December 1993

Facilitating Transformations in a Human Genome Project Database

Susan B. Davidson
University of Pennsylvania, susan@cis.upenn.edu

Anthony S. Kosky
University of Pennsylvania

B. Eckman
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-93-94.

Facilitating Transformations in a Human Genome Project Database

Abstract

Human Genome Project databases present a confluence of interesting database challenges: rapid schema and data evolution, complex data entry and constraint management, and the need to integrate multiple data sources and software systems which range over a wide variety of models and formats. While these challenges are not necessarily unique to biological databases, their combination, intensity and complexity are unusual and make automated solutions imperative. We illustrate these problems in the context of the Human Genome Database for Chromosome 22 (Chr22DB), and describe a new approach to a solution for these problems, by means of a deductive language for expressing database transformations and constraints.

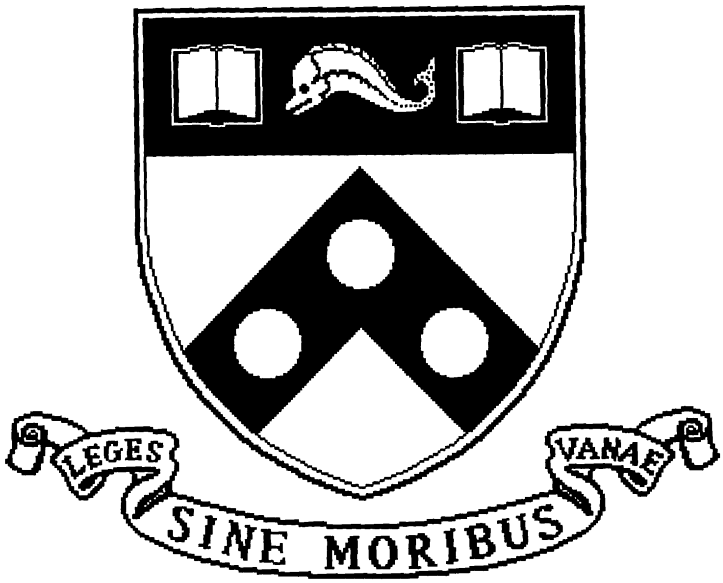
Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-93-94.

Facilitating Transformations In A Human Genome Project Database

MS-CIS-93-94
LOGIC & COMPUTATION 74

S.B. Davidson
A.S. Kosky
B. Eckman



University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

December 1993

Facilitating Transformations in a Human Genome Project Database *

S. B. Davidson, A. S. Kosky
Dept. of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389
Email: susan@cis.upenn.edu, kosky@saul.cis.upenn.edu

B. Eckman
Department of Genetics
Dept. of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6145
Email: eckman@cbil.humgen.upenn.edu

Contact author: Susan B. Davidson, Phone (215) 898-3490, Fax (215) 898-0587

December 8, 1993

Abstract

Human Genome Project databases present a confluence of interesting database challenges: rapid schema and data evolution, complex data entry and constraint management, and the need to integrate multiple data sources and software systems which range over a wide variety of models and formats. While these challenges are not necessarily unique to biological databases, their combination, intensity and complexity are unusual and make automated solutions imperative. We illustrate these problems in the context of the Human Genome Database for Chromosome 22 (Chr22DB), and describe a new approach to a solution for these problems, by means of a deductive language for expressing database transformations and constraints.

1 Introduction

Human Genome Project databases present a confluence of interesting database challenges: rapid schema and data evolution, complex data entry and constraint management, and the need to integrate multiple data sources and software systems which range over a wide variety of models and formats. These challenges are common to laboratory notebook databases in general, within the Human Genome Project as well as within the broader realm of biological databases. While these challenges are not necessarily unique to biological databases, their combination, intensity and complexity are unusual and make automated solutions imperative. Furthermore, techniques to aid in their solution either do not exist or are inadequate in this domain. This paper illustrates these problems in the context of the database developed at the Philadelphia Genome Center for

*This research was supported in part by the following grants: NSF IRI9004137, ARO DAAH0493G0, and NIH P50-HG-00425.

Chromosome 22, and describes a first step to solving what we perceive to be the core of these problems: a language in which to express data transformations.

The goal of the Human Genome Project (HGP) is to sequence the 24 distinct chromosomes comprising the human genome. Each chromosome is composed of a long, double-stranded molecule of DNA (deoxyribonucleic acid). This double-stranded DNA is made up of complementary pairs of four different nucleotides or *bases* (A, G, C, T), arranged like beads on a string. *Sequencing* DNA means discovering the exact sequence of A's, C's, T's, and G's on the string. Although there are techniques for directly sequencing short DNA strings (approximately 400 bases), current methods are not practical for sequencing the entire genome (3 billion bases) or even a single chromosome (60-200 million bases) at one time. Consequently, while researchers continue to attempt to develop faster sequencing technologies, the HGP has set mapping the chromosomes as a less ambitious intermediate goal. *Mapping* involves the ordering of identifiable DNA fragments as markers along the chromosome, and anchoring markers at known positions to serve as landmarks. Genome center databases typically serve as laboratory notebooks for mapping efforts. The database discussed in this paper, Chr22DB, is the laboratory notebook for the Philadelphia Genome Center for Chromosome 22, located at the University of Pennsylvania and Children's Hospital of Philadelphia.

One of the major problems faced in HPG databases is rapid schema evolution and the resulting need to modify existing applications. As in many research efforts, genome centers and other organizations doing genome-related research are vying for increasingly limited resources. The urgent need to produce data cost-effectively, and its potential application in medical diagnostics and therapeutics such as gene therapy, combine to produce a fast-paced, high-pressure environment. Consequently, new and better experimental techniques are constantly being developed and the experimental data being modeled is constantly changing, forcing evolution of the laboratory notebook database schema. Rather than having the luxury of a two to three month design and application development phase, the schema evolution and adaptation of existing applications to the evolving schema must occur extremely rapidly, since investigators consult the database to plan and guide ongoing experimentation. Furthermore, since the direction of future experimentation is based on existing data, data integrity is crucial. As later sections will illustrate, however, the data is very complex, hierarchically organized, and contains an unusually large number of links among tables (inclusion dependencies). This gives rise to a number of complex, non-standard constraints that need to be specified and enforced in order for the data to be correct.

Another major problem is that access to multiple, heterogeneous remote databases and software packages is frequently needed to augment the contents of the laboratory notebook databases and to answer queries posed by researchers. These databases include archival databases, such as the nucleic acid sequence database, Genbank, the protein sequence data base, PIR [1], the biomedical bibliographic data base, Medline, and the human genome map data base, GDB [2]; a growing number of laboratory notebook databases; as well as software systems such as BLAST [3], FASTA [4], and Staden which perform complex data analysis involving such computational problems as pattern-matching search and string comparison. However, at the present time there is no industry standard data model or DBMS for computational biology; these databases therefore include flat relational databases (Sybase), object-oriented databases (Object Store, GemStone) and complex-relational databases (e.g., the National Center for Biotechnology Information's use of the ASN.1 data transmission format as a data storage format [5]), PC- and Macintosh-based databases (4th Dimension, *inter alia*). In addition, individual investigators use simple spreadsheet databases such

as Microsoft Excel to meet immediate needs. As an example of how laboratory notebooks incorporate data from other sources, in populating Chr22DB (a Sybase database) we imported as much shared data as possible from other electronic sources, including GDB and other archival databases, as well as preexisting spreadsheet databases administered by individual investigators. As part of our collaborations with other genome centers, we have also imported data from ObjectStore to Sybase and vice versa. Importing this data presented significant data transformation challenges.

It should be noted that this heterogeneity in schemas and models within the HGP is likely to persist. Contrary to standard database dogma, no single conceptual schema is suitable for all applications: As data complexity increases, different schemas may capture only partial, and perhaps significantly different views of the data as a whole; as analysis tasks increase in complexity beyond simple queries, it is often necessary to organize the data to optimize a specific application to achieve acceptable system performance; as the expressive power of the representation language grows and the complexity of the data objects increases, the possibilities for describing the same data in a variety of equivalent and equally valid ways also grows. Similarly, no single data model is universally superior to all other data models for all applications. Thus, we find numerous independent structurings of the same or similar information. The GenBank family is a case in point: there is the "standard" flat-file version with numerous trivial syntactic variants, a relational version developed at the Los Alamos National Laboratory [6], the ASN.1 version developed at NCBI, a relational version developed from the ASN.1 version by the Philadelphia Center for Chromosome 22 [7], and at least one knowledge base version, also developed within our group [8], which transforms the data from a sequence entry view to a biological concept view. Each of these has its own advantages and disadvantages that include issues of representation and query language expressiveness, and portability, among others.

Two recent papers underscore the problems that we have been alluding to, and indicate that they are pervasive to HPG databases:

- A recent report of a Department of Energy Informatics "summit" [9] listed a number of simple queries that were impossible to answer with the current data sources, not because of the unavailability of the data, but because the sources are distributed among various databases, programs and structured files, and there is no effective technique for combining these sources.
- Goodman [10] in an appraisal of his attempt to create a genome information system listed two major issues that he faced: (a) the lack of an adequate query language for the DBMS he was using; and (b) the fact that the underlying schema was constantly evolving.

An obvious conclusion is that most of these problems would be at least partly solved with a sufficiently expressive query language. However, we believe that there is an important prior problem: that of *transforming data* into some form that is understandable by users, a query language, or an applications program. A fixed query language will not solve the problem of integrating data sources unless we can also be sure that when a new data source becomes available, we can transform it into a structure acceptable to existing queries or applications. The problem of schema or data evolution calls for flexible tools for rapidly re-mapping databases – again so that they can be used as input to existing applications. Thus the first issue that we must face is a principled approach to data transformations: Transformations between schemas in a single data model (as with schema

evolution), between different data models (as with data entry screens, and as in the Genbank family of databases), or across multiple data models (as in the integration of data from multiple sources).

The purpose of this paper is to describe our approach to specifying data transformations, and illustrate it using a sample problem of data transformations that has arisen in Chr22DB. The approach we describe is declarative, a variant of existing proposals [11, 12]. While declarative query languages (datalog and its extensions) have not yet gained universal acceptance as query languages, we believe they *are* the right approach to data transformations.[1] The reason for this is that while a data transformation can be thought of as a query, it is one in which the computational forms used are very simple and whose output is rather large and structurally complex – a whole database rather than a single relation. Furthermore, it is highly desirable to have the query in a form that is easy to analyze and to reason about in light of rapid evolution: it is easier to rewrite and manipulate a declarative program than it is to modify embedded SQL or C code.

The remainder of this paper is organized as follows: Section 2 illustrates a part of Chr22DB and a data transformation problem that has been encountered. Section 3 describes our transformation language and shows how it is used to capture the sample problem. We conclude by arguing how current techniques fail to address the problems we have encountered, and discussing future research.

2 A Sample Data Transformation in Chr22DB

The data in the archival and laboratory notebook databases for the HPG is sufficiently complex that even researchers in molecular biology have trouble understanding terms and structures in schemas they are not already familiar with. For those who know little to nothing about molecular biology, they are almost completely unintelligible. To understand the sample data transformation problem encountered in Chr22DB, we must therefore start off by explaining a bit about what is being modeled and what some of the terms used mean.

2.1 A Databaser's View of the Biological Background

As mentioned in the introduction, the HGP's intermediate goal is *mapping*: ordering markers (fragments of DNA) along the chromosome and locating them at known positions. A variety of techniques are used to anchor markers to specific locations on the chromosome. For the sake of simplicity, we will consider only one, *physical mapping* using cloned probes and Sequence Tag Sites (STS's).

To begin, the chromosome of interest is cut randomly into overlapping pieces of experimentally manipulable size (50,000-1 million bases). These pieces are then reassembled into a linear ordering representing their order in the original DNA string. To discover the relative ordering of fragments, it is crucial to be able to ascertain when the sequence of two pieces of DNA overlaps, that is, when the pieces come from neighboring sites in the original string. One technique for detecting sequence overlap between two pieces of DNA is to demonstrate that their sequence contains the sequence of a third, much shorter fragment, called a *probe*. The linear ordering on the pieces yields a linear

[1] It is interesting to note that Goodman also arrived at the same conclusion in [10].

ordering on the probes whose sequence is contained in them, and vice versa. The ordered set of probes become the desired map landmarks. The landmarks and ordered fragments may then used to sequence areas of special interest such as regions thought to be related to inheritable disease. For example, we can divide a fragment of interest into pieces of sequenceable length, sequence them, and reassemble the fragment; or choose a landmark near the region of interest and sequence from it into the desired region.

Physical mapping and its relationship to DNA sequence is illustrated in Figure 1. At the top of this figure, a chromosome is depicted with the banding patterns visible under a microscope, which themselves function as landmarks at the coarsest level of granularity. Vertical lines denote markers (probes). Horizontal lines denote larger, overlapping DNA fragments whose sequence contains marker sequence. Below, the sequence of a tiny substring of DNA is shown.

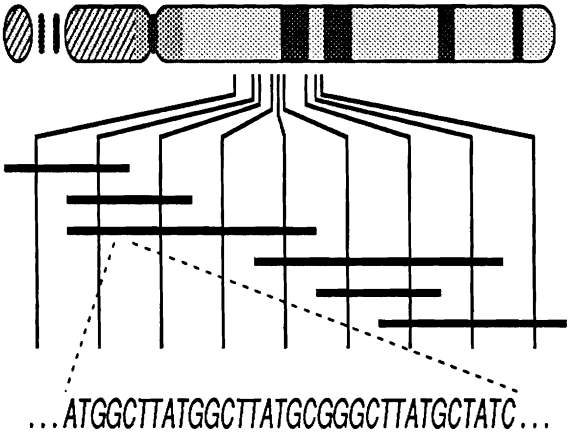


Figure 1: Physical Mapping of a Chromosome

Two types of probes used in physical mapping are represented in Chr22DB: 1) Cloned probes and 2) Sequence Tag Sites (STS's). Cloned probes are actual physical reagents stored in freezers, and STS's are information stored in a database. In what follows, we briefly describe some of the information maintained about probes by Chr22DB.

Cloned Probes. In cloning, a fragment or *interval* of human DNA is inserted into carrier or *vector* DNA in bacterial or yeast cells. When the host cells are cultured, many exact replicas of the human DNA are produced, to be used in future experiments. Important data items about a clone are: the size in kilobases of the inserted DNA; the type of vector used; the insertion sites in the vector, specified by the enzymes used to cut the vector before inserting the human DNA; the clone's name, including the laboratory which named it; the clone's chromosomal location (given in bands, e.g., "22q11.2", which are based on the banding patterns visible under a microscope); and a cross-reference to the Genome Data Base (GDB), the archival database of the Human Genome Mapping Project, located at Johns Hopkins University.

Sequence Tag Sites (STS's). An STS is an interval of DNA defined by a *primer pair*: a pair of sequenced nucleic acid intervals used as primers to start a chemical reaction called *amplification by the polymerase chain reaction* (PCR amplification). The entire reaction comprises several stages, each proceeding at a different temperature. An amplification reaction will not occur unless the primer sequences are found, properly spaced, within the test sequence; therefore, a successful reaction demonstrates sequence containment. Important data items about an STS are: its name, including the laboratory which named it; the name, sequence, and melting temperature of each of the primers; the expected size range of the amplified product; the temperature and time required for each stage of the process (PCR conditions); a cross-reference to GDB; the name of the cloned probe from which the primers were derived; and the chromosomal location of the site.

2.2 A Sample Database Transformation

The data in Chr22DB comes from a variety of sources: archival databases such as GDB, preexisting spreadsheet databases, object-oriented laboratory notebook databases from other centers, as well as directly from experiments being carried out at the center. Importing these sources of information involve data transformations, and has to date been done largely by hand. In order to simplify the exposition but illustrate some complexity of structural transformation, we will focus on the direct data entry performed. While this does not seem particularly glamorous, it should be noted that data entry is a special case of data transformations, and for our purposes perhaps the best illustration. It is also one of the transformations which initially motivated our work: When Chr22DB was started, the schema evolved extremely rapidly, experimental data was produced at an alarming rate, and there was a backlog of data to be entered. Rewriting the data entry applications was enormously time consuming since application generator tools could not handle the complexity of the data involved, and the modification had to be done by hand.

In data entry applications, the data are captured on a screen form that provides a specialized view of the underlying database. The view and the database may differ widely in structure, and the application must map between these two schemas. An example of a form used to enter STS lab notebook data is shown in Figure 2. It consists of a complex relation (`STS`) with three sub-relations (`Primers`, `PCR_conditions`, `Location`):

```
STS(STS_name, lab, GDB_locus, used_here, tech_lab, PCR_product_size_low,
   PCR_product_size_high, polymorphic, probe_type, comments, Primers,
   PCR_conditions, Location)
Primers(name, sequence, melting temp, pmethod, datepicked, strand)
PCR_conditions(machine, init_temp, init_time, denat_temp,..,
   final_time, buffer)
Location(chromosome, startposition, endposition, units, verified, location,
   notebook, comments)
```

Since each screen enters a single STS, there will be one row in the `STS` relation, two rows in the `Primers` relation denoting the primer pair, and multiple rows in the `PCR_conditions` and `Location` relations.

To enter the information in the data-entry screen, it must be transformed to the underlying (relational) Chr22DB database. A conceptual (EER) schema of the relevant portion of Chr22DB is

shown in Figure 3; this is merely introduced to convey the linkages between relations rather than to give a precise semantics of the schema. In this EER schema, *Has* arcs are many-to-one in the direction of the arrows; the prefix *Mand-* on an arc label designates that the relationship is mandatory; and *ISA** denotes a generalization relationship in which the primary key of the specialization entity-set differs from the primary key of the generic entity-set.[2] The relevant relational tables and attributes for this schema are given below. Uppercase attribute names denote primary keys.

```
lab(CODE)
names(MATERIAL_ID, LAB_CODE, NAME, public_name)
material(ID, material_type)
interval(ID, interval_type)
na_interval(ID, material_id, probe_type, is_polym,...)
map_loc(NA_INTERVAL_ID, CHROMOSOME, STARTPOS, ENDPOS, units,
   verif_method, expt_location, notebook, comments)
sequence(ID, na_interval_id, seq_string...)
primer(ID, pname, picked_from_na_interval_id, melting_temp,
   pick_method, date_picked, strand)
STS(ID, pr1_primer_id, pr2_primer_id, PCR_prod_size_lo,
   PCR_prod_size_hi, used_here, tech_lab, comments)
PCR_conditions(STS_ID, AMPL_MACHINE, ANNEAL_TEMP, init_denat_temp,
   init_denat_time, denat_temp, denat_time, ...)
```

In this database transformation a complex relation with nested subrelations is flattened into a standard relational schema with value-based pointers linking related tables. The atomic attributes of the top-level screen relation are distributed over 6 relational tables in the target schema: `names`, `lab`, `material`, `interval`, `na_interval`, and STS. The `Primers` subrelation is decomposed into 5 target tables: `na_interval`, `interval`, `material`, `primer`, `sequence`. The two name fields in the entry screen (`STS_name` and `GDB_locus`) are mapped to two separate rows in the target `names` table, which are linked by the internal identifier of the object being inserted.

In order to accomplish the data transformations, appropriate insert statements must be generated. The normalized target schema relies on internal system-generated integer identifiers to accomplish the links among related tables. These identifiers must be generated (primary keys of `material`, `interval`, and `lab` tables) or retrieved (foreign keys in all other tables, e.g., primer ids in the `STS` table) when the data are transformed.

To maintain data integrity, the transformed data must conform to the integrity constraints of the target database. Preeminent are key and inclusion dependency constraints, but more complex constraints may also hold. For example, each material must have at least one GDB name (i.e., `names.lab_code` = "GDB") and at least one non-GDB name (i.e, `names.lab_code` $\neq$ "GDB").

3 A Language for Database Transformations and Constraints

Although there are many choices of languages in which to express data transformations, we believe that a deductive approach is best. There are several reasons for this, most of which were mentioned

[2]The schemas in this paper were all drawn using ERDRAW [13].

in the introduction: the language should be declarative, so that transformations are easy to modify and reason about; the language should allow expression of structural complexity, since this is more important than expression of computational complexity; a deductive language requires only a small number of inference rules to build a complete proof system on which implementations can be based; and finally, the language should unify transformations and integrity constraints since there is a significant level of interaction between the two. Not only do constraints play a part in determining transformations between databases, but a transformation may imply certain constraints on the source and target database.

Our language is therefore based on Horn-clause logic and has a simple formal semantics, allowing for formal reasoning about database transformations, constraints and the interactions between the two. Not only can transformations be expressed in this language, but unambiguous and nonrecursive transformation programs can be implemented using code generators for a variety of database programming languages and having constraints checked at run time. The proposed code generators will work in two stages: First rules are converted to a *normal form*, each rule specifying how a complete entry for the target database is generated from the source database. The normalised rules are then converted into code for the appropriate DBMS. This approach means that logical inferences are performed only once at the rule level, rather than many times at the data level. Further it is straightforward to re-use the core of the program, allowing easy addaption of the code generator for a variety of database systems.

In presenting the language, we start by explaining the underlying data model, then giving the syntax of the language with several examples of constraints and transformation clauses that have been generated for the data entry screen application described in the previous section. Finally, we describe what normal forms are, and how they will be used to implement transformation programs.

3.1 Data Model

Since one of our objectives was to design a language which allowed us to describe and implement transformations between as wide a range of data models as possible, the language is based around a nested relational data model. The model is similar to that of [14], and allows nested relations, set-valued attributes and object identity. This is a natural extension of the relational data model, but also allows us to represent the complex data structures found in the various semantic and object-oriented data models that are currently gaining popularity. The model also allows the attributes of records to be either *required* or *optional* (*not-null* or *null*), a feature that is common in many data models, and which allows us indirectly to represent variants.

We use Skolem functions to generate object identities as in [12]. Skolem functions can be applied to a group of values in order to create an entirely new value, which can then be used as an object identifier, or as a way of referencing a row in some particular relation from other relations. Our semantics makes no assumptions about what range of values these Skolem functions should have other than that they be distinct, though they can easily be implemented using integers or some other base type. The type system for the language ensures that the values generated by two distinct Skolem functions can not be confused.

Many established data models incorporate various kinds of constraints as primitives: relational data models may support keys or other functional dependencies, certain existence decencies and

inclusion constraints, while object oriented databases may support some concept of inheritance and object identity. While such constraints are often important and useful, there remain many important dependencies which occur in biological and other databases but do not fall into any of these categories. Indeed choosing such specific subclasses of dependencies and treating them separately seems a rather *ad hoc* approach. We do not make any such class of constraints primitive in our data model. Instead our language provides a means to express a very general family of constraints, including but not limited to those mentioned above.

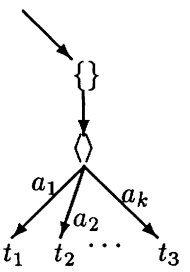
3.2 The Language

Our deductive language allows independent access to the the individual components of a relation or tuple, and variables can be bound to simple values, tuples in a relation or entire relations (sets of tuples). Individual clauses describe one conceptual part of a transformation, rather than describing the construction of an entire tuple in a relation, thus making transformation programs easy to understand and maintain. In this respect it differs from established logic-based database query languages, such as Datalog and ILOG, in which relation names are used as predicates, and variables are bound to base values only; a practice which becomes awkward when relations with many attributes are being considered. To some extent this language could be considered to be an extension of ILOG ([12]) to nested relations, or as a restriction of IQL ([11]). However such comparisons are difficult since the nature of the implementations and intended purposes of these languages are very different.

Types

Types in our language are taken to be regular trees with nodes marked either by $\langle\rangle$ (tuple types), $\{\}$ (set types), or some base type such as *int*, *string* and so on. $\{\}$-nodes must have exactly one outgoing edge, while $\langle\rangle$-nodes must have one or more outgoing edges, each marked with an attribute label taken from a set $\mathcal{A}$, and also marked as either *required* or *optional*. Nodes marked by a base type have no outgoing edges.

A type tree with a $\{\}$-node at the root represents a set type, while one with a $\langle\rangle$-node at the root represents a tuple type. A relation is considered to be a set of tuples: a *relation type* has a $\{\}$-node at the root with outgoing edge going to a $\langle\rangle$-node. So a relation with attributes $A_1, \ldots, a_k$, of types $t_1, \ldots, t_k$ respectively, would have the form:



In a *flat relation type* the outgoing edges from the $\langle\rangle$-node go to base type nodes.

As well as individual relations, we consider databases as a whole to have types. A *database type* will have a $\langle\rangle$-node at the root with outgoing edges labeled by all the names of the relations or classes in the database and going to the types of those relations. For example a database type for the database whose schema is shown in Figure 3, with relations such as `STS`, `primer`, `sequence` and `na-interval`, will be a tuple type with attributes `STS`, `primer` and so on, each of which would go to an appropriate relation type.

Our language is strongly typed, in that, given types for the source and target databases of a transformation, a unique type can be inferred for each term in a transformation program.

Terms and Atomic Formulae

The main syntactic elements of our language are *terms*, ranged over by $P, Q, \ldots$, and *atoms*, ranged over by $\phi, \psi, \ldots$. *Terms* represent values in a database, while *atoms* are the basic building blocks of formulae. An atom represents one simple statement about some values. They are defined by the following abstract syntax:

$$
\begin{array}{rcll}
P & ::= & \mathsf{Src} & \text{— } \textit{source database} \\
 & | & \mathsf{Tgt} & \text{— } \textit{target database} \\
 & | & \mathsf{c} & \text{— } \textit{constant} \\
 & | & X & \text{— } \textit{variable} \\
 & | & a & \text{— } \textit{attribute} \\
 & | & P.a & \text{— } \textit{projection} \\
 & | & f(P_1, \ldots, P_{r_f}) & \text{— } \textit{Skolem function} \\
 & | & P\langle \Phi_1, \ldots, \Phi_k \rangle & \text{— } \textit{compound term}
\end{array}
$$

and

$$
\begin{array}{rcll}
\Phi & ::= & P \dot{=} Q & \text{— } \textit{equality} \\
 & | & P \dot{\neq} Q & \text{— } \textit{inequality} \\
 & | & P \dot{\in} Q & \text{— } \textit{set-inclusion} \\
 & | & P \dot{\leq} Q \mid P \dot{\geq} Q & \text{— } \textit{arithmetic predicates} \\
 & | & \mathsf{Undef}(a) & \text{— } \textit{undefined optional attribute} \\
 & | & \mathsf{False} & \text{— } \textit{contradiction}
\end{array}
$$

Here the Src and Tgt represent the source and target databases in a transformation, which are regarded as tuples of relations. Constants are always of base type, while variables can be bound to sets and tuples as well as to values of base type.

If P is a term representing a tuple, then $P.a$ is the value of the a attribute of the tuple (if it is defined).

A compound term of the form $P\langle \phi_1, \ldots, \phi_n \rangle$ has the same value as the term P. However it carries with it the atoms $\phi_1, \ldots, \phi_n$ which are to be interpreted relative to P: so any attribute term, a, occurring in one of $\phi_1, \ldots, \phi_n$ (but not in any smaller compound term) is evaluated as $P.a$. An attribute term, a, must occur within some compound term. The use of compound terms and attribute terms can be regarded as a notational shorthand, but are necessary in order to group together atoms describing a single tuple, and to make the syntax comprehensible.

Atoms are built using the binary predicates = (equality), $\neq$ (inequality) and $\in$ (set inclusion), and also the Undef predicates which check for the definedness of an optional attribute of a tuple. Also there is a nullary predicate False which represents an error situation, and which is used in checking the validity of a transformation. In an implementation occurrences of False would have some message attached to them to indicate the source of an error.

For example an atom $X \in \texttt{STS}$ would mean that X is tuple in the relation STS. We could use a compound term to put further restrictions on X:

$$X\langle \texttt{id} = I, \texttt{pr1\_primer\_id} = P1, \texttt{pr2\_primer\_id} = P2\rangle \in \mathsf{Tgt}.\texttt{STS}$$

means that X is a tuple in the target relation STS with id attribute I, pr1_primer_id attribute $P1$ and pr2_primer_id attribute $P2$.

Clauses

A *clause* has the form

$$\psi \leftarrow \phi_1, \ldots, \phi_n$$

The atom ψ is called the *head* of the clause, while $\phi_1, \ldots, \phi_2$ form the *body* of the clause.

Not all syntactically correct clauses are meaningful. A clause is said to be *well-formed* for source database type T_{src} and target database type T_{tgt} if it is *well-typed* with respect to T_{src} and T_{tgt}, meaning that all the types of terms occurring in the clause make sense when we take the term Src to have the type T_{src} and Tgt to have the type T_{tgt}, and it is *range-restricted.* The concept of range-restriction is taken from Datalog ([15]), and means that each variable in the clause is restricted to range over some finite set of values. The formal definitions of these restrictions, together with a more detailed presentation of the semantics of the language, can be found in [16]. All the clauses considered in this paper will be well-formed for the relevant types.

Suppose a clause $\psi \leftarrow \phi_1, \ldots, \phi_n$ is well-formed for types T_{src} and T_{tgt}. The meaning of the clause is that if, for some instantiation of the variables in the body, $\phi_1, \ldots, \phi_n$ are true, then there is an instantiation of the remaining variables in the head of the clause such that ψ is also true. Clearly the truth of a clause is dependent on the values of the source and target databases for which it is being evaluated. A pair of database values μ and ν, of types T_{src} and T_{tgt} respectively, are said to *satisfy* a clause if it is true when we take the term Src to denote the value μ and Tgt to denote ν.

For example the clause

$$X = Y \;\leftarrow\; X\langle \texttt{id} = I\rangle \in \mathsf{Tgt}.\texttt{STS},\; Y\langle \texttt{id} = I\rangle \in \mathsf{Tgt}.\texttt{STS}$$

says that, for any two tuples X and Y in the relation STS, if X and Y have the same value, I, on their id attributes then they are equal. In other words the attribute id is a key for STS. This clause is an example of a *constraint*: a clause which concerns only one database rather than the connection between a pair of databases.

The terms in a clause can be classified as *source terms* which denote values in the source database, and *target terms* which denote values in the target database. A *target constraint* is then a clause containing only target terms, while a *source constraint* contains only source terms. Constraints

on the target database are tested after a transformation is carried out, and, if the constraints are violated, the transformation is rolled back and an appropriate error is reported to the user. Constraints on the source database are not interesting for the purpose of transformations, since the source database must already have been populated.

We will now look at some more examples of constraints for the database shown in Figure 3. Firstly an inclusion dependency, that for every `primer id` in the STS table there is a corresponding entry in the `primer` table:

$$X\langle \texttt{id} = P\rangle \in \textsf{Tgt.primer} \leftarrow Y\langle \texttt{pr1\_primer\_id} = P\rangle \in \textsf{Tgt.STS}$$

Next that each material has exactly one GDB name:

$$\begin{aligned} X = Y \quad \leftarrow \quad & X\langle \texttt{material\_id} = M, \texttt{lab\_code} = \text{“GDB”}\rangle \in \textsf{Tgt.names}, \\ & Y\langle \texttt{material\_id} = M, \texttt{lab\_code} = \text{“GDB”}\rangle \in \textsf{Tgt.names} \end{aligned}$$

And, finally, that a public name cannot be a GDB name:

$$\textsf{False} \leftarrow \langle \texttt{publicname} = \text{“Yes”}, \texttt{lab\_code} = \text{“GDB”}\rangle \in \textsf{Tgt.names}$$

Note that the last two of these constraints could not be expressed using the traditional functional and existence dependencies for the relational model.

In determining a transformation between two databases, we are interested in a special class of clauses called *transformation clauses*. A *transformation clause* is one which contains only target terms in its head, and which does not contain any Undef atoms for target terms. Note that a clause can be both a transformation clause and a target constraint. Indeed target constraints will often play an important part in determining a transformation.

For example, the following is a transformation clause generating part of the STS relation of the schema shown in Figure 3 from the data entry screen shown in Figure 2:

$$\begin{aligned} & \langle \texttt{id} = \texttt{f\_STS}(PI1, PI2), \\ & \ \texttt{pr1\_primer\_id} = PI1, \\ & \ \texttt{pr2\_primer\_id} = PI2, \\ & \ \texttt{id} = \texttt{f\_STS}(PI1, PI2), \\ & \ \texttt{PCR\_prod\_size\_lo} = SL, \\ & \ \texttt{PCR\_prod\_size\_hi} = SH\rangle \in \textsf{Tgt.STS} \\ & \quad \leftarrow \langle\langle \texttt{pname} = PN1\rangle \in \texttt{primer\_group}, \\ & \qquad \langle \texttt{pname} = PN2\rangle \in \texttt{primer\_group}, \\ & \qquad \texttt{PCR\_prod\_size\_lo} = SL, \\ & \qquad \texttt{PCR\_prod\_size\_hi} = SH\rangle \in \textsf{Src.STS\_screen}, \\ & \quad \langle \texttt{pname} = PN1, \texttt{id} = PI1\rangle \in \textsf{Tgt.primer} \\ & \quad \langle \texttt{pname} = PN2, \texttt{id} = PI2\rangle \in \textsf{Tgt.primer} \\ & \quad PI1 \leq PI2 \end{aligned}$$

Notice that the Skolem function `f_STS` is used to generate ids for the STS relation. Also notice that the body of this clause makes use of the target database relation `primer` in order to look up the

primer_id's. The tuples for this relation are in turn generated by another clause:

$$
\begin{array}{l}
\langle \texttt{id} = \texttt{f\_primer}(PN), \\
\ \texttt{pname} = PN, \\
\ \texttt{melting\_temp} = MT, \\
\ \texttt{pick\_method} = PM, \\
\ \texttt{date\_picked} = DP, \\
\ \texttt{strand} = ST\rangle \in \mathsf{Tgt.primer} \\
\quad \leftarrow \langle\langle \texttt{pname} = PN, \\
\qquad\quad \texttt{melting\_temp} = MT, \\
\qquad\quad \texttt{pmethod} = PM, \\
\qquad\quad \texttt{date\_picked} = DP, \\
\qquad\quad \texttt{strand} = ST\rangle \in \texttt{primers}\rangle \in \mathsf{Src.STS\_screen}
\end{array}
$$

We will see in Section 3.3 that it is necessary to unfold clauses like this, in order to get a clause that refers only to source relations in its body and only to target relations in its head. Clauses of this form can be processed in one-pass without referring to the target database.

Transformation Programs

A *transformation program*, from database type T_{src} to database type T_{tgt}, consists of a set Δ of transformation clauses that are well formed for T_{src} and T_{tgt}.

If Δ is a transformation program from T_{src} to T_{tgt}, and μ is a database value of type T_{src} and ν is a database value of type T_{tgt}, then ν is said to be a Δ-*transformation* of μ iff, for each clause $C \in \Delta$, μ and ν satisfy C.

A transformation program Δ from T_{src} to T_{tgt} is said to be *complete* iff, for any database value μ of type T_{src}, if there exists a Δ-transformation of μ then there is a *unique smallest* such transformation. The *smallest* Δ-transformation is important because it represents the data generated by the transformation program Δ from the source database: in general a transformation program will imply that certain data should be in the target database but does not exclude other additional data from being in the database as well. If a transformation program is complete then there is no ambiguity about what this smallest transformation is. It is these *unique smallest* transformations that we wish to compute.

We are particularly interested in transformations that can be done in "one pass": that is transformations that can be carried out by reading the source database and inserting values into the target database, as opposed to *recursive* transformations in which data which is inserted in to the target database is then used to create more data for the target database. This means that we restrict our attention to *non-recursive* transformation programs. However the problem of testing whether a transformation program is recursive in our nested relational model is a little more delicate than the problem for the flat relational model and Datalog. Details can be found in [16].

3.3 Normal Forms

We now limit our attention to the special case of database transformations where the target database is flat relational, as is the case with our data enty application. In this case, we first convert a transformation program into a *normal form*, which can in turn easily be converted into a program in some (non-recursive) query language.

Suppose our target database contains a relation R. A transformation clause is said to be in *normal form* if it has the form

$$X\langle a_1 = P_1, \ldots, a_k = P_k, b_1 = Q_1, \ldots, b_l = Q_l \rangle \in R \leftarrow \phi_1, \ldots, \phi_n$$

where

1. $a_1, \ldots, a_k$ are the required attributes of the relation R, and $b_1, \ldots, b_l$ are a subset of the optional attributes of the relation R,
2. the atoms $\phi_1, \ldots, \phi_n$ contain only source terms and constants, and
3. the terms $P_1, \ldots, P_k, Q_1, \ldots, Q_l$ are built using only variables, constant symbols and function symbols (so no attribute labels).

A transformation program is said to be in normal form if all its clauses are in normal form.

We have an algorithm [16] which given a non-recursive transformation program for a flat relational target database type, if the program is complete will return an equivalent program in normal form, and if the program is not complete will fail, reporting an error. This algorithm forms the central part of our code generators for transformation programs. If the source database type is also flat relational then clauses in normal form can be directly translated into a join-and-project expression in relational calculus or a "select-from-where" expression in SQL. If the source database is not flat-relational then normal form clauses can be converted into CPL ([17, 18]) or some other suitable query language.

The normal-form clauses are built by combining and unfolding clauses of a transformation, in order to form clauses which provide a complete description of a tuple in the target database in terms of the elements of the source database. Because our transformation programs are not recursive it follows that this process will terminate. If it is possible to build only a partial description of a tuple for some relation, then it follows that the transformation program is not complete.

For example a normal-form clause for the `STS` table in the transformation from the STS data-entry

screen (Figure 2) to Ch22DB (Figure 3) formed from the clauses in section 3.2 would be:

$$
\begin{aligned}
&\langle \texttt{id} = \texttt{f\_STS}(PI1, PI2),\\
&\ \texttt{pr1\_primer\_id} = PI1,\\
&\ \texttt{pr2\_primer\_id} = PI2,\\
&\ \texttt{PCR\_prod\_size\_lo} = SL,\\
&\ \texttt{PCR\_prod\_size\_hi} = SH,\\
&\ \texttt{used\_here} = UH,\\
&\ \texttt{tech\_lab} = TL\rangle \in \texttt{Tgt.STS}\\
&\qquad \leftarrow \langle\langle \texttt{pname} = PN1\rangle \in \texttt{primers},\\
&\qquad\quad \langle \texttt{pname} = PN2\rangle \in \texttt{primers},\\
&\qquad\quad \texttt{used\_here} = UH, \texttt{tech\_lab} = TL,\\
&\qquad\quad \texttt{PCR\_prod\_size\_lo} = SL,\\
&\qquad\quad \texttt{PCR\_prod\_size\_hi} = SH\rangle \in \texttt{Src.STS\_screen},\\
&\qquad\ PI1 = \texttt{f\_primer}(PN1),\\
&\qquad\ PI2 = \texttt{f\_primer}(PN2),\\
&\qquad\ PI1 < PI2
\end{aligned}
$$

Notice that this clause gives a complete description of a tuple in the STS relation, and does not call on any of the target relations in the body of the clause. In particular the calls to the primer relation which were in the body of the clause in section 3.2 have been replaced by applications of the Skolem function f_primer.

3.4 Transformation Tools

To automate the transformation process, we are planning to implement transformations using code-generators which convert transformation programs into programs in a variety of database programming languages. Our initial efforts will be to write code-generators for SYBASE and CPL ([17]). However the core of the tools will be the convert-to-normal-form algorithm, and further interfaces to generate code from normal-forms into a variety of other languages can be constructed easily at a later stage. SYBASE applications are the most immediate requirement for Chr22DB, while CPL seems like an ideal target language due to its ability to handle complex datatypes, and to connect to a wide variety of heterogeneous database systems.

Initially we see transformation programs coming from a combination of user input and meta-data stored in a database. For example, SYBASE uses meta-relations to store type information and key constraints for the relations in a database, while schema-design tools, such as ER-draw and SDT ([13]), record a variety of constraints on a schema that may not be reflected by the underlying DBMS. By accessing this information directly, we avoid the requirement for the user to enter it all by hand. Ultimately we would like to build graphical schema-manipulation tools which automatically generate the relevant constraints and transformation clauses for a schema evolution.

4 Conclusions

Human Genome Project databases present many difficult database management problems. The complexity of the data structures involved together with the frequency of schema evolutions and the large number of incompatible heterogeneous databases with which data must be exchanged mean that existing solutions – including those for schema evolution and integrated access to heterogeneous databases – are inadequate.

For example, although much has been written on the subject of schema evolution (see [19]), existing works concentrate on the problem of manipulating database schemas, and in some cases on the co-existence of instances for old and new schemas, rather than on the corresponding transformations of the underlying data. However, maintaining distinct instances for each incarnation of a rapidly evolving database schema is not a practical solution, and it is essential to have all the previously entered data available for the current schema. Clearly re-entering all the data from scratch is not a viable option, and so it is necessary to transform data from the old to the new schema.

Some of these issues have also been addressed in [20], although in the context of database integration and for a more limited data model. Our proposed language allows us to specify database transformations in a clear and formal manner, allowing for formal analysis and reasoning, and then implement the transformation for a variety of database systems by means of a code generator. In addition our language allows for the specification of constraints that arise from the complex data structures necessary for representing Human Genome data, which are not representable using established constraint languages.

Related work also includes that of schema merging in heterogeneous databases (see [21, 22, 23, 24, 25]). Central to all of these approaches is the need to have some user manipulation of the underlying schemas to indicate how the underlying databases are related to the merged schema. To our knowledge, there has been no principled, systematic approach proposed to do this other than our proposed constraint language. Note that our approach can be used to define a merged schema: each of the underlying schemas is a source schema, and the merged schema is the target schema.

There are many areas of future research, some of which we have already indicated, such as the completion of normal form algorithms for target databases which are not flat relational; the implementation of code generators from normal form transformation programs to languages of interest (initially SYBASE, then a language for collection types called CPL); and the eventual development of a window driven interface for specifying transformation programs.

Another issue is that of *composing transformations*: while some transformations will be applied only once, many transformation programs will be applied repeatedly. The most frequent of these is probably data-entry transformation programs; others involve transformation programs which import data from other archival databases, such as GDB, which are run routinely in order to reflect the continuous updates of the archival databases. We do not want to rewrite these transformation programs every time there is a minor schema evolution on the Chromosome 22 database, hence the need to *compose* the transformations.

To see why this is a composition of transformations, recall that a schema evolution is reflected by an underlying database transformation, taking data for the first schema and transforming it into

data for the second, updated schema. Since our data-entry transformation programs generate data for the first schema, we like to be able to apply the schema-update transformation in order to get data for the current schema. Alternatively, we would like a way to automatically incorporate the schema changes into our existing transformation programs in order to come up with transformation programs which target the new schema.

Unfortunately simple composition will not always be possible. A transformation will often imply constraints on its source database which are not incorporated in the source database schema. While these constraints must be satisfied in order for the transformation to go through, we can not be sure that new data generated for the source database will continue to satisfy these constraints. The problem is then one of determining what constraints are implied on a source database by a transformation, and then checking that these constraints will continue to be satisfied by new data generated for that source database. This interaction of transformations and constraints is one we believe to be very important, and any tools that could help automate the process will be useful.

We have currently completely specified the data entry transformation, and have partially specified a transformation from another archival genomic database, GDB, to Chr22DB.[3] Our experience is that the approach is extremely useful, since the relationships between structures in the source and target is clearly indicated in the clauses of the program. Knowing first-hand how laborious it was to transform SYBASE code for data entry as Chr22DB evolved, this is an extremely important practical gain. It is also interesting to observe that while declarative (deductive) query languages have not gained universal acceptance as query languages, this is one arena in which we believe they will play a central role.

Acknowledgements: We are indebted to Peter Buneman, Chris Overton and David Searles for their help and advise in developing and presenting these ideas. The really cool postcript diagram in Figure 1 is also due to David Searles.

References

[1] W. Barker, D. George, L. Hunt, and J. Garavelli, "The PIR Protein Sequence Database," *Nucleic Acids Res*, vol. 19, pp. 2231–2236, 1991.

[2] P. Pearson, "The genome data base (GDB)— a human genome mapping repository," *Nucleic Acids Res*, vol. 19, no. 2237-2239, 1991.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, pp. 403–410, 1990.

[4] W. R. Pearson, "Rapid and Sensitive Sequence Comparison with FASTP and FASTA," *Proc. Natl. Acad. Sci. U.S.A.*, vol. 85, pp. 2444–2448, 1990.

[5] NCBI Staff, "ENTREZ: Sequences Users's Guide," Tech. Rep. Release 1.0, National Center For Biotechnology Information, National Library of Medicine, Bethesda, MD, 1992.

[3] For reasons of complexity and space, this example was not included in the paper.

[6] M. J. Cinkosky, J. Fickett, D. Nelson, and T. G. Marr, "The Restructuring of GenBank," Oct 1987.

[7] K. Hart, D. B. Searls, and G. C. Overton, "SORTEZ: A relational translation of NCBI's ASN.1 database." Submitted for publication to CABIOS 12/16/92., 1993.

[8] G. C. Overton, J. Aaronson, J. Haas, and J. Adams, "QGB: A system for querying sequence database fields and features," *Computational Biology*, 1994. Accepted for publication.

[9] "Meeting Report — DOE Informatics Summit." Unpublished report available via ftp, April 1993.

[10] N. Goodman, S. Rozen, and L. Stein, "Requirements for a deductive query language in the mapbase genome-mapping database," in *Proc. Workshop on Programming with Logic Databases,* Vancouver, BC, October 1993.

[11] S. Abiteboul and P. Kanellakis, "Object identity as a query language primitive," in *Proceedings of ACM SIGMOD Conference on Management of Data*, (Portland, Oregon), pp. 159–173, 1989.

[12] R. Hull and M. Yoshikawa, "ILOG: Declarative creation and manipulation of object identifiers," in *Proceedings of International Conference on Very Large Data Bases*, pp. 455–468, 1990.

[13] E. Szeto and V. M. Markowitz, "Erdraw 4.0: A graphical editor for extended entity-relationship schemas. reference manual," Tech. Rep. LBL–PUB–3084, Lawrence Berkeley Laboritory, Berkeley, California, 1993.

[14] S. Abiteboul and C. Beeri, "On the power of languages for the manipulation of complex objects," in *Proceedings of International Workshop on Theory and Applications of Nested Relations and Complex Objects*, (Darmstadt), 1988. Also available as INRIA technical report 846.

[15] J. D. Ullman, *Principles of Database and Knowledgebase Systems II: The New Technologies.* Rockvill, MD 20850: Computer Science Press, 1989.

[16] A. S. Kosky, "A language for database transformations and constrains." Manuscript available from author at `kosky@saul.cis.upenn.edu`, 1993.

[17] L. Wong, "Querying nested collections: A dissertation proposal." Manuscript available from `limsoon@saul.cis.upenn.edu`, August 1993.

[18] V. Breazu-Tannen, P. Buneman, and L. Wong, "Naturally embedded query languages," in *LNCS 646: Proceedings of 4th International Conference on Database Theory, Berlin, Germany, October, 1992* (J. Biskup and R. Hull, eds.), pp. 140–154, Springer-Verlag, October 1992. Available as UPenn Technical Report MS-CIS-92-47.

[19] J. F. Roddick, "Schema evolution in database systems — an annotated bibliography," *SIGMOD Record*, vol. 21, pp. 35–40, December 1992.

[20] S. Widjojo, D. S. Wile, and R. Hull, "Worldbase: A new approach to sharing distributed information," tech. rep., USC/Information Sciences Institute, February 1990.

[21] A. Motro, "Superviews: Virtual integration of multiple databases," *IEEE Transactions on Software Engineering*, vol. SE-13, pp. 785–798, July 1987.

[22] A. Sheth, J. Larson, J. Cornellio, and S. Navethe, "A tool for integrating conceptual schemas and user views," in *Proceedings of 4th International Conference on Data Engineering*, pp. 176–183, 1988.

[23] S. Navathe, R. Elmasri, and J. Larson, "Integrating user views in database design," *IEEE Computer*, vol. 19, pp. 50–62, January 1986.

[24] C. Batini, M. Lenzerini, and S. Navathe, "A comparative analysis of methodolgies for database schema integration," *ACM Computing Surveys*, vol. 18, pp. 323–364, December 1986.

[25] A. Sheth and J. Larson, "Federated database systems for managing distributed heterogeneous and autonomous databases," *ACM Computing Surveys*, vol. 22, pp. 183–236, September 1990.

Jun 28 1993 CHROMOSOME 22 GENOME CENTER STS DATA

STS name R32 BUDARF
lab

GDB locus D22S259 DNA Segment, single copy probe R32

Used here Y Tech Lab BUDARF

PCR product size (bp) 141 141 Polymorphic N Probe type ANONYMOUS
low high
Comments

PRIMERS

Name	Sequence (5' to 3')	Melting temp	Pmethod	Date picked	Strand
R32FB	CACCATCTGCTGGTGCAG	56	LANDER	11/03/92	RV
R32FF2	GGGGATTGATAGAATTAAGCCC	55	LANDER	12/15/92	FW

PCR CONDITIONS

PCR Machine	Initial.. temp	time	Denature. temp	time	Anneal... temp	time	Extend... temp	time	Cycles	Final.... temp	time	Buffer
PCR-9600	95	120	94	15	55	15	72	82	30	72	420	1.5 MgCl2

CHROMOSOMAL LOCATION

Chr	Start position	End position	Units	Verified	Location	Notebook	Comments
22	Q11.1	Q11.2	BANDS	SO BLOT	BUDARF		

Figure 2: STS Data Entry Screen

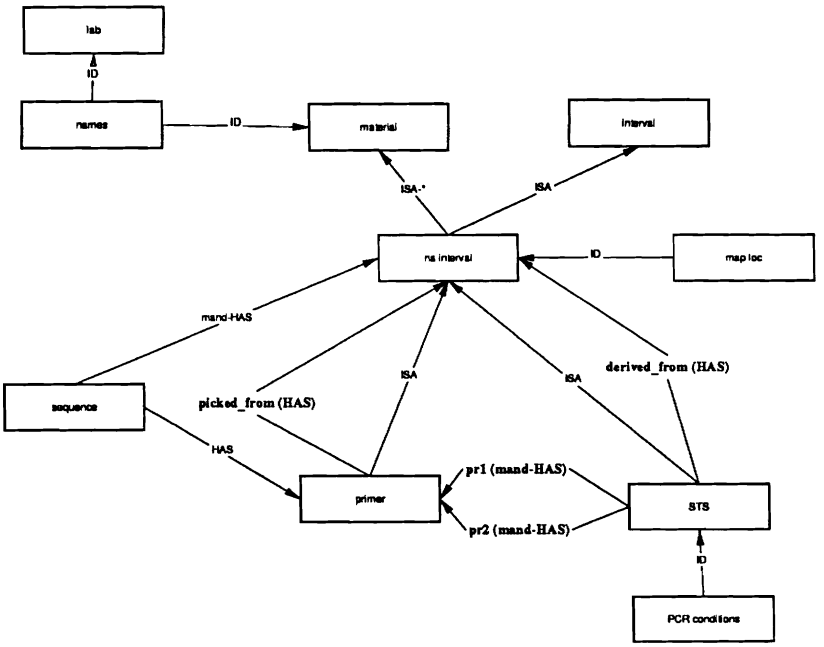


Figure 3: Schema of Target Database for STS's.