edges consist of two nodes and one edge label, $2^3 = 8$ index tables could be used. In practice, the same performance can be obtained with four or five index tables. In return for the slight increase in space, the matching of an edge in a query to a graph in the database can be performed in constant time.

Segmenting of the database into the different packages, as described in section 6.3, has been constant across implementations. We found that a graph size of 10,000-100,000 edges formed a useful package of data. Much larger data sets can be usefully decomposed into this size. In a couple of the implementations, we relaxed the restriction presented here that an edge can only be formed from vertices and labels in the same graph. It is possible to create a split indexing hierarchy where each vertex or edge label tracks (1) the other graphs in which it occurs as part of an edge, and (2) the edges in which it occurs only for edges within the same graph. This reduces the size of the indexing tables, and simplifies the addition and deletion of edges.

The implementations rely on indexing the edges for efficient implementation of querying. One of the advantages of the complete indexing is that more complex queries can be efficiently retrieved. However, complete indexing is generally not considered for most data models. An advantage of representing graphs as a set of edges is that complete indexing can be obtained with a small, fixed number of index tables.

7. Graph Templates

Genome data consists of real world entities and human-defined concepts, which must be accurately and flexibly represented in systems supporting ongoing research. Accurately defining these entities and concepts is a vital step in producing systems which can be used by biologists. Each genome researcher has a “mind’s-eye” view of information. When listening to two biologists talk, one notices that these views may overlap but are seldom identical. When speaking, these ambiguities can be ignored or redefined through discussion, but in databases, the same ambiguity must be eliminated or specified clearly. Often the choice is made to define a view which represents a minimum definition of a concept rather than deal with representing multiple definitions. Our approach to the problem of needing multiple descriptions of concepts which can be broken apart into individual views is to define graph templates for each user view. From a graph description of a concept, individual views are abstracted. Since all views originate from the same representation, consistency and accuracy are maintained.

Graph templates are created from the database schema and are used as updatable views on a database. They define what the data in the database looks like to the end user. Some templates may be specified by the schema developer as query-only when they describe an incomplete view of the database and should not be used for entering or manipulating data, though the query-only graph templates are defined in the same way as the templates used for both creation and querying.

7.1 Introduction

Graph templates describe the types of data that can be stored in a database. A graph template is a (usually small) graph where some of the vertices or edge labels may be variables, and a graph template is defined by its name, parameters, and edges.

For example, to store information about clones used in a laboratory, a user might want to create a graph template where a clone has a name, a type (such as YAC, cosmid, or M13), and a library from which the cosmid originates. The graph template called “simple-clone” can be defined with four parameters: clone, name, clonetype, and library and three edges. A graph template for a “simple-clone” might be:

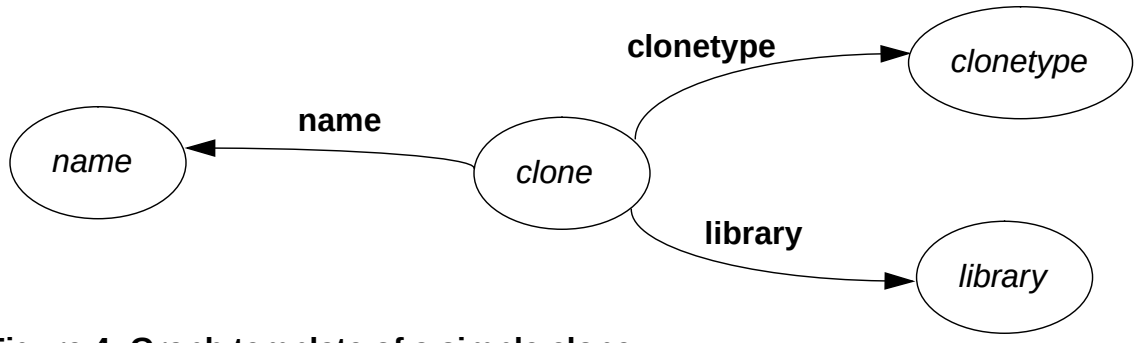

Figure 4: Graph template of a simple clone

In this figure of a graph template, the italicized nodes correspond to variables in the graph template while the bold edge labels are constants.

A template has two presentations: visual and textual. The visual presentation is used by the user to design, modify, or examine graph templates. The textual presentation is useful to exchange graph templates between software components. A textual presentation of the graph template “simple-clone” is given below which corresponds to the graphical presentation given above.

```
simple-clone clone name clonetype library
clone        ‘name’     name
clone        ‘clonetype’ clonetype
clone        ‘library’  library
```

The graph template is presented with the first line containing the name “simple-clone” followed by the four parameters and each additional line corresponds to one edge. The four parameters are:

clone, name, clonetype, and library. These correspond to the variable vertices in the graphical presentation above. An edge is a tab-delimited line with three columns where the first column corresponds to the source vertex, the middle column corresponds to the edge label, and the last column corresponds to the destination vertex. Constants are denoted by ‘single quotes’, and in this example all the edge labels are constants. The first edge initiates at the vertex “clone”, has a constant label of “name”, and has a destination vertex which is the parameter “name”. The other two edges are defined similarly.

There are six data types to support graph templates: vertices, labels, variables, edges, and graph templates. A graph template is a name, a parameter list and a collection of vertices, labels, variables, and edges where:

- Vertices are nodes in the graph which model the simple concepts and n-ary relations of the domain.
- Labels on the edges define the type of relation which holds between the two vertices. Thus, an edge is a relation between two vertices and one label.
- Variables are used as placeholders for either vertices or labels. Variable vertices are variables used as vertices in the graph. Variable labels are variables used as labels in the graph.
- Edges connect two vertices with a label. The vertices and/or label may be a variable.
- A parameter list is an ordered subset of the variables defined in the graph template.

There are eight operations in the data model to create, access, and manipulate the graph templates. The database operations create, list, verify existence, or delete graph templates in the database. Graph template access operations retrieve the parameters or edges of a graph template. A build operation instantiates a graph template and adds the resulting edges to the database. A query

operation matches a graph template against the database and returns a report of the matching sub-graphs.

The operations are:

create_template: name, list of parameters, list of edges -> ()

constraint: graph templates are uniquely named.

list_templates: () -> list of names

delete_template: name -> ()

template_exists: name -> True iff it exists in database

template_parameters: name -> list of parameters

template_edges: name -> list of edges

build: template name, list of arguments -> ()

constraint: list of arguments must be same length as template's list of parameters.

query: template name -> list of records

7.2 Building

A graph template may be considered to be a declarative procedure which can be used for querying and building graphs. An advantage of using graph templates is that the user need only describe a structure for the concept to be captured by the database and operations for entry and querying are automatically defined.

A graph template is used to build graphs in the database by binding the variables of the template and adding the edges of the resulting graph to the database.

For example, to build a graph capturing a clone using the template of the previous section, a user specifies as arguments the name, clonetype, and library of the clone. A vertex which connects the arguments must be also be created. In the graph template, it is labeled “clone”. This vertex could be specified by the user or created automatically by the database. These are the vertices which provide the internal structure of the graph(s) which makes up the database. It corresponds to the relation ids which are often added to table definitions in relational databases or to the object ids which are automatically created within object-oriented ones. The command:

build simple-clone [new_vertex] YWXD1000 YAC STLouis db_graph

adds nodes and edges to the graph db_graph which correspond to:

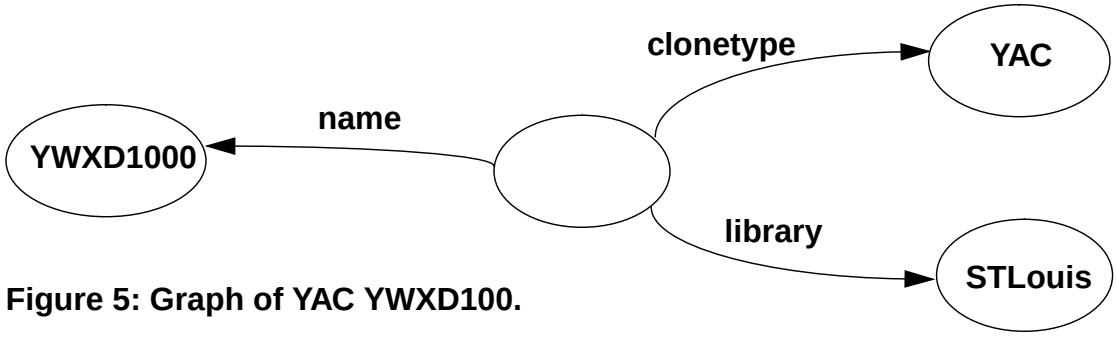

Figure 5: Graph of YAC YWXD100.

7.3 Querying

A user queries the database by defining a graph template which describes the desired query. Each graph template can define several queries depending upon which parameters to the graph template are bound. In a specific query, some of the arguments to graph templates may be specified and some left unbound. The partially-specified graph templates are given to a graph query al-

gorithm which retrieves matching graphs from the database. The matching graphs are passed to a report generator which formats the graphs into a report for the user.

The graph query algorithm takes a graph template and matches it against a graph, returning a report of all subgraphs in the database which match the graph template. The algorithm works by comparing the graph template to the database and then returning the graphs in the database which match the query. Mathematically, the graphs which match the query contain the information in the query and may contain more information.

The graph query algorithm works by decomposing the graph template into its edges and matching them against the edges stored in the database. The retrieved edges are recombined into graphs which form the query result. The query operations consists of four steps:

1. decompose the graph template into a series of edge queries,

2. for each edge, query the stored data for matching edges,

3. build the complete query result using the individual query results from each edge, and

4. return the complete query result as a report.

To match a graph template against a graph in the database, the algorithm first decomposes the graph template into edges. Each edge is queried against the database and the results of the edge queries are combined into the graph query results. An edge query is the mechanism for retrieving data from storage. All graph data is stored as edges, and the graph query algorithms retrieves data as edges. The edges retrieved from storage are combined into graphs as specified by the query graph. These resulting graphs are the output of the graph query.

The algorithm iterates over each edge in the graph template and queries storage for matching edges. An edge consists of a relation, a source, and a destination. Any one of the three can be either specified or unspecified in a query. Thus, there are 2^3 different kinds of edge queries, and a

template edge can have 0, 1, 2, or 3 variables. The values found in the database for the edge variables are stored in a temporary table. We have developed data structures for storing and completely indexing edges in a database. By creating index tables for the all edge queries, complex queries can be asked more efficiently. Thus, each edge query consists of an index table lookup.

Each edge in the graph template is used to retrieve the matching set of edges from the database, and the resulting set of edges is added to a data structure which contains the result of the previously matched edges in the graph template. The data structure stores the data in tables. The result of the query of a template edge to a graph is combined with the previous intermediate results through either a join, a restriction on previous values, or a Cartesian product depending upon whether the variable(s) in the edge query have previously occurred. For space and time efficiency, the data structure which captures intermediate results stores the data as tables which form suspended Cartesian products. The algorithm continues to update the data structure through restrictions and joins.

As each edge in the template graph is used to retrieve matching edges from the database, its edges are included in the query result by creating a new suspended Cartesian product, performing a restriction, or performing a natural join depending upon whether the template edge introduces all new variables, no new variables, or some new variables, respectively. Suspended Cartesian products may be evaluated when the template edge being considered shares variables with two tables in the suspension. When this occurs, the suspended cartesian product is removed and a natural join is performed between the two existing tables and the new table which contains data matching the template edge.

The result of the complete query is a data structure which stores all subgraphs of the graph in the database that match the edges of the graph template. The data structure consists of a set of ta-

bles that are in a suspended cartesian product. From this data structure, a report is generated. To generate the final report, the suspended Cartesian products are evaluated by iterating through the tables. This occurs in time linear with the size of the output.

7.4 More on Views

Graph templates also form a foundation for other view mechanisms on graphs. Three possibilities we have examined are abstract data types, objects, and type constructors. Views may be implemented as abstract data types which include additional operations to manipulate the data. For example, a collection of abstract data types for physical mapping could be developed which would allow biologists with some programming experience to develop small applications to manipulate maps. Another approach is to include inheritance in the types which compose a view. Objects are a mechanism for encapsulating the behavior of computational processes. They typically include mechanisms for describing data abstractly, for hiding the behavior of object operations (or methods), and for making the object's internal structure and operations available to a restricted class of objects which inherit them. In a system, they may be designed around the data which they store or around the behavior which they provide. Using objects as views on graphs is a practical approach which makes graph representations available within readily available programming environments [Bergeman et al. 1995]. A more general technique is to define type constructors as views which would allow the development of domain-specific data models. Domain-specific (or application-specific) data models were first proposed by Brodie [1984] and would include genome-specific type constructors, such as ordered contiguous sequence, partial order, or overlapping interval collections (contigs). Some genome-specific data models were described in section 3.2, and one general approach to developing domain-specific data models is to use constructs from constructive type theory [Graves 1993a].

8. Conclusion

We have investigated using graphs as the foundation for a genome database systems by developing a data model based on graphs. We have demonstrated that:

1. Graph-based representations are useful for representing genome data.
2. A representation language tailored to the requirements of genome data can be used as the basis of a graph data model.
3. A graph data model is a viable foundation for genome databases.

We have found that graph-based technologies are useful in multiple aspects of database development and plan to expand on the capabilities of graphs for developing genome databases.

9. Acknowledgments

The author thanks Ellen Bergeman, Dan Davison, Charlie Lawrence, and Bill Rounds for frequent discussions of the ideas in this paper.

This work was begun while the author was at the Artificial Intelligence Laboratory, Dept. of Electrical Engineering and Computer Science, University of Michigan and was supported by NSF grant ISI-9120851. The research was supported by an appointment to the Human Genome Distinguished Postdoctoral Fellowships sponsored by the U.S. Department of Energy, Office of Health and Environmental Research, and Administered by the Oak Ridge Institute for Science and Education. A portion of the work was also supported by the W.M. Keck Center for Computational Biology and the Baylor Human Genome Center funded by the NIH National Center for Human Genome Research. The author is currently supported by DOE grant DE-FG03-94ER61818.

Appendix

A. Graph Data Model

There are four data types in this graph data model: vertices, edges, labels, graphs. A graph is a collection of vertices, edges and labels where:

- Vertices are nodes in the graph which model the simple concepts and n-ary relations of the domain.
- Edges connect two vertices. There cannot be multiple edges between two vertices with the same label.
- Labels on the edges define the type of relation which holds between the two vertices. Thus, an edge is a relation between two vertices and one label.
- Symbols are vertices in the graph which identify external objects.

Operators are needed for defining data in a database, updating a database, and querying a database. In a graph data model, these operators access and manipulate graphs.

There are operations in the data model to create, access, and manipulate the graphs and graph templates. General database operations create, list, verify existence, or delete graphs and graph templates in the database. Graph and graph template access operations retrieve the edges of a graph or the parameters or edges of a graph template. A build operation instantiates a graph template and adds the resulting edges to a specified graph. A query operation matches a graph template against a graph and returns a report of the matching subgraphs.

The basic operations are:

create_graph: name -> ()

constraint: graphs are uniquely named.

new_vertex: graph name-> vertex

new_label: graph name, label name -> label

constraint: edge labels in a graph are uniquely named.

new_edge: graph name, source vertex, label, destination vertex -> ()

constraint: edges are composed of the labels and vertices of the graph in which the edge occurs.

graph_label: graph name, label name -> label

graph_edges: graph name, source vertex or null, label or null, dest vertex or null -> list of edges

There are two possible operations to support symbols. The operation to support symbols is:

new_symbol: symbol name -> symbol

constraint: symbols in the database are uniquely named.

new_symbol: graph name, symbol name -> symbol

constraint: symbols in a graph are uniquely named.

The operations to support indexed relations are:

new_indexed_label: label -> (indexed) label

label_index_size: label -> non-negative integer

label_get_index_label: label, number -> label

constraint: number must be less than or equal to the index size.

graph_edges_indexed_only: graph name, source vertex or null, label, dest vertex or null-> list of edges

graph_edges_all: graph name, source vertex or null, label, dest vertex or null -> list of edges

The operations to handle multiple graphs are:

list_graphs: () -> list of names

graph_exists: name -> True iff it exists in database

graph_edges: name -> list of edges -- all edges in graph

The operations to support templates are:

create_template: name, list of parameters, list of edges -> ()

constraint: graph templates are uniquely named.

list_templates: () -> list of names

delete_template: name -> ()

template_exists: name -> True iff it exists in database

template_parameters: name -> list of parameters

template_edges: name -> list of edges

build: template name, list of arguments, graph name -> ()

constraint: list of arguments must be same length as template's list of parameters.

query: template name, graph name -> list of records

References

1. Ait-Kaci H. *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Type Structures*. Ph.D. Thesis, Computer and Information Science, University of Pennsylvania, Philadelphia, PA, 1984.
2. Bergeman ER, Graves M, Lawrence CB. "Viewing Genome Data as Objects for Application Development". Proceedings, Third International Conference on Intelligent Systems for Molecular Biology (ISMB-95). AAAI Press. July, 1995.

3. Brodie, Michael, John Mylopoulos, and Joachim Schmidt, editors. *On conceptual modelling: Perspectives from artificial intelligence, databases, and programming languages.* Springer-Verlag, New York, 1984.

4. Chen IA, Markowitz VM. An overview of the Object Protocol Model (OPM) and OPM Data Management Tools, TR LBL-33706, Lawrence Berkeley Laboratory, 1994.

5. Cinkosky MJ, Fickett JW, Barber WM, Bridgers MA, and Troup CD. SIGMA: A system for integrated genome map assembly. *Los Alamos Science* 20:267-269, 1992.

6. Codd EF. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6): 377-387, June 1970.

7. Codd EF. Data models in database management. In M. Brodie and S.N. Ziles, eds, *Proc. Workshop on Data Abstraction, Databases, and Conceptual Modelling*, June 1980. Also, *ACM SIGMOD Record 11(2).*

8. Coleman D, Arnold P, Bodoff S, Dollin C, Gilchrist H, Hayes F, and Jeremaes P. *Object-Oriented Development the Fusion Method.* Prentice Hall, Englewood, NJ, 1994.

9. Connell J, Shafer LB. *Structured Rapid Prototyping*. Yourdon Press (Prentice Hall), Englewood, NJ, 1989.

10. Consens MP, Mendelzon AO. *GraphLog: A Visual Formalism for Real Life Recursion*. In H Garcia-Molina and HV Jagadish, editors, Proc of the 1990 ACM SIGMOD International Conference on Principles of Database Systems, page 404-416, May 1990a.

11. Consens M, Mendelzon A. *The G+/GraphLog visual query system*. In H Garcia-Molina and HV Jagadish, editors, Proc of the 1990 ACM SIGMOD International Conference on Management of Data, page 388, May 1990b.

12. Durbin R, Thierry-Meig J. A C. elegans database. Available via anonymous ftp from lirmm.lirmm.fr, cele.mrc-lmb.cam.ac.uk and ncbi.nlm.nih.gov 1991.

13. Goodman N, Rozen S, and Stein L. LabBase: A Database to Manage Laboratory Data in a Large-Scale Genome-Mapping Project. *IEEE Engineering in Medicine and Biology.* Special issue on Managing Data for the Human Genome Project. 11(6) 1995.

14. Graves M. *Theories and tools for designing application-specific knowledge base data models*. Ph.D. Thesis, University of Michigan, Dept. of Electrical Engineering and Computer Science. University Microfilms, Inc. April, 1993a.

15. Graves M. Integrating order and distance relationships from heterogeneous maps. *In* L. Hunter, D. Searls and J. Shavlik, eds., *Proceedings of the First International Conference on Intelligent Systems for Molecular Biology (ISMB-93),* AAAI/MIT Press, Menlo Park, CA, July 1993b.

16. Graves M, Bergeman E, Lawrence CB. A Graph-Theoretic Data Model for Genome Mapping Databases. In the Proceedings of the Hawaii International Conference on System Sciences-28, Biotechnology Computing Track. January, 1995.

17. Graves M, Bergeman E, Lawrence CB. A graph conceptual model for developing human genome center databases. *Computers in Biology and Medicine.* Special issue on Information Retrieval and Genomics. 26(3), pp 183-197. 1996.

18. Gyssens M, Paradaens J, and D Van Gucht. *A graph-oriented object database model.* In Proceedings of ACM Conference on Principles of Database Systems, 1990a.

19. Gyssens M, Paradaens J, and D Van Gucht. *A graph-oriented object model for database end-user interfaces.* In Proceedings of 1990 ACM SIGMOD Conference on Management of Data, 1990b.

20. Harley E, Bonner AJ. A Flexible Approach to Genome Map Assembly. *Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology (ISMB-94)*, AAAI/MIT Press, Menlo Park, CA. pp 161-169, August 1994.

21. Hofestaedt R. Modeling and Visualization of Metabolic Bottlenecks. In H. Lim, ed., *Proceedings of The Third International Conference on Bioinformatics and Genome Research.* World Scientific Publishing. June, 1994.

22. Karp P, Paley S. Automated drawing of metabolic pathways. In H. Lim, ed., *Proceedings of The Third International Conference on Bioinformatics and Genome Research.* World Scientific Publishing. June, 1994.

23. Lee AJ, Rundensteiner E, Thomas S, Lafortune S. An information model for genome map representation and assembly. *ACM 2nd Int. Conf. on Information and Knowledge Management.* Nov, 1993.

24. Levene M, Poulovassilis A. *The hypernode model and its associated query language.* In Proc Fifth Jerusalem Conference on Information Technology, pages 520--530, 1990.

25. Mirkin BG, Rodin SN. *Graphs and Genes*, volume 11 of *Biomathematics*. Springer-Verlag, 1984.

26. Nebel B, Smolka G. Representation and reasoning with attributive descriptions. In K.H. Blasius, U. Hedstuck and C.R. Rollinger, editors *Sorts and Types in Artificial Intelligence*, volume 418 of LNAI, pages 112-139. Springer-Verlag, 1990

27. Ochs R. A relational database model for metabolic information. In H. Lim, ed., *Proceedings of The Third International Conference on Bioinformatics and Genome Research.* World Scientific Publishing. June, 1994.

28. Prieto-Diaz R, Arango G, eds. *Domain Analysis and Software Systems Modeling.* IEEE Press, 1991.

29. Robbins B. An Information Infrastructure for the Human Genome Project. *IEEE Engineering in Medicine and Biology.* Special issue on Managing Data for the Human Genome Project. 11(6) 1995.

30. Sowa J. *Conceptual Graphs.* Addison-Wesley, Reading, MA, 1984.

31. Thieffry D, Thomas R. Logical Analysis of Genetic Regulatory Networks. In H. Lim, ed., *Proceedings of The Third International Conference on Bioinformatics and Genome Research.* World Scientific Publishing. June, 1994.