DATABASE AND QUERY ANALYSIS TOOLS FOR MYSQL: EXPLOITING HYPERTREE AND HYPERGRAPH DECOMPOSITIONS

A thesis presented to
the faculty of
the Russ College of Engineering and Technology of Ohio University

In partial fulfillment
of the requirements for the degree
Master of Science

Selvameenal Chokkalingam
November 2006
This thesis entitled

DATABASE AND QUERY ANALYSIS TOOLS FOR MYSQL:
EXPLOITING HYPERTREE AND HYPERGRAPH DECOMPOSITIONS

by

SELVAMEENAL CHOKKALINGAM

has been approved for
the School of Electrical Engineering and Computer Science
and the Russ College of Engineering and Technology by

David W. Juedes
Associate Professor of Electrical Engineering and Computer Science

Dennis Irwin
Dean, Russ College of Engineering and Technology
A database is an organized collection of data. Database systems are widely used and have a broad range of applications. It is thus essential to find efficient database query evaluation techniques. In the recent years, new theories and algorithms for database query optimization have been developed that exploit advanced graph theoretic concepts. In particular, the graph theoretic concepts of hypergraphs, hypergraph decompositions, and hypertree decompositions have played an important role in the recent research.

This thesis studies algorithms that employ hypergraph decompositions in order to detect the cyclic or acyclic degree of database schema, and describes implementations of those algorithms. The main contribution of this thesis is a collection of software tools for MySQL that exploit hypergraph properties associated with database schema and query structures.

Approved:

David W. Juedes

Associate Professor of Electrical Engineering and Computer Science
This work is dedicated to my favorite ever loving eternal father, Christ
Acknowledgments

I am very thankful to my advisor, Dr. David W. Juedes for his endless patience, support and commitment. I am greatly indebted to him for his indispensable advice and constant encouragement. I would like to thank my thesis committee members Dr. Cynthia R. Marling, Dr. Frank Drews, and Dr. Xiaoping Shen. I would like to thank my husband, Arunachalam, for being a constant driving force behind me to finish my thesis and earn my masters degree. I am very thankful to my family and friends for their support and understanding.
Table of Contents

Abstract ......................................................... 3
Dedication ....................................................... 4
Acknowledgments ............................................... 5
List of Tables ................................................... 8
List of Figures ................................................... 9
1 Introduction .................................................. 10
   1.1 A Simple Motivating Example .......................... 11
   1.2 Related Work ........................................... 14
   1.3 Thesis Overview ....................................... 16
2 Relational Databases and MySQL .......................... 17
   2.1 Relational Data Model ................................. 17
   2.2 Relational Algebra Operators ......................... 18
   2.3 Conjunctive Queries .................................. 24
3 Hypergraphs and Hypergraph Decompositions ............... 26
   3.1 Hypergraphs ............................................ 26
   3.2 Hypergraph Decompositions ......................... 30
4 Degrees of Acyclicity and Their Detection Algorithms 36
   4.1 α-Acyclicity ............................................ 37
   4.2 β-Acyclicity ............................................ 38
   4.3 γ-Acyclicity ............................................ 42
   4.4 Berge-Acyclicity ....................................... 47
   4.5 Normalization and Acyclicity ......................... 48
5 Database Analysis Tools For MySQL ......................... 50
   5.1 A Database to Relation File Converter ............... 51
   5.2 The Relation File to Hypergraph Converter ........... 53
   5.3 Hypergraph Analyzer ................................... 53
   5.4 The SQL Query Parser ................................ 54
   5.5 High Level Query to Normal Query Converter .......... 55
   5.6 Software Implementation Details .................... 56
   5.7 Software Usage ........................................ 56
   5.8 Software Results ...................................... 57
Bibliography ...................................................... 62
List of Tables

1.1 Relations for University Schema ........................................... 12
4.1 Subset of University Schema ............................................... 38
4.2 Subset of University Schema with Primary Keys ....................... 49
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Hypergraph for University Schema</td>
<td>13</td>
</tr>
<tr>
<td>1.2</td>
<td>Hypergraph for Conjunctive Query</td>
<td>15</td>
</tr>
<tr>
<td>2.1</td>
<td>Employee</td>
<td>18</td>
</tr>
<tr>
<td>2.2</td>
<td>Department</td>
<td>18</td>
</tr>
<tr>
<td>2.3</td>
<td>Company Database</td>
<td>18</td>
</tr>
<tr>
<td>2.4</td>
<td>Relation R</td>
<td>19</td>
</tr>
<tr>
<td>2.5</td>
<td>Relation S</td>
<td>19</td>
</tr>
<tr>
<td>2.6</td>
<td>Union</td>
<td>19</td>
</tr>
<tr>
<td>2.7</td>
<td>Difference</td>
<td>20</td>
</tr>
<tr>
<td>2.8</td>
<td>Cartesian Product</td>
<td>21</td>
</tr>
<tr>
<td>2.9</td>
<td>Projection</td>
<td>22</td>
</tr>
<tr>
<td>2.10</td>
<td>Selection</td>
<td>22</td>
</tr>
<tr>
<td>2.11</td>
<td>Equi-join $R \Join S_{C_3=C_7}$</td>
<td>23</td>
</tr>
<tr>
<td>3.1</td>
<td>Acyclic Hypergraph</td>
<td>29</td>
</tr>
<tr>
<td>3.2</td>
<td>Cyclic Hypergraph</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>Reduced Cyclic Hypergraph</td>
<td>30</td>
</tr>
<tr>
<td>3.4</td>
<td>The Hypertree Decomposition for the Hypergraph in Figure 1.1</td>
<td>35</td>
</tr>
<tr>
<td>4.1</td>
<td>$\alpha$-Acyclic Hypergraph</td>
<td>39</td>
</tr>
<tr>
<td>4.2</td>
<td>Pure Cycle</td>
<td>40</td>
</tr>
<tr>
<td>4.3</td>
<td>Weak-$\beta$ Cycle</td>
<td>41</td>
</tr>
<tr>
<td>4.4</td>
<td>$\beta$-Acyclic Hypergraph</td>
<td>41</td>
</tr>
<tr>
<td>4.5</td>
<td>$\gamma$-Cycle</td>
<td>44</td>
</tr>
<tr>
<td>4.6</td>
<td>$\gamma$-Acyclic Hypergraph</td>
<td>44</td>
</tr>
<tr>
<td>4.7</td>
<td>Cyclic Hypergraph</td>
<td>49</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Database systems are ubiquitous in our current society. Large organizations such as Amazon.com, Google, and even Ohio University rely heavily on databases for their day-to-day operations. Therefore, the ability to process database queries and optimize database access is an important problem in computer science with a broad range of applications. Due to the great importance of these tasks, significant recent research has concentrated on finding efficient database query evaluation techniques. Much of this research has exploited graph theory concepts in order to achieve efficient algorithms. In particular, the graph theory concepts of hypergraphs, hypergraph decompositions, and hypertree decompositions have played an important role in this recent research.

The connection between databases and hypergraphs is straightforward. A given relational database schema (a set of attributes and a set of relations on those attributes) can easily be represented by a hypergraph where the set of nodes in the hypergraph correspond to the set of attributes in the database schema, and each hyperedge corresponds to a set of attributes included in a relation in the database schema. It has been known for at least 25 years (Yannakakis 1981) that various NP-complete problems on database schema with “simple” (acyclic) hypergraphs admit polynomial-time algorithms. Similar results hold for “simple” database queries.

Recent research has concentrated on expanding the scope of “simple” database schema (and database queries) to schema whose hypergraphs have hypergraph decom-
positions with bounded hypertree width. It is now known that various NP-complete problems on database schema (and database queries) are solvable in polynomial-time if the given object has bounded hypertree width. The objective of this research is to provide analysis tools for future research in database schema and database query optimization. In this vein, the popular open-source database system MySQL was chosen as the target platform for database schema and database query information. The contribution of this thesis is a collection of software tools that exploit hypergraph properties associated with database schema and query structures. Given a database schema or a conjunctive query, this software determines whether the hypergraph associated with the input schema is acyclic or cyclic. Furthermore, if the database schema is acyclic its precise degree of acyclicity is determined. The software constructs a hypertree decomposition for every acyclic database schema. Whenever the given input database schema is $\gamma$-acyclic, such a database schema supports high level queries where specifications on how to join relations can be omitted. Our software tool accepts such high level meta queries and converts them to standard queries that can be processed by most of the available commercial query parsers. Finally, the software validates basic MySQL select-project-join queries.

1.1 A Simple Motivating Example

To begin, a simple motivating example is provided that explains the basic terminology that is used in this thesis. Consider the simple relational database schema provided in Table 1.1. This database schema stores information about students, classes, faculty, departments, and relatives at a university. This database schema maps naturally to the hypergraph found in Figure 1.1. This hypergraph is acyclic as is explained next.

In the literature (Yannakakis 1981; Beeri et al. 1981), a hypergraph is said to be acyclic if the repeated application of the following three reduction rules results in an empty hypergraph. In the following three reduction rules, $V$ is used to represent the set of vertices in the hypergraph and $E$ is used to represent set of hyperedges.
Faculty Relation

| Fac_FirstName | Fac_LastName | Fac_Id | Dept_Id |

Student Relation

| Std_FirstName | Std_LastName | Std_Id | Major_Dept_Id |

Course Relation

| Course_Name   | Course_Id    | Fac_Id | Dept_id |

Class Relation

| Class_Id      | Course_Id    | Std_Id |

Department Relation

| Dept_Id       | Dept_Name    | Chair_FName | Chair_LName |

Relatives Relation

| Person1_FName | Person1_LName | Person2_FName | Person2_LName |

Table 1.1: Relations for University Schema

1. **Sole Node Removal**: If a vertex \( v \in V \) appears in only one edge \( e_1 \in E \), delete \( v \) from \( V \) and from that edge \( e_1 \).

2. **Ear Removal**: Delete edge \( e_1 \) from \( E \) if there is another edge \( e_2 \in E \) such that \( e_1 \subseteq e_2 \).

3. **Sole Edge Removal**: If there is only one edge \( e \in E \) remaining, that edge can be removed along with its associated vertices.

Notice that Rule 3 is essentially derived from Rule 1 since if only a single edge remains in the graph, all of its vertices will be deleted by Rule 1 until only an empty edge remains. This process is called the *Graham* or *GYO* reduction.

To see that the hypergraph in Figure 1.1 is acyclic, observe that *Std_FirstName*, *Std_LastName*, *Major_Dept_ID* can be deleted from the top hyperedge via Rule 1. This leaves an edge with just the attribute *Std_ID*. This edge can be deleted by Rule 2 since the edge containing *Std_ID*, *Class_ID*, *Course_ID* is a superset of that edge.
This process continues until the single edge \(\{\text{Course\_ID}, \text{Course\_Name}, \text{Fac\_ID}, \text{Dept\_ID}\}\) remains. This edge can be deleted via Rule 3.

As explained in Chapter 4, the hypergraph in Figure 1.1 is actually \(\gamma\)-acyclic, and hence more powerful meta queries for this relational database can be given. The transformation of high level meta query to normal query is explained in the following example. Let \(R\) be the university relational schema already shown in Table 1.1. The associated hypergraph \(H_R\) is \(\gamma\)-acyclic and so is \(R\). Consider the following high level meta query.

\[
\text{SELECT S\_FName, S\_LName, F\_FName, F\_LName}
\text{FROM Student, Faculty;}
\]

Intuitively, we would like to find all the students who are taking classes from an individual faculty member. However, this query will return the Cartesian product of the Student and Faculty relations since there are no common attributes between the two relations and there is no "where" clause in this query. Moreover, the above query does not explain how to join relations Student and Faculty.
In the schema $R$, there is no common attribute that relates Student and Faculty. However, since this relational schema is $\gamma$-acyclic, any two attributes can be linked only in a single way. The analysis tool presented here can find the exact join path which is shown below.

```sql
SELECT S_FName, S_LName, F_FName, F_LName
FROM Student, Faculty, Class, Course
WHERE Student.S_Id = Class.S_Id
AND Class.Co_Id = Course.Co_Id
AND Course.F_id = Faculty.F_Id;
```

Database queries can also be represented as hypergraphs using an approach similar to the one used to map a database schema to its hypergraph. Let us consider conjunctive queries for this discussion. Every variable in the query corresponds to a vertex in the hypergraph and each atom in a conjunctive query forms a hyperedge. Consider the following conjunctive query from database schema shown in Table 1.1 to find the list of students who are enrolled in the EECS Department. A query against a given database schema can contain any existing relation in the schema. Every attribute in the relation could be either a variable or constant. Consider the following query:

\[
S(fn, ln) : \neg Student(fn, ln, s1, D1) \& Department(D1, "EECS", cn)
\]

The hypergraph representation of the above query is shown in Figure 1.2. Notice that the query hypergraph is actually a sub-hypergraph of the database schema hypergraph which is already shown in Figure 1.1. Sub-hypergraphs of $\beta$, $\gamma$, and Berge acyclic hypergraph are always acyclic. This assures that queries against $\beta$, $\gamma$, and Berge acyclic database schemas will always be acyclic. But it should be observed that acyclic database queries can be built from a cyclic database schema. So, even if the database schema as such is hard to analyze, its sub-hypergraphs can be acyclic and thus easier to evaluate. Recently, many related works have concentrated on query structures rather than on the database.

### 1.2 Related Work

In an early paper, Chandra and Merlin showed that answering conjunctive queries in their general case is NP Complete (Chandra and Merlin 1977). Since then, finding
tractable conjunctive query classes has become an active area of research. In 1981, Yannakakis showed that acyclic conjunctive queries can be evaluated in polynomial time (Yannakakis 1981). Also in 1981, Beeri et al. summarized various properties of acyclic database schemes (Beeri et al. 1981). In 1983, Fagin gave various equivalent definitions for $\beta$ and $\gamma$ acyclicity and provided polynomial-time algorithms for determining the degree of acyclicity of a hypergraph. (Fagin 1983). In the same paper, Fagin also proved the additive nature of the hierarchical acyclicity degrees. In 1982, D’Atri and Moscarini (D’Atri and Moscarini 1982) provided a recursive pruning algorithm to determine the acyclicity degree ($\alpha, \beta, \gamma$ or Berge) for hypergraphs. Algorithms for computing projections, minimizing joins, inferring dependencies, and testing for dependency satisfaction for acyclic database schemes are given in (Yannakakis 1981).

Much recent research concentrates on efficient evaluation of database queries. Gottlob et al. showed that the precise complexity of acyclic Boolean conjunctive query evaluation is LOGCFL-Complete (Gottlob et al. 1998). Flum et al. extended results on conjunctive query evaluation to a large class of formulas through tree decompositions (Flum et al. 2002). Papadimitriou and Yannakakis proved that acyclic conjunctive queries with inequalities can be evaluated in fixed parametrized polynomial time in its input and output (Papadimitriou and Yannakakis 1997). Rajaram and Ullman proved that there is a natural outer join sequence producing the full disjunction if and only if the set of relation schemes forms a connected, $\gamma$-acyclic hypergraph (Rajaraman and Ullman 1996). Grohe et al. showed that the evaluation of conjunctive queries with bounded tree width is tractable (Grohe et al. 2001). Decision problems like evaluation of Boolean conjunctive queries and query containment were shown to be tractable for acyclic queries (Gottlob et al. 2002). Gottlob et al. also
showed that the evaluation of conjunctive queries having hypertree width at most $w$, where $w \geq 1$, is solvable in polynomial time.

1.3 Thesis Overview

In this thesis, attention is restricted to the relational database model. Some of the popular relational DBMS packages are Oracle (ora 2006), MySQL (mys 2006), SQLServer (sql 2006), DB2 (DB2 2006), PostgreSQL (pos 2006) etc. Among the query types, conjunctive query structures and validate SPJ(Select, Project and Join) queries using SQL query parser are explored. MySQL relational database is used in the database tool implementation.

In this thesis, algorithms that employ hypergraph decomposition to detect the acyclic (cyclic) degree of database schema are studied and implemented. The discussion is divided into 5 chapters. Chapter 2 describes the necessary background knowledge on the relational database model specifically about the MySQL database, relational database operators, query types, and the conjunctive query evaluation problem. Chapter 3 discusses hypergraph structures and their natural correspondence with relational schema and queries. Chapter 3 classifies a hypergraph as cyclic or acyclic using the well known GYO reduction algorithm. Chapter 4 describes various degrees of acyclicity, their detection algorithms, and how to build hypergraph decompositions. Chapter 5 discusses the software tools for database analysis. Finally, Appendix A provides the software analysis of the directory relational schema used for the EECS department at Ohio University.
Chapter 2

Relational Databases and MySQL

This chapter reviews the relational database model that is used throughout the thesis. In particular, this chapter discusses how the relational model is used in the MySQL database system. MySQL is one of the most widely used open source relational database. MySQL standard database server for 64 bit sun solaris sparc machines (Version 4.19) is used for this discussion.

2.1 Relational Data Model

The relational data model is the most popular and widely used data model. The relational model is based on the mathematical concept called relational calculus. E.F.Codd set the rules to develop databases based on relational models (Codd 1970). A relational database scheme $R$ is a collection of relations $(r_1, r_2, \ldots r_n)$, and the current values of all relations form the database. Every relation $r$ has a finite set of attributes $a_1, a_2, \ldots a_m$. A finite domain $D_{a_i}$ of atomic values is associated with each attribute $a_i$. In the relational database terminology, the relational schemes are known as tables and every attribute is a column in that table. For example, Figures 2.1 and 2.2 show the relational schemes Employee and Department respectively. The Employee relation has the attribute set $\{\text{EmpID, Name, DeptID}\}$. Figure 2.3 shows the current values of the relational schemes that form a company database.
2.2 Relational Algebra Operators

The main operations of the relational algebra are union, set difference, Cartesian product, selection (allowing only some lines of a table), and projection (allowing only some columns). Details are provided about these relational algebra operators in this section. The operands of relational algebra are either constant relations or variables denoting relations of a fixed arity. Let \( R \) and \( S \) be the two relations as shown in Figures 2.4 and 2.5, respectively.

2.2.1 Union

As is standard in set theory, the union operator is denoted by \( \cup \). \( R \cup S \) is the set of tuples that are in \( R \) or \( S \) or both, i.e., \( R \cup S = \{ x : x \in R \text{ or } x \in S \} \).

If a relation \( R \) had \( n \) tuples and \( S \) had \( m \) tuples, then \( R \) union \( S \) would have \( \leq n + m \) tuples. It is logical to take the union among relations with the same arity.

<table>
<thead>
<tr>
<th>Employee</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPID</td>
<td>Name</td>
<td>DeptID</td>
</tr>
<tr>
<td>1236</td>
<td>Bill Joe</td>
<td>MF12</td>
</tr>
<tr>
<td>3478</td>
<td>Cary Gol</td>
<td>QS36</td>
</tr>
<tr>
<td>1356</td>
<td>Bridge Hyde</td>
<td>QS36</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Department</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DeptID</td>
<td>DeptName</td>
<td>Location</td>
<td></td>
</tr>
<tr>
<td>MF12</td>
<td>Manufacturing</td>
<td>Denver</td>
<td></td>
</tr>
<tr>
<td>QS36</td>
<td>Quality Service</td>
<td>Houston</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.3: Company Database
For the relations $R$ and $S$ shown in Figures 2.4 and 2.5, $R$ union $S$ is shown in Figure 2.6. MySQL has a union operator (Keyword: UNION) which accepts two queries with the same arity.

The MySQL query for performing the union of all student names and faculty names would be as follows:

```
SELECT Fac_FirstName, Fac_LastName FROM Faculty
UNION
SELECT Std_FirstName, Std_LastName FROM Student
```

Notice that the attributes names in this union do not have to match.
2.2.2 Set Difference

Again, as is standard in set theory, set difference is denoted by the symbol $-$. $R - S$ is the set of tuples that are in $R$ but not in $S$, i.e., $R - S = \{x : x \in R \text{ and } x \notin S\}$. Figure 2.7 shows both $R - S$ and $S - R$. Note that while the union operator is commutative, the set difference is not commutative.

MySQL does not have an explicit operator to implement set difference. However, set difference can be implemented using sub queries as the following example illustrates. The MySQL query to find faculty who are not personally related to any of the students enrolled in the university would be as follows.

```sql
SELECT DISTINCT Fac_FirstName, Fac_LastName FROM Faculty
WHERE (Fac_FirstName, Fac_LastName)
    NOT IN
    SELECT Person1_FName, Person1_LName FROM Relatives
```

This query essentially uses the `NOT IN` keyword as the set difference operator.

2.2.3 Cartesian Product

As is standard in set theory, the Cartesian product is denoted using the product symbol $\times$. $R \times S$ is the result set with all possible combinations from $R$ and $S$, i.e., $R \times S = \{(a, b) : a \in R, b \in S\}$. If $R$ had $n$ rows and $c$ columns and $S$ had $m$ rows and $d$ columns, then $R \times S$ would typically have $m \times n$ rows with $c + d$ columns where the first $c$ columns form a tuple from $R$ and the next $d$ columns form a tuple from $S$. Figure 2.8 shows the Cartesian product of $R$ and $S$. 
When attributes are selected from relations without specifying the way to link the involved relations, the result set would be typically a Cartesian product of all rows from input relations. The following query is an example from MySQL that produces a Cartesian product result set:

```sql
SELECT Fac_FirstName, Fac_LastName,
     Std_FirstName, Std_LastName
FROM Faculty, Student
```

2.2.4 Projection

In database theory, the projection operator is denoted by $\pi$. Assume that there is a relation $R$ with attribute set $X$. Moreover, assume that $Y \subseteq X$. Then, the projection of $R$ by $Y$ (denoted by $\pi_Y$) is the relation $R'$ with attribute set $Y$ such that a tuple $T'$ is in $R'$ if and only if there is a tuple $T$ in $R$ where all of the values of the attributes in $T$ match the corresponding attributes in $T'$. In layman’s terms, the projection $\pi_Y$ strips all of the columns labeled by attributes from $Y$ out of the table for $R$ and removes all duplicate table entries. The projection can typically restrict the columns appearing in the result set. For example, $\pi_{1,4}(R)$ would project only column number 1 and 4 from relation $R$. Note that the attributes can either be appearance order numbers like the one shown here or the attribute’s definition name itself. Figure 2.9 shows the projection of columns $C_5$ and $C_8$ from relation $S$ denoted by $\pi_{C_5,C_8}(S)$. In a MySQL query, the projection of required attributes is provided in the attribute
list after the SELECT keyword. A MySQL query to project the first and last names of the faculty is as follows:

\[
\text{SELECT Fac_FirstName, Fac_LastName FROM Faculty}
\]

<table>
<thead>
<tr>
<th>C5</th>
<th>C8</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>18</td>
</tr>
<tr>
<td>c</td>
<td>86</td>
</tr>
<tr>
<td>d</td>
<td>11</td>
</tr>
</tbody>
</table>

Figure 2.9: Projection

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>3</td>
<td>123</td>
<td>11</td>
</tr>
<tr>
<td>c</td>
<td>4</td>
<td>350</td>
<td>37</td>
</tr>
</tbody>
</table>

Figure 2.10: Selection

### 2.2.5 Selection

The selection operator is denoted by \( \sigma \). Selection is an operation that selects specific tuples from the database based on a list of conditions. The condition argument to a select operator can include the attribute list, constants, logical operators and arithmetic comparison operators. The select operation \( \sigma_C(S) \), where \( C \) is the condition imposed on the relation \( S \), is the set of tuples that satisfy the formula \( C \) after substituting the corresponding tuple values for the appropriate attributes in the formula \( C \). For example, \( \sigma_{1 \neq 2}(S) \) would select the tuples from relation \( S \) where its first attribute is not equal to its second attribute. For instance, Figure 2.10 shows the tuples from relation \( R \) that satisfy the condition \( C_3 > 120 \) ( \( \sigma_{C_3>120}(R) \)).

MySQL uses the \textit{WHERE} keyword to indicate selection. For example, the MySQL query to select students having last name of Smith is as follows:

\[
\text{SELECT Std_FirstName, Std_LastName FROM Student}
\text{WHERE Std_LastName = 'Smith'}
\]
2.2.6 Join

In relational algebra, the join operator ($\bowtie$) in many ways behaves like the Cartesian product operator. Given two relations $R_1$ and $R_2$, the join of $R_1$ and $R_2$ is the relation whose attributes are the union of the attributes $A_1$ and $A_2$ from $R_1$ and $R_2$, respectively, and whose tuples correspond to merged tuples from $R_1$ and $R_2$, i.e., $T$ is tuple from $R_1 \bowtie R_2$ if and only if there exist tuples $T_1$ and $T_2$ from $R_1$ and $R_2$ whose tuples match on $A_1 \cap A_2$ and $T$ matches $T_1$ and $T_2$ on all attributes from $T_1$ and $T_2$. The join described above is referred to as a natural join since the attribute names must match. In general, joins do not have to be natural joins. One attribute in one relation is equated to another attribute in another relation.

A more general join operation is called a theta-join, and it involves a predicate using comparisons operators $\{=, \neq, >, <, \geq, \leq\}$ among attributes from the two relations. A theta join is the relation of all combined tuples from the two relations $R_1$ and $R_2$ satisfying the predicate. An equi-join is a $\theta$-join where $\theta$ is equality. Figure 2.11 shows the result of equi-join operation $R \bowtie S_{C_3=C_7}$

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
<th>C7</th>
<th>C8</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>3</td>
<td>123</td>
<td>11</td>
<td>d</td>
<td>9</td>
<td>123</td>
<td>11</td>
</tr>
<tr>
<td>c</td>
<td>8</td>
<td>100</td>
<td>86</td>
<td>c</td>
<td>8</td>
<td>100</td>
<td>86</td>
</tr>
</tbody>
</table>

Figure 2.11: Equi-join $R \bowtie S_{C_3=C_7}$

MySQL can specify various types of joins via the SELECT and WHERE keywords. For example, a MySQL query to find students taking one or more courses under faculty Anna Bell is given as follows:

```sql
SELECT S_FName, S_LName, F_FName, F_LName
FROM Student, Faculty, Class, Course
WHERE Student.S_Id = Class.S_Id
AND Class.Co_Id = Course.Co_Id
AND Course.F_id = Faculty.F_Id
AND F_FName = 'Anna'
AND F_LName = 'Bell';
```
Notice that the relations Student and Faculty are joined through a series of relations (Class and Course). MySQL does not provide an implicit natural join. The links between relations have to be explicitly stated.

2.3 Conjunctive Queries

A query $Q$ is a mapping of a database DB to a relation $Q(n)$, where $n$ is the arity (# of attributes) of the query $Q$. Various classes of queries have been studied in the literature, including:

1. conjunctive queries,
2. first order queries,
3. positive queries, and
4. datalog queries.

Conjunctive queries are the most basic query class. In the literature, CQ denotes the class of all conjunctive queries. Conjunctive queries can be viewed as those queries that use relational algebra without union and set difference. A conjunctive query builds a new relation $Ans$ of arity $k$ from the collection of relations $R_1, R_2, \ldots, R_n$ via the conjunction (AND). Let $V_1, V_2, \ldots, V_m$ be $m$ variables, then $Ans$ is the table of $k$-tuples $\langle V_{i1}, V_{i2}, \ldots, V_{ik} \rangle$ satisfying $R_1(a_{i11}, a_{i12}, \ldots, a_{i1\lambda_1}) \land \ldots \land R_n(a_{n11}, a_{n21}, \ldots, a_{n\lambda_n})$ where each $a_{ij}$ is either a variable or a constant. A conjunctive query $Q$ is a Boolean conjunctive query (BCQ) if its result relation does not contain any variables. A Boolean conjunctive query evaluates to true if and only if there exists a setting of the variables $V_1, V_2, \ldots V_m$ satisfying $R_1(a_{111}, a_{112}, \ldots, a_{11\lambda_1}) \land \ldots \land R_n(a_{n11}, a_{n21}, \ldots, a_{n\lambda_n})$.

2.3.1 Some Conjunctive Query Examples

This section provides examples of MySQL queries and their corresponding conjunctive queries.

A MySQL query to select a list of students with the last name "Smith" is as follows:
SELECT Std_FirstName, Std_LastName FROM Student
WHERE Std_LastName = ‘Smith’

This is a conjunctive query. In the notation given above, the conjunctive query for the above selection query is \( S(fn, ln) : = \neg \text{Student}(fn, ”Smith”, s1, d1) \). Next, consider a Boolean conjunctive query to answer if there is any student taking a course under a faculty who is a relative of that student:

\[
S() : = \text{Student}(Sfn, Sln, S1, d1) \land \text{Class}(C1, Co1, S1) \land \text{Course}(cn, Co1, F1, d2) \\
\land \text{Faculty}(Ffn, Fln, F1, d3) \land \text{Relative}(Sfn, Sln, Ffn, Fln).
\]

The MySQL query for the above conjunctive query would be as follows.

```
SELECT S_FName, S_LName, F_FName, F_LName
FROM Student, Faculty, Class, Course, Relative
WHERE Student.S_Id = Class.S_Id
AND Class.Co_Id = Course.Co_Id
AND Course.F_id = Faculty.F_Id
AND Relative.Person1_FName = Student.S_FName
AND Relative.Person1_LName = Student.S_LName
AND Relative.Person2_FName = Faculty.F_FName
AND Relative.Person2_LName = Faculty.F_LName
```

Notice that, in a conjunctive query, when a variable occurs in more than one clause its corresponding SQL query includes an equi-join operation. Notice that the various equi-joins are connected with keyword "AND" in SQL similar to the conjunction operator \( \land \) in the language of conjunctive queries.

On a final note, the other query classes mentioned earlier add various levels of expressive power to database queries. A positive query is the union of conjunctive queries. Positive queries add disjunction (OR) to conjunctive queries. First order queries add set difference. Datalog queries add recursion to positive queries.
Chapter 3

Hypergraphs and Hypergraph Decompositions

This chapter provides definitions for the notions of hypergraphs, acyclicity in hypergraphs, tree decompositions, and hypertree decompositions. As hypergraphs are a somewhat natural graph representation of database schema and queries, it is essential to explore the properties of hypergraphs.

3.1 Hypergraphs

We begin by providing a precise, mathematical definition of a hypergraph. Intuitively, a hypergraph is a graph where edges can connect one or more vertices.

Definition 3.1 A hypergraph is a tuple $H = (V, E)$, where $V$ is a non-empty set of vertices and $E \subseteq P(V) - \emptyset$ is the edge set, i.e., the elements of $E$ are non-empty sets of vertices.

An element $e \in E$ is referred to as a hyperedge. Notice that if the cardinality of all edges in a hypergraph $H$ are equal to 2, such a hypergraph is a simple graph.
3.1.1 Hypergraphs, Database Schema, and Conjunctive Queries

As mentioned in the introduction, given any database relational schema $R$, there is a corresponding hypergraph $H_R$. There is a natural correspondence between database relational schema and hypergraphs. Each attribute $a_i$ in $R$ corresponds to a vertex $v_i$ in the hypergraph $H_R$. Each relation $r_i(a_{i1}, \ldots, a_{i\lambda_i})$ in $R$ forms a hypergraph edge $e_i = \{v_{i1}, \ldots, v_{i\lambda_i}\}$. Notice that the common attributes among relations connect corresponding hypergraph edges. As shown in the introduction, Figure 1.1 gives the hypergraph representation for the university schema listed in Table 1.1.

 Conjunctive queries can also be represented by hypergraphs. To see this, let $Q$ be a conjunctive query with $m$ variables $V_1, \ldots, V_m$. Every variable $V_i$ in a conjunctive query $Q$ corresponds to a vertex $v_i$ in the hypergraph $H_Q$. Then, for each relation $R_i$ mentioned in the query $Q$, we have a hyperedge $e_i$ containing all of the vertices $v_j$ associated with the variables $V_j$ listed in the query $Q$ with relation $R_i$. For example, consider the following conjunctive query for the university relation in Table 1.1 to find the list of students who are enrolled in the EECS Department:

$$S(fn, ln) : -Student(fn, ln, s1, D1) \land Department(D1, "EECS", cf_n, cln)$$

The hypergraph representation of above query is shown in Figure 1.2.

In this example, notice that the constant “EECS” is included in the hypergraph as a vertex. Since this constant appears in only one hyperedge, it can be deleted without changing the underlying structure of the hypergraph.

3.1.2 Acyclic and Cyclic Hypergraphs

The introduction mentioned that acyclic hypergraphs are “simple” hypergraphs since many computational problems can be solved in polynomial-time on acyclic hypergraphs. In regular graph theory, acyclic graphs are either trees (or forests) in the undirected case or directed acyclic graphs (DAG) in the directed case. Directed acyclic graphs have important applications in various computer science problems. In graph theory, an acyclic graph is a graph with no path of length 3 or more that starts with a node and leads back to itself. The acyclic notion in hypergraphs differs from
that used by graph theorists. Acyclic hypergraphs are hypergraphs that could be represented by a tree or a forest with one hyperedge in every tree node and every vertex in a hyperedge forms an induced subtree. Acyclic and cyclic hypergraphs can be detected in polynomial time using a reduction algorithm that we refer to as the GYO reduction that was discovered independently by Graham (Graham Sept 1979) and by Yu and Ozsoyoglu (Yu and Ozsoyoglu Nov. 1979). This reduction process is given in the introduction. The formal definition of a acyclic hypergraph is given as follows.

Definition 3.2 A hypergraph \( H \) is acyclic if its GYO reduction is empty. Otherwise, the hypergraph \( H \) is cyclic.

Acyclic hypergraphs are preferred widely due to a number of desired computational properties that make query optimization easier than in the general case (Yannakakis 1981). In fact, it has been posited that acyclic schema are the natural ones, and any database scheme that is not acyclic could be a database design error.

As the introduction explains, the GYO reduction of a hypergraph is obtained by performing a series of ear and sole node removals until no more edges can be removed. When \( H \) equals its reduction, then \( H \) is reduced. After the reduction, if \( H \) is empty then that hypergraph is acyclic. Otherwise \( H \) is cyclic. The GYO reduction of a hypergraph is unique, independent of the order of ear removals.

Some Examples

Two examples are given below to demonstrate the GYO reduction reduction process. To begin, consider the hypergraph \( H_1 \) in Figure 3.1. As you can see, \( H_1 \) has 4 edges \( \{A, E, F\}, \{A, C, B\}, \{E, C, D\} \) and \( \{A, E, C\} \). We pick \( ear_1 \) as \( \{A, E, F\} \) and \( consume_1 \) as \( \{A, E, C\} \). In edge \( ear_1 \), node F does not belong to any other hyperedge and can be deleted from \( ear_1 \) using the sole node removal operation. Our new \( ear'_1 = \{A, E\} \) is a subset of \( consume_1 \) edge \( \{A, E, C\} \). By edge removal rule \( ear'_1 \) can be deleted with the consuming edge as \( \{A, E, C\} \). Similarly, edge \( \{A, E, C\} \) serves as the consuming edge for the remaining edges \( \{A, B, C\} \) and \( \{C, D, E\} \). After removing all
the possible ears, we are left with our sole consuming edge \( \{A, E, C\} \). By rule 3, the sole remaining edge can be deleted and we are left with a empty hypergraph. So, our hypergraph \( H_1 \) is acyclic. Consider the same example with the edge \( \{A, E, C\} \) removed as in Figure 3.2. In the new hypergraph \( H_2 \), the node removal rule can be applied to all three edges. This will reduce \( H_2 \) to \( H'_2 \) as shown in Figure 3.3. None of the edges can be deleted further from \( H'_2 \) since no edge is a subset of another edge. In other words there is no consuming edge. As \( H'_2 \) cannot be reduced to an empty hypergraph, \( H_2 \) is a cyclic hypergraph.

An Implementation of the GYO reduction

This section provides some details of how the GYO reduction is implemented in the software. The performance of our algorithm is measured in terms of \( n \), the number...
of vertices in the hypergraph, and \( M = \sum_{e \in E} |e| \), the total number of vertices in the edges of the hypergraph.

The GYO reduction is implemented by first pre-processing the graph to count, for every vertex \( v \in V \), the total number of edges in which \( v \) occurs. This pre-processing step can be performed in \( O(M) \) steps.

As a second step, the GYO algorithm operates as follows. Every edge of the hypergraph \( H \) is compared to the remaining edges to check for a possible ear-consumer relationship. Let \( e_j \) be an edge in the hypergraph, and let \( c_i \) be another edge in the hypergraph. For every \( \{e_j, c_i|(i \neq j)\} \) pair, we compute \( x = \{e_j - c_i\} \), and check if \( e_j \) is the only edge containing all the elements of \( x \). This can be performed in linear time by consulting the preprocessed table mentioned above. If the nodes of \( x \) are only contained in \( e_j \), then \( e_j \) is an ear that can be consumed by \( c_i \). In such a case \( e_j \) is deleted from \( H \). Then we update the vertex-incidence table by decrementing the edge count for all vertices in \( e_j \).

This process continues for each pair of edges. The outer ear iteration loop is incremented by 1 and the inner loop is reset to the first edge. If \( e_j \) is not an ear for \( c_i \) then \( i \) is incremented to point to the next edge until a consumer node is found or until all the edges are compared. In the later case, \( e_j \) is not an ear for any of the edges and so \( j \) is incremented to the next edge and the comparison proceeds similarly. When the outer iteration ends, if our reduced hypergraph is empty then the original hypergraph is an acyclic hypergraph. The above implementation of the GYO reduction algorithm runs in the order of \( M^2 \) steps.
3.2 Hypergraph Decompositions

As mentioned earlier, it is well-known that NP hard problems are tractable on input structures having bounded cyclicity measures such as tree width, query width, cut-set width and so on (Flum et al. 2002; Grohe et al. 2001; Kolaitis and Vardi 1998; Chekuri and Rajaraman 2000). This section gives basic definitions for tree and hypertree decompositions. The discussion begins by defining tree decompositions, a well known cyclic measure on regular graphs.

3.2.1 Tree Decomposition for Regular Graphs and Hypergraphs

Tree decompositions of regular graphs were first introduced by Robertson and Seymour (Robertson and Seymour 1986), and are now widely used in the literature on graph algorithms.

Definition 3.3 A tree decomposition of a graph \( G = (V, E) \) is a pair \( \langle T, X \rangle \) where \( T \) is a tree \( T = \langle I, f \rangle \) and \( X : I \rightarrow P(V) \) is a mapping of the nodes of the tree to subsets of \( V \), such that the following conditions are satisfied.

1. \( \bigcup_{i \in I} X(i) = V \), i.e., all the vertices appear somewhere in the tree,
2. For every edge \( \{u, v\} \in E \), there exists an \( i \in I \) such that \( \{u, v\} \in X(i) \), and
3. For every vertex \( v \in V \), the subtree induced by \( I_v = \{i \in I | v \in X(i)\} \) is connected.

The width of a tree decomposition is given by \( \max\{|X(i)| - 1, i \in I\} \). The tree width of a graph \( G \) is the minimum width of any tree decomposition of \( G \).

Since this study works primarily with hypergraphs, an explanation of how tree decompositions can be extended to hypergraphs is merited. A first approach converts a hypergraph to a simple graph called the graph primal and builds a tree decomposition for the graph primal. The graph primal of a hypergraph is defined as follows.
Definition 3.4 The graph primal $G' = (V', E')$ of a hypergraph $H = (V, E)$ is a simple graph with vertex set $V' = V$ and edge set $E' = \{(u, v) \mid u, v \in e \text{ for some } e \in E\}$, i.e., any two vertices in $V'$ are connected if they appear in some hyperedge in $H$.

We define the tree width of a hypergraph $H$ to be the treewidth of the graph primal of $H$. While this notion of the treewidth of a hypergraph is somewhat straightforward, there are several disadvantages to this definition.

The main disadvantage of this approach is that a hypergraph graph edge with $n$ nodes is converted into a clique of size $n$ in the graph primal. Since it is well-known that graphs with cliques of size $n$ have treewidth at least $n - 1$, this means that hypergraphs with “big” hyperedges have large treewidth. Notice that this is true even if the hypergraph is acyclic. Since small treewidth is generally associated with graphs that are “almost” acyclic, this notion of the “tree width” of a hypergraph may not help us to develop fast algorithms for problems on hypergraphs.

Since this first attempt at using treewidth with hypergraphs has several disadvantages, other notions of “hypertree width” were developed.

### 3.2.2 Hypertree Decompositions

As observed in the previous section, the treewidth measure does not work well for hypergraphs since even simple hypergraphs can have an unbounded tree width. Therefore, it is essential to use a different cyclic measure width that fits hypergraphs naturally. Recently, Gottlob et al. defined a notion of a hypertree width using the concept of a hypertree decomposition (Gottlob et al. 1999). Hypertree decompositions are similar to tree decompositions in the sense that the graph is broken into a tree. However, a hypertree decomposition adds an extra bag containing hyperedges to each node of the tree. The hypertree width of a hypergraph is the maximum cardinality of the set of hyperedges associated with each node in the tree. This definition gives acyclic hypergraphs a hypertree width of one. We now provide the precise definitions.

Definition 3.5 (Gottlob et al. 2002) A hypertree for a hypergraph $H = (V, E)$ is a triple $\langle T, X, \lambda \rangle$, where $T = (I, f)$ is a rooted tree, $X : I \rightarrow P(V) - \emptyset$ and $\lambda : I \rightarrow$
The functions $X$ and $\lambda$ label each vertex $p \in I$ in the tree with a set of vertices and hyperedges, respectively.

In a hypertree $\langle T, X, \lambda \rangle$, we write $T_i$ for the subtree of $T$ rooted at $i$. If $T' = (I', f')$ is a subtree of $T$, then $X(T') = \bigcup_{v \in T'} X(v)$.

**Definition 3.6** *(Gottlob et al. 2002)* A hypertree decomposition of a hypergraph $H = (V, E)$ is a hypertree $HD = \langle T, X, \lambda \rangle$ for $H$ that satisfies the following conditions.

1. For each edge $e \in E$, there exists $i \in I$ such that $e \subseteq X(i)$;
2. For each vertex $v \in V$, the set $\{i \in I | v \in X(i)\}$ induces a subtree of $T$;
3. For each $i \in I$, $X(i) \subseteq \bigcup_{e \in \lambda(i)} e$;
4. For each $i \in I$, $(\bigcup_{e \in \lambda(i)} e) \cap X(T_i) \subseteq X(i)$.

Condition 1 requires that nodes in each hyperedge of $H$ should be contained in at least one bag of $T$. Condition 2 states the vertex connectedness property, i.e., if tree nodes $I_i$ and $I_k$ have vertex $v_a$, then all nodes in the unique path between $I_i$ and $I_k$ should contain $v_a$. Conditions 3 and 4 give the relationships among the hyperedge and vertex bags.

The width of a hypertree decomposition $HD = \langle T, X, \lambda \rangle$ is $\max\{(|\lambda(i)|), i \in I\}$. The hypertree width $hw(H)$ of a hypergraph $H = (V, E)$ is the minimum width of any hypertree decomposition of $H$. Gottlob and et al. have shown a LOGCFL algorithm to decide whether a given conjunctive query has a $k$–bounded hypertree width *(Gottlob et al. 2002)*.

### 3.2.3 Acyclic Hypergraph and Hypertree Decomposition

As mentioned above, acyclic hypergraphs have a hypertree width of 1. This requires a bit of an explanation. This connection was first made using the notion of a join tree.
**Definition 3.7** A join tree for hypergraph $H$ is a tree whose nodes are labeled by hypergraph edges such that (i) the set of tree nodes that contain any vertex $v \in V$ induces a connected subtree, and (ii) two tree nodes are connected if and only if the associated hypergraph edges share common vertices.

The connection between join trees and acyclic hypergraphs is well-known.

**Claim 3.1** A hypergraph is acyclic if and only if it has a join tree. (Beeri et al. 1983)

The connection between join trees, acyclic hypergraphs, and hypertree decompositions is given in the following claim that we provide without proof.

**Claim 3.2** A hypergraph $H = (V, E)$ is acyclic if and only if there is a join tree or a hypertree decomposition $\langle T, X, \lambda \rangle$ such that for every tree vertex $i \in I$ there exists a hyperedge $e \in E$ such that $\lambda(i) = e$ and $X(i) = e$.

Figure 3.4 shows a hypertree decomposition for the university schema example in Figure 1.1. This hypertree decomposition (join tree) was constructed via the GYO reduction process. The hypertree decomposition was built in bottom up fashion, i.e., the root node was constructed last. During the GYO reduction process, whenever a ear-consumer relationship occurs, such an ear-consumer pair is transformed into a child-parent node pair in our hypertree. Since reduction Rule 2 implies that any two edges that have some common nodes can be reduced only through the subset relation forming a ear-consumer relationship, the hypertree decomposition built using these rules preserves the vertex connectedness property. As the last step of the reduction process, Rule 3 deletes the remaining edge. This edge becomes the root node of our hypertree. Observe that, at this time, all the other edges would be connected to the root node either directly or through series of other nodes in the hypertree.

### 3.2.4 Acyclic Hypertree Decompositions in Literature

Acyclic hypergraphs and their associated join trees/hypertree decompositions have been employed in the literature on database algorithms for at least the last 25 years.
Figure 3.4: The Hypertree Decomposition for the Hypergraph in Figure 1.1.

Tarjan and Yannakakis showed that determining whether a CQ is acyclic and computing its join tree is possible in linear time (Tarjan and Yannakakis 1984). Yannakakis showed that acyclic BCQ can be evaluated in time polynomial in the input and output using join trees (Yannakakis 1981). Yannakakis used a sequential algorithm that processes the join tree in a bottom up fashion, joining two nodes at a time starting with leaves doing a series of \((n - 1)\) joins for a join tree with \(n\) nodes. Recent work by Gottlob et al. employed a highly parallel algorithm on join trees to show that the precise computation complexity of evaluating BCQ is LOGCFL complete (Gottlob et al. 1998).
Chapter 4

Degrees of Acyclicity and Their Detection Algorithms

In the last chapter, it was shown that acyclic hypergraphs have hypertree decompositions with a hypertree width of 1. However, it is possible for an acyclic hypergraph to have a subset of hyperedges, called a sub-hypergraph, with a hypertree width greater than 1. Since such a subset might correspond naturally to a query to the associated database, this means a query on a simple, acyclic database schema could result in a cyclic hypergraph for the query.

For example, consider the hypergraph shown in Figure 3.1. This hypergraph is acyclic, but one of its sub-hypergraphs, shown in Figure 3.2, is cyclic. In database terms, though a database schema is acyclic, conjunctive queries based over such acyclic databases could be cyclic. One approach to solving this problem is to restrict certain configurations or by adding special conditions to the definition of acyclicity.

In an early work, Fagin explored a number of straightforward modifications of the definitions of “acyclic” hypergraphs, and identified four degrees of acyclicity that were appropriate for work in databases (Fagin 1983). Fagin referred to these four degrees of acyclicity as $\alpha$, $\beta$, $\gamma$, and Berge acyclicity. Here, we list these degrees in the increasing order of their desirable database properties and restrictiveness. These four degrees of acyclicity have a number of desirable properties that have already been established in the hypergraph literature.
1. Each of the four degrees of acyclicity has a decision algorithm that runs in polynomial-time (Graham Sept 1979; Yu and Ozsoyoglu Nov. 1979; Fagin 1983), i.e., given a hypergraph, it is possible to efficiently determine whether that hypergraph is α, β, γ, or Berge acyclic.

2. Each of the last three degrees of acyclicity (β, γ, and Berge) have the property that every sub-hypergraph of such an acyclic hypergraphs are at least α-acyclic. This condition ensures that queries based on such acyclic databases are acyclic themselves. Since every sub-hypergraph of a β-acyclic hypergraph is α-acyclic, every sub-hypergraph is sure to enjoy at least the desirable properties of α-acyclicity.

3. Each of the last two degrees (γ and Berge) has the property that in the associated database schema, any two attributes can be related only in a single unique way. This property provides significant advantages to γ and Berge-acyclic database schema, since simple, high-level queries may be written that omit details on how to join various relations (Fagin 1983).

As one might expect, these four degrees of acyclicity are related to one another via the subset relation. Every Berge acyclic hypergraph is γ-acyclic. Every γ-acyclic hypergraph is β-acyclic. Finally, every β-acyclic hypergraph is α-acyclic. This relation is proper in the sense that there are γ-acyclic hypergraphs that are not Berge acyclic, etc. This chapter provides precise definitions for these various degrees of acyclicity. Also provided are implementation details on these algorithms for identifying degrees of acyclicity. While the algorithm implementations are original to this thesis, the core algorithmic ideas are due to Fagin’s original paper (Fagin 1983).

4.1 α-Acylicity

Following Fagin’s paper, we begin by defining α-acyclicity, the least restrictive degree of acyclicity. In the literature, the terms α-acyclic hypergraph and acyclic hypergraph are synonymous. Hence, a hypergraph $H$ is α-acyclic if and only if GYO reduction on $H$ results in an empty hypergraph.
Table 4.1: Subset of University Schema

<table>
<thead>
<tr>
<th>Faculty Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>F_Name</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Faculty_Course Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>F_Name</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Course Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Course_Name</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Faculty_Course_Dpt Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>F_Name</td>
</tr>
</tbody>
</table>

As we have seen before, the GYO reduction process identifies $\alpha$-acyclicity. For the sake of illustrating various degrees of acyclicity, a subset of the university schema in Table 4.2 is shown. The hypergraph $H$ shown in Figure 4.1 is $\alpha$-acyclic. Notice that, in this $\alpha$-acyclic example, a sub-hypergraph of $H$ (\{F_Name, F_Major, F_Address, Dept\}, \{Course_Name, Grader, F_Name\}, \{Course_Name, Room_No, Time, Dept\}) is cyclic. Formally, the notion of a subhypergraph is defined as follows.

**Definition 4.1** A sub-hypergraph $H_s = (V_s, E_s)$ of a hypergraph $H = (V, E)$ satisfies the following properties (i) $E_s \subset E$ and (ii) $V_s = \bigcup_{e \in E_s} e$.

Since the advantages and detection process for $\alpha$-acyclic hypergraphs have been explained previously, detailed discussion of the other degrees of acyclicity is merited.

### 4.2 $\beta$-Acyclicity

$\beta$-acyclicity is explained next. $\beta$-acyclicity is the next less restrictive degree of acyclicity after $\alpha$-acyclicity. Fagin defines $\beta$-acyclicity through $\beta$-cyclicity as follows.

**Definition 4.2** A hypergraph $H$ is $\beta$-acyclic if and only if it is not $\beta$-cyclic.

Fagin gives five equivalent definitions for $\beta$-cyclicity (Fagin 1983). Fagin’s second definition that defines $\beta$ cyclicity in terms of whether some subhypergraph is cyclic is used here.
**Definition 4.3** A hypergraph $H$ is $\beta$-cyclic if some sub-hypergraph of $H$ is cyclic.

For example, consider the hypergraph $H_1$ shown in Figure 4.1. This hypergraph is $\alpha$-acyclic but $\beta$-cyclic. This is because the sub-hypergraph $\{(F \text{\_Name}, F \text{\_Major}, F \text{\_Address, Dept}), \{\text{Course \_Name, Grader, F \_Name}\}, \{\text{Course \_Name, Room \_No, Time, Dept}\}\}$ is cyclic since its GYO reduction is non-empty.

Notice that this first definition is attractive since it guarantees that every sub-hypergraph of a $\beta$-acyclic hypergraph is $\alpha$-acyclic. However, since there are an exponential number of sub-hypergraphs, this definition does not immediately help to derive an efficient detection algorithm.

An equivalent definition for $\beta$-cyclicity identifies a new configuration called a $\beta$-cycle.

**Definition 4.4** A hypergraph $H$ is $\beta$-cyclic if and only if $H$ has a $\beta$-cycle.

Before the notion of a $\beta$-cycle is defined, the notion of a pure cycle must be defined. Hyperedges that share common nodes are called neighboring hyperedges. Let $S_1, S_2, \ldots, S_m$ be a sequence of neighboring hyperedges. (The last hyperedge $S_m$ and the first hyperedge $S_1$ are neighbors.) The value of $m$ should be greater than or equal to 3 to form a cycle. In such a sequence, if every vertex in the hypergraph is shared by at most two hyperedges then the sequence is called a pure cycle. In other words, nodes in the neighboring hyperedges of a pure cycle can be either isolated or shared only by two hyperedges. Figure 4.2 shows a pure cycle with 5 hyperedges.
A $\beta$-cycle is similar to a pure cycle. Consider a sequence $S_1, S_2, \ldots, S_m$ of edges. Exclude all the common nodes $X$ from $S_1, S_2, \ldots, S_m$, where $X = S_1 \cap S_2 \cap \ldots S_m$. Let the new sequence after excluding $X$ from $S_1$ through $S_m$ be $S_1', S_2', \ldots, S_m'$. If the new sequence $S_1', S_2', \ldots, S_m'$ forms a pure cycle then our sequence $S_1, S_2, \ldots, S_m$ is a $\beta$-cycle. Put more simply, a $\beta$-cycle is a pure cycle after excluding common nodes in all the hyperedges that occur in the sequence. Notice that, a pure cycle is a $\beta$-cycle where $X = \emptyset$.

Fagin’s fourth definition for a $\beta$-cycle involves a weak $\beta$-cycle. Fagin defines a weak $\beta$-cycle in a hypergraph $H$ as follows.

**Definition 4.5** A weak $\beta$-cycle in a hypergraph $H$ is a sequence $(S_1, x_1, S_2, x_2, \ldots, S_n, x_n, S_{n+1})$ such that

1. $\{x_1, \ldots, x_n\}$ are distinct nodes of $H$,
2. $\{S_1, \ldots, S_n\}$ are distinct hyperedges of $H$, and $S_{n+1} = S_1$,
3. $m \geq 3$, and
4. $x_i$ is in $S_i$ and $S_{i+1}(1 \leq i \leq m)$ and in no other $S_j$.

Therefore in a weak $\beta$-cycle that has $m$ hyperedges and $(m + n)$ nodes, according to rule 4, $m$ shared nodes should occur only in $S_i$ and $S_{i+1}(1 \leq i \leq m)$ and in no other $S_j$. There is no restriction for the other $n$ nodes. Thus a weak $\beta$-cycle can have other nodes apart from the special $m$ shared nodes that could be shared by two or
more hyperedges. The sequence \((S_1, A, S_2, B, S_3, C, S_4, D, S_1)\) in Figure 4.3 is a weak \(\beta\)-cycle. Note that the node \(E\) occurs in more than 2 hyperedges \((S_1, S_2\) and \(S_3)\).

**Definition 4.6** A hypergraph \(H\) is \(\beta\)-cyclic if \(H\) has a weak-\(\beta\)-cycle.

For other definitions and their equivalence, please consult Fagin’s original paper (Fagin 1983). The hypergraph \(H\) shown in Figure 4.4 is \(\beta\)-acyclic.

![Figure 4.3: Weak-\(\beta\) Cycle](image)

4.2.1 An Algorithm to Detect \(\beta\)-acyclicity

Fagin’s \(\beta\)-cycle algorithm tests the hypergraph \(H\) for a \(\beta\)-cycle. Fagin approaches the problem by examining all triples of edges in the hypergraph. All possible hyper-edge triples in the hypergraph are checked for a “\(\beta\)-cycle start.” If a triple satisfies
the $\beta$-cycle start condition, it is further analyzed using connected components to see if the $\beta$-cycle is completed.

Consider a $\beta$-cycle $(S_1, S_2, \ldots, S_m)$. Let $X = \bigcap_{i=1}^{m} S_i$. It can be shown that $X = S_1 \cap S_2 \cap S_3$ as follows. Assume that $S_1 \ldots S_n$ is a $\beta$-cycle. Let $X = S_1 \cap S_2 \ldots S_n$. Let $X' = S_1 \cap S_2 \cap S_3$. It suffices to prove that $X = X'$ by showing subsets in both directions. It is easy to see that $X$ is a subset of $X'$. The reverse subset is harder to see. So, assume that $X'$ is not a subset of $X$, i.e., that there exists a common element of $x$ in $S_1, S_2, S_3$ that is not in $X$. In this case, $x$ is in $S_1 \cap S_2$ and $S_2 \cap S_3$, but not in $X$. Hence, $S_1 - X, S_2 - X, \ldots, S_n - X$ is not a pure cycle. This is a contradiction to the fact the $S_1 \ldots S_n$ is a $\beta$-cycle. Hence, $X'$ is a subset of $X$. It follows that $X = X'$.

Let $S_i, S_j, S_k$ be a triple of edges in the hypergraph $H$. Then, $S_i, S_j, S_k$ are said to be a $\beta$-cycle start if $X = S_i \cap S_j \cap S_k$, $(S_i - X) \cap (S_j - X) \neq \emptyset$ and $(S_j - X) \cap (S_k - X) \neq \emptyset$. For each $\beta$-cycle start triple $S_i, S_j, S_k$, let $X = S_i \cap S_j \cap S_k$. With this $X$, Fagin's algorithm constructs a hypergraph from the edges

$$T' = \{e - X|e \in E, (e = S_i) \text{ or } (e = S_k) \text{ or } (X \subset e \text{ and } e \cap (S_j - X) = \emptyset)\}.$$  

As Fagin showed, $S_i, S_j, S_k$ begins a $\beta$-cycle in $H$ if and only if $S_i - X$ is connected (via a sequence of hyperedges) to $S_k - X$ in the hypergraph formed by $T'$. Hence, Fagin’s algorithm can be implemented via a depth first search for each $T'$ for all possible $\beta$-cycle start triples.

### 4.3 $\gamma$-Acyclicity

This section explains about the next degree of acyclicity in the sequence, which is $\gamma$-acyclicity. Database schema that are $\gamma$-acyclicity have many advantages. For example, Rajaram and Ullman proved that there is a natural outerjoin sequence producing the full disjunction if and only if the set of relation schemes forms a connected, $\gamma$-acyclic hypergraph (Rajaraman and Ullman 1996). As mentioned earlier, in a $\gamma$-acyclic database any two attributes or relations can be joined only in a single way. Since there is only a single way to join relations, data ambiguity is removed. Fagin defined $\gamma$-acyclicity as follows.
**Definition 4.7** A hypergraph $H$ is $\gamma$-acyclic only when it is not $\gamma$-cyclic.

Fagin gives several equivalent definitions for $\gamma$-cyclicity similar to $\beta$-cyclicity. The first one defines $\gamma$-cyclicity in terms of a $\gamma$-cycle, i.e., a hypergraph $H$ is $\gamma$-cyclic if and only if it has a $\gamma$-cycle.

**Definition 4.8** Fagin (1983) A $\gamma$-cycle in a hypergraph $H$ is a sequence of edges $S_1, \ldots, S_{n+1}$ with $S_1 = S_{n+1}$ and a corresponding sequence of distinct vertices $y_1, \ldots, y_n$ such that

1. The first $n$ edges are distinct, i.e., $\{S_1, \ldots, S_n\}$ are distinct hyperedges in $H$,
2. each $y_i$ is in $S_i \cap S_{i+1}$ ($1 \leq i \leq n$),
3. if $1 \leq i < n$, then $y_i$ is not in any $S_j$ except $S_i$ and $S_{i+1}$,
4. and, finally, $n$ is at least 3.

From the above definition we can infer that in a $\gamma$-cycle only one node ($y_n$) out of the $m$ shared nodes can be contained in more than two hyperedges. Figure 4.5 shows a $\gamma$-cycle in which node $A$ occurs in 3 hyperedges. Nodes $B$ and $C$ are shared by 2 hyperedges and $D$, $E$, and $F$ are isolated nodes. The hypergraph shown in Figure 4.4 is $\beta$-acyclic but $\gamma$-cyclic. This hypergraph is $\gamma$-cyclic since $\{\{\text{F\_Name}, \text{F\_Major}, \text{F\_Address, Dept}\}, \text{Dept}, \{\text{F\_Name, Course\_Name, Dept}\}, \text{Course\_Name, } \{\text{Course\_Name, Grader, F\_Name}\}, \text{F\_Name, } \{\text{F\_Name, F\_Major, F\_Address, Dept}\}\}$ forms a $\gamma$-cycle. Note that the node $\{\text{F\_Name}\}$ occurs in more than two hyperedges that participate in $\gamma$-cycle. Also, $\{\text{F\_Name, room, Time}\}$ relation can be obtained in more than one way. Both $\{\text{Course\_Name, Grader, F\_Name}\} \Join \{\text{Course\_Name, Room\_No, Time}\}$ and $\{\text{F\_Name, Course\_Name, Dept}\} \Join \{\text{Course\_Name, Room\_No, Time}\}$ can answer the query to list faculty names, lecture room and time details. As there is more than one way to answer the same query, there is a possibility for data ambiguity. Also, while constructing the query, the user must choose the right path to yield the correct answer.
The hypergraph $H$ shown in Figure 4.6 is $\gamma$-acyclic. Notice that any two attributes in $H$ can be related only in a single way. As there is no ambiguity involved, in $\gamma$-acyclic databases it is possible to use a high level query where the user is not required to specify which relations must be joined to obtain the answer one desires.

### 4.3.1 An Algorithm to Detect $\gamma$-acyclicity

Next, an algorithm to detect $\gamma$-acyclicity is described. The original algorithm is due to D’Atri and Moscarini (D’Atri and Moscarini 1982), but it is also described in Fagin’s work (Fagin 1983).

The algorithm to detect whether a hypergraph is $\gamma$-acyclic is very similar to the GYO reduction process. A hypergraph is determined to be $\gamma$-acyclic if the repeated application of the following five reduction rules results in an empty hypergraph. Here, we use $V$ for the set of vertices in the hypergraph and $E$ for the set of edges.
1. **Rule 1: Isolated Node Deletion**: If a vertex \( v \in V \) belongs to only one hyperedge then delete that node.

2. **Rule 2: Singleton Edge Deletion**: If an hyperedge \( e \in E \) contains only one node \( v \in V \) then delete that hyperedge \( e \). Do not delete \( v \) from other edge occurrences.

3. **Rule 3: Equivalent Edge Node Deletion**: If two nodes occur together in the same hyperedges they are said to be edge equivalent, and one of these vertices may be deleted. For example, if we have two hyperedges \( \{A, B, C, D\} \) and \( \{A, B, F, Q\} \), then either \( A \) or \( B \) can be deleted.

4. **Rule 4: Empty Edge Deletion**: Delete any empty hyperedges that result from previous applications of rules.

5. **Rule 5: Repeated Edges**: If there are two equivalent hyperedges, delete any one of the hyperedges.

Notice that the application of rules 1-5 always either deletes an edge or deletes a vertex from the graph. If the application of rules 1-5 does not delete an edge or a vertex, then repeated applications of these rules will never delete any additional edges or vertices. Hence, this process will halt at some point. Furthermore, this process must end after at most \( O(|V| + |E|) \) applications of the reduction rules.

The algorithm applies the 5 rules to the source hypergraph \( H \) to form a reduced hypergraph \( H' \). After an iteration, if \( H' = \emptyset \) or \( H = H' \), the algorithm terminates, otherwise another iteration is applied with \( H' \) as its source. Similar to the GYO reduction implementation, this algorithm is also implemented by first pre-processing the graph to count, for every vertex \( v \in V \), the total number of edges in which \( v \) occurs.

The first two rules are very straightforward to implement. Rule 1 is implemented by detecting isolated nodes using the pre-processed vertex incidence table. The second rule to find singleton edges is implemented by checking the size of every hyperedge set. The third rule is implemented by checking every pair of vertices for edge equivalency. Let us denote the vertex pair by \( P_{ij} = \{v_i, v_j\} \), where \( v \in V \). The intersection result
of $P_{ij}$ with every hyperedge $e \in E$ is found. If $P_{ij} \cap e = P_{ij}$ or $P_{ij} \cap e = \emptyset$ for every $e \in E$, then the vertex pair in $P_{ij}$ are edge equivalent. When two nodes $\{v_i, v_j\}$ are edge equivalent, the first node in the pair $v_i$ is deleted from all occurring hyperedges. The final two rules will be automatically applied in the set based algorithm. If the end result is an empty hypergraph, then the original hypergraph is $\gamma$-acyclic; otherwise it is $\gamma$-cyclic.

Let us apply the above algorithm to the $\gamma$-cyclic example in Figure 4.4. As you can see, hypergraph $H$ in the example consists of following four relations:

$$
\begin{align*}
&\text{F\_Name} \quad \text{F\_Major} \quad \text{F\_Address} \quad \text{Dept} \\
&\text{F\_Name} \quad \text{Grader} \quad \text{Course\_Name} \\
&\text{F\_Name} \quad \text{Dept} \quad \text{Course\_Name} \\
&\text{Course\_Name} \quad \text{Room\_No} \quad \text{Time}
\end{align*}
$$

The isolated nodes in $H$ (F\_major, F\_Address, Grader, Room\_No, Time) are deleted by Rule 1. This results in the following reduced relations.

$$
\begin{align*}
&\text{F\_Name} \quad \text{Dept} \\
&\text{F\_Name} \quad \text{Course\_Name} \\
&\text{F\_Name} \quad \text{Dept} \quad \text{Course\_Name} \\
&\text{Course\_Name}
\end{align*}
$$

By Rule 2, the singleton edge $\{\text{Course\_Name}\}$ is deleted from the above reduced hypergraph.

$$
\begin{align*}
&\text{F\_Name} \quad \text{Dept} \\
&\text{F\_Name} \quad \text{Course\_Name} \\
&\text{F\_Name} \quad \text{Dept} \quad \text{Course\_Name}
\end{align*}
$$

In the above listed relations, no two hyperedges are edge equivalent. None of the rules can be applied anymore and thus the algorithm terminates with a non empty reduction result. Therefore, hypergraph $H$ in Figure 4.4 is $\gamma$-cyclic.

Let us apply the same algorithm to our $\gamma$-acyclic example in Figure 4.6.
By applying Rule 1, all isolated nodes are removed which leads to the following two relations.

F_Name Dept
F_Name Dept

By applying Rule 5, as both the relations are alike, one hyperedge is removed.

F_Name Dept

Again by Rule 1, both F_Name and Dept are removed from the hypergraph. As the end reduction result is empty, the hypergraph in Figure 4.6 is $\gamma$-acyclic.

### 4.4 Berge-Acyclicity

This section explains about the last degree of acyclicity called Berge-acyclicity. Similar to the $\beta$ and $\gamma$ acyclic definitions, the definition of Berge-acyclicity also involves its counterpart the Berge-cycle. The definition for Berge-acyclicity is as follows.

**Definition 4.9** A hypergraph $H$ is Berge-cyclic if and only if it has a Berge cycle; otherwise, it is Berge-acyclic.

Now, we define a Berge-cycle following the notation of Fagin (Fagin 1983).

**Definition 4.10** A Berge cycle in a hypergraph $H = (V, E)$ is a sequence $S_1, \ldots, S_{n+1}$ of hyperedges where $S_1 = S_{n+1}$ and a sequence $y_1, \ldots, y_n$ of distinct vertices such that

1. $\{S_1, \ldots, S_n\}$ are distinct hyperedges elements of $E$,

2. $n \geq 2$, and

3. $x_i$ is in $S_i$ and $S_{i+1}(1 \leq i \leq m)$.
From the above definition it can be inferred that, since the value of \( n \) can be equal to 2, a hypergraph with some hyperedges that share more than one node satisfies the Berge-cycle condition. In other words, in a Berge-acyclic hypergraph \( H \) no two hyperedges share more than one node. The hypergraph in Figure 4.6 is Berge-cyclic as it has the following Berge-cycle: \( \{\text{F_Name, F Major, F Address, Dept}\}, \text{Dept}, \{\text{F_Name, Course_Name, Dept, Room_no, Time, Grader}\}, \text{F_Name}, \{\text{F_Name, F Major, F Address, Dept}\} \). Notice that every \( \gamma \)-cycle is a Berge-cycle but the converse is not always true. The definitions of \( \gamma \) and Berge-cycles differ in their minimum value of \( n \) (\( n \geq 2 \) in Berge-cycle and \( n \geq 3 \) in \( \gamma \)-cycle). We note that database designers consider the Berge-acyclic condition to be too restrictive to model real life entities.

4.4.1 An Algorithm to Detect Berge-Acyclicity

Next, an algorithm to detect Berge-acyclicity is described. The Berge-acyclic condition can be detected by checking that no two hyperedges in a \( \gamma \)-acyclic hypergraph \( H \) share more than one node. This can be easily done by first finding out all unique pairs of hyperedges in \( H \). If there are \( n \) hyperedges in \( H \), then the number of pairs would be \( n \times (n - 1)/2 \). Each of these \( n \times (n - 1)/2 \) pairs \((e_i, e_j)\) are examined for their intersection set \( X_{ij} = (e_i \cap e_j) \). If any of the \( X_{ij} \) contains two or more nodes, then \( H \) is Berge-cyclic; otherwise \( H \) is Berge-acyclic.

4.5 Normalization and Acyclicity

To avoid confusion, the connection between normalization and acyclicity is presented. Normalization process removes cycles from the relational schema created by the primary keys. However, when the cycle involves non-primary keys, it is possible for a normalized relational schema to be cyclic. The following is an example database schema that is at least in 3\(^{rd}\) normal form. The primary keys in each relation are underlined. Figure 4.7 shows the hypergraph \( H \) associated with the given database schema. Notice that, the hypergraph \( H \) is cyclic.
When a hyperedge \( \{ \text{F\_Name}, \text{Course\_Name}, \text{Dept} \} \) is added to \( H \), hypergraph \( H \) becomes \( \alpha \)-acyclic while remaining normalized. The \( \alpha \)-acyclic hypergraph is shown in Figure 4.1. Hence, even though a relational database schema is normalized, it could be acyclic or cyclic. There is no correlation between acyclicity and normalization.
Chapter 5

Database Analysis Tools For MySQL

In the previous chapters, hypergraphs, hypergraph decompositions, acyclic hypergraphs, various degrees of acyclicity and algorithms to detect them were explained. In this chapter, software tools that exploit hypergraph properties associated with database schema and query structures are described. For this thesis, two pieces of software were implemented that, in combination, provide a variety of services. Briefly, the following database and query analysis software tools were developed.

1. A database to relation file converter: – `dbextractor.sh`. The database to relation file converter is a shell script called `dbextractor.sh` that connects to a MySQL server, extracts details about all the database schema, and writes each database schema information to a separate file called a ”relation file”.

2. A database analyzer and query optimizer: – `AnalyzeDB`. The program `AnalyzeDB` is a database analyzer and query optimizer that performs a variety of functions, including the following.

   (a) A relation file to hypergraph converter: This part of `AnalyzeDB` converts the database schema in the relation file format to a hypergraph file format.
(b) **A hypergraph analyzer:** This portion of AnalyzeDB is the most important part of the software. The hypergraph analyzer accepts numeric hypergraph input and analyzes the hypergraph to test if it is acyclic. If the input hypergraph is acyclic, its hypertree decomposition is constructed and its exact degree of acyclicity is found using the algorithms described in the previous chapters.

(c) **A SQL query parser:** AnalyzeDB contains a SQL query parser that can parse basic SQL queries. The parser determines if the SQL query is syntactically correct in the sense that the arrangement of SQL keywords and variable names matches the SQL grammar and that the relation names and column names are correct.

(d) **A high level query to normal query converter:** Finally, AnalyzeDB contains a high level query converter. As explained in the previous chapter, databases that have $\gamma$ and Berge-acyclic hypergraph representations can support a high level query where the user can omit details on how to join relations. This section of the tool accepts such high level meta queries and converts them to standard queries that can be processed by most of the available commercial query parsers.

In this chapter, further details concerning input, output and implementation for the software tools listed above are provided.

### 5.1 A Database to Relation File Converter

This section begins by explaining the individual software tools that are listed above. The first tool, a database to relation file converter, is implemented using a shell script. The shell script, known as "dbextractor.sh", connects to the MySQL server using normal user permissions and executes the "show database" command to get the list of databases available. The script then connects to all available databases one-by-one and executes the "show tables" command that describes all the table information such as their names, their attribute names, and attribute data types.
Table names and attributes names are parsed and written to a text file which is
referred to as a "relation file". The details for every database are written in a sep-
arate relation file. For example, if there are 3 databases such as University, Test,
and Company, in the MySQL database server their relation text files would be Data-
base_University_Rel.txt, Database_Test_Rel.txt, and Database_Company_Rel.txt. It
should be noted that every MySQL database server has its own repository tables
stored in a database called mysql. This system generated database is omitted when
creating the relation file. After creating the relation files, the shell script closes the
MySQL connection. The following list is the relation file extracted by dbextractor.sh
for the database shown in Table 4.2:

----
faculty
Field
  f_name
  f_major
  f_address
  dept
----
course
Field
  course_name
  room_no
  time
  dept
----
faculty_course
Field
  f_name
  grader
  course_name
----
faculty_course_dept
Field
  f_name
  course_name
  dept
----
EOF
5.2 The Relation File to Hypergraph Converter

In AnalyzeDB, the database schema given by a relation file is converted to a hypergraph. As shown in the introduction, this conversion is very straightforward. To enable efficient coding, however, the table names and attribute names in string format are converted to numerical values. Every attribute is assigned a unique number starting at 1. In a hypergraph, attributes can occur in more than one hyperedge. Such attributes are assigned the same number in all hyperedges.

This part of AnalyzeDB is demonstrated with an input-output example. For the relation file input shown in the previous section, the hypergraph output would be as follows: \(\{1,2,3,4\} \{5,6,7,4\} \{1,8,5\} \{1,5,4\}\). Notice that attributes in every relation are grouped together using the curly brackets. In the above hypergraph, 1 corresponds to F_Name, 2 to F_Major, 3 to F_Address, 4 to Dept, 5 to Course_Name, 6 to Room_No, 7 to Time, and 8 to Grader. Observe that in the final relation \(\{\text{F\_Name, Course\_Name, Dept}\}\), since all the attributes already appear in earlier relations, no new attribute number is generated.

5.3 Hypergraph Analyzer

The hypergraph analyzer in AnalyzeDB examines the hypergraph to test whether it is acyclic. If the hypergraph is acyclic, its hypertree decomposition is constructed and its precise degree of acyclicity is found using various algorithms described in the previous chapters. The hypergraph analyzer can take either a hypergraph that represents a database schema or a conjunctive query. As there are two different kinds of inputs, it is necessary to inform AnalyzedDB about the kind of input by using input mode flags. This is detailed further in the software tool program usage section.

5.3.1 Conjunctive Query to Numeric Hypergraph Conversion

As it was mentioned before, the input to the hypergraph analyzer can be either a hypergraph that represents a database schema or a conjunctive query. Database
schema input to the tool were also explained in the previous sections. In the conjunctive query input case, the user is expected to input the conjunctive query in its numerical hypergraph format. Presently this software does not automatically convert the conjunctive query to its numeric hypergraph format. An example for such conversion is given below.

Consider a query over the university schema in Table 1.1 to find list of students that takes courses offered by a department other than the student’s major department. A conjunctive query for above question might be as follows.

\[
\text{Student}(SFN, SLN) \leftarrow \text{Class}(Class, Co1, S1) \land \text{Course}(CN, Co1, F, D1) \land S(SFN, SLN, S1, D2) \]

The above query can be represented in a numeric hypergraph format as \{1,2\} \{3,4,5\} \{6,4,7,8\} \{1,2,5,9\}. The mapping of attributes to numerals is as follows: SFN to 1, SLN to 2, Class to 3, Co1 to 4, S1 to 5, CN to 6, F to 7, D1 to 8, and D2 to 9.

### 5.4 The SQL Query Parser

The SQL query parser inside of AnalyzeDB validates basic SQL queries which are also called "select-project-join" queries. If the SQL query syntax is correct, the parser proceeds to validate its semantics (variable names, etc.) against the database. The parser validates the query by matching the the relation and attribute names in the query against the database schema. The semantic checker also verifies that the columns in the column_list correspond to the table_list, i.e. the column_list cannot have column names that are not in the tables listed in table_list. Also, if any attribute name occurs in more than one relation, an access identifier is required. An access identifier identifies the exact table name to a repeated column name by prefixing the table name to the column name as tablename.columnname. The output of the SQL parser inside of AnalyzeDB is a result that indicates whether the input SQL query is valid or invalid. In the case of semantic errors, the user is notified about the first syntax or semantic error that occurs. The basic SQL query syntax that have been implemented in our parser is as follows.

\[
<\text{query}|\text{subquery}> ::= 
\]
\[
\{ \text{SELECT} \ast | \text{column\_list} \\
\text{FROM} \text{table\_list} \\
[\text{WHERE search\_condition}] \\
[\text{GROUP BY} \text{column\_list}] \\
[\text{HAVING search\_condition}] 
\}
\]

<search\_condition> ::= 
\[
\{ \ [\text{NOT}] \ \{<\text{predicate}> | ( \text{<search\_condition>} ) \} \]
\[
[\{\text{AND} | \text{OR}\} \ [\text{NOT}] \ \{<\text{predicate}> | ( \text{<search\_condition>} ) \} \} \]
\]

<predicate> ::= 
\[
\{ \text{expression} \ {= | <> | != | > | >= | !> | < | <= | !<} \text{ expression} \\
| \text{expression} \ [\text{NOT}] \text{ LIKE} \text{ expression} \\
| \text{expression} \ [\text{NOT}] \text{ BETWEEN} \text{ expression AND expression} \\
| \text{expression} \ [\text{NOT}] \text{ NULL} \\
| \text{expression} \ [\text{NOT}] \text{ IN} \ (\text{subquery} | \text{expression} \ [,...n]) \\
| \text{expression} \ {= | <> | != | > | >= | !> | < | <= | !<} \ \\
{\text{ALL} | \text{SOME} | \text{ANY}} \ (\text{subquery}) \\
| \text{EXISTS} \ (\text{subquery}) 
\}
\]

All keywords like select, from, etc. should be given in upper case. Identifiers such as table names and attributes should start with an upper case letter. We used Lex and Yacc to split the input query into tokens and to find the structure of the query. We used the SQL parser lex/yacc rules developed by Leroy Cain (Cain 2006). On top of Cain’s rules, we built the necessary code that validates the query semantics.

### 5.5 High Level Query to Normal Query Converter

One of the most useful features of AnalyzeDB is the ability to transform a high level query for $\gamma$ and Berge acyclic databases to a normal query. In the introduction, an example for such a transformation was provided. Now, details are given on how such a transformation is implemented. The software maintains a two dimensional link table to store link details between every pair of relations. The link information
is either a numeric relation index or a list of attribute names. If two relations are connected directly by some common attributes, the link information would be a list of those attribute names; otherwise, the first table index in the series of tables that connects those relations is stored. The link table is constructed in a sequential way by first connecting the pair of relations that are directly connected. Then the relations that are connected with a single level of connection (1 intermediate table) are linked. This link process is repeated up to \( n \) levels of connections, where \( n \) is the number of relations in the given database. When a query contains two relations which are not directly connected, the link table for their connection is sought. The link table is recursively searched to find series of tables that would connect those two relations. For the sake of simplicity, queries that have only two relations in the high level query’s table list are transformed.

### 5.6 Software Implementation Details

The MySQL database server is deployed on a Solaris 64 bit OS on a SPARC system. A MySQL standard server version 4.1.9 is used. As mentioned before, Lex and Yacc are used for SQL parsing. The various modules listed before are implemented using C++ and C. Various set operations are implemented using STL classes.

### 5.7 Software Usage

In this section, the software usage is explained. All the above listed modules except the shell script are linked into a single program. The software program is called ”AnalyzeDB”. The command line structure to invoke AnalyzeDB is shown as follows.

\[
\text{AnalyzeDB [-d(r|q)] RelationFile|CQFile [Queryinputfile]}
\]

options : d - debug; r - relation file mode;
q - queryfile mode; r is default if no mode chosen

In the above usage, the input argument ”DatabaseFile” or ”CQFile” is mandatory. AnalyzeDB determines whether the input is ”DatabaseFile” or ”CQFile” through the
mode bit; Mode bit ‘r’ is for the relation file and ‘q’ represents the query file mode. The next argument, ”Queryinputfile” is optional. When ”Queryinputfile” is not provided, AnalyzeDB prompts the user to enter an SQL query. Notice that the Queryinputfile can be provided only in relation file mode since the SQL query parser requires a relation file to perform query validation.

## 5.8 Software Results

This section shows the software analysis results for the familiar database example shown throughout this thesis which is in Table 1.1. The relation file created by dbextractor.sh for this example is as follows.

```plaintext
----
faculty
Field
faculty_fname
faculty_lname
faculty_id
department_id
----
student
Field
std_fname
std_mname
std_lname
std_id
major_dept_id
----
course
Field
course_name
course_id
faculty_id
department_id
----
class
Field
class_id
course_id
```
The results of AnalyzeDB for the above relation file and a query file, which is listed along with the results, are shown below.

Relations and Attributes in the given Database are:
Relation faculty: {faculty_fname,faculty_lname,faculty_id, dept_id }.
    Its Numerical Representation:{1,2,3,4}
Relation student: {std_fname,std_mname,std_lname,std_id, major_dept_id }.
    Its Numerical Representation:{5,6,7,8,9}
Relation course: {course_name,course_id,faculty_id,dept_id }.
    Its Numerical Representation:{10,11,3,4}
Relation class: {class_id,course_id,std_id }.
    Its Numerical Representation:{12,11,8}
Relation dept: {dept_id,dept_name,chair_fname,chair_lname }.
    Its Numerical Representation:{4,13,14,15}
Relation relatives: {person1_fname,person1_lname,person2_fname, person2_lname }.
    Its Numerical Representation:{16,17,18,19}
Hypergraph notation of the given input is
There are 6 Sets
1 2 3 4
5 6 7 8 9
10 11 3 4
Hypergraph is ACYCLIC

Hypertree Decomposition of given acyclic hypergraph is as follows:

```
16 17 18 19
  |____8 11 12
    |_____3 4 10 11
      |_____4 13 14 15
        |_____1 2 3 4
          |_____5 6 7 8 9
```

Hypergraph is also BETA ACYCLIC

Hypergraph is also GAMMA ACYCLIC

Hypergraph is BERGE CYCLIC

Press any key to continue for Query Parsing...

SQL QUERY PARSER:

Usage: SqlParser DatabaseFile [Queryinputfile]

SQL Statement Format: SELECT [ALL|SOME] * | ColumnList
FROM TableList
  [WHERE Searchcondition]
  [GROUP BY ColumnList]
  [HAVING Search Condition]

Keywords are ALL CAPS ( SELECT , FROM etc )
Identifiers like Table name and Column name start with Upper case
Enter exit to quit

SELECT Std_lname , Fac_lname FROM Student , Faculty ;
This is a high level meta query.
Following Where clause will be added to make a regular query:
WHERE faculty.dept_id = course.dept_id
AND course.course_id = class.course_id
AND class.std_id = student.std_id

ALL TABLES AND COLUMNS ARE IN DATABASE
-- Query Number:1 SQL Syntax Correct

SELECT Std_fname , Std_lname , Course_name FROM Student , Course ;
This is a high level meta query.
Following Where clause will be added to make a regular query:
WHERE course.course_id = class.course_id
AND class.std_id = student.std_id

ALL TABLES AND COLUMNS ARE IN DATABASE
-- Query Number:2 SQL Syntax Correct

SELECT Fac_fname, Fac_lname, Chair_fname, Chair_lname
FROM Faculty, Dept;
This is a high level meta query.
Following Where clause will be added to make a regular query:
WHERE dept.dept_id = faculty.dept_id

ALL TABLES AND COLUMNS ARE IN DATABASE
-- Query Number:3 SQL Syntax Correct

SELECT College_name FROM Dept;
Column College_name does not exists in database
-- Query Number:4 SQL Syntax Correct

SELECT Std_id Student;
-- Query Number:5 SQL Syntax WRONG

SELECT Course_name, Faculty.fname, Faculty.lname
FROM Cause, Faculty
Course_id BETWEEN
-- Query Number:6 SQL Syntax WRONG

5.8.1 Conjunctive Query Input

As it is mentioned earlier, the input to AnalyzeDB is either a relation file or a
numeric conjunctive query input. Here, a few input-output examples for conjunctive
query input cases are shown. As mentioned before, the user is required to convert
the conjunctive query to its numeric representation before feeding it to this system.

Input: \{1,3\} \{1,2,4\} \{2,3,4\}
Output: There are 3 Sets
1 3
1 2 4
2 3 4
Hypergraph is CYCLIC
Input: \{1,2,3\} \{3,4,5\} \{5,6,1\} \{1,3,5\}
Output: There are 4 Sets
1 2 3
3 4 5
5 6 1
1 3 5
Hypergraph is ACYCLIC
Hypertree Decomposition of given acyclic hypergraph is as follows:
\[
\begin{array}{c}
3 4 5 \\
|______1 3 5 \\
|______1 2 3 \\
|______1 5 6
\end{array}
\]
Hypergraph is BETA CYCLIC

Input: \{1,2,3,4\}\{5,6,7\}\{8,9,7\}\{9,4,10\}
Output: There are 5 Sets
1 2 3
1 6 5
1 5 4 3
1 3 5
1 3
Hypergraph is ACYCLIC
Hypertree Decomposition of given acyclic hypergraph is as follows:
\[
\begin{array}{c}
1 5 6 \\
|______1 3 5 \\
|______1 3 4 5 \\
|______1 3 \\
|______1 2 3
\end{array}
\]
Hypergraph is also BETA ACYCLIC
Hypergraph is GAMMA CYCLIC
Bibliography


Appendix A

EECS Directory Schema Analysis Report

This appendix provides an analysis, using AnalyzeDB, of the directory relational schema used for the EECS department at Ohio University. The relational database structure used in the EECS department is as follows.

```
BIBTEX_CONFIGURATION
(
CONFIGURATION Memo/Hyperlink,
DATE_VALID_FROM DateTime (Short) (8),
DATE_VALID_TO DateTime (Short) (8),
PERSON_CREATOR Text (20),
PERSON_INVALIDATOR Text (20)
)
```

```
BIBTEX_PUBLICATION
(
PUBLICATION_ID Long Integer (4),
ENTRY_TYPE Text (40),
CITATION_KEY Text (100),
DATE_CREATED DateTime (Short) (8),
DATE_MODIFIED DateTime (Short) (8),
PERSON_CREATOR Text (20),
PERSON_MODIFIER Text (20)
)
```

BUILDING
( 
BUILDING_FAC_KEY Long Integer (4),
BUILDING_SIS_KEY Text (4),
BUILDING_NAME Text (500),
BUILDING_CODE_WEB Text (20),
BUILDING_NAME_WEB Text (500) )

COLLEGE
(
COLLEGE_CODE Text (8),
COLLEGE_NAME Text (200),
DATE_VALID_FROM DateTime (Short) (8),
DATE_VALID_TO DateTime (Short) (8),
PERSON_CREATOR Text (20),
PERSON_INVALIDATOR Text (20) )

CONTACT_TYPE
(
CONTACT_TYPE_CODE Text (2),
CONTACT_TYPE Text (40),
DATE_VALID_FROM DateTime (Short) (8),
DATE_VALID_TO DateTime (Short) (8),
PERSON_CREATOR Text (20),
PERSON_INVALIDATOR Text (20) )

DEPARTMENT
(
DEPARTMENT_ID Long Integer (4),
DEPARTMENT_CODE Text (16),
DEPARTMENT_NAME Text (200),
DEPARTMENT_TYPE_CODE Text (8),
COLLEGE_CODE Text (8),
WEBSITE Text (510),
EMAIL Text (200),
DATE_VALID_FROM DateTime (Short) (8),
DATE_VALID_TO DateTime (Short) (8),
PERSON_CREATOR Text (20),
PERSON_INVALIDATOR Text (20) )
DEPARTMENT_CONTACT_CAMPUS
(
  DEPARTMENT_ID Long Integer (4),
  CONTACT_ORDER Long Integer (4),
  CONTACT_TYPE_CODE Text (2),
  BUILDING_FAC_KEY Long Integer (4),
  OFFICE Text (20),
  PHONE Text (20),
  FAX Text (20)
)

DEPARTMENT_GROUP
(
  DEPARTMENT_ID Long Integer (4),
  GROUP_CODE Text (20),
  GROUP_NAME Text (500),
  DESCRIPTION Memo/Hyperlink,
  DATE_VALID_FROM DateTime (Short) (8),
  DATE_VALID_TO DateTime (Short) (8),
  PERSON_CREATOR Text (20),
  PERSON_INVALIDATOR Text (20)
)

DEPARTMENT_PUBLICATION
(
  DEPARTMENT_ID Long Integer (4),
  PUBLICATION_ID Long Integer (4),
  DATE_VALID_FROM DateTime (Short) (8),
  DATE_VALID_TO DateTime (Short) (8),
  PERSON_CREATOR Text (20),
  PERSON_INVALIDATOR Text (20)
)

DEPARTMENT_TYPE
(
  DEPARTMENT_TYPE_CODE Text (100),
  DEPARTMENT_TYPE Text (100),
  DATE_VALID_FROM DateTime (Short) (8),
  DATE_VALID_TO DateTime (Short) (8),
  PERSON_CREATOR Text (20),
  PERSON_INVALIDATOR Text (20)
TABLE PERSON
(
PERSON_CODE Text (20),
FIRST_NAME Text (60),
MIDDLE_NAME Text (60),
LAST_NAME Text (80),
NAME_PREFIX Text (40),
NAME_SUFFIX Text (40),
DEGREE_SUFFIX Text (100),
TITLE Text (500),
WEBSITE Text (500),
EMAIL Text (500),
PERSON_TYPE_CODE Text (16),
DATE_VALID_FROM DateTime (Short) (8),
DATE_VALID_TO DateTime (Short) (8),
PERSON_CREATOR Text (20),
PERSON_INVALIDATOR Text
)

PERSON_CONTACT_CAMPUS
(
PERSON_CODE Text (20),
CONTACT_ORDER Long Integer (4),
CONTACT_TYPE_CODE Text (2),
BUILDING_FAC_KEY Long Integer (4),
OFFICE Text (20),
PHONE Text (20),
FAX Text (20)
)

PERSON_CONTACT_EXTERNAL
(
PERSON_CODE Text (20),
CONTACT_ORDER Long Integer (4),
CONTACT_TYPE_CODE Text (2),
ADDRESS_LINE1 Text (500),
ADDRESS_LINE2 Text (500),
ADDRESS_LINE3 Text (500),
ADDRESS_LINE4 Text (500),
CITY Text (500),
STATE_CODE Text (4),
ZIP_CODE Text (20),
PHONE Text (20),
FAX Text (20)
)

PERSON_DEPARTMENT
(
PERSON_CODE Text (20),
DEPARTMENT_ID Long Integer (4),
TITLE Text (200),
DATE_VALID_FROM DateTime (Short) (8),
DATE_VALID_TO DateTime (Short) (8),
PERSON_CREATOR Text (20),
PERSON_INVALIDATOR Text (20)
)

PERSON_DEPARTMENT_GROUP
(
PERSON_CODE Text (20),
DEPARTMENT_ID Long Integer (4),
GROUP_CODE Text (20),
DATE_VALID_FROM DateTime (Short) (8),
DATE_VALID_TO DateTime (Short) (8),
PERSON_CREATOR Text (20),
PERSON_INVALIDATOR Text (20)
)

PERSON_NAME_PREFIX
(
PREFIX_CODE Text (20)
)

PERSON_PUBLICATION
(
PERSON_CODE Text (20),
PUBLICATION_ID Long Integer (4),
DATE_VALID_FROM DateTime (Short) (8),
DATE_VALID_TO DateTime (Short) (8),
PERSON_CREATOR Text (20),
PERSON_INVALIDATOR Text (20)
)
PERSON_RESEARCH
(
PERSON_CODE Text (20),
RESEARCH Memo/Hyperlink,
DATE_VALID_FROM DateTime (Short) (8),
DATE_VALID_TO DateTime (Short) (8),
PERSON_CREATOR Text (20),
PERSON_INVALIDATOR Text (20)
)

PERSON_TYPE
(
PERSON_TYPE_CODE Text (16),
DATE_VALID_FROM DateTime (Short) (8),
DATE_VALID_TO DateTime (Short) (8),
PERSON_CREATOR Text (20),
PERSON_INVALIDATOR Text (20)
)

PUBLICATION
(
PUBLICATION_ID Long Integer (4),
PUBLICATION_TYPE_CODE Text (4),
TITLE Memo/Hyperlink,
AUTHORS Memo/Hyperlink,
VENUE Memo/Hyperlink,
PUB_MONTH Long Integer (4),
PUB_YEAR Text (8),
NOTES Memo/Hyperlink,
DATE_VALID_FROM DateTime (Short) (8),
DATE_VALID_TO DateTime (Short) (8),
PERSON_CREATOR Text (20),
PERSON_INVALIDATOR Text (20)
)

SIS_COLLEGE
(
SIS_SCHOOL_CODE Text (8),
COLLEGE_CODE Text (8)
)
SIS_DEPARTMENT
(
SIS_DEPT_CODE Text (8),
DEPARTMENT_CODE Text (16),
COLLEGE_CODE Text (8)
)

BIBTEX_PUBLICATION_FIELD
(
PUBLICATION_ID Long Integer (4),
FIELD_NAME Text (40),
FIELD_TEXT Memo/Hyperlink
)

DEPARTMENT_RESEARCH_TOPIC
(
DEPARTMENT_ID Long Integer (4),
RESEARCH_TOPIC_ORDER Long Integer (4),
RESEARCH_TOPIC Text (500),
DATE_VALID_FROM DateTime (Short) (8),
DATE_VALID_TO DateTime (Short) (8),
PERSON_CREATOR Text (20),
PERSON_INVALIDATOR Text (20)
)

PERSON_BIOGRAPHY
(
PERSON_CODE Text (20),
BIOGRAPHY Memo/Hyperlink,
DATE_VALID_FROM DateTime (Short) (8),
DATE_VALID_TO DateTime (Short) (8),
PERSON_CREATOR Text (20),
PERSON_INVALIDATOR Text (20)
)

PERSON_RESEARCH_TOPIC
(
PERSON_CODE Text (20),
RESEARCH_TOPIC_ORDER Long Integer (4),
RESEARCH_TOPIC Text (500),
DATE_VALID_FROM DateTime (Short) (8),
DATE_VALID_TO DateTime (Short) (8),
Most of the relations have common fields such as DATE_VALID_FROM, DATE_VALID_TO, PERSON_CREATOR, PERSON_INVALIDATOR, PHONE, FAX, TITLE, WEBPAGE. These fields are required to validate data in individual relations, and they are not meant to be used to join relations. In order to avoid automatic natural joins on these columns, they were changed to unique names by adding a unique numerical value to the end of each column name like DATE_VALID_FROM1, DATE_VALID_TO1, PERSON_CREATOR1, PERSON_INVALIDATOR1, etc.

The AnalyzeDB software shows that this schema is cyclic. Three cycles have been identified that make this schema cyclic. Those three cycles are listed as follows.

Cycle 1:

Department_Publication:
{DEPARTMENT_ID,PUBLICATION_ID,DATE_VALID_FROM7, DATE_VALID_TO7,PERSON_CREATOR7,PERSON_INVALIDATOR7 }

Person_Publication:
{PERSON_CODE,PUBLICATION_ID,DATE_VALID_FROM12, DATE_VALID_TO12,PERSON_CREATOR12,PERSON_INVALIDATOR12 }

Person_Department_Group:
{PERSON_CODE,DEPARTMENT_ID,GROUP_CODE,DATE_VALID_FROM11, DATE_VALID_TO11,PERSON_CREATOR11,PERSON_INVALIDATOR11 }

Cycle 2:

Person_Department_Group:
{PERSON_CODE,DEPARTMENT_ID,GROUP_CODE,DATE_VALID_FROM11, DATE_VALID_TO11,PERSON_CREATOR11,PERSON_INVALIDATOR11 }

PERSON_RESEARCH_TOPIC:
{PERSON_CODE,RESEARCH_TOPIC_ORDER, RESEARCH_TOPIC,DATE_VALID_FROM18,DATE_VALID_TO18,PERSON_CREATOR18, PERSON_INVALIDATOR18 }
DEPARTMENT_RESEARCH_TOPIC : {DEPARTMENT_ID, RESEARCH_TOPIC_ORDER, RESEARCH_TOPIC, DATE_VALID_FROM16, DATE_VALID_TO16, PERSON_CREATOR16, PERSON_INVALIDATOR16 }

Cycle 3:
--------

Department_Publication :
{DEPARTMENT_ID, PUBLICATION_ID, DATE_VALID_FROM7, DATE_VALID_TO7, PERSON_CREATOR7, PERSON_INVALIDATOR7 }

Person_Publication :
{PERSON_CODE, PUBLICATION_ID, DATE_VALID_FROM12, DATE_VALID_TO12, PERSON_CREATOR12, PERSON_INVALIDATOR12 }

PERSON_RESEARCH_TOPIC : {PERSON_CODE, RESEARCH_TOPIC_ORDER, RESEARCH_TOPIC, DATE_VALID_FROM18, DATE_VALID_TO18, PERSON_CREATOR18, PERSON_INVALIDATOR18 }

DEPARTMENT_RESEARCH_TOPIC : {DEPARTMENT_ID, RESEARCH_TOPIC_ORDER, RESEARCH_TOPIC, DATE_VALID_FROM16, DATE_VALID_TO16, PERSON_CREATOR16, PERSON_INVALIDATOR16 }