Permitting Constraint Violations in Data Storage for Integrated Data Repositories

A Thesis Proposal Presented to the
Department of Computer Science
at Brigham Young University

In Partial Fulfillment of the Requirements
for the Degree of Master of Science

Lars E. Olson
April 17, 2002

This material is based upon work supported by the National Science Foundation under Grant #0083127

I. Introduction

While trying to integrate data from multiple sources, a user often finds uncertain or incomplete data, conflicting data, or data that does not satisfy a predetermined set of constraints. Most simple databases either require the conflicts to be resolved or ignore the constraint entirely and depend on the user to find and correct conflicting data.

Some situations require a more robust behavior than this, such as when the user cannot resolve the conflicting data and wishes to keep all the possible values to indicate uncertainty. Such a database should be able to keep the desired data model and constraints, while at the same time it should be able to store exceptions to the model. In addition, the database should allow the user to add degrees of certainty to uncertain data so that the most probable values can be identified.

As an example, suppose we are collecting genealogical information. Since people can only have one birth date, it makes sense to include a constraint in the database to permit only one birth date for each person. Suppose, however, that for a particular person we find two possible dates of birth (such as “4 or 5 December 1394") or an imprecise date of birth (such as “December 1394,” with a month and year but no day). There may be no way of ever finding out the true date of birth. Since we may never know and since we typically want to have as much identifying information as possible, the database should store as much as it can, even though doing so may violate a constraint.

The method for storing this information needs to be scalable. For example, suppose that for the same person, we have not only an imprecise date of birth (such as “December 1394,” as in the last example), but also two values for a marriage date (such as “2 February 1423 or 1424”), and two values for a death date (such as “21 February 1436 or 1437”). The actual set of dates could be any one combination of the unknown values (e.g. birth date: 5 December 1394, marriage date: 2 February 1423, and death date: 21 February 1437). Since the number of possible combinations can get very large for even a single person (in this case, 31 birth dates * 2 marriage dates * 2 death dates = 124 possible combinations), an efficient method for storing the multiple combinations must be used.

In some cases we might be able to eliminate some of the combinations. For example, suppose we know that a certain person’s surname is either Purcell or Loveridge, and that her place of birth was either Cambridge or Oxford. Thus, there are four possible combinations of these values, any of which might be the correct value. Suppose further, however, that we do know that the Purcell family lived in Cambridge and the Loveridge family lived in Oxford at the time of her birth. The database should allow us to assert the correct combinations (namely, surname: Purcell— birth place: Cambridge; and surname: Loveridge— birthplace: Oxford) and eliminate the other incorrect combinations. In general, the proposed database must be able to correlate some of the unknown values.

In each of the previous examples, we can store the conflicting data as a disjunction (the “OR”) of the uncertain values. Databases that allow a disjunction of values in place of single values are called *disjunctive databases*. Some theoretical foundations and proposed models of disjunctive databases already exist (see for example [IV89], [AG85], and [KW85]). Some of these models, however, are not expressive enough to solve all of the described problems. For example, the OR-tables described in [IV89] can store multiple data such as birth dates in a single field of one record, but cannot correlate data between disjunctions, and thus cannot model the data correlating the Purcell family with Cambridge and the Loveridge family with Oxford in the example previously described. Others are not scalable, or require automated theorem-proving to answer queries. For example, the conditional tables described in [AG85] can represent the data for the Purcell-Loveridge problem, but only by introducing first-order logic statements as conditions. This may not be acceptable because most database users do not know how to use first-order logic to correlate data. In addition, answering queries would require a theorem prover and could raise some time complexity issues.

As a further complication, not all constraint violations can be modeled by disjunctive data. In our genealogy example, we may find that a child, father, and mother all have one recorded birth date each, but the child's birth date is before the father's or the mother's. Children cannot be born before their parents, so we should be able to recognize an error in the data, even if we have no alternative values that would make more sense. Other constraints that must be considered for genealogical data can be found in [Bro00] and [BAFS01].

This thesis will describe a model that better suits the problem, and examines some of the implications of using this model to allow constraint violations and measures of uncertainty. Storage schemas and methods for queries and updates will be described, along with analysis of their space and time complexities.

II. Thesis Statement

What is an appropriate database model for representing incomplete, uncertain, and inconsistent data? Given such a model, how can we efficiently map the data and certainty values to physical storage? As part of this mapping, we must also answer the following questions: How can inconsistent data be efficiently correlated across records? What is the complexity of making updates (e.g. insertion and deletion) and of answering queries about constraint violations? Can normal database queries still be answered efficiently? How do we find a valid subset of the data that is consistent with the constraints, and if more than one such subset exists, can we efficiently determine which is most likely?

III. Methods

This project will involve creating components of a database management system capable of handling the problems stated in the introduction. Several distinct problems arise from allowing constraint violations, storing disjunctive values, and correlating data across records. The following paragraphs contain examples of some of the foreseen problems.

Further Examples of Correlating Data Values

Such data correlations can exist not only in relation to a single entity (such as the Purcell-Loveridge problem) but also between separate people. We might know a certain person's surname is either Bernard or Barnett. If we know that the family is from a culture that traditionally passes surnames from fathers to children, then we also know that her father's name was also either Bernard or Barnett. In fact, we know that either both surnames are Bernard or both are Barnett, and can safely eliminate the possibilities that the daughter's name is Bernard and the father's is Barnett, and vice-versa.

As a consequence of allowing multiple values as exceptions to database constraints, other records in the database might be affected. For example, we might want a database table of all the people who were alive for a particular time period, along with references to personal journals containing information about them. Suppose, for a particular person, we have multiple possible birth dates, one of which is within the target time period and one of which is not. Depending on the true birth date, this person may or may not appear in this journal table. This example is significant because the data correlation is not between two unknown values, but rather between an unknown value and the existence of an entire record in a database table.

The database constraints themselves might depend on an unknown or inconsistent value. For example, we may have two addresses recorded for a particular contact, one in the United States and the other in Canada. The address structure is different for these two countries (the format of the postal code, for example), so the address constraints will change based on the country of origin. This example is significant because the correlation is not between actual values in the database, but rather between an actual value and a part of the metadata.

The Sub-relation Data Model

Conceptually, the database model to be used will store inconsistent values in a “sub-relation.” These sub-relations may appear in any column of the data table (as Figure 1 shows), may span multiple columns (as Figure 2 shows), or indeed may span portions of different records (as Figure 3 shows). This will allow data that does not violate any constraints to be stored normally.

Table *Person*:

Name	Birth	Marriage	Death
James I	{u}	{v}	{w}

Sub-relation {u}:

Birth
1 Dec 1394
⋮
31 Dec 1394

Sub-relation {v}:

Marriage
2 Feb 1423
2 Feb 1424

Sub-relation {w}:

Death
21 Feb 1436
21 Feb 1437

Figure 1: Example containing three sub-relations. Note that sub-relation {u} conceptually contains all dates between 1 Dec. and 31 Dec. 1394, but doesn't necessarily have to contain each date explicitly in the implementation.

Table *Person*:

First Name	Surname	Birthplace
Priscilla	{x}	{x}

Sub-relation {x}:

Surname	Birthplace
Purcell	Cambridge
Loveridge	Oxford

Figure 2: Sub-relation spanning multiple columns.

Table *Person*:

ID#	Given Name	Surname
26DP	Catherine	{y}
26DS	William	{y}

Sub-relation {y}:

26DP.Surname	26DS.Surname
Bernard	Bernard
Barnett	Barnett

Figure 3: Sub-relation spanning multiple records. Note that since both attributes in the sub-relation refer to the Surname attribute in table *Person*, the metadata must also identify the tuple associated with each value.

There will be cases where constraint violations take place, without having multiple values for a single field (such as the example of finding that a child's birth date is before the father's or the mother's). Thus the project will also require a method for checking the data against a set of constraints, both for existing data and for database modifications such as insertion and deletion of records. This type of constraint-checking is a well-developed area of database theory. The code written for [BAFS01] includes checks for many genealogy-specific constraints and will be integrated with this project.

Avoiding CoNP-Completeness

[IV89] presents a proof that queries on disjunctive databases in general have CoNP-complete time complexity. Based on a theorem presented in [LYY95], however, we present a way to handle genealogical data so that many queries, if not most, become tractable. Furthermore, the theorem also gives us a way to determine which queries remain intractable, which gives us the opportunity to handle these queries heuristically or under reasonably bounded extents.

The idea of the theorem is based on the notion of disjunctive graphs; that is, graphs with hyperarcs that represent disjunctions. Figure 4 shows an example of a disjunctive graph, along with one of its possible interpretations, where each hyperarc is replaced by one of the possible arcs it can represent. ([LYY95] uses the term *model* instead of *interpretation*, but *interpretation* will be used to avoid confusion with other connotations of *model* that are used in this paper.) Disjunctions can occur on the head side of the arc (such as the arc from *a* to {*b*,*c*}), on the tail side (such as the arc from {*c*,*d*} to *f*), or on both sides. Since only one of the disjunctive heads or tails can hold, each disjunction gives rise to a different interpretation. There are multiplicatively many interpretations, which is why disjunctive data naturally leads to intractability.

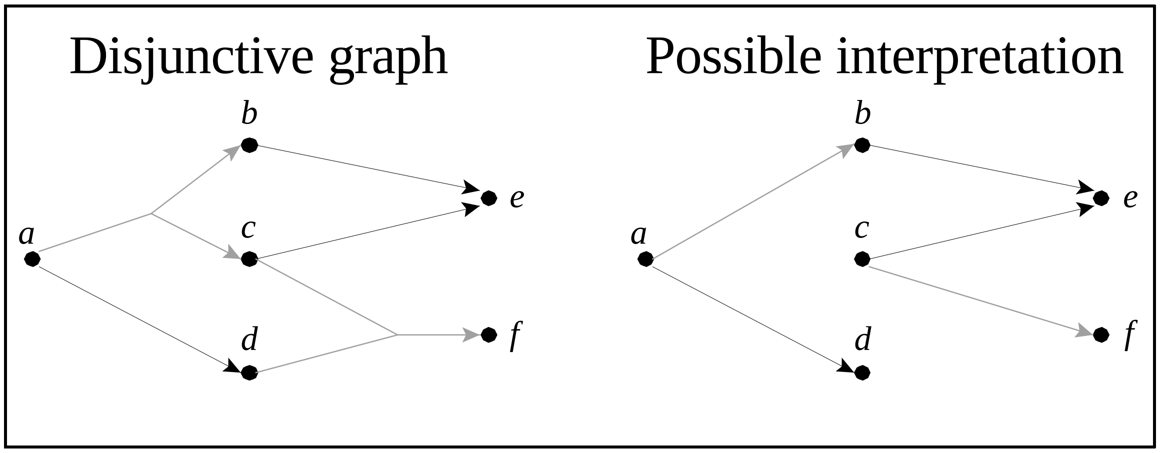

Figure 4: An example of a disjunctive graph (left) containing two disjunctive arcs, and one of the four possible interpretations of the graph (right).

[LYY95] considers the problem of computing the transitive closure of a node *x* in a disjunctive graph, which is defined as the set of all nodes *y* such that in every valid interpretation of the disjunctive graph, there exists a non-trivial path from *x* to *y*. For the graph shown in Figure 4, for example, the transitive closure of node *a* is the set {*a*, *d*, *e*}. The theorem states that if each disjunctive arc of the graph contains a disjunction only in the head of the arc (rather than in the tail), computing the transitive closure is a polynomial-time algorithm; otherwise, it is CoNP-complete.

Table *Person*:

ID#	Name	Birth Date	Birth Place ID#	Marriage Date
1	John Doe	12 Mar. 1840 or 12 Mar. 1841	1 or 2	15 Jun. 1869 or 16 Jun. 1869
⋮	⋮	⋮	⋮	⋮

Table *Place*:

ID#	City	State
1	Commerce or Nauvoo	Illinois
2	Quincy	Illinois
⋮	⋮	⋮

Figure 5: Genealogy database to be converted to a disjunctive graph. Note that attribute Birth Place ID# is a foreign key referencing table *Place*. Optional certainty measures can also be associated with some of the values (including possibly the non-disjunctive values).

To see how to use disjunctive graphs to answer database queries, consider the disjunctive database shown in Figure 5. We can transform this data into a disjunctive graph by drawing arcs from each of the object identifiers (or equivalently, tuple identifiers) to their corresponding attributes, using disjunctive arcs where necessary. We also add arcs representing foreign keys to the actual nodes in the table being referenced. We can attach labels to each arc to represent the attribute labels. Figure 6 shows the portion of the transformed graph corresponding to the values shown in Figure 5.

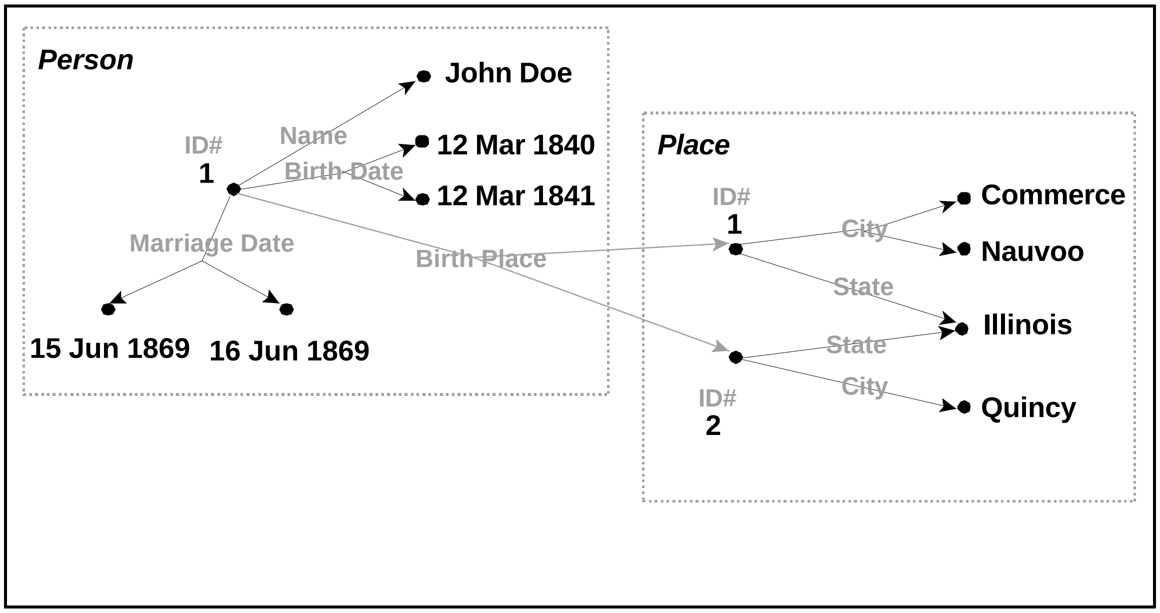

Figure 6: Database from Figure 5 transformed into a disjunctive graph.

Consider the query, “In what state was the person with ID#1 born?” (which can be written as $\pi_{\text{State}}(\sigma_{\text{ID=1}}\text{Person} \bowtie \text{Place})$ in relational algebra notation.) To answer this query, we compute the transitive closure of the node labeled ID#1 and return the node corresponding to the attribute “State,” which in this case is Illinois. We can guarantee that no arcs with disjunctive tails will appear for a query such as this if we insist that no disjunctions for object identifiers occur (e.g. the relation will never contain “ID#1 or ID#2” for a single object-identifier attribute) and that foreign keys only reference these object identifiers in other tables. Since the only arcs created in the transformation originate from these object identifiers, we will never have a disjunctive tail, and therefore this type of query can be answered in polynomial time.

Many other queries can be answered in polynomial time this way, but in some cases the result introduces a new problem. If we consider the query, “In what city and state was the person with ID#1 born?” (which can be written as $\pi_{\text{City,State}}(\sigma_{\text{ID=1}}\text{Person} \bowtie \text{Place})$), we still return Illinois as the state, but we find no city in the closure (as [LYY95] defines it). The correct response to this query depends on what the user really means in the query, which could be one of three desired answers:

- What values do we know without a doubt?
- What are all the possible values for each attribute?
- What are the most likely values?

For the first question, since we do have doubt about the correct city, the correct response is to return nothing for the city and Illinois for the state, and thus the transitive closure does indeed give the correct solution. For the second question, it should return Commerce, Nauvoo, and Quincy as possible cities and Illinois as the state. To return this answer, we perform another simple transformation on the graph by replacing all disjunctive arcs with regular arcs to all the possible attributes, as Figure 7 shows. This becomes a degenerate case in which there are no disjunctions, and thus computing the closure is still in polynomial time. To answer the third question, we must first answer the question of which values in each disjunction are the most likely values, which will be discussed in the next section.

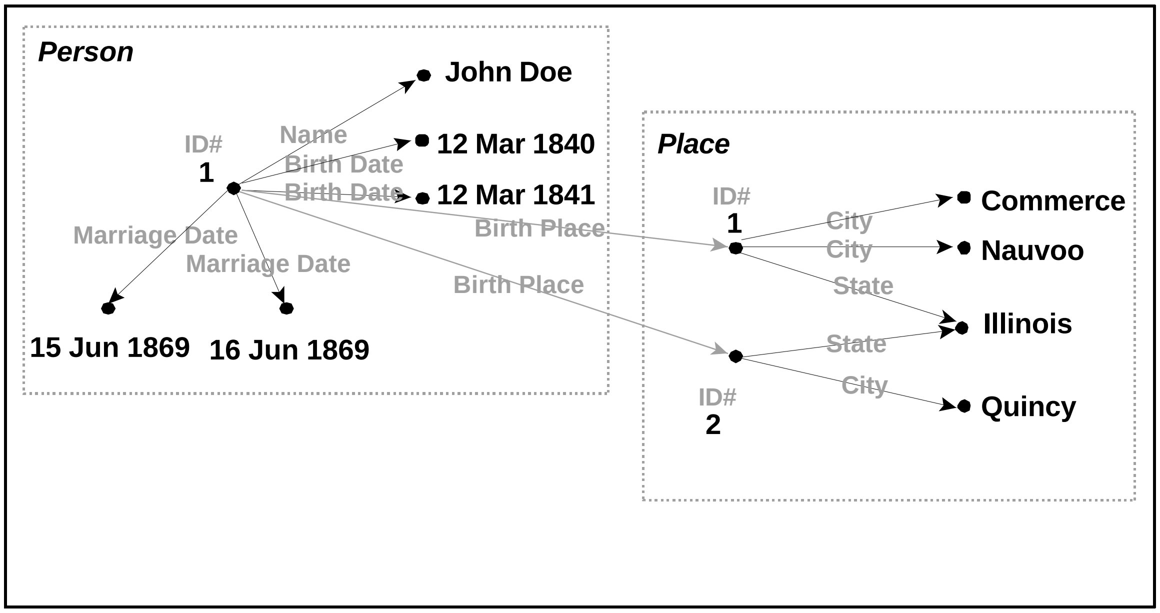

Figure 7: Graph from Figure 6 transformed to consider all possible values as true.

When a query requires selection on non-key attributes, such as, "Find the names of all people born in Nauvoo between 1841 and 1844" ($\pi_{\text{Name}}\ \sigma_{\text{BirthDate} \geq 1841 \wedge \text{BirthDate} \leq 1844 \wedge \text{City} = \text{Nauvoo}}$ (Person $\bowtie$ Place)), rather than on a key attribute, such as the queries already discussed, we can still guarantee polynomial running time. We compute the closure for every possible ID# (which is bounded by the number of nodes *n* in the graph, and thus the running time is bounded by *n* * P(*n*) where P(*n*) represents the time required to compute the closure for one node). For each ID#, if the Date falls within the specified range and the City attribute is "Nauvoo," then we add the Name attribute to the answer. Again, this answer will vary depending on whether the user wants what is definitely known, what all the possible answers are, or what the most likely answer is.

Most queries can be handled in this manner to achieve polynomial running time, but there are some exceptions. As long as tables are joined on object identifiers, the disjunctions will always be in the heads of the arcs, but some queries require joins on other attributes. Consider, for example, the query “Which people have the same birth date?” ($\pi_{P1.Name, P2.Name}$(Person P1 $\bowtie_{P1.BirthDate = P2.BirthDate}$ Person P2).) A portion of the graph needed to solve this query might look like Figure 8. Notice that to answer this query, since the join attribute is not the object identifier, we must add disjunctive arcs from the join attributes to the corresponding object identifiers. This creates arcs with disjunctions in the tail, and cannot therefore be solved using the polynomial-time algorithm in [LYY95]. Again, the correct response to this query depends on whether the user wants what is definitely known, what all the possible answers are, or what the most likely answer is. If the user wants what is definitely known, we can offer partial answers by simply removing all the disjunctive arcs entirely, or by asking the user to limit the search space. If a partial answer is not acceptable, we can at least detect when such a query is CoNP-complete, warn the user if the size of the graph is too large, and ask how we should proceed. If the user wants an enumeration of all the data for each attribute, we can make the same transformation we performed in Figure 7, and the query will be polynomial.

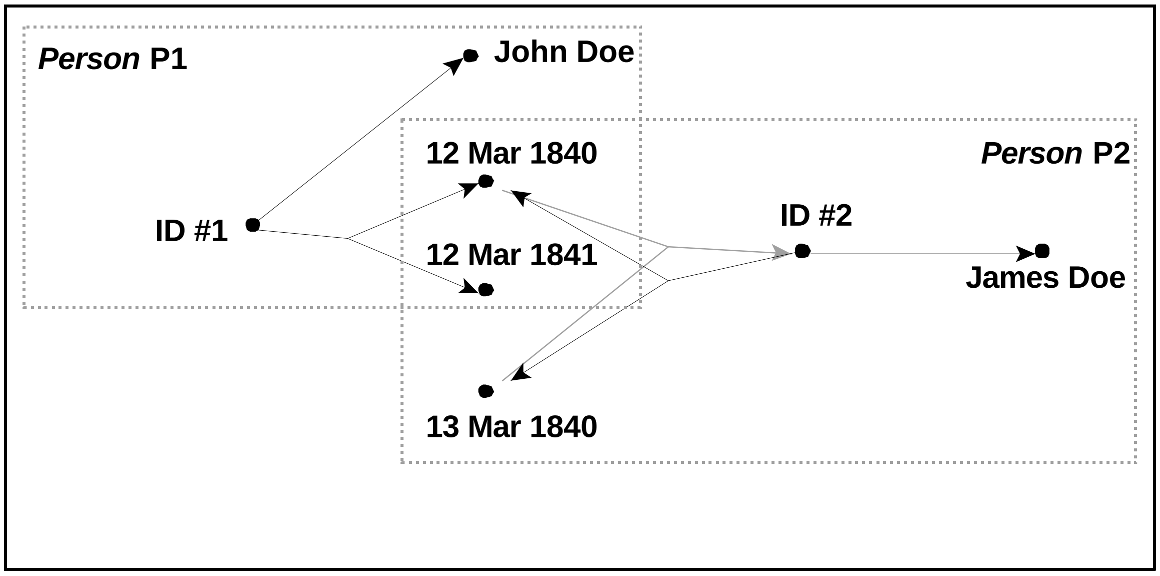

Figure 8: Query including a join on a non-key attribute.

Determining the Most Likely Interpretations

An approach to finding a valid subset of the data will also be developed. In many cases, however, there will be multiple subsets that could be valid. In the birthday example previously given, in fact, any one of the possible combinations could theoretically be valid. The database model might therefore be asked to find the "most likely" combination, provide some sort of ranking for each combination so that the user can decide which is most likely, or at least use a heuristic to find a "good enough" combination. To facilitate this, it should also include a way to specify a degree of certainty to the possible values. We can, for example, use an approach similar to the one found in [GC97].

In general, determining which values are the most likely, based on certainty measures and given constraints, can be a difficult problem to solve. In our particular application of genealogy, most disjunctions appear to be mutually independent. Determining whether John Doe's birth date was in 1840 or 1841, for example, probably will not affect the choice of whether his great-grandfather's name was Joshua or Jacob. Disjunctions that are not mutually independent can usually be limited to immediate family relations. For example, determining the correct birth dates of a grandchild and a grandparent can be decided by comparing the possible birth dates of the grandchild with those of the grandchild's parents (an immediate family relation) and of the grandchild's parents with those of the grandparents (also immediate family relations). Thus we can limit our search space to parents, siblings, and children. It is important to note that limiting the search space will only yield locally optimal answers, which could be the same as (or very close to) the globally optimal answer, but cannot be guaranteed to be so.

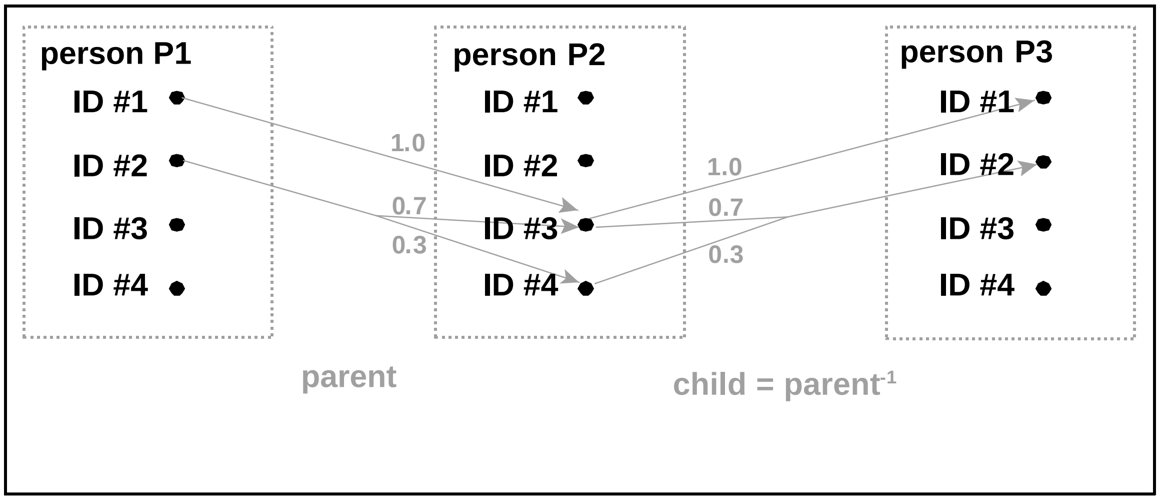

Figure 9: Graph creating all immediate family connections, including certainty measures. Other attributes, such as Birth Date, have been omitted to keep the diagram simple.

Assuming there are *n* values in the database, each arc in the disjunctive graph can have a branching factor of *k*, which is at most (and probably considerably less than) *n*. We can make connections between all the people in the database with their parents in a graph of depth 2. We can make connections with their siblings by inverting the connections on the parent graph to find all the children. The resulting graph has a depth of 3, as Figure 9 shows. Since each person appears once at each depth level, there are at most *n* arcs between nodes of depth 1 and nodes of depth 2. Similarly, there are at most *n* arcs between depth 2 and depth 3. Since each arc has a branching factor of *k*, we have at most *nk* possible choices between depth 1 and depth 2, and another *nk* between depth 2 and depth 3. The total possible choices are n^2k^2, which is bounded by n^4. Thus, a brute-force backtracking algorithm to calculate the likelihood of each possibility (and pick the most likely choices), using standard techniques for processing uncertainty such as those explained in [GC97], can be done in polynomial time.

Physical Storage and User Interface

Since the sub-relation approach is mostly hierarchical in nature (as demonstrated in Figures 1-3), and since XML easily represents hierarchical data, XML files will be used to represent the input data. In cases when disjunctive data is not entirely hierarchical, we can use pointers within each base relation and define the sub-relation externally. In other words, the values in the relation are polymorphic— they could be simple values, or they could be pointers to sub-relations that are defined separately. For example, the data from Figure 3 can be represented in XML as Figure 10 shows. Since the GEDCOM format is commonly used to store genealogical data, the GEDCOM parser developed in [AS01] will be extended to convert GEDCOM data into XML.

```
<Relation Name="Person">
   <Person>
      <ID>26DP</ID>
      <GivenName>Catherine</GivenName>
      <Surname pointer={y}/>
   </Person>
   <Person>
      <ID>26DS</ID>
      <GivenName>William</GivenName>
      <Surname pointer={y}/>
   </Person>
      ⋮
</Relation>

<SubRelation Name={y}>
   <Disj>
      <Surname reference="Person" ID="26DP">Bernard</Surname>
      <Surname reference="Person" ID="26DS">Bernard</Surname>
   </Disj>
   <Disj>
      <Surname reference="Person" ID="26DP">Barnett</Surname>
      <Surname reference="Person" ID="26DS">Barnett</Surname>
   </Disj>
</SubRelation>
```

Figure 10: Possible XML representation of the data from Figure 3.

The main program will be a prototype that parses the resulting XML file and stores the data using sub-relation data structures where necessary. The database schema and constraints will be fixed, so as to concentrate on the problem of uncertain and incomplete data. Once the data is stored, the program will provide an interface to allow the user to add or delete data (which may require sub-relations to be added or removed). The interface will also allow the user to output the modified data into a new XML file.

The interface will also allow the user to make queries on the data. Rather than using a query language, the queries will have a set fill-in-the-blank format where the user will only need to provide the specific search parameters. The types of query formats provided will include searching for a specific value or set of values, finding constraint violations within a subset of the database, and returning all certain data, all possible data, or the most likely combinations.

The prototype will also serve to test the theories developed by this thesis. It will perform timing measurements on the queries it evaluates, which should follow a polynomial curve for the polynomial queries and an exponential curve for the CoNP-complete queries.

IV. Contribution to Computer Science

This thesis will describe the theoretical foundations of a model for representing disjunctive information in a database, and will aid in the process of data integration when inconsistency is expected and a resolution of the inconsistency will be difficult or impossible. As an example for the application of the theories presented, a prototype back-end database for genealogical data will be developed.

V. Delimitations of the Thesis

No attempt will be made to do the following:

- Match schemas (see for example [BE99]) or otherwise define how to extract the data from the different sources (see for example [Ull97]).

- Perform data cleaning; that is, decide which of the different values is the correct or most likely value (see for example [Sub94] or [LM98]). While the project will use certainty measures associated with each uncertain value and attempt to provide a “most likely” interpretation, this conclusion will depend highly on the certainty measures, which are assumed to be already provided. An automated process for determining these certainty values is beyond the scope of this thesis.

- Provide a process whereby the validity of information sources is evaluated. This thesis will assume that all information being integrated either has some possibility of being true or has some significance in identifying a record.

- Decide whether the conflicting values imply different records (e.g. two birthdays for John Doe actually means there are two separate John Doe’s). Papers such as [HS95] cover this topic.

- Decide whether two different values are actually two different representations of the same value (such as forms of abbreviations or data from sources in different languages).

- Provide a generalized application capable of storing and checking any user-specified constraints. The high-level concepts will hopefully be presented in a general manner so as to apply to any arbitrary application, but the concrete examples will be specifically for a genealogy application.

VI. Thesis Outline

c. Finding Constraint Violations
d. Finding Views that Satisfy the Constraints

5. Analysis and Results (3 pages)

6. Conclusions, Limitations, and Future Work (2 pages)

VII. Thesis Schedule

A tentative schedule of the thesis is as follows:

Literature search and reading	May 2001 – March 2002
Chapter 3	March 2002 – May 2002
Chapter 4	May 2002 – August 2002
Chapters 1 & 2	August 2002 – September 2002
Chapters 5 & 6	September 2002 – November 2002
Thesis Revision and Defense	December 2002

VIII. Bibliography

[AG85] S. Abiteboul and G. Grahne, “Update Semantics for Incomplete Databases”, *Proceedings of the 11th International Conference on Very Large Databases (VLDB)*, Aug. 21-23, 1985, Stockholm, Sweden, pp. 1-12.

Contains a brief description of Codd Tables, Naïve Tables, and Conditional Tables, also describes update operations that can be performed on disjunctive databases.

[AS01] A. Ard and T. Sederberg, “GenDatabase”, July 2001, undergraduate project for Computer Science at Brigham Young University, Provo, Utah.

C++ program that parses standard GEDCOM data from text files.

[BAFS01] A.J. Bobo, A. Ard, T. Finnigan, and T. Sederberg, “Gena”, summer 2001, undergraduate project for Computer Science at Brigham Young University, Provo, Utah.

Program that provides a visual representation of genealogical data and includes several constraint checks on the validity of the data, marking the segments of the visualization where the violations occur.

[BE99] J. Biskup and D. Embley, "Extracting Information from Heterogeneous Information Sources Using Ontologically Specified Target Views", *Information Systems*, to appear.

Describes a process for database integration using ontologies to describe the database schemas. A mapping between the ontologies is created so that queries on the source database can extract data in the form of the target ontology.

[Bro00] V. Brox, "Date Estimation in Lineage-Linked Databases", http://home.no.net/gedcom/dissertation/dissertation.html, May 2000, undergraduate project for Computing Science at University of Newcastle upon Tyne, United Kingdom.

Presents a dissertation on methods for estimating unknown dates in genealogical databases based on other known or approximate dates for the person, his parents, or his children. Also includes a set of constraints used for making the approximations.

[FM92] J. A. Fernández and J. Minker, "Disjunctive Deductive Databases", *Proceedings of the Logic Programming and Automated Reasoning Conference (LPAR)*, July 15-20, 1992, St. Petersburg, Russia, pp. 332-352.

Presents a review of the work done in disjunctive databases, describes "model trees," a method of answering queries, and presents some open problems in disjunctive database theory.

[GC97] N. van Gyseghem and R. de Caluwe, "The UFO Database Model: Dealing with Imperfect Information", *Fuzzy and Uncertain Object-Oriented Databases: Concepts and Models*, ed. R. de Caluwe, World Scientific Publishing Co. Pte. Ltd., 1997, pp. 123-185.

Defines an object-oriented database model and modeling concepts for dealing with fuzzy information and uncertain information.

[HS95] M. A. Hernández and S. J. Stolfo, "The Merge/Purge Problem for Large Databases", *Proceedings of the 1995 ACM Special Interest Group on Management of Data (SIGMOD)*, May 23-25, 1995, San Jose, California, pp. 127-138.

Addresses the problem of determining when two records from different sources actually refer to the same person, and describes an algorithm for detecting such duplicate records. The concepts apply to information integration in general, but the examples that they give are for detecting duplicate records of people.

[IV89] T. Imielinski and K. Vadaparty. "Complexity of Query Processing in Databases with OR-Objects," *Proceedings of the Eighth ACM Symposium on Principles of Database Systems (PODS)*, March 29-31, 1989, Philadelphia, Pennsylvania, pp. 51-65.

Describes a model of disjunctive database called an OR-table, which allows disjunctions for single attributes in the database. Also gives a proof that queries on OR-tables have CoNP-complete time complexity.

[KW85] A. M. Keller and M. W. Wilkins, "On the Use of an Extended Relational Model to Handle Changing Incomplete Information", *IEEE Transactions on Software Engineering*, Vol. 11, No. 7, July 1985, pp. 620-633.

Contains a description of a disjunctive database model using set nulls, marked nulls, and *possible* conditions. This paper draws a distinction between knowledge-adding and change-recording updates, how these updates would be performed in this data model, and what some of its limitations are.

[LM98] J. Lin and A. Mendelzon. "Merging Databases under Constraints," *International Journal of Cooperative Information Systems (IJCIS)*, Vol. 7, No. 1, March 1998, pp. 55-76.

Describes a formal semantics for merging first-order logic statements and provides proofs of some of the properties of these semantics. Also briefly discusses representations of disjunctive data, and the problem of merging databases with conflicting schemas.

[LYY95] J. Lobo, Q. Yang, C. Yu, G. Wang, and T. Pham, "Dynamic Maintenance of the Transitive Closure in Disjunctive Graphs," *Annals of Mathematics and Artificial Intelligence*, Vol. 14, 1995, pp. 151-176.

Introduces disjunctive graphs and includes a polynomial-time algorithm for solving the transitive closure for a particular category of disjunctive graphs.

[Sub94] V. S. Subrahmanian, "Amalgamating Knowledge Bases", *ACM Transactions on Database Systems (TODS)*, Vol. 19, No. 2, June 1994, pp. 291-331.

Describes a framework for integrating knowledge bases using various languages for annotated logic, including true/false, true/false/maybe/inconsistent, and real-valued truth values. Also presents some axioms and theorems for fixpoints in the amalgamated knowledge base. In this framework, a supervisory database is given a set of user-provided mediating rules to specify what action to take when data conflicts are encountered.

[Ull97] J. D. Ullman, "Information Integration Using Logical Views", *Proceedings of the Sixth International Conference on Database Theory (ICDT)*, Jan. 8-10, 1997, Delphi, Greece, pp. 19-40.

Provides an overview of approaches to answering queries from heterogeneous data sources, including creating queries based on the views of the data sources. Gives examples using the Information Manifold project and the TSIMMIS project.

IX. Artifacts

Other than the written thesis, this project will produce a program that demonstrates storage of and queries on inconsistent data in genealogy. This program will be written in Java, and will use the Xerces package written by Apache Software for parsing XML files. Since most genealogical data is stored in GEDCOM format, a program to convert GEDCOM to XML will also be produced. This will be used to create input XML files of my own genealogy, as well as some input XML files of my own creation to test the capabilities of the database framework. To enumerate the constraints on a genealogical database, an XML Schema for the input files will need to be found or created.

X. Signatures

This proposal, by Lars Olson, is accepted in its present form by the Department of Computer Science of Brigham Young University as satisfying the proposal requirement for the degree of Master of Science.

David W. Embley, Committee Chairman

Dennis Ng, Committee Member

Aurel Cornell, Committee Member

David W. Embley, Graduate Coordinator