2013

Missing Data in the Relational Model

Marion Morrissett
Virginia Commonwealth University

Dedication

This research is dedicated to content,

data with missing values that represent the always-complete real world.

And to structure,

the relational model created and developed by the scientists, researchers, teachers,

and practitioners who populate my test case database.

MISSING DATA IN THE RELATIONAL MODEL

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

by

MARION R. MORRISSETT
Bachelor of Arts, University of Virginia, 1972
Mathematical Sciences Certificate in Computer Science,
Virginia Commonwealth University, 1987
Master of Science, Virginia Commonwealth University, 1994
Doctor of Philosophy, Virginia Commonwealth University, 2013

Director: LORRAINE M. PARKER
ASSOCIATE PROFESSOR, DEPARTMENT OF COMPUTER SCIENCE

Virginia Commonwealth University
Richmond, Virginia
May, 2013

Acknowledgments

Many people have provided help and support during this work. My friends and family listened to my dissertation status reports, the programmers among them heard the technical details and all were patient. John Cookson, Tom Nicholls, and Paul Bruggeman shared their experience with problems created and solved by computers. Mac Kerfoot, Donnie Bergh, and Shin Adcox were always there to talk about work, networks, problem solving, and life. Susan Campbell, Peter Bacque, Randy Green, John Gibney, and Peter Kohn carried me through this with conversation and emotional support.

My children Melissa and Jeffrey and their mother, Linda followed my progress with interest bringing me joy and purpose. My siblings Mike, Drew, Sydney, Leslie, and Courtney were always there for me, our many points of view and various ideas making us the family that Evelyn and Andy parented with love and grace. Thank you for your faith and support.

The role of the Virginia Commonwealth University School of Engineering, School of Business, and College of Humanities and Sciences faculties is significant. I thank all of those who teach, and especially my dissertation advisory committee for your patience, advice and ideas.

Larry Williams, fellow grad student and database lab partner, thanks for walking through this with me. Dr. Charles A. Bell, VCU's first computer science PhD, contributed his insight, expertise and book on MySQL. Chuck, your example inspired me and your observations reminded me that this would be difficult. I remain inspired, thanks.

Without Dr. Susan S. Brilliant and Dr. Lorraine M. Parker this research would not have been possible. Dr. Brilliant encouraged me to pursue the ideas for my master's thesis, taught me technical writing, and showed me great teaching. Dr. Parker contributed significant ideas to this dissertation with encouragement and accurate criticism, improved my writing by telling me when to leave out unwarranted explanations and when to explain the obvious, and taught me to teach. I cannot thank you both enough.

Dr. Mary Elizabeth Glade, historian, scholar and teacher, my friend and confident who patiently listens when I need to talk and hears what I say. Thank you Betsy; for showing me how to listen and sharing life's grand ideas.

Table of Contents

List of Tables

List of Figures

Abstract

MISSING DATA IN THE RELATIONAL MODEL

By Marion R. Morrissett, Ph.D.

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

Virginia Commonwealth University, 2013

Director: Lorraine M. Parker
Associate Professor, Department of Computer Science

This research provides improved support for missing data in the relational model and relational database systems. There is a need for a systematic method to represent and interpret missing data values in the relational model. A system that processes missing data needs to enable making reasonable decisions when some data values are unknown. The user must be able to understand query results with respect to these decisions. While a number of approaches have been suggested, none have been completely implemented in a relational database system. This research describes a missing data model that works within the relational model, is implemented in MySQL, and was validated by a user feasibility study.

CHAPTER 1 Introduction

1.1 The need to represent missing data

The relational model of data is based on set theory and first-order predicate logic. The relational model provides an intuitive way to view data and serves as a specification for a relational database management system [10]. Relational databases are time-varying collections of relation variables (relvars) that present the user with tables (relations) of data values in columns (attributes) and rows (tuples). A relation is composed of two parts. The first is the heading, which is a set of attributes, their domains, and domain type triples. The heading is a predicate in which the attribute names represent a set of parameters that range over the attribute values of the specified domains. The second part of the relation is its body, which is a set of tuples. Each tuple is a proposition, which is true within the closed world of the database. A tuple of attribute values not present in the database is a false proposition. Thus a query which returns an empty results relation, shows no data that can indicate a true proposition. This mathematical model of data makes it possible to represent real-world objects using a consistent collection of information and to perform operations on this information using relational algebra and relational calculus. [23, pp.67-68]

The real world is always complete and accurate, but data used to construct a representation of a real-world object may not be. The relational model permits data values to be missing [15, p.39]. Data values that exist in the real world, but are not yet available must be represented as missing until known. Data values that do not apply to a real-world object are either inapplicable to the object and missing or the attributes for these data values should be eliminated. Because the latter approach is not always feasible, a distinction among the kinds of missing data is required. If the relational model is to accurately represent real-world objects, an adequate representation of missing data is necessary.

Missing data must not cause a database operation to produce anomalous results. The system may report that an answer is not known within the database, but it must be clear that this is different from a query that returns no data. A model is needed to represent information missing from databases in a way that maximizes completeness of representation, minimizes loss of accuracy, is consistent with the relational model of data, and makes sense to users.

1.2 Unknown values in relational databases

Codd identified data dependence and data inconsistency as two problems in large computer application development environments that could be solved by the relational model of data [10, p.377]. The relational model separates the logical representation of data from its physical storage. This allows the abstraction presented to the user to remain consistent if it is necessary to change the physical structure of data storage. To support data independence, the relational model relies on a programming language neutral data sublanguage to connect application programs to data. This approach shifts responsibility for managing the physical data from the application to the database management system in a way that reduces software development cost and minimizes software maintenance [63, pp.52-53].

The relational model eliminates duplicate data through data normalization and uses constraints to maintain data integrity during databases operations. This approach moves responsibility for managing data consistency from the application to the database management system in a way that allows new data requirements to be met without introducing data redundancy, contradiction, and inconsistency. [10, pp.383-387]

While the relational model solved the problems of data dependence and data inconsistency, the solution created an unexpected problem in the case of missing data. Previously, system analysts identified valid data values and programs verified these values during data input and record update. Data exceptions were processed as necessary. If unknown data values were permitted, the application included programming logic to identify and process each case of missing data. Otherwise, invalid data was rejected and an exception process issued an error message or report. Corrected data could then be entered. When the application was responsible for handling missing data at input it had the logic to interpret the meaning of missing data values. With the shift of responsibility for handling data verification from the application to the DBMS, came the responsibility for interpretation of verified missing data values at output.

Currently the relational model uses nulls to represent missing data values and interprets nulls using 3-valued logic. This approach is similar to the methods used by non-relational database management systems, but it is not fully implemented in any relational database management system. Incomplete information in the relational model is described in section 3.2.

1.3 Research overview

A complete problem statement is presented in chapter 2. Different kinds of missing data are identified in chapter 3. Different theories for handling missing data are discussed in chapter 4. While theoretical solutions are important, solutions implemented in a relational database management system either as an experiment or a commercial product are of special interest. The impact of missing data on relational database management systems and applications is considered in chapter 5. Chapter 6 is a specification for a solution to the missing-data problem. The design, implementation, and verification of this specification is described in chapter 7. Because user understanding is part of any solution, chapter 8 presents the results of a feasibility study conducted with volunteer participants. Chapter 9 summarizes the research and future work is considered in chapter 10.

1.4 Contributions of this research

This research examines how applications and databases share responsibility for data interpretation to develop an improved representation for missing data that better meets the needs of the DBMS and the user.

1.4.1 New model for missing data

The KNOWN/UNKNOWN model is a new model for representing missing data. While an attribute missing data may contain an invalid value or no value, the KNOWN/UNKNOWN approach stores information about what is missing and why as data values in relations. This representation complies with the relational model.

1.4.2 Metadata about missing data

The KNOWN/UNKNOWN model is able to capture information about missing data available at the time and point of data entry, maintain and update this information throughout the data's life cycle, and make it available when and where it is needed. This metadata is a subsystem component for each relational database that implements the model. The DBMS uses the KNOWN/UNKNOWN metadata to make missing data processing decisions. Users can rely on metadata to explain what is missing from query results and why it is missing.

1.4.3 Compatibility with existing databases

The KNOWN/UNKNOWN model is backward compatible with the SQL standard for missing data using nulls and 3-valued logic. The metadata relations may be constructed for an existing database and over time populated with the information needed to make them useful. During and after this transition, queries relying on nulls will continue to return the expected results using 3VL. This provides a straightforward migration path from the SQL standard to the KNOWN/UNKNOWN model.

1.4.4 Support for application and database design

The well-defined metadata component of the KNOWN/UNKNOWN model supports database development during application design. Documented categories for missing data and the capability for expansion encourages planning for missing data. If missing data is needed in the application, the use cases developed during project planning contribute to application validation and may be used in regression testing.

1.4.5 Database metrics for missing data

The data about what is missing and why in the KNOWN/UNKNOWN model provides a way to categorically measure missing data. A set of metrics that measure the amount of missing data in a database can be used as a benchmark of completeness and used as a component of database status reporting. Ad hoc metrics can evaluate the overall condition of an application's data and help improve the methods used to gather data.

CHAPTER 2 Problem

2.1 The problem of incomplete information

The missing data problem is: How should incomplete information be represented in a relational database management system?

There are four categories of incomplete information identified in chapter 3. Data is missing because it is applicable but unknown at this time, the value is unreliable (invalid), it is not applicable to this entity (inapplicable), or it is unknowable for an understood reason such as law or policy.

Applicable, invalid, and unknowable are similar and often require the same processing. In some cases, unknowable data may be processed as if it were inapplicable. If identified during database design, an inapplicable attribute should be eliminated through normalization. After a database is implemented a policy change may require an attribute to be inapplicable for some tuples. In some cases, inapplicable attributes are made applicable, but remain unknown if a data value is missing.

The most common type of missing data may be applicable but unknown [1], but all types of incomplete information, including those not yet identified, must be available to describe missing data in a database.

[1]An investigation of missing data types from the literature in chapter 3 discovered “applicable, but unknown” to be common to all references including those that identify one type of missing data [8][3][50][11][34][14][19][64, pp.13]. If missing data values were typed, it would be possible to count these types and identify the most common reasons for missing data.

2.1.1 What is the problem?

The missing data problem ranges from how to represent missing data in a relational database management system (DBMS) to how the user represents missing data in a specific application design. On one extreme, the DBMS provides an interpretation of missing data values and at the other, interpretation of missing data is the application's responsibility.

The greater the distance between the physical storage of data and the user's level of abstraction, the more a well defined data model is needed to interpret the data's meaning. There should be a way to represent, process, and interpret missing data with responsibility shared so that the DBMS returns results that the user application can interpret correctly. The relational model is a well defined model for complete information, but what is lacking are clear rules to interpret incomplete information.

2.1.2 Why is it a problem?

In the first version of the relational model (RM/V1) [15, p.iv] and the current SQL standard [65, p.24], most of the responsibility for handling missing data is located within the relational model and DBMS, with the user expected to understand nulls and 3-valued logic (3VL). However, the use of nulls and 3VL may be counterintuitive [19, p.233]. This results in an unbalanced division of processing responsibility which the user is not always aware that they have agreed to when using the system. Thus the response to a user query may be incorrect.

In this context, the missing data problem is a database query problem. The DBMS must return correct answers to all queries. To do this, there must be an equivalent accuracy for the case of complete information and for the case of incomplete information. A correct answer is one that is not misleading, anomalous, or confusing from the user's point of view. Failure to do this leaves the missing data problem unsolved.

2.2 Purpose of this research

This research presents a practical solution to the missing data problem that can be implemented in a DBMS to the extent possible and in the realm of the user to the extent necessary.

2.2.1 Why a solution is important

The possibility of an incorrect answer to a query represents DBMS failure to the user. Codd's third rule for fully relational databases requires a systematic treatment of null values as the solution to the missing data problem.

> *Codd's Third Rule* "Null values (distinct from the empty character string or a string of blank characters or any other number) are supported in the fully relational DBMS for representing missing information in a systematic way, independent of data type." [36, p.133]

Existing relational DBMS products do not provide a systematic treatment of null in a way that complies with Codd's third rule. Null is implemented in most DBMS and is used to represent both unknown data and inapplicable attributes, which is confusing. A boolean truth value of “unknown” is not implemented in any DBMS [2]. Null is not evaluated systematically within the DBMS as it is implemented in SQL. For example, built in functions such as COUNT and SUM treat null differently from what might be expected. The aggregate function COUNT returns 0 if its argument is NULL, but SUM returns NULL if its argument is NULL. Date argues both COUNT and SUM should return zero. [21, pp.302-3] While this may be a flaw in SQL and not the DBMS, there are queries that are known to return incorrect results when one or more missing data values are involved in the match criteria and/or the database [34] [19]. For these reasons, a solution to the missing data problem is important if the relational model is to reach its full potential.

2.2.2 What a practical solution must do

A solution must be found within the constraints of the problem space defined by the relational model. A solution to the missing data problem must determine how a relational database management system represents missing data in the logical schema, maps data to physical storage using this schema, uses this representation when processing data and executing database operators to create a correct interpretation, and presents this interpretation to the user interface.

[2]In 2013, a search of the world wide web, product documentation, and experiments using DBMS products indicate that Microsoft SQL Server, Microsoft Access, Oracle, Ingres, PostgresSQL, and MySQL either implement a boolean type capable of no more than 2-valued logic or have no boolean type at all. This requires the user to designate some other data type and a set of values to represent false, true and unknown. (see section 3.2.3)

Backward compatibility with RM/V1 as it is partially implemented in existing DBMS is desirable, but it may be necessary to extend the relational model and/or modify the DBMS. If this is the case, changes should be minimized. No change to the relational model may interfere with the representation of data as relations, data independence, data consistency, or the relational algebra or calculus. Codd created his 12 rules [15, pp.500-501] [36, 99.129-142] as a way to measure compliance with the relational model [15, p.29]. If a change to one of these rules is needed, it should be carefully considered. Extensions to SQL should leave existing statements and operators unchanged and functional.

A data model is sound if it is able to return correct answers, but no incorrect answers. It is complete if it is able to return all correct answers. The DBMS response to each query and the user's understanding of the results must be sound and complete.

2.3 Problem context and solution space

The problem solution space is defined by the relational model of data, the SQL standard as implemented by the MySQL DBMS, and the practical aspects of designing and maintaining databases (i.e. enterprise models and conceptual schema revisions caused by policy changes or new laws). The relational model is to be considered from both the model-theoretic and proof-theoretic points of view within the framework of the closed-world assumption. An extension to the relational model no more intrusive than Codd's solutions to the missing data problem is considered desirable

[11] [12].

2.3.1 Closed world assumption

The closed world assumption must be retained, where each tuple in a relation is a proposition defined by the conjunction of its data values. Each tuple of complete information in a relation is a true proposition. A query that returns an empty results relation represents a false proposition. A tuple that matches search arguments but also has one or more missing data values represents a proposition that is true, but not entirely known. A tuple that matches some search arguments with known data values and may match others with missing data values represents a proposition that may be true. This last type of tuple violates the law of the excluded middle and leads to 3-valued logic.

2.3.2 A truth-bearer that makes sense

The problem with using null to represent missing data is its dependency on 3-valued logic. When the proposition defined by a tuple containing a null is evaluated using 3VL the result is neither true or false, its conjunctive truth-value is *unknown.* Date argues that a logic based on 3VL is not intuitive and should be avoided [19, p.233]. The elimination nulls and 3-valued logic is intended to make a representation of missing data more intuitive to the user. But what is needed is a truth-bearer that can interpret a tuple that is missing data in a way that makes sense to the user.

CHAPTER 3 Types of Missing Data

There are more than one kind of missing data and it is necessary to identify them if they are to be represented accurately. Missing data types have been identified by published papers and standards reports.

3.1 Early investigation

3.1.1 Language Structure Group

In 1962, the Language Structure Group (LSG) of the Development Committee of the Conference on Data Systems Languages (CODASYL) published a report towards developing a theory of data processing. Three concepts at the center of a data processing system were the entity, property, and value. This data processing system model is not the relational model, but has certain similarities. An entity has one or more properties and each property has a single value taken from an associated set of property values. In this model, property value sets function like domains in the relational model and are the source of an entity's property values. At a minimum a property value set must include two values to represent "non-applicable data" using *Omega* (Ω) and "missing data" using *Theta* (Θ). Representing incomplete data with symbols provides consistency and avoids confusion with data values such as spaces or zero. [8, p.191]

3.1.2 ANSI DBMS model

The American National Standards Institute (ANSI) Computers & Information Processing (X3) Standards Planning and Requirements Committee (SPARC) for Data Base Management System (DBMS) created a study group to investigate and document an architecture for database management systems. This report concentrates on interface specifications between system components and user roles. It uses a 3-level DBMS design with an internal schema at the physical layer, a conceptual schema as the logical database design, and an external schema as the user and application interface.

The data administrator creates a conceptual schema of an enterprise that the database is to represent. Domains are defined as components of the conceptual schema and each domain of eligible values has a list of characteristics. The list of domain characteristics includes a name, meaning, type, rules for edit, comparison, or validity, and the "manifestation of null." Missing data classifications identified in the "manifestation of null" in Table 1 are a collection of ideas from the committee and allow for expansion. [3, pp.55-56]

Table 1 identifies several kinds of imperfect information, but not all of these are incomplete information. One classification, "available, but of suspect validity" is uncertain with various explanations as to why the data may not be trustworthy, but the data is not necessarily missing. Other classifications may be combined into one using the notion of data values that are not known or are unknowable for the time being.

Table 1: ANSI/X3/SPARC Manifestation of Null

not valid for this individual
e.g. maiden name of male employee
valid, but does not yet exist for this individual
e.g. married name of female unmarried employee
exists, but not permitted to be logically stored
e.g. religion of this employee
exists, but not knowable for this individual
e.g. last efficiency rating of an employee who worked for another company
exists, but not yet logically stored for this individual
e.g. medical history of newly hired employee
logically stored, but subsequently logically deleted
logically stored, but not yet available
available, but undergoing change (may be no longer valid)
change begun, but new values not yet computed change incomplete, committed values are part new, part old, may be inconsistent change incomplete, but part of new values not yet committed change complete, but new values not yet committed
available, but of suspect validity (unreliable)
possible failure in conceptual data acquisition possible failure in internal data maintenance
available, but invalid
not too bad too bad
secured for this class of conceptual data
secured for this individual object
secure at this time
derived from null conceptual data (any of the above)

3.1.3 CODASYL approach

The CODASYL data model uses an internal literal called null to represent missing data. Programmers are expected to test for this literal and use appropriate logic to process missing data. [50, pp.186-187] This is a simple mechanism using a single value to represent all kinds of missing data.

3.2 Relational model

Although other databases used null to represent missing data, there was no representation for missing data in Codd's first proposal of the relational model [10].

3.2.1 Codd's inclusion of null in the relational model

In 1975 Codd answered questions about the relational model's handling of nulls [11]. One question [1] asked how relational retrieval operations are affected by nulls and another [2] asked about the impact of nulls on arithmetic operations and library functions. Codd answered with an overview of how nulls and 3-valued logic can represent data currently unknown and described how arithmetic results should be determined when values are null.

[1] "Little attention has been given to the treatment of null values in relations when retrieval operations are being executed. Is it not necessary to extend both the relational algebra and the relational calculus to accommodate null values?" [11, p.24]

[2] "How do arithmetic operations and library functions treat this type of null value?" [11, p.28]

3.2.2 RM/V1

In 1979, Codd extended the relational model to increase the ability of relational databases to express the meaning of data. Nulls and 3-valued logic were formally added to the relational model as the solution to the missing-data problem. [12] This is version RM/V1 [15, pp.169-171]. In RM/V1 the interpretation of null was limited to “value at present unknown” [12, p.403].

3.2.3 SQL and null

The use of null to represent missing data and the resulting 3-valued logic are included in the SQL standard [64, pp.13-14]. Although null is implemented in most database systems, few products implement a corresponding boolean data type that includes an *unknown* value to support 3-valued logic as RM/V1 and the SQL standard require [65, p.24].

Microsoft SQL Server 2012 documents a bit data type that takes the values of 0, 1, or NULL with an explanation that string values TRUE and FALSE can be converted to bit-values of 1 and 0 [44]. Microsoft Office Access 2003 uses the same approach as SQL Server [43].

Oracle does not document a boolean data type in its SQL reference [53]. However, on its website Oracle explains that a boolean is not needed because an integer with a function that evaluates 0 or 1 as FALSE or TRUE gives the same results [51].

Ingres accepts literals TRUE or FALSE for boolean columns [2, p.78] and defines a boolean variable using an integer that represents unknown using NULL [2, p.86-88].

PostgresSQL represents a boolean using a byte and allows it to contain literal values *true*, *t*, *yes*, *y*, *on*, or *1* for true, literal values *false*, *f*, *no*, *n*, *off*, or *0* for false, with a NULL meaning unknown [57, p.133].

MySQL has BOOL and BOOLEAN data types which are synonyms for a TINYINT (signed char) with values on [-128,127] [52, pp.798-799]. Boolean literals defined as case insensitive are TRUE as 1 and FALSE as 0 [52, p.726]. A boolean variable that can be NULL may represent a truth-valued 3VL as TRUE, FALSE, and a NULL for unknown.

These DBMS products approach 3-valued logic using some data type that takes on one of two states and maps literals for true and false to these states. If this variable is declared as nullable, unknown is represented by a NULL. SQL users now have another interpretation of null stored for a boolean data type. A stored null can represent either missing data or the unknown boolean state. For example, if a survey asks for a response of true, false or unknown to a question, null meaning unknown is a valid response. In this case, the application needs logic that interprets null as data (i.e. a valid response) rather than missing data.

3.2.4 RM/V2

In 1986, Codd proposed an expanded solution to the problem of missing data and added support for the case where properties were inapplicable to some items in a relation. The term null was deprecated and missing data is identified by marks. The missing data "value at present unknown" (previously null) became an A-mark and "property not applicable to this object" is an I-mark. [14] The relational model now had two kinds of missing data, but not a 4-valued logic (4VL) [14, p.62]. According to Codd, using the A-mark and I-mark in place of null required an evolutionary path for the relational model from RM/V1 to RM/V2 [15]. Part of this path required that users accept responsibility for learning to use the more complex 4VL of RM/V2. So far, the advance of relational DBMS beyond partial implementation of RM/V1 has been halted by inertia and the burden of 4VL.

3.2.5 Date's seven types of null

Date [19] determined that a systematic solution to the missing data problem needed to account for more than one kind of missing data and identified seven distinct types of null.

1. *Value not applicable*

   This data value is not and should not be present. This instance of the attribute should be ignored because it represents a property that its object does not have. For example, the attribute for sales commission does not apply to non-sales people. This is equivalent to Codd's I-mark in RM/V2.

2. *Value unknown*

   This data value is missing. The property represented by the attribute is applicable to the object represented by this tuple. For example, the attribute salary is unknown for a newly hired employee until supplied by the employee's department. This is equivalent to Codd's null in RM/V1 and A-mark in RM/V2.

3. *Value does not exist*

   In this case when the attribute is applicable and the data is not yet known, it does not exist. For example, all employees are expected to have an individual taxpayer identification number as a requirement for employment. If an employee is hired before applying for this identifier, the missing data is applicable, but does not exist. Once the id number is issued, the data value exists and can be known.

4. *Value undefined*

   This is the case when an attribute's data value is assigned by a computation that depends on other data values in other attributes. For as long as a divisor used in this computation is equal to zero, its result is undefined.

5. *Value not valid*

   This is the case when a data value outside of a permissible range might be stored with the expectation of being corrected later. For example, an employee age exceeds the mandatory retirement age. This data is not missing, but it requires exception processing before it can be considered a valid, known, and applicable data value. Either the data is corrected or the data integrity constraint must be revised before a data value can be present for this property.

6. *Value not supplied*

   This is the case when a data value was withheld. It may be anticipated that it will be supplied later. This data is missing and it is unknown. This could be the case where the property is applicable, but the data value is unknown or it may be the case of data not being supplied because it is inapplicable.

7. *Value is the empty set*

   This is the case when a relational operator creates a new relation that is missing one or more data values. For example, an outer join may create tuples with missing attributes. SQL fills in these missing values with nulls, but the correct interpretation of this missing data is an empty set.

3.3 Imperfect information

Zadeh invented fuzzy logic to deal with imperfect information in a way humans would understand using terms such as imprecise, uncertain, incomplete, unreliable, vague, and partially true [73]. The data elements in fuzzy logic are often described as being imprecise, vague, ambiguous, subjective, unclear, uncertain, inconsistent, or incomplete. Definitions for these words and a background search of the published literature on fuzzy logic places the meaning of these terms in context and suggests the kinds of information that are well expressed using fuzzy concepts. Selecting precise definitions, in combination with fuzzy data, provides a set of working definitions that can be used to define fuzzy domains to extend the relational database model. [71] [41] [56]

Ma presents five classifications of imperfect data (i.e. inconsistent, imprecise, vague, uncertain, and ambiguous) for use when modeling fuzzy data [40, p.47]. An analysis of these five classifications and the eight commonly used terms to describe fuzzy data suggests using pairs of descriptive terms where one describes the data and the other the data query. This works well for terms that are fuzzy in the context of imperfect information, but fails for terms that have precise definitions in the relational model of data (i.e. inconsistent data and incomplete data). Care must be taken not to confuse inconsistent data and incomplete or missing data with other kinds of imperfect information.

3.3.1 Imprecise and vague

Imprecise was the first term used by Zadeh to describe fuzzy data. Imprecise is defined as not exact; vague or indefinite in nature. Both imprecise and vague describe values that may be known approximately, but not exactly. There may be an exact data value in the real-world, but it could not be obtained and may be unmeasurable using existing technology. Imprecise numeric input is best represented as a data range that is likely to contain the accurate, but undetermined value. These imprecise numeric values are fuzzy numbers that are represented as approximations mapped within a minimum/maximum range. An indeterminable value is not missing, but it cannot be measured accurately and a fuzzy number may be the most accurate representation possible. [71] [49, p.12-13]

Given imprecise data, vague querying is necessary. Vague is defined as being stated in general or indefinite terms; not having an exact or precise meaning. There are other definitions implying that the cause of vagueness is a lack of understanding, clouded thoughts or a hazy mental state. While the first definition describes imprecise data, the latter describes the mental state of the user who is searching the data. If identified natural language terms can be mapped to approximated ranges of values, imprecise data input and vague querying can be defined using these specific natural language terms. The user intuitively knows if an answer is close or acceptable and can judge the query results. A vague query is the search for something similar to the query match criteria. A vague value is not missing because it exists and the user will recognize it when he sees it. [40, p.47]

3.3.2 Ambiguous and subjective

Ambiguous is defined as capable of being classified in two or more categories, which reflects subjectivity. Membership in one, another or both sets is a matter of opinion. Ambiguity is closely related to subjectivity, opinion, and perception. [40, p.48]

Subjective is defined as being determined from opinions, intuition, or feelings rather than observation and reason. Subjectivity represents preconceptions derived from within the observer; not necessarily based on the external environment. This is similar to the idea that vagueness may originate in the mind of the user rather than within the data. In the context of a database query, this suggests that the user may have expectations that acceptable results must match.

If applications are to support data ambiguity and user subjectivity, data may require multiclass classification. An ambiguous domain is represented by a data value of some appropriate type and one or more fuzzy membership weights associated with the attribute as classifications. [71] Each classification is a set in which the data value has partial membership. Subjective querying allows the user to search for an object using multiple classifications in a way that resolves data ambiguity, but ambiguous data values are known values, not missing data.

3.3.3 Unclear and uncertain

Unclear is defined as ambiguous (explained above), but also means not explicitly defined (lacking a value), or indecipherable. *Undefined data may be missing until its value is known.* This definition suggests a missing data category. [19, p.220] Data may be indecipherable if there is confusion on the part of the person who gathered and/or added the data to the database. *If data is not understood well enough to add to the database, it may be considered undefined, lacking a value, and missing.* This definition suggests a missing data category.

Uncertain has a number of relevant definitions that depend on context. A fact may be uncertain if it is doubtable (i.e. the source of the data is not trusted). If data values are valid, this is an opinion and must be resolved outside of the database before data input. Database integrity relies on data and referential integrity rules to enforce database consistency by constraining data values to those allowed by the database design. If data values are invalid and do not comply with integrity constraints, they may be applicable, but uncertain and missing. This is a category of missing data. Events in the past are certain and are either known or unknown. *For example, while everyone living has a date of birth, for some this may be approximately known and for others it may be unknown.* This is a category of missing data. [34] Events in the future are uncertain with an indefinite date and time because they may or may not occur. These events may be possible with an unknowable possibility or inevitable with an unknowable date and time. *For example, everyone dies and a date of death for a living person is applicable, but does not exist.* This is a category of missing data. [40, p.48] [49, p.13]

Application design and queries must consider data clarity and certainty. Entity properties that may be undefined represent potential missing data. A search for those who are no longer living, is a query for those whose date of death is equal-to-or-less than the current date. A search for those who are still living, is a query for those whose date of death is missing and undefined.

3.3.4 Inconsistent and incomplete

Inconsistent is defined as showing contradiction (i.e. a proposition with component propositions that cannot all be true). Data inconsistency is a problem solved in the relational model by allowing relations to be normalized in a way that eliminates data redundancy. If applications need information from a single entity and duplicate this entity in different databases, a failure to update all copies of the data creates data inconsistency. Correct database design can eliminate this problem and allow all applications to share one copy of the data. [40, p.47] [49, p.12]

After a database is designed and implemented it is possible that a change in enterprise policy may cause a change in applicability of a property. If the change applies to all entities, the attribute can be removed from the database. If the change applies to some entities, the attribute must allow “data value missing because property inapplicable.” [34]

Incomplete is defined as not being finished or not having all of its components. Incomplete information is missing data and is often identified as unknown or inapplicable. Incomplete information must be stored using a meaningful representation until the missing data is available. It is necessary for the database management system and the application to process this representation while actual data is absent. [12]

3.4 Summary of missing data types

Applicable, but unknown data values must be included as one of the missing data types. This type appears to occur most often.

Invalid data values believed to be known, but not able to pass input verification or comply with data integrity constraints are missing. In this case, rather than drop the data value and record missing data “applicable, but unknown,” the invalid value could be stored in the database and used in the data correction process. The DBMS must process this data value as if it were not present, by definition it is not correct and is not known.

Inapplicable data values must be included as one of the missing data types. This type is important because it must be processed and interpreted in a way that is fundamentally different than many other types. If it is not possible to mark an attribute as not applicable, it is likely to be treated as unknown. This can cause an error in a query result set.

Unknowable data values are of two types. A restriction by law or policy is a security issue, but it could be a reason to remove data values from a database to meet a security mandate. A missing result from an SQL operator such as an outer join must be clearly indicated.

CHAPTER 4 Background

4.1 Null and 3-valued logic

Codd represented a missing data value as a null stored in the database [11, p.24]. A null is a missing data indicator rather than a value [11, p.25]. Component attributes of a primary key must be data values, while other attributes may be null [12, p.403]. The evaluation of missing data by the DBMS and its interpretation by the user relies on 3-valued logic (3VL) (see Figure 1) [11, p.25] [12, p.403]. If one or more data values compared using a relational operator is null, the truth value is "unknown." When the result of 3VL evaluation is "unknown," Codd's "null substitution principle" [12, p.404] allows a null to stand for any valid domain value while not being part of that domain and relational operations can include tuples that may match ("maybe-tuples" [7, p.608]) in the result set. The null substitution principle includes a non-duplication rule that allows one null be equal to another for the purpose of duplicate removal. [11, p.26] [1] The first version of the relational model (RM/V1) included this approach to processing missing data using a "truth-functional" [67, p.163] system.

Codd defined RM/V1 as a preliminary treatment of missing data. Nulls repre-

[1]The ANSI SQL standard allows duplicate rows as a default requiring users to specifically request duplicate removal from query results [65, p.263]. While a relation is a set of tuples, a table with duplicate rows is a multiset. Codd and Date agree that the relational model does not include duplicate tuples [10] [24].

AND	t	u	f
t	t	u	f
u	u	u	f
f	f	f	f

OR	t	u	f
t	t	t	t
u	t	u	u
f	t	u	f

NOT	
t	f
u	u
f	t

$MAYBE$	
t	f
u	t
f	f

Figure 1: 3-valued logic truth tables for AND, OR, NOT, and MAYBE

sent "value at present unknown," but "property inapplicable" is also missing data. Queries that embed a tautology (see section 4.3.2) return an incorrect result that confuses users. The "outer" variants of relational operators union and join can create an inapplicable missing value. [12, p.403].

The theory behind the relational model provides a practical foundation for data management and programmer productivity [13]. Codd described a relational processing capability as relationally complete when it included support for 2-valued logic without nulls and as having a fully relational algebra when it supported a 3-valued predicate logic using a single kind of null [13, p.112]. This definition describes the RM/V1 model for missing data. Codd defines a relational system as fully relational when it has better support for domains and primary keys [13, p.113]. This makes support for RM/V1 in existing relational DBMS products a necessary benchmark in the evolution of the relational model. Yet existing DBMS products do not support nulls to the extent required by the 1999 SQL standard [65] (see section 3.2.3), and most databases have missing data [11, p.24].

Codd's extension of RM/V1 to RM/V2 represented missing data using a "mark" instead of null. An applicable, but missing data value (null) is an A-mark and data missing because it does not apply to an item is an I-mark. Attributes are now described as either marked or unmarked and the term null is no longer used. [14, pp.56-58] Replacing null with marks leads to 4-valued logic (4VL) (see Figure 2). [15] There has been no migration from RM/V1 to RM/V2 in the implementations of relational DBMS.

AND	*t*	*a*	*i*	*f*
t	*t*	*a*	*i*	*f*
a	*a*	*a*	*i*	*f*
i	*i*	*i*	*i*	*f*
f	*f*	*f*	*f*	*f*

OR	*t*	*a*	*i*	*f*
t	*t*	*t*	*t*	*t*
a	*t*	*a*	*a*	*a*
i	*t*	*a*	*i*	*f*
f	*t*	*a*	*f*	*f*

NOT	
t	*f*
a	*a*
i	*i*
f	*t*

MAYBE	
t	*f*
a	*t*
i	*f*
f	*f*

Figure 2: 4-valued logic truth tables for AND, OR, NOT, and MAYBE

4.2 A foundation for maybe-operators

Biskup [7] provides a formal foundation for Codd's RM/V1 representation of missing data using nulls ("value at present unknown"), maybe-tuples determined by the null substitution principle, and rules for processing nulls. His foundation is built on three fundamental assumptions. The first is the "appropriate scheme assumption" that the real-world is time-independent and can be represented by attributes which take values from a domain to characterize real-world events as tuples in a time-varying relation. Next, the "incomplete information assumption" states it is appropriate that a relation scheme describe real-world events, but detailed knowledge of this scheme is not always be complete. This means the time-varying relation must represent missing data. Finally, the “closed world assumption” [58] assumes that a tuple that cannot be derived from a relation by appropriate application of the null substitution principle does not hold in the real world.

Biskup’s method extends Codd’s representation for missing data by adding a special attribute to each tuple indicating “definite” or “maybe” status to support maybe-tuple processing. Definite-tuples contain only known data values. Maybe-tuples include attributes represented as missing data values.

Codd referred to Biskup’s published work in his 1981 ACM Turing Award Lecture [13, p.116] as an example of research leading to improved handling of nulls, but there is no indication that Biskup’s ideas were tested or implemented.

4.3 The problem with null

Grant [34] identified two problems with Codd’s null and 3VL approach to missing data.

4.3.1 The null debate

Grant [34] determined that there were at least two kinds of null. While Codd addressed the problem of data values applicable yet unknown, Grant suggested that another null be included for data values missing and not applicable.

Date [18] asserted that nulls should not be supported in the relational model because null is not a value. Date and Darwen [24, p.193] observed that relations are created from tuples and attributes that contain values, but null is not a value. The Structured Query Language (SQL) allows DBMS tables to contain nulls, but the relational model does not [64, p.13].

Codd includes a requirement for systematic support of missing data in the relational model that does not store a representation for the missing data as a value in the represented attribute using any kind of default or special value [15, p.39-40]. This requirement is one of Codd's twelve rules for relational database. This clearly shows that Codd supported the use of nulls or marks to represent missing data.

4.3.2 The null paradox

A second problem identified by Grant [34] is the paradox for queries with an embedded tautology. These queries should match all data values in a domain and return all tuples with a domain value or an applicable missing data value. But the truth table for the Kleene 3VL used by Codd does not allow a tautology when one of the variable values is "unknown" [33, p.222]. For example, a query for people age fifty or younger, or older than fifty should includes every person in the relation. All living people should match this query, even if a person's age is not known, missing from the database, and marked with a null. However, people with a null date of birth will not be retrieved when using 3VL.

Codd observed that excluding individuals of unknown age is reasonable because the DBMS reports what it knows about the real world, not necessarily what is true about the real world [12, pp.403-408]. Another suggestion was to to warn users not to write queries that are tautologies [15, p.385].

Grant's solution is a "non-truth-functional" 3VL [34, p.156] that substitutes domain values for the null and if a 2-valued logic (2VL) tautology is detected, the expression containing a null is true otherwise it is unknown [33, pp. 222-223]. This approach ensures that if an expression is a tautology in 2VL, it is also a tautology in 3VL. There is no evidence of Grant's method for evaluating queries was implemented in a DBMS.

4.4 Non-truth-functional systems

Non-truth-functional systems do not use truth tables to determine the veracity of comparisons made using relational operators. This approach attempts to resolve the paradox caused by missing data and many-valued logics. [33]

4.4.1 Denotational semantics for applicable and inapplicable

Vassiliou [67] proposed a "non-truth-functional" system based on denotational semantics to interpret missing data within the relational model. Denotational semantics use an approximation function to map a data type from one interval to another data type in a more precise interval. There are data types for which no finite approximation is possible. Approximations of these data types define the domain limits as the greatest lower bound and the lowest upper bound. These limits are used to extend attribute domains to include missing values that are applicable, but unknown or inapplicable. Additionally, the approximation function can detect queries with embedded tautologies.

4.4.2 Partially known data and set-valued attributes

Grant [35] suggested a modified relational model that allowed partially known data values stored as numeric ranges, partially given strings, or set-valued attributes. This data model does not allow duplicates of tuples when each attribute contains a single value, but duplication of tuples is implicit for the case of partially known values. Nulls are allowed for information that is not known, but partially known values can be stored as a range over an entire domain. While predicates may take the truth values “true,” “unknown,” and “false,” the unknown truth value is a placeholder so that there is no truth-functional 3-valued logic. Grant’s true-predicate and maybe-predicate correspond to Lipski’s external interpretation of the real world and internal interpretation of real world modeled by the data [38, p.263].

The primitive predicates in Grant’s model include negation so there is no need for a boolean NOT. If P is a predicate for all possible values in an attribute, a true-predicate, P_T, is defined as the predicate that holds true for all proper substitutions for the entries in P and a maybe-predicate, P_M, is defined as the predicate that holds true for at least one proper substitution for the entries in P. [35]

Using Grant’s representation of partially known data a person whose age is unknown would be inserted into the database as a null. A constraint on the representation of age would record this information as a null, but in the range of 1 to 110. Using Grant’s notation, such a data value for AGE appears as <null, (1,110)>. From

Grant's definition of a true-predicate, tautology (AGE $\leq 50 \vee$ AGE > 50) evaluates an unknown AGE attribute as true because every substitution in the range of 1 to 110 is less than, equal to, or greater than 50.

Others [1] [61] [17] have suggested models for partially known data to represent missing data values. This allows the unusual case where a data value is known to be one of number of values. The usual case is that unknown data is completely unknown. A set-valued attribute creates a predicate that is both conjunctive and disjunctive. This characteristic violates the set-theoretic model used by the relational model of data. These ideas have not been implemented and can be considered beyond the scope of database [67].

4.4.3 Statistical probability

Wong [69] proposed a statistical approach to determine probable values for missing data using statistical inference and prior information stored in a separate database. In the case of missing attributes (i.e. applicable but unknown), a probability distribution could be used to determine the mean or median as an approximate data value, but missing data (i.e. inapplicable) is flagged so that it can be ignored. This system was implemented using the INGRES relational database system and QUEL data sublanguage. This approach uses large datasets of raw data or samples rather than determining specific missing data values. While theoretically sound, Wong concluded that the source of data used to create the prior distribution information was a potential database implementation problem.

Others [31] [4] [26] [27] have suggested probabilistic models for incomplete and missing data. These probabilistic system use statistical models that make a general estimation from specific details in large data samples rather than being database management systems that use the relational model to store and retrieve data.

4.4.4 Fuzzy possibility

Fuzzy logic generalizes the truth value of a proposition from a 2-valued logic {0,1} to a many-valued logic using the interval [0,1]. A membership function determines a value from the interval [0,1] to indicate a weight or degree of membership of a data value in a fuzzy set. Using fuzzy set theory to define fuzzy domains, allows fuzzy databases a flexible way to represent imperfect information in a database. In simple terms, fuzzy sets allow objects to have a varying weight of truth and/or to belong to more than one classification.

A fuzzy-weight attribute may provide a mechanism to represent missing data. Zadeh gives an example of 3-valued logic built on a fuzzy set using two value levels α and β [71]. These value levels create three ranges; from 0 to β, between β and α, and from α to 1. This function shown in Equation (1), returns "true," "false," or "unknown" for the value in attribute u of domain F using the membership weight from function $\mu_F()$ and the α and β level values associated with the attribute type of domain F.

$$\text{3VL truth value for fuzzy domain } F = \begin{cases} \text{false} & \text{when } \mu_F(u) = [0, \beta] \\ \text{unknown} & \text{when } \mu_F(u) = (\beta, \alpha) \\ \text{true} & \text{when } \mu_F(u) = [\alpha, 1] \end{cases} \qquad (1)$$

While the meaning of the three values may be application specific, this approach could allow applications to define the necessary processes for an n-valued logic derived from a fuzzy-weight attribute. An application could determine how to process each case of missing data by matching a classification weight to an α-cut at a level appropriate for the needs of the application. In this case, the criteria for membership in a fuzzy unknown classification is an application responsibility. If the DBMS is to generalize processing of missing data using an n-valued logic, there is a need to supply relevant parameters to the process.

Medina [42] proposed the generalized fuzzy relational database (GEFRED) model of data. Its representation for missing data is based on an equivalency between degree of membership in fuzzy sets and possibility distributions [72]. Missing data values that are applicable to an object are UNKNOWN with a possibility distribution of 1 indicating certainty that a value from the domain represents the missing data. Missing data values that are inapplicable to an object are UNDEFINED with a possibility distribution of 0 indicating that no value from the domain possibility represents the missing data. Dubois and Prade [29] extended Zadeh's possibility theory to include the concept of necessity. de Tre [25] built on GEFRED, modeling unknown information as uncertainty about its propositional truth using possibility theory.

4.4.5 Logic database and knowledge based systems

Reiter [59] argues that logic provides a way to represent database relations, integrity constraints, and queries using well-formed formulas of first order predicate logic. The relation heading represents a predicate and each tuple in the relation body represents a proposition that is true within the closed world of the database. This approach supports query languages able to reason deductively.

Reiter [60] developed a method based on first-order logic for querying relational databases that have missing data represented by nulls. This method is sound and in certain cases complete, but in all cases compatible with the relational model of data. This approach is essentially a specification of a predicate logic for a knowledge based system that is complete for two classes (i.e. positive and universally conjunctive) queries of databases with missing data values defined as null (i.e. applicable, but unknown) and for databases not missing data values.

The relational model implemented in existing DBMS is complete for databases with complete information. The problem of missing data are the cases of incompleteness when a query result does not include all correct answers and the user is not able to accurately interpret the query result.

Yuan et al. [70] proposed an extension to the relational model and Reiter's algorithm to support sound and complete query evaluation for relational databases containing nulls. Completeness requires tuples to include disjunctive data. This is achieved using attributes that contain relations of data values one of which may make the tuple a true predicate. This is a model of a non-first-normal form database.

Lipski [38] proposed a mathematical model to study the semantics of incomplete information in databases. This model, called an "information system," was based on modal logic, stored information about properties of objects, and allowed this information to be incomplete. Incomplete information extended the notion of a null value and was represented by a subset of an attribute's domain. Lipski's information system was not based on the relational model of data, but it is similar to a relation. It used a single table where columns represent properties and rows represent objects. In the case of incomplete information, it may not be known if an object has a particular property (i.e. property is inapplicable) or it may be known that the object has a property, but its particular value is not known (i.e. property applicable, but value unknown). If a property is applicable, the missing information must be a value from a subset of the property's domain and the property may hold a set of possible values.

An information system has a query language and can answer questions in either of two ways. It can find and list the set of objects that match a property. It can also determine if it is true or false that a particular object has a property. Queries may have one of two interpretations. An external interpretation is about the real world as it is incompletely modeled by the information system. An internal interpretation refers to the information system's data content. In terms of modal logic, an external interpretation is what is necessary and an internal interpretation is what is possible. This follows modal logic, the external interpretation is that which is true and necessary while the internal interpretation is that which may or may not be and is possible.

Others [39] [66, pp.100-115] have suggested how logic and knowledge based systems might represent and process missing data. Datalog [62] is a subset of Prolog that can reason deductively, is designed to serve as an SQL query sublanguage, and has been implemented using Prolog as an experimental learning system. Logic databases and knowledge based systems may be similar to database management systems, but use data and rules to infer the specific from the general rather than being database management systems that use the relational model to store and retrieve data.

4.5 Defaults and special values

4.5.1 Avoidance

A current introductory guide to SQL suggests that nulls be avoided. The recommended approach is to set each attribute in the database as NOT NULL unless data is expected to be initially missing, but supplied later. This approach suggests the origin of special values and encapsulates the missing data problem in a nutshell. [5, pp.44-50].

4.5.2 Special values

Date [19] [22] proposed an alternative to nulls and 3VL which uses a special data value selected to represent missing data. This method requires a valid, but unused domain value for each type of missing data. One of these special values may be configured as the SQL attribute default value to be inserted when no value is supplied. While users and applications must be aware of each missing data type, interpret the meaning of each special value, and dedicate actual data values from the domain to serve as special values, using a domain data value to represent missing data eliminates the need for 3VL [22]. The advantage of this approach and the elimination of 3-valued logic is that comparison operators always return true or false even when a special value is compared to a data value or another special value [20, p.245]. Arithmetic operations will always return meaningful values, but only if the special value for a missing data type was carefully chosen. This could also simplify DBMS processes that rely on the results of comparisons and arithmetic [19, pp.223-231]. This approach can be easily implemented in current database systems with support from the DBA.

An issue with Date's approach is its focus on a single type of missing data (i.e. applicable, but unknown) provides an overly simplified model. Although Date suggests that special values can be extended to deal with other missing data types [22, p.351] and this seems feasible, each type of missing data requires an unused value from the attribute's domain and it is not clear that it will always be possible to dedicate these data values to represent missing data.

In *The Third Manifesto*, Date and Darwen observe that null is not a value and while SQL allows its tables to contain null, this is a violation of the relational model because relations are created from tuples and attributes that contain values [24, p.193]. Using null to represent missing data is a simple solution to a complex problem caused by data that is unavailable or inapplicable. While bad database design may result in an attribute inapplicable to an object, data that is not yet available is supported by the relational model. For this reason, *The Third Manifesto* excludes nulls from the relational model and recommends user selected default values instead.

4.5.3 Default values with truth tables

Gessert [32] extended the concept of replacing nulls with special values in a way that allows an existing DBMS to implement 4VL using 2VL. This approach uses a logical status table in parallel with each data table that allows missing data. Both tables use identical attribute names and data values as a primary key. Data tables store non-key missing data values as user selected default values. Logical status tables store a validity code for each non-key attribute in the data table. These codes indicate if the data is valid, invalid, unknown or inapplicable.

A summary of theses validity codes (truth values) follows:

1. Not applicable (NA) stored value 0

   This is an RM/V2 I-mark (the attribute does not apply to the object and should be ignored).

2. Applicable and false (AF) stored value 1

   A value in the attribute's domain failed input validation and is to be corrected.

3. Applicable and maybe true (AM) stored value 2

   This is an RM/V1 null or RM/V2 A-mark and the data value is unknown.

4. Applicable and true (AT) stored value 3

   The stored data value is complete information.

This method eliminates the need for nulls, requires no modification to the DBMS, and allows an application to determine 4VL interpretations for missing data using 2VL. This is a practical concept that may have been used in a real application. However, it doubles the size of each relation and requires user applications to implement all processing logic for missing data. The default values used in this approach are supplied by the user rather than the DBMS.

The DBMS considers the data table and the logical status table both to be data. The application must manage both tables correctly to manage missing data. If two application programs use the same database, each must be configured to manage the missing data information in the same way. If there is a need to explain missing data using metadata, this must be implemented in the application.

There is no transparent support for nulls that can be used with this method. If the data table stores a non-key missing data value as null, the application must handle the null as a default value.

4.6 Decomposition

The binary relation led to the n-ary relation in the relational model. A binary relation with a primary key and a single data value cannot be incomplete [16, p.6]. The notion of an irreducible relation suggests a technique to eliminate attributes that do not have data values. Advocates for decomposition reject the use of nulls and 3VL to represent missing data, consider inapplicable data not to be missing, and solve the problem of missing data by iteratively decomposing a relation into smaller relations until there are no attributes missing a value. In the following sections decomposition of a relation using normalization is explored as a method to remove nulls from attributes.

4.6.1 Vertical and horizontal decomposition

Darwen [16] relies on a tuple being a proposition with the relation's heading as its predicate. Attributes in a tuple are conjunctive propositions connected by a logical AND. Tuples in a relation are disjunctive propositions separated by a logical OR. Using these characteristics, the missing data values in a relation can be eliminated by a normalization process. Recomposition of the original relation by querying the decomposed relations is essential if this method is to work.

Vertical decomposition uses projection to eliminate attributes of unknown values or inapplicable properties. An attributes with missing data values is decomposed into a relation of just the key. This relation is named to indicate which attribute it represents and why data is missing (e.g. unknown or inapplicable).

Horizontal decomposition uses restriction to collect propositions with different meaning into separate relations. Tuples with incomplete information necessarily have an abbreviated meaning (loss of information) when unknown or inapplicable attributes are removed.

4.6.2 Iterative decomposition

Pascal [54] proposed an approach in which the database stores only what it knows about the real world. An item in the real world may have properties that exist in the real world, but are not known to the database. An item in a database that does not have a particular property belongs in a relation that does not have an attribute for this property. This method requires the application include logic to construct an interpretation from this collection of relations.

As more becomes known, the model is updated and at any time it can be queried about what it does know. The results of a query may be a collection of relations each of which is either a subset or superset of another and of a different degree. However, each tuple in each relation is unequivocally a true proposition and its approach to missing data uses 2VL. Pascal describes this approach as feasible and implies that it may be implemented [54, p.25], but there is no indication when this may be.

4.7 Summary of previous work

Codd's third rule establishes a need for systematic representation of missing data. RM/V1 provides a specification for how to do this using nulls and 3-valued logic. Biskup's formal foundation for RM/V1 describes how it might work if implemented. The SQL standard includes the features needed by RM/V1, but existing relational DBMS do not fully implement this standard.

The proposed solutions to the missing data problem focus on eliminating or mitigating the impact of 3-valued logic required by a representation of missing data that uses NULL. However, the existing DBMS have not implemented these approaches either. In this context, the problem of representing missing data values in relational DBMS is an unsolved problem.

CHAPTER 5 Impact of Missing Data

Missing data impacts relational database design as well as the application software and users that rely on the design. The issues identified in chapter 4 contribute to a better understanding of the broad impact of missing data on relational databases. A complete analysis of missing data's effect on the relational model is necessary before a workable solution to the problem can be fully identified.

5.1 Database design

Database design creates a schema or view of data at the logical level. At this level incomplete information must be represented so that a relational DBMS can correctly execute relational operators and the results of processing can be correctly interpreted when presented to the user.

The realization that data must be allowed to be absent is made during database design. The user's application requirements identify attributes that may be temporarily unknown. The database design process determines if it is feasible to maintain database integrity when a particular data value is missing. This is the case for an attribute that is always applicable to its entity and while the data value is currently unknown, it is expected to be available later. If an attribute is not applicable to its entity in some cases, the need to flag the attribute as missing should be eliminated by further data normalization. Once made, these decisions are incorporated in the database and application implementation.

5.2 Database management systems

Database management systems bind the logical data schema to the physical data definitions used by storage devices. The DBMS must store the database schema and use the metadata from this representation to process relational operators and return results.

A DBMS that supports missing data values, must check each attribute involved in an operation and if the attribute allows missing values, adjust processing to the extent that it can.

5.3 SQL data sublanguage

Missing data should require the DBMS to process data comparisons and arithmetic operations as special cases. When a database is designed for a new system of application programs, use cases for the application should be examined to identify SQL queries that will refer to attributes allowed to be represented by missing data. These queries are expected to be used by the applications and are part the application's verification testing.

5.4 Application programs

The user's view of the data is an interface at the logical level. Access to this interface may be through an application program or through an end user query tool. The user must be able to determine a correct interpretation of data defined by the schema and processed by the DBMS.

Data that was identified as potentially incomplete must be documented in the application's detailed design specification.

When it becomes necessary to alter a database to add support for missing data, the change is transparent to the user if the database management system is responsible for processing all kinds of missing data. But if the application program is responsible for processing missing data, the application must be reengineered.

The change may require support for data that is temporarily unavailable or for data that is not always applicable to an entity. Each kind of change requires modification to the database and to the application programs.

CHAPTER 6 Hypothesis

The solution to the missing data problem is specified in this chapter. The interpretation of missing data uses information stored in database tables. There are clearly defined roles for the DBMS, database administrator, and user. This solution has been implemented in a client interface using an embedded MySQL server.

6.1 The KNOWN/UNKNOWN model

In this model, attributes which are part of keys must contain data values. Other attributes are allowed to have missing data values of various types. Values that are unknown may be applicable and missing, or invalid. Values that are optional may be inapplicable or unknowable.

6.2 Metadata for missing data types

The missing data types described in chapter 3 are summarized in Table 2. The four classes determine how a DBMS processes missing data, the seven types indicate how the application or user derives an interpretation, and the class-type "tag" is a unique representation for the kinds of missing data. This metadata is used by the KNOWN/UNKNOWN missing-data model. It can be expanded or customized for a given database, and is available to applications and users.

Properties that are applicable to an item are expected to have values. A missing

Table 2: Metadata for missing data types

Class Name	**Type Description**	**Class**	**Type**	**Tag**
Applicable	property applicable - value unknown	1	2	UNK
Applicable	property applicable - value does not yet exist	1	3	NYE
Invalid	property applicable - value is undefined	2	4	UND
Invalid	property applicable - value input is invalid	2	5	INV
Invalid	property applicable - value withheld at input	2	6	MIS
Inapplicable	property not applicable to this item	3	1	N/A
Unknowable	value declared unknowable; withheld or removed	4	3	REM
Unknowable	value result from SQL operation is empty set	4	7	NIL

value that is unknown may exist in the real world, but not in the database. A value that does not yet exist in the real world cannot exist in the database at this time. Invalid values can be corrected by the data input/update process, but until made valid these values must be processed as if unknown.

Inapplicable data values are significant because they must be processed and interpreted in a way that is fundamentally different from applicable or invalid data. These data are not expected to exist in the real world or the database. A missing result from an SQL operation such as an outer join must be clearly indicated as unknowable. While unknowable values are processed as if the attributes are inapplicable, it is important that users know why these values are missing.

6.3 KNOWN and UNKNOWN data values

A relation variable (relvar) which allows missing data uses three relations to represent complete and incomplete information. This relvar has a name by which it may be referenced (e.g. my_names) and its constituent relations are named by appending a qualifier (i.e. _KNOWN, _UNKNOWN, and _MISSING). The KNOWN relation stores only tuples with complete information (see Table 3). The KNOWN relation has a shadow that stores tuples that are missing data values as the UNKNOWN relation (see Table 4). Information about what data is missing from the UNKNOWN relation is stored in the MISSING relation (see Table 5) with a class-type tag giving a foreign key to the metadata in Table 2.

Table 3: my_names_KNOWN

first	middle	last
Edgar	F	Codd
Chris	J	Date

Table 4: my_names_UNKNOWN

first	middle	last
Hugh		Darwen
Andrew		Warden

Table 5: my_ names_MISSING

key	attr	tag
Darwen	middle	UNK
Warden	middle	N/A

The key of the UNKNOWN relation, the name of the attribute that is missing data, and the missing data class-type tag are stored in the MISSING relation. These attributes form the key of the MISSING relation.

6.4 Integrity independence

The KNOWN/UNKNOWN missing-data representation must be enabled for the entire database. By default, all non-key attributes may contain incomplete information. Attributes may be flagged to disallow missing data. If data can be missing, the database allows each type of missing data represented by the class and type from Table 2. It would also be possible to implement a DBMS system for which allowed missing data types are specified during table creation. However, this is not in the scope of this research.

6.4.1 Entity integrity

Because no data value may be missing from a primary key component, both the KNOWN and UNKNOWN relations will have primary keys that are complete. The KNOWN/UNKNOWN model requires that no key value be duplicated in the two relations. An attempt to insert a tuple with complete information into the KNOWN relation will fail if its key is present in the UNKNOWN relation and vice-versa.

6.4.2 Referential integrity

Referential integrity is represented as a dependency between a referenced relvar (parent) and a referencing relvar (child). Constraints are defined on the foreign keys in child relvars and refer to the primary key of the parent relvar. The semantics of referential integrity require that foreign keys not be missing as invalid, inapplicable or unknowable. If a constraint sets a child tuple's foreign key to a missing value it is "applicable, but does not yet exist."

6.4.3 Database integrity

The Golden Rule [23, p.261] requires that all relations in a database remain consistent with the integrity constraints defined for the database at all times. The KNOWN/UNKNOWN missing-data model maintains the Golden rule for its constituent relations. This requires that the conjunction of all integrity constraints over the KNOWN and UNKNOWN relations must not evaluate to false. The KNOWN relation will evaluate to true. The UNKNOWN relation will evaluate to true, maybe-true (unknown) or inapplicable, but not to false.

6.5 Relational operations using missing data

There are five primary operations required by the relational model: Cartesian product, set union, project, set difference, and restrict. All other operations can be defined in terms of these [66, p.55] [23, p.192]. This section uses the four test cases described in section 6.5.3 to illustrate these five primary relational operations.

6.5.1 Expression evaluation

If a relational operation requires the evaluation of missing data in an expression, it is possible to compare known and unknown values of the same data type when the missing data class is either applicable or invalid. For example, a known integer and an applicable, but unknown integer may be equal or one may be less/greater than the other. However, if the missing data class is inapplicable or unknowable, a comparison is meaningless.

If a relational operation requires the evaluation of missing data for duplicate removal, one missing value may be a duplicate of another when the metadata in the MISSING relation indicates that attribute, class, and type are the same.

6.5.2 3-valued logic (unknown and MAYBE)

In the case of the KNOWN relation, the comparison operators are truth-valued functions that depend on data values and return either true or false. In the case of the UNKNOWN relation, a non-truth-valued function is needed to compare missing data values. The value of an unknown but applicable attribute is some value from the attribute's domain and within the database's integrity constraints. Until this missing data value is known, an evaluation using a truth-valued function is "unknown," but a non-truth-valued function can determine if a maybe-match is possible. The KNOWN/UNKNOWN missing-data model supports querying for tuples that are maybe-matches.

A DBMS that uses 3-valued logic can be made more intuitive by including a MAYBE modifier for its comparison operators. While the SQL comparison operator “IS NULL” is used to select attributes that do not have data values, the MAYBE match modifier is different. The MAYBE modifier relaxes comparisons allowing applicable and invalid unknown values from a domain to match known values and other applicable missing data from the same domain. Maybe-matches expand the UNKNOWN relation in a result set.

6.5.3 Four test cases for missing data

Four variants of the my_names and your_names relvars are used to examine the behavior of relational operations.

The test cases are shown in Tables 6 through 13. They are designed to demonstrate how a DBMS applies the relational operations using the KNOWN/UNKNOWN missing-data model.

Sequence numbers are used as pseudo-keys. The KNOWN/UNKNOWN representation for missing data depends on keys to connect the missing data in the UNKNOWN relation with its metadata in the MISSING relation. Result sets require keys to reference a missing data value’s metadata. These keys may be domain keys composed from the relation’s attributes or pseudo-keys generated sequentially during querying. Equivalent pseudo-keys do not indicate matching tuples nor do unequal pseudo-keys indicate mismatching tuples.

Table 6: Case 1 (a) - complete and incomplete information in my_names

my_names_KNOWN			
key	first	mi	last
1	Edgar	F	Codd
2	Chris	J	Date

my__names_UNKNOWN			
key	first	mi	last
3	Hugh		Darwen
4	Andrew		Warden

my_names_MISSING		
key	attr	tag
3	mi	UNK
4	mi	N/A

Table 7: Case 1 (b) - complete and incomplete information in your_names

your_names_KNOWN			
key	first	mi	last
5	Jeffrey	D	Ullman
6	Margo	I	Seltzer

your__names_UNKNOWN			
key	first	mi	last
7	Fabian		Pascal
8	David	?	McGoveran

your_names_MISSING		
key	attr	tag
7	mi	UNK
8	mi	INV

My_names in Table 6 and your_names in Table 7 are a database that includes both complete and incomplete information. Data is missing for various reasons, but there are no duplicate or possibly duplicate names in these relvars.

Table 8: Case 2 (a) - a tuple from your_names_KNOWN is duplicated in my_names_KNOWN

my_names_KNOWN			
key	first	mi	last
1	Edgar	F	Codd
2	**Jeffrey**	**D**	**Ullman**

my__names_UNKNOWN			
key	first	mi	last
3	Hugh		Darwen
4	Andrew		Warden

my_names_MISSING		
key	attr	tag
3	mi	UNK
4	mi	N/A

Table 9: Case 2 (b) - a tuple from my_names_KNOWN is duplicated in your_names_KNOWN

your_names_KNOWN			
key	first	mi	last
5	**Jeffrey**	**D**	**Ullman**
6	Margo	I	Seltzer

your__names_UNKNOWN			
key	first	mi	last
7	Fabian		Pascal
8	David	?	McGoveran

your_names_MISSING		
key	attr	tag
7	mi	UNK
8	mi	INV

My_names in Table 8 and your_names in Table 9 have the name "Jeffrey D Ullman" common to both KNOWN relations. These are exact duplicate names, but the pseudo-keys created by the DBMS do not indicate knowledge of this duplication.

Table 10: Case 3 (a) - a tuple from your_names_UNKNOWN may be duplicated in my_names_KNOWN

my_names_KNOWN			
key	first	mi	last
1	Edgar	F	Codd
2	**Chris**	**J**	**Date**

my__names_UNKNOWN			
key	first	mi	last
3	Hugh		Darwen
4	Andrew		Warden

my_names_MISSING		
key	attr	tag
3	mi	UNK
4	mi	N/A

Table 11: Case 3 (b) - a tuple from my_names_KNOWN may be duplicated in your_names_UNKNOWN

your_names_KNOWN			
key	first	mi	last
5	Jeffrey	D	Ullman
6	Margo	I	Seltzer

your__names_UNKNOWN			
key	first	mi	last
7	**Chris**		**Date**
8	David	?	McGoveran

your_names_MISSING		
key	attr	tag
7	mi	UNK
8	mi	INV

My_names in Table 10 and your_names in Table 11 have a name with an unknown middle initial that may be a duplicate of a name with a known middle initial. The name “Chris Date” in the UNKNOWN relation of the your_names relvar is a maybe-match with the name “Chris J Date” in the KNOWN relation of the my_names relvar.

It is possible, but not certain that these are the same person because the key for each relvar is a pseudo-key created by sequential number generators. If the keys matched and were a domain key (e.g. employee ID number), it would be certain that “Chris Date” and “Chris J Date” are the same person.

Table 12: Case 4 (a) - tuples in your_names_UNKNOWN may be duplicated in my_names_UNKNOWN

my_names_KNOWN			
key	first	mi	last
1	Edgar	F	Codd
2	Chris	J	Date

my__names_UNKNOWN			
key	first	mi	last
3	**Hugh**		**Darwen**
4	**Andrew**		**Warden**

my_names_MISSING		
key	attr	tag
3	mi	UNK
4	mi	N/A

Table 13: Case 4 (b) - tuples in my_names_UNKNOWN may be duplicated in your_names_UNKNOWN

your_names_KNOWN			
key	first	mi	last
5	Jeffrey	D	Ullman
6	Margo	I	Seltzer

your__names_UNKNOWN			
key	first	mi	last
7	**Hugh**		**Darwen**
8	**Andrew**		**Warden**

your_names_MISSING		
key	attr	tag
7	mi	UNK
8	mi	MIS

My_names in Table 12 and your_names in Table 13 have names common to both UNKNOWN relations. "Hugh Darwen" is missing the middle initial for the same reason (applicable, but unknown) in both relvars. It is possible that this name is a duplicate, but the assumption that "Hugh Darwen" is one person cannot be determined without more information.

The name "Andrew Warden" is missing the middle initial for different reasons (inapplicable vs. applicable but withheld) in each relvar. It must be assumed there is a person named "Andrew Warden" who has no middle initial and an "Andrew Warden" who did not provide his middle initial.

These test cases show how the inclusion of the MISSING metadata in KNOWN/UNKNOWN model provides useful information about possible tuple matches to users and the DBMS when data is missing. Involvement with the meaning of missing data encourages data gathering and supports accurate data analysis.

6.5.4 Cartesian Product

The Cartesian product of two relvars, shows each tuple from the first relvar paired with each tuple from the second relvar. Cartesian products using the four test cases for my_names and your_names are shown in Tables 14 through 25.

If the product for case 1 shown in Table 15 is used to project and select my_names and your_names on middle initial, there are maybe-matches on middle initial in the DERIVED_UNKNOWN table.

If the product for case 2 shown in Table 17 is used to project and select my_names and your_names on middle initial, there is an exact match on first name, middle initial and last name for "Jeffrey D Ullman."

If the product for case 3 shown in Table 20 is used to project and select my_names and your_names on middle initial, there is an exact match on first name and last name between "Chris J Date" in my_names_KNOWN relation and "Chris Date" whose middle initial is missing in your_names_UNKNOWN, a maybe-match on middle initial implying these may be the same "Chris Date."

If the product for case 4 shown in Table 24 is used to project and select my_names and your_names on middle initial, there are exact matches on first name and last name for "Hugh Darwen" and "Andrew Warden" in the DERIVED_UNKNOWN relation. But "Andrew Warden" is missing a middle initial for two different reasons (i.e. it is missing because it is known not to exist or because it is non-applicable) and is not a match.

Table 14: Case 1 - (*my_names* × *your_names*)KNOWN

derived_KNOWN							
m_key	m_first	m_mi	m_last	y_key	y_first	y_mi	y_last
1	Edgar	F	Codd	5	Jeffrey	D	Ullman
1	Edgar	F	Codd	6	Margo	I	Seltzer
2	Chris	J	Date	5	Jeffrey	D	Ullman
2	Chris	J	Date	6	Margo	I	Seltzer

Table 15: Case 1 - (*my_names* × *your_names*)UNKNOWN

derived_UNKNOWN							
m_key	m_first	m_mi	m_last	y_key	y_first	y_mi	y_last
1	Edgar	F	Codd	7	Fabian		Pascal
1	Edgar	F	Codd	8	David	?	McGoveran
2	Chris	J	Date	7	Fabian		Pascal
2	Chris	J	Date	8	David	?	McGoveran
3	Hugh		Darwen	5	Jeffrey	D	Ullman
3	Hugh		Darwen	6	Margo	I	Seltzer
3	Hugh		Darwen	7	Fabian		Pascal
3	Hugh		Darwen	8	David	?	McGoveran
4	Andrew		Warden	5	Jeffrey	D	Ullman
4	Andrew		Warden	6	Margo	I	Seltzer
4	Andrew		Warden	7	Fabian		Pascal
4	Andrew		Warden	8	David	?	McGoveran

Table 16: Case 1 - MISSING

derived_MISSING			
m_key	y_key	attr	tag
1	7	y_mi	UNK
1	8	y_mi	INV
2	7	y_mi	UNK
2	8	y_mi	INV
3	5	m_mi	UNK
3	6	m_mi	UNK
3	7	m_mi	UNK
3	7	y_mi	UNK
3	8	m_mi	UNK
3	8	y_mi	INV
4	5	m_mi	N/A
4	6	m_mi	N/A
4	7	m_mi	N/A
4	7	y_mi	UNK
4	8	m_mi	N/A
4	8	y_mi	INV

Table 17: Case 2 - (*my_names* × *your_names*) KNOWN

DERIVED_KNOWN							
m_key	m_first	m_mi	m_last	y_key	y_first	y_mi	y_last
1	Edgar	F	Codd	5	Jeffrey	D	Ullman
1	Edgar	F	Codd	6	Margo	I	Seltzer
2	Jeffrey	D	Ullman	5	Jeffrey	D	Ullman
2	Jeffrey	D	Ullman	6	Margo	I	Seltzer

Table 18: Case 2 - (*my_names* × *your_names*) UNKNOWN

DERIVED_UNKNOWN							
m_key	m_first	m_mi	m_last	y_key	y_first	y_mi	y_last
1	Edgar	F	Codd	7	Fabian		Pascal
1	Edgar	F	Codd	8	David	?	McGoveran
2	Jeffrey	D	Ullman	7	Fabian		Pascal
2	Jeffrey	D	Ullman	8	David	?	McGoveran
3	Hugh		Darwen	5	Jeffrey	D	Ullman
3	Hugh		Darwen	6	Margo	I	Seltzer
3	Hugh		Darwen	7	Fabian		Pascal
3	Hugh		Darwen	8	David	?	McGoveran
4	Andrew		Warden	5	Jeffrey	D	Ullman
4	Andrew		Warden	6	Margo	I	Seltzer
4	Andrew		Warden	7	Fabian		Pascal
4	Andrew		Warden	8	David	?	McGoveran

Table 19: Case 2 - MISSING

DERIVED_MISSING			
m_key	y_key	attr	tag
1	7	y_mi	UNK
1	8	y_mi	INV
2	7	y_mi	UNK
2	8	y_mi	INV
3	5	m_mi	UNK
3	6	m_mi	UNK
3	7	m_mi	UNK
3	7	y_mi	UNK
3	8	m_mi	UNK
3	8	y_mi	INV
4	5	m_mi	N/A
4	6	m_mi	N/A
4	7	m_mi	N/A
4	7	y_mi	UNK
4	8	m_mi	N/A
4	8	y_mi	INV

Table 20: Case 3 - (*my_names* × *your_names*) KNOWN

DERIVED_KNOWN							
m_key	m_first	m_mi	m_last	y_key	y_first	y_mi	y_last
1	Edgar	F	Codd	5	Jeffrey	D	Ullman
1	Edgar	F	Codd	6	Margo	I	Seltzer
2	Chris	J	Date	5	Jeffrey	D	Ullman
2	Chris	J	Date	6	Margo	I	Seltzer

Table 21: Case 3 - (*my_names* × *your_names*) UNKNOWN

DERIVED_UNKNOWN							
m_key	m_first	m_mi	m_last	y_key	y_first	y_mi	y_last
1	Edgar	F	Codd	7	Chris		Date
1	Edgar	F	Codd	8	David	?	McGoveran
2	Chris	J	Date	7	Chris		Date
2	Chris	J	Date	8	David	?	McGoveran
3	Hugh		Darwen	5	Jeffrey	D	Ullman
3	Hugh		Darwen	6	Margo	I	Seltzer
3	Hugh		Darwen	7	Chris		Date
3	Hugh		Darwen	8	David	?	McGoveran
4	Andrew		Warden	5	Jeffrey	D	Ullman
4	Andrew		Warden	6	Margo	I	Seltzer
4	Andrew		Warden	7	Chris		Date
4	Andrew		Warden	8	David	?	McGoveran

Table 22: Case 3 - MISSING

DERIVED_MISSING			
m_key	y_key	attr	tag
1	7	y_mi	UNK
1	8	y_mi	INV
2	7	y_mi	UNK
2	8	y_mi	INV
3	5	m_mi	UNK
3	6	m_mi	UNK
3	7	m_mi	UNK
3	7	y_mi	UNK
3	8	m_mi	UNK
3	8	y_mi	INV
4	5	m_mi	N/A
4	6	m_mi	N/A
4	7	m_mi	N/A
4	7	y_mi	UNK
4	8	m_mi	N/A
4	8	y_mi	INV

Table 23: Case 4 - (*my_names* × *your_names*) KNOWN

derived_KNOWN							
m_key	m_first	m_mi	m_last	y_key	y_first	y_mi	y_last
1	Edgar	F	Codd	5	Jeffrey	D	Ullman
1	Edgar	F	Codd	6	Margo	I	Seltzer
2	Chris	J	Date	5	Jeffrey	D	Ullman
2	Chris	J	Date	6	Margo	I	Seltzer

Table 24: Case 4 - (*my_names* × *your_names*) UNKNOWN

derived_UNKNOWN							
m_key	m_first	m_mi	m_last	y_key	y_first	y_mi	y_last
1	Edgar	F	Codd	3	Hugh		Darwen
1	Edgar	F	Codd	8	Andrew		Warden
2	Chris	J	Date	3	Hugh		Darwen
2	Chris	J	Date	8	Andrew		Warden
3	Hugh		Darwen	5	Jeffrey	D	Ullman
3	Hugh		Darwen	6	Margo	I	Seltzer
3	Hugh		Darwen	3	Hugh		Darwen
3	Hugh		Darwen	8	Andrew		Warden
4	Andrew		Warden	5	Jeffrey	D	Ullman
4	Andrew		Warden	6	Margo	I	Seltzer
4	Andrew		Warden	3	Hugh		Darwen
4	Andrew		Warden	8	Andrew		Warden

Table 25: Case 4 - MISSING

derived_MISSING			
m_key	y_key	attr	tag
1	7	y_mi	UNK
1	8	y_mi	MIS
2	7	y_mi	UNK
2	8	y_mi	MIS
3	5	m_mi	UNK
3	6	m_mi	UNK
3	7	m_mi	UNK
3	7	y_mi	UNK
3	8	m_mi	UNK
3	8	y_mi	MIS
4	5	m_mi	N/A
4	6	m_mi	N/A
4	7	m_mi	N/A
4	7	y_mi	UNK
4	8	m_mi	N/A
4	8	y_mi	MIS

6.5.5 Set Union

The result of set union relies on duplicate removal. RM/V1 evaluates one null as equal to another and a duplicate [12, p.405]. The KNOWN/UNKNOWN model evaluates missing data values using metadata from the MISSING relation and if missing values for an attribute have the same class and type, they are considered duplicates. The union of my_names and your_names relvars using the four test cases are shown in Tables 26 through 29.

Table 26: Case 1 - $my\_names \cup your\_names$

DERIVED_KNOWN			
key	first	mi	last
1	Edgar	F	Codd
2	Chris	J	Date
5	Jeffrey	D	Ullman
6	Margo	I	Seltzer

DERIVED_UNKNOWN			
key	first	mi	last
3	Hugh		Darwen
4	Andrew		Warden
7	Fabian		Pascal
8	David	?	McGoveran

DERIVED_MISSING		
key	attr	tag
3	mi	UNK
4	mi	N/A
7	mi	UNK
8	mi	INV

The result shown in Table 26 includes all tuples from the my_names relvar in Table 6 and the your_names relvar in Table 7. There are no duplicate or possibly duplicate tuples with or without including the keys in the set union result.

Table 27: Case 2 - $(\pi_{first,mi,last}(my\_names) \cup \pi_{first,mi,last}(your\_names))$

DERIVED_KNOWN		
first	mi	last
Edgar	F	Codd
Jeffrey	D	Ullman
Margo	I	Seltzer

DERIVED_UNKNOWN			
pseudo-key	first	mi	last
101	Hugh		Darwen
102	Andrew		Warden
103	Fabian		Pascal
104	David	?	McGoveran

DERIVED_MISSING		
key	attr	tag
101	mi	UNK
102	mi	N/A
103	mi	UNK
104	mi	INV

The result shown in Table 27 is a relvar created using the projection onto first name, middle initial, and last name from the my_names relvar in Table 8 and the your_names relvar in Table 9. A duplicate of “Jeffrey D Ullman” from the KNOWN relation was removed. See section 6.5.6 for explanation of pseudo-key in UNKNOWN and MISSING.

Table 28: Case 3 - $(\pi_{first,mi,last}(my\_names) \cup \pi_{first,mi,last}(your\_names))$

DERIVED_KNOWN		
first	mi	last
Edgar	F	Codd
Chris	J	Date
Jeffrey	D	Ullman
Margo	I	Seltzer

DERIVED_UNKNOWN			
pseudo-key	first	mi	last
101	Hugh		Darwen
102	Andrew		Warden
103	Chris		Date
104	David	?	McGoveran

DERIVED_MISSING		
key	attr	tag
101	mi	UNK
102	mi	N/A
103	mi	UNK
104	mi	INV

The result shown in Table 28 is a relvar created using the projection onto first name, middle initial, and last name from the my_names relvar in Table 10 and the your_names relvar in Table 11. There is a possibly duplicated tuple.

"Chris J Date" from my_names_KNOWN identifies a single specific individual. "Chris Date" from your_names_UNKNOWN identifies one or more individuals with a middle initial from the domain of 'A' through 'Z'. If the former is removed, the result set is less specific. If the latter is removed, the information content of the result set is reduced. These tuples may be duplicates, but the user must determine if these are the same person.

Table 29: Case 4 - $(\pi_{first,mi,last}(my\_names) \cup \pi_{first,mi,last}(your\_names))$

DERIVED_KNOWN		
first	mi	last
Edgar	F	Codd
Chris	J	Date
Jeffrey	D	Ullman
Margo	I	Seltzer

DERIVED_UNKNOWN			
pseudo-key	first	mi	last
101	Hugh		Darwen
102	Andrew		Warden
103	Andrew		Warden

DERIVED_MISSING		
key	attr	tag
101	mi	UNK
102	mi	N/A
103	mi	MIS

The result shown in Table 29 is created using the projection onto first name, middle initial, and last name from the my_names relvar in Table 12 and the your_names relvar in Table 13.

A duplicate of "Hugh Darwen" from the UNKNOWN relation was removed because his middle initial is missing for the same reason. "Andrew Warden" appears in both UNKNOWN relations and twice in the set union result because his middle initial is missing for different reasons. In the case of "Andrew Warden," each occurrence of the name appears to be distinct. If these tuples are duplicates, it is the result of a data entry error and can be corrected.

6.5.6 Project

Projection onto attributes from a KNOWN/UNKNOWN relvar is consistent with the relational model. If the key is projected, duplicate tuples are not possible. If the key is not projected, missing data values in the UNKNOWN relation must be supported by metadata in the MISSING relation using generated pseudo-keys. Using known data values and the metadata, duplicate tuples identified as exact matches or maybe-matches are removed. Projection onto middle initial using set union of my_names and your_names with the four test cases are shown in Tables 30 through 33.

Table 30: Case 1 - $(\pi_{mi}(\pi_{first,mi,last}(my\_names) \cup \pi_{first,mi,last}(your\_names)))$

DERIVED_KNOWN
mi
F
J
D
I

DERIVED_UNKNOWN	
pseudo-key	mi
101	
102	
103	?

DERIVED_MISSING		
key	attr	tag
101	mi	UNK
102	mi	N/A
103	mi	INV

The result shown in Table 30 projects attribute *mi* from the union of my_names and your_names in Table 26. The missing the middle initial of "Hugh Darwen" is applicable, but unknown. In the projected relvar "Hugh Darwen" is represented in the UNKNOWN and MISSING relations by pseudo-key 101. A projected middle initial for "Fabian Pascal" duplicates this tuple and is also represented by pseudo-key 101. "Andrew Warden" is a fictional person who does not have a middle initial (inapplicable) and is identified by pseudo-key 102. The question mark represents applicable, but invalid input for "David McGoveran."

Table 31: Case 2 - $(\pi_{mi}(\pi_{first,mi,last}(my\_names) \cup \pi_{first,mi,last}(your\_names)))$

DERIVED_KNOWN
mi
F
D
I

DERIVED_UNKNOWN	
pseudo-key	mi
101	
102	
103	?

DERIVED_MISSING		
key	attr	tag
101	mi	UNK
102	mi	N/A
103	mi	INV

The result shown in Table 31 projects attribute *mi* from the first name, middle initial, last name union of my_names and your_names in Table 27. A duplicate of "Jeffrey D Ullman" was removed by the set union operation from the KNOWN relation. The representation of missing data values in the UNKNOWN and MISSING relations is the same as above in Table 30.

Table 32: Case 3 - $(\pi_{mi}(\pi_{first,mi,last}(my\_names) \cup \pi_{first,mi,last}(your\_names)))$

DERIVED_KNOWN
mi
F
J
D
I

DERIVED_UNKNOWN	
pseudo-key	mi
101	
102	
103	?

DERIVED_MISSING		
key	attr	tag
101	mi	UNK
102	mi	N/A
103	mi	INV

The result shown in Table 32 projects attribute *mi* from the first name, middle initial, last name union of my_names and your_names in Table 28. Some of the data values in this test case have changed, but the projection onto middle initial is identical to those shown above in Table 31.

Table 33: Case 4 - $(\pi_{mi}(\pi_{first,mi,last}(my\_names) \cup \pi_{first,mi,last}(your\_names)))$

DERIVED_KNOWN
mi
F
J
D
I

DERIVED_UNKNOWN	
pseudo-key	mi
101	
102	
103	

DERIVED_MISSING		
key	attr	tag
101	mi	UNK
102	mi	N/A
103	mi	MIS

The result shown in Table 33 projects attribute *mi* from the first name, middle initial, last name union of my_names and your_names in Table 29. A duplicate tuple for "Hugh Darwen" was removed by the set union operation. The missing middle initial of "Hugh Darwen" is applicable, but unknown. In the projected relvar "Hugh Darwen" is represented in the UNKNOWN and MISSING relations by pseudo-key 101. The middle initial for "Andrew Warden" is projected twice. "Andrew Warden" is a fictional person who does not have a middle initial (inapplicable) and is identified by pseudo-key 102. Another "Andrew Warden" has a middle initial, but withheld it and is identified by pseudo-key 103.

6.5.7 Set Difference

The set difference between two relvars, is created by duplicate removal. If a tuple in the first relvar is duplicated in the second, it is removed. KNOWN values are compared for equality and UNKNOWN values are evaluated using metadata from the MISSING relation. The difference between my_names and your_names relvars using the four test cases are shown in Tables 34 through 37.

Table 34: Case 1 - ($my\_names - your\_names$)

DERIVED_KNOWN			
key	first	mi	last
1	Edgar	F	Codd
2	Chris	J	Date

DERIVED_UNKNOWN			
key	first	mi	last
3	Hugh		Darwen
4	Andrew		Warden

DERIVED_MISSING		
key	attr	tag
3	mi	UNK
4	mi	N/A

The result shown in Table 34 includes all tuples from the my_names relvar in Table 6 because both keys and names are unique it is not possible for a tuple to be duplicated in the your_names relvar from Table 7.

Table 35: Case 2 - $(\pi_{first,mi,last}(my\_names) - \pi_{first,mi,last}(your\_names))$

DERIVED_KNOWN		
first	mi	last
Edgar	F	Codd

DERIVED_UNKNOWN			
pseudo-key	first	mi	last
101	Hugh		Darwen
102	Andrew		Warden

DERIVED_MISSING		
key	attr	tag
101	mi	UNK
102	mi	N/A

The result shown in Table 35 is derived using the projection onto first name, middle initial, and last name from my_names in Table 8. “Jeffrey D Ullman” is an exact match with a tuple in the projection onto first name, middle initial, and last name from your_names in Table 9 and is removed.

Table 36: Case 3 - $(\pi_{first,mi,last}(my\_names) - \pi_{first,mi,last}(your\_names))$

DERIVED_KNOWN		
first	mi	last
Edgar	F	Codd
Chris	J	Date

DERIVED_UNKNOWN			
pseudo-key	first	mi	last
101	Hugh		Darwen
102	Andrew		Warden

DERIVED_MISSING		
key	attr	tag
101	mi	UNK
102	mi	N/A

The result shown in Table 36 is the projection onto first name, middle initial, and last name from the my_names relvar in Table 10. No tuple in your_names from Table 11 matches a tuple in my_names.

The result includes “Chris J Date” from my_names_KNOWN because “Chris Date” in your_names_UNKNOWN is a maybe-match, but not a duplicate in the KNOWN/UNKNOWN model. The user may use external information or domain keys to determine these tuples are duplicates and correct the your_names database.

Table 37: Case 4 - $(\pi_{first,mi,last}(my\_names) - \pi_{first,mi,last}(your\_names))$

DERIVED_KNOWN		
first	mi	last
Edgar	F	Codd
Chris	J	Date

DERIVED_UNKNOWN			
pseudo-key	first	mi	last
101	Andrew		Warden

DERIVED_MISSING		
key	attr	tag
101	mi	N/A

The result shown in Table 37 is derived using the projection onto first name, middle initial, and last name from my_names in Table 12 and from your_names in Table 13.

"Hugh Darwen" in my_names_UNKNOWN is a duplicate of a tuple in your_names_UNKNOWN with the same first name, last name, and reason for a missing middle initial and is removed. "Andrew Warden" is also in both UNKNOWN relations, but is not removed as a duplicate because the middle initial is missing for different reasons.

6.5.8 Restrict

Restriction of tuples in a KNOWN/UNKNOWN relvar can be constrained to exact matches (consistent with the relational model) or a MAYBE comparison operator can be used to select only maybe-matches. If exact matches and maybe-matches are specified by a query, the result set includes both kinds of matches. Restrictions of my_names or the union of my_names and your_names from the four test cases are shown in Tables 38 through 45.

Table 38: Case 1 - ($\sigma_{mi\ =\ 'F'}(my\_names)$)

DERIVED_KNOWN			
key	first	mi	last
1	Edgar	F	Codd

DERIVED_UNKNOWN			
key	first	mi	last

DERIVED_MISSING		
key	attr	tag

The result shown in Table 38 restricts tuples from my_names in Table 6 to those with middle initials equal to 'F'. "Edgar F Codd" has the middle initial 'F' and is an exact match. "Hugh Darwen" has an unknown middle initial and "Andrew Warden" does not have a middle initial (inapplicable).

Table 39: Case 1 - ($\sigma_{mi\ MAYBE\ =\ 'F'}(my\_names)$)

DERIVED_KNOWN			
key	first	mi	last

DERIVED_UNKNOWN			
key	first	mi	last
3	Hugh		Darwen

DERIVED_MISSING		
key	attr	tag
3	mi	UNK

The result shown in Table 39 restricts tuples from my_names in Table 6 to those with middle initials that may be equal to 'F'. "Hugh Darwen" has an unknown middle initial that could be 'F' and is a maybe-match. "Andrew Warden" does not have a middle initial (inapplicable) and cannot match.

Table 40: Case 1 - $(\sigma_{mi\ =\ 'F'\ \vee\ mi\ MAYBE\ =\ 'F'}(my\_names))$

DERIVED_KNOWN			
key	first	mi	last
1	Edgar	F	Codd

DERIVED_UNKNOWN			
key	first	mi	last
3	Hugh		Darwen

DERIVED_MISSING		
key	attr	tag
3	mi	UNK

The result show in Table 40 shows that a restriction of tuples that are exact matches or maybe-matches gives the set union of the two result sets in Tables 38 and 39.

Table 41: Case 2 - ($\sigma_{m\_mi\ =\ y\_mi}(my\_names\ \times\ your\_names)$) KNOWN

DERIVED_KNOWN							
m_key	m_first	m_mi	m_last	y_key	y_first	y_mi	y_last
2	Jeffrey	D	Ullman	5	Jeffrey	D	Ullman

Table 42: Case 2 - ($\sigma_{m\_mi\ =\ y\_mi}(my\_names\ \times\ your\_names)$) UNKNOWN

DERIVED_UNKNOWN							
m_key	m_first	m_mi	m_last	y_key	y_first	y_mi	y_last
1	Edgar	F	Codd	7	Fabian		Pascal
1	Edgar	F	Codd	8	David	?	McGoveran
2	Jeffrey	D	Ullman	7	Fabian		Pascal
2	Jeffrey	D	Ullman	8	David	?	McGoveran
3	Hugh		Darwen	5	Jeffrey	D	Ullman
3	Hugh		Darwen	6	Margo	I	Seltzer
3	Hugh		Darwen	7	Fabian		Pascal
3	Hugh		Darwen	8	David	?	McGoveran

Table 43: Case 2 - MISSING

DERIVED_MISSING			
m_key	y_key	attr	tag
1	7	y_mi	UNK
1	8	y_mi	INV
2	7	y_mi	UNK
2	8	y_mi	INV
3	5	m_mi	UNK
3	6	m_mi	UNK
3	7	m_mi	UNK
3	7	y_mi	UNK
3	8	m_mi	UNK
3	8	y_mi	INV

The result shown in Table 41 restricts tuples from the Cartesian product of my_names and your_names in Table 17 to those with a middle initial in my_names that is equal or maybe equal to a middle initial in your_names. “Jeffrey D Ullman” is in both relvars and is the only exact match. “Andrew Warden” does not have a middle initial and cannot match.

Table 44: Case 3 - $(\sigma_{first\ ='Chris'\ \wedge\ last='Date'\ \vee\ mi\ MAYBE\ =\ 'J'}(my\_names \cup your\_names))$

DERIVED_KNOWN			
key	first	mi	last
2	Chris	J	Date

DERIVED_UNKNOWN			
key	first	mi	last
3	Hugh		Darwen
7	Chris		Date
8	David	?	McGoveran

DERIVED_MISSING		
key	attr	tag
3	mi	UNK
7	mi	UNK
8	mi	INV

The result shown in Table 44 restricts tuples in the set union of names and keys from my_names in Table 10 and your_names in Table 11 to those with a first name equal to “Chris” and a last name equal to “Date” or those with a middle initial that may be ’J’.

Including the key in the union means there can be no duplicate tuples, but the user may determine from the metadata in the MISSING relation that “Chris J Date” and “Chris Date” are the same person.

Table 45: Case 4 - $(\sigma_{my\_names.key\ =\ \pi_{key}(\sigma_{attr\ =\ 'mi'\ \wedge\ tag\ =\ 'N/A'}(my\_names\_MISSING))}(my\_names))$

DERIVED_KNOWN			
key	first	mi	last

DERIVED_UNKNOWN			
key	first	mi	last
4	Andrew		Warden

DERIVED_MISSING		
key	attr	tag
4	mi	N/A

The result show in Table 45 restricts the names from my_names in Table 12 to those that do not have a middle initial. These are the names missing a middle initial because the attribute *mi* does not apply to the name. “Andrew Warden” matches the query because he is a fictional character and does not have a middle initial.

6.6 Arithmetic operations using missing data

There are five SQL aggregate functions: count, sum, avg, min, and max. While data values may be missing for different reasons and for different attributes, aggregate values are determined using the known values for a specified attribute from the KNOWN and the UNKNOWN relations. Using the metadata from the MISSING relation applicable, invalid, inapplicable and unknowable missing data values can be counted, but cannot be used in a meaningful way to determine a summation, average, minimum or maximum value. All missing data values are ignored in arithmetic. The relvar triple "ages" shown in Table 46 contains test data for the SQL aggregate functions.

Table 46: ages Relation Variable

ages_KNOWN		
first	last	age
Edgar	Codd	79
Chris	Date	71
Margo	Seltzer	49
Jeffrey	Ullman	69

ages_UNKNOWN		
first	last	age
Hugh	Darwen	
David	McGoveran	-1
	Parker	27
Fabian	Pascal	
Andrew	Warden	

ages_MISSING		
key	attr	tag
Darwen	age	UNK
McGoveran	age	INV
Parker	first	UNK
Pascal	age	UNK
Warden	age	N/A

6.6.1 Count

The known data values can be counted for the column *age* from the ages_KNOWN and ages_UNKNOWN relations giving Table 47.

The unknown data values can be counted for the attribute *age* using metadata

Table 47: $count_{age}(ages)$

COUNT_KNOWN(age)
5

COUNT_UNKNOWN(age)

COUNT_MISSING(age)

"Parker" is missing a first name and is in the unknown relation, but has a known age. There are five rows with known *age* values.

from the MISSING relation giving Table 48 or a similar query can count those who have no age giving Table 49

Table 48: $count_{tag}(\sigma_{attr\ ='age'\ \wedge\ tag\ \neq\ 'N/A'}(ages\_MISSING))$

COUNT_KNOWN(tag)
3

COUNT_UNKNOWN(tag)

COUNT_MISSING(tag)

There are three rows with a *tag* that indicates an applicable but unknown *age* value. "Parker" is missing a first name with an 'UNK' *tag*, but has a known age.

Table 49: $count_{tag}(\sigma_{attr\ ='age'\ \wedge\ tag\ =\ 'N/A'}(ages\_MISSING))$

COUNT_KNOWN(tag)
1

COUNT_UNKNOWN(tag)

COUNT_MISSING(tag)

There is one row with a *tag* that indicates an inapplicable *age* value. “Andrew Warden” is a fictional character who does not have a date of birth or age.

More complex queries to match missing data values can be written selecting tags for classes of missing data using system metadata from Table 2.

6.6.2 Sum, Avg, Min, and Max

The known *age* attributes in the “ages” relvar are totaled using the five rows with known *age* values as shown in Table 50 .

Table 50: $sum_{age}(ages)$

SUM_KNOWN(age)
295

SUM_UNKNOWN(age)

SUM_MISSING(age)

The average derived from the count and summation of known data values and the minimum or maximum selected from known values work in a similar way giving only a known result using known data values.

CHAPTER 7 Solution Implementation and Verification

In order to implement the research hypothesis described in chapter 6, SQL needs an enhanced comparison operator to select rows with unknown data values as possible matches (maybe-matches). Other enhancements are needed to create, update, and delete tables using the KNOWN/UNKNOWN model. The capabilities and constraints of an implementation of these extensions are specified in Appendix B.

7.1 Design

The KNOWN/UNKNOWN model for missing data can be implemented using either one table to hold both known and unknown rows of data or separate tables that closely follow the hypothesis model. The metadata for missing data values must be stored in a separate table. Either approach will meet the conceptual model from the user's point of view, but there is a difference in processing one table twice versus each of two smaller tables once.

Each approach can be implemented to produce a result set of three relations. The advantages and disadvantages relate to performance and compatibility with MySQL, the target DBMS. In all cases database client programs must be aware of the model and able to process a relation variable result set containing the three relations. The integrated known and unknown table design requires less intrusive modification to MySQL because it allows changes to be made at a higher level of abstraction requiring fewer changes to MySQL's implementation.

7.1.1 Integrated known and unknown tables

An implementation that uses one table to store the complete and incomplete data is well suited for the case when queries for exact matches are prevalent. If there are no missing data values in a result set, the unknown and missing relations are returned as empty tables. This approach may also reduce the necessity for temporary tables used to hold intermediate results.

1. *Initialize query result set relvar*

    - Create empty integrated table for RESULT_KNOWN and RESULT_UNKNOWN
    - Create empty RESULT_MISSING table

2. *Check for Attributes Projected*

    - Compare the columns specified in the query to the attributes defined for the integrated table.
    - Create a list of columns to be projected.
    - Create a list of columns not to be projected.
    - Restrict the missing data table using the list of columns to be projected and store these rows in the RESULT_MISSING table.

3. *Check for Exact match Restrictions*

   - If the search query has an exact match criteria, search the integrated table for exact matches.
   - Use the list of projected attributes to eliminate columns in each exact match row.
   - Store exact match rows not missing any data values and exact match rows with missing data values in the integrated known and unknown table of the result set.

4. *Check for Maybe-match Restrictions*

   - If the search query has a maybe-match criteria, the missing table is searched by attribute name for appropriate missing data tags and keys which are retrieved for maybe-matches.
   - Use these keys to retrieve maybe-match rows from the integrated table and store the key, attribute, and tag triples in the RESULT_MISSING table.
   - Use the list of projected attributes to eliminate columns in each row of maybe-matches.
   - Store these rows in the integrated known and unknown table of the result set.

5. *Check for additional processing in the case of intermediate results*

    - If the integrated and missing tables are an intermediate result, continue processing the query.
    - If the integrated and missing tables are a complete result set, return the results to the user.

6. *Return result set relvar to the user*

    - Select rows from the integrated known and unknown table without a key in the RESULT_MISSING table as the RESULT_KNOWN table.
    - Select rows from the integrated known and unknown table with a key in the RESULT_MISSING table as the RESULT_UNKNOWN table.
    - The missing table created by this process is the RESULT_MISSING table.

The additional processing to present data to the user as separate tables is done once. Separate tables of complete information and incomplete information are presented to the user as derived tables. Using this approach, the extensions to SQL may be implemented in MySQL or its client by automated rewriting of the queries into standard SQL.

Advantages

Queries search a single table for both exact matches and maybe-matches. If an item is updated and made complete, it does not have to be deleted from the unknown table and added to the known table. It is possible and straightforward to support both the KNOWN/UNKNOWN model and SQL nulls in the same database. Backward compatibility with missing data represented using null provides a migration path for databases in which existing queries function indefinitely.

Disadvantages

All query result sets must be separated into known and unknown derived tables for presentation to the user even for the case when the entire relvar is retrieved. If the extensions to SQL are implemented in the client and rewritten into standard SQL, intermediate results must be managed outside of MySQL.

7.1.2 Separate known and unknown tables

An implementation that strictly follows the conceptual model stores rows with missing data values in a table separate from known data values. This requires more processing than the integrated table approach.

1. *Initialize query result set relvar*
   - Create empty RESULT_KNOWN table
   - Create empty RESULT_UNKNOWN table
   - Create empty RESULT_MISSING table
2. *Check for Attributes Projected*
   - Compare the columns specified in the query to the attributes defined for the known and unknown tables.

- Create a list of columns to be projected.
- Create a list of columns not to be projected.
- Restrict the missing data table using the list of columns to be projected and store these rows in the RESULT_MISSING table.

3. *Check for Exact match Restrictions in the known table*

    - If the search query has an exact match criteria, search the known table for exact matches.
    - Use the list of projected attributes to eliminate columns in each exact match row.
    - Store exact match rows in the RESULT_KNOWN table.

4. *Check for Exact match Restrictions in the unknown table*

    - If the search query has an exact match criteria, search the unknown table for exact matches.
    - Use the list of projected attributes to eliminate columns in each exact match row.
    - Use the RESULT_MISSING table's key, attribute, and tag information to determine if there is a missing data column and store the row in the RESULT_KNOWN table or RESULT_UNKNOWN table.

5. *Check for Maybe-match Restrictions*

- If the search query has a maybe-match criteria, the metadata table is searched by attribute name for appropriate missing data tags and keys are retrieved for maybe-matches.
- Use the list of projected attributes to eliminate columns in each maybe-match row.
- Use the RESULT_MISSING table's key, attribute, and tag information to determine if there is a missing data column and store the row in the RESULT_KNOWN table or RESULT_UNKNOWN table.

6. *Check for additional processing in the case of intermediate results*
   - If the known, unknown and missing tables are an intermediate result, continue processing the query.
   - If the known, unknown and missing tables are a complete result set, return the results to the user.
7. *Return result set relvar to the user*
   - RESULT_KNOWN table in the result set is presented to the user.
   - RESULT_UNKNOWN table in the result set is presented to the user.
   - RESULT_MISSING table in the result set is presented to the user.

Advantages

The separate table model is less complex once the required modifications that implement it are made to MySQL. MySQL efficiently manages intermediate result tables needed for duplicate removal and subquery. If the key columns are not part if the table projection and not in the result, synchronizing the required pseudo-keys shared by the unknown and missing tables is less complex. Data presentation is straightforward. The modified MySQL will accept a single query and return a result set of known, unknown, and missing tables even when one or more of these is empty.

Disadvantages

Storing known and unknown information in separate tables and managing both tables as a single data store requires modification to MySQL. The separate table method depends on keys being unique across both the known and unknown tables and in the case of a key that auto_increments, it must be shared by both tables. If a column of missing data is removed from a table many rows of data may need to be moved from the unknown table to the known table. Queries that do not rely on table indexes must scan two tables for rows that match the search criteria. If missing data values are classified as non-applicable or unknowable for security reasons, all rows are in the unknown table and represented by one or more rows in the missing data table. In this case, storage requirements for the database increase and may impact performance.

7.1.3 Missing values metadata table

There is a tuple in this relation for each attribute with a missing data value. Information explaining the classification of the missing data and why it is missing is part of the database schema. Creating and maintaining this relvar component can be done using either the integrated table or the separate table approach.

7.2 Implementation

A subset of the KNOWN/UNKNOWN model for missing data using the MySQL embedded server was implemented using the C programming language [6]. This implementation integrates the known and unknown tables in a single relation and allows either an invalid data value or the SQL standard null as a place holder for missing data. For tables that allow only null as a missing data indicator, queries that refer to null as well as those that use the metadata in the _MISSING table are supported. The default for non-key attributes is to allow missing data including an invalid value. The MISSING metadata table is available as a component of each relvar and result. SQL is extended to include a MAYBE modifier for comparison operators to allow the user to include maybe-matches in the query result set.

7.2.1 MyKU client

MyKU is a client program with a user interface that accepts SQL for exact and/or maybe-matches and presents results as a KNOWN/UNKNOWN relvar or a single relation for tables that use nulls. It has a lexical scanner generated using Flex [55] and a parser generated using Bison [28] for SQL select statements derived from a grammar written for a subset of SQL [37]. As SQL statements are parsed, MyKU builds an abstract syntax tree (AST) which is used by the MyKU select query rewrite logic to transform extended SQL into standard SQL. For every extended SQL statement entered MyKU sends one, two or three queries to the MySQL embedded server. A query may be for the KNOWN/UNKNOWN model's relvar triple or for one of its component relations. The result relvar for the KNOWN/UNKNOWN model is presented to the user as a relvar triple which may include empty component relations. MyKU appears to users much like the MySQL client with output in the same basic format [30].

The MyKU flowchart for client component input and result presentation is shown in Figure 3 and the source code is given in Appendix C. At program start, MyKU connects to the embedded MySQL server and enters an input loop that continues until an exit statement is entered. Each user statement is sent to the scanner and parsed to determine if it is extended SQL. If the input is not extended SQL, it is sent unchanged to MySQL. Otherwise, the input is sent to the query rewrite component of MyKU and transformed into multiple SQL statements that are each sent to MySQL. MyKU accepts the result tables from MySQL and presents the output to the user.

The MyKU flowchart for select query rewrite is shown in Figure 4 and the source code is given in Appendix D. To transform extended SQL into standard SQL, branches of the AST for *select*, *from*, and *where* are visited and information is gathered about columns, tables, and rows. All queries that return data have columns for projection and may or may not restrict rows. Unions and products have more than one table input and require additional handling. The rewritten queries for the known, unknown, and missing tables are returned to be sent to MySQL.

MyKU client component flowchart

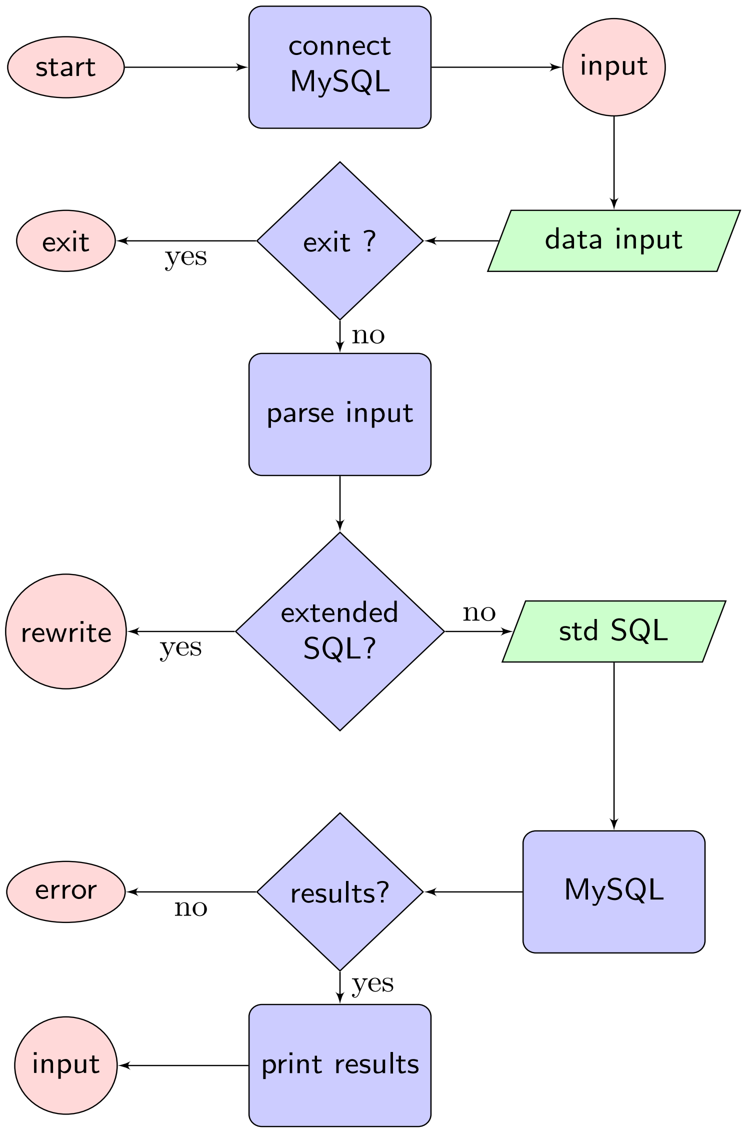


Figure 3: MyKU user input component flowchart

MyKU select query rewrite flowchart

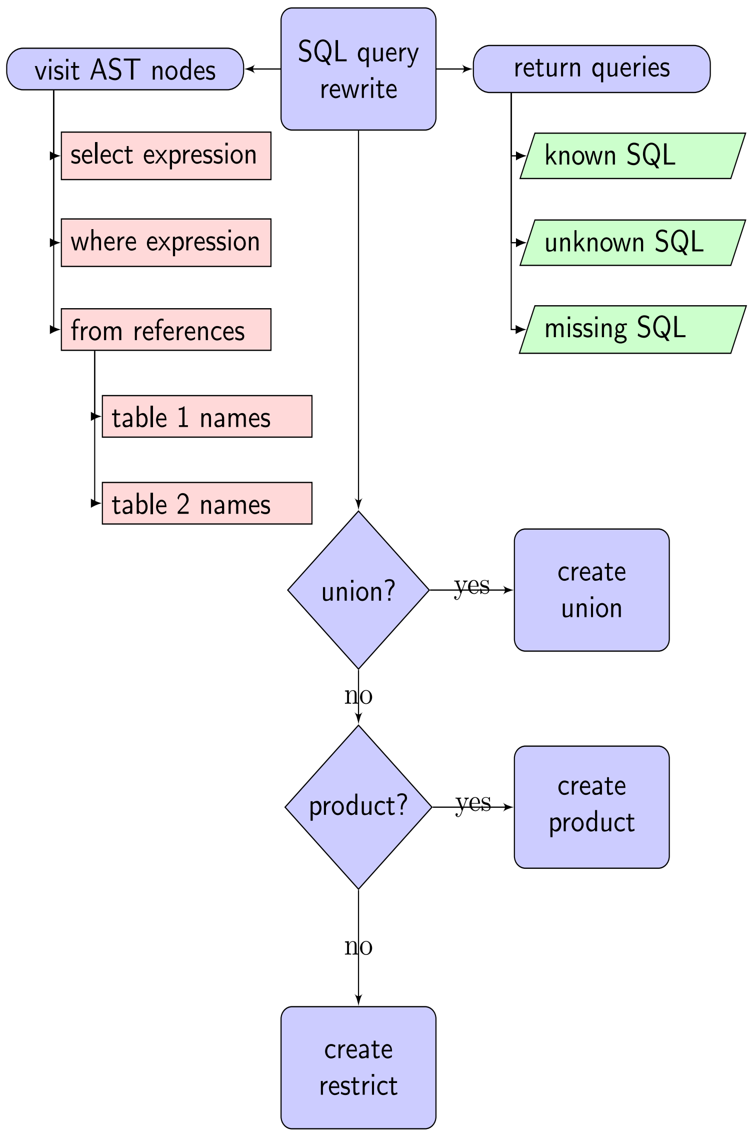


Figure 4: MyKU query rewrite component flowchart

Extended SQL using standard SQL

The KNOWN/UNKNOWN model uses the term “maybe” to modify the relational comparison operators if possible matches between query search terms and attributes missing data values are sought. These maybe-matches only make sense for missing values that are applicable and could match once the missing data is known. This is implemented using the keys returned from a search of the _MISSING table for the attribute named in the query with appropriate missing data tag values in order to retrieve maybe-matches for the _UNKNOWN table.

The extended SQL query for the test case from Table 39 in section 7.3.5 is shown in Figure 5.

```
select * from my_names where mi MAYBE = ’F’;
```

Figure 5: Restrict my_names to middle initials that maybe ’F’

The match criteria of this query is for middle initial (mi) values that are missing, but could be equal to ’F’ if they were known. All rows that may match are in the integrated known and unknown data table, but to find these rows a key value is needed from the _MISSING table where the missing attribute is “mi” and the missing data tags match UNK, NYE, UND, INV or MIS. These rows are used to create the _UNKNOWN table in the result. The rows from the _MISSING table are also selected using the same attribute and tags to create the _MISSING table in the result. There are no exact matches so the _KNOWN table in the result is empty.

The tags UNK, NYE, UND, INV and MIS are used to select applicable missing data and to avoid inapplicable or unknowable data represented by tags N/A, REM or NIL. A query that can find these keys is used as a subquery in standard SQL shown in Figure 6 and the MyKU result of this query from section 6.5.8 is shown in Figure 28.

```
# result_UNKNOWN
select * from my_names
 where pk in (select pk from my_names_missing
               where attr = 'mi'
                 and tag in ('UNK', 'NYE', 'UND', 'INV', 'MIS'));

# result_MISSING
select * from my_names_missing
 where attr = 'mi' and
       tag in ('UNK', 'NYE', 'UND', 'INV', 'MIS'));
```

Figure 6: Restrict my_names to middle initials that maybe 'F'

7.2.2 MyKU intermediate results

Each extended SQL query processed by MyKU needs to be rewritten as a query for the _KNOWN table, as a similar query for the _UNKNOWN table, and as a query for the _MISSING table often with a subquery. In some cases more than one subquery result may be needed as part of an intermediate results, but the extended SQL query processing functions are not recursive.

One solution to this problem is to have MyKU generate standard SQL queries that create derived tables (see Appendix E) and views (see Appendices F and G) to serve as intermediate results, but this has the potential to create tables out of synch with the MySQL integrated and missing tables. While MyKU was under development, these tables were created using SQL scripts and changed as needed, but were not incorporated into the MyKU program. This approach works well when MyKU is used to search for matches without subqueries, but will not be feasible when the integrated KNOWN/UNKNOWN relvar is to be updated using inserts and deletes. The fix for the intermediate results problem is planned as future work in chapter 10.1.1 and requires implementing the KNOWN/UNKNOWN model at a lower level within the DBMS.

7.2.3 MyKU duplicate removal

While SQL does not require tables to have a key, the KNOWN/UNKNOWN model requires a primary key for each table and it depends on the key to connect rows with missing data values to the _MISSING table.

The presence of unique keys in MyKU's intermediate results means all rows are unique during duplicate removal. Using projection to remove key attributes breaks the connections between the rows in the _UNKNOWN and _MISSING tables. To avoid this problem, if not present in a query, the primary key is forced into the projection and results include duplicate rows. What is needed to solve this problem is an implementation of the KNOWN/UNKNOWN model within the DBMS server where the intermediate results are stored in data structures easily accessible by the C programming language. This approach would allow actual keys or pseudo-keys to be stored with intermediate results, but not processed as data during duplicate removal. The fix for the duplicate removal problem is planned as future work in chapter 10.1.2 and requires implementing the KNOWN/UNKNOWN model at a lower level within the DBMS.

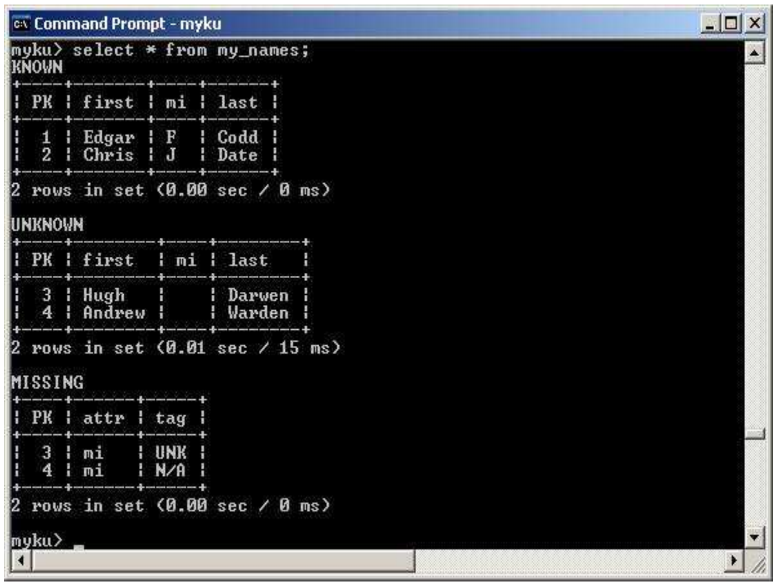
7.3 Verification

The verification of the KNOWN/UNKNOWN implementation compares the results defined in chapter 6 to results from MyKU using the same test data shown in section 6.5.3 and as queried by MyKU in Figures 7 through 14 below.

Four of the five basic relational operations implemented in MySQL are implemented in MyKU and are represented in the verification process. The missing operation is set difference.

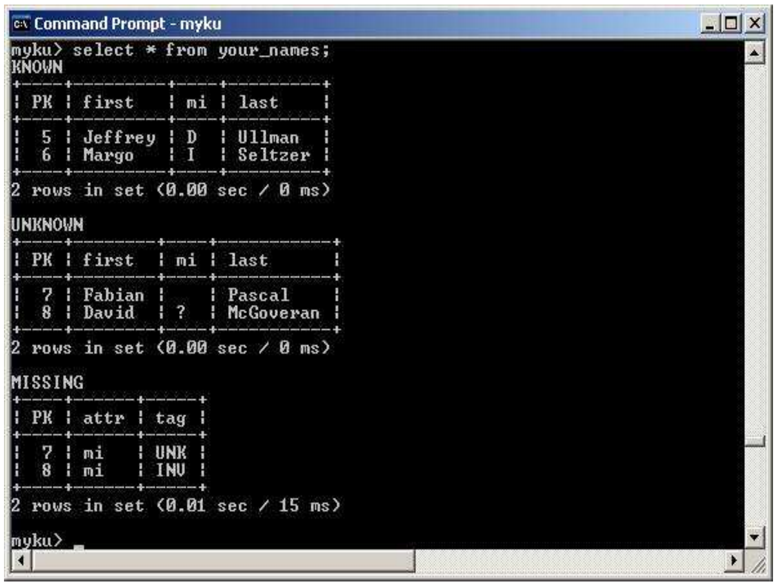
MySQL does not have a set difference operator and there is no reference to set difference in the MySQL manual [52]. A search of the Internet for how to take the difference between two tables using MySQL found several suggested workarounds, but the INTERSECT and EXCEPT options for SQL JOIN [65] are not available in MySQL. Set difference is an issue to be resolved as future work.

7.3.1 Four test cases



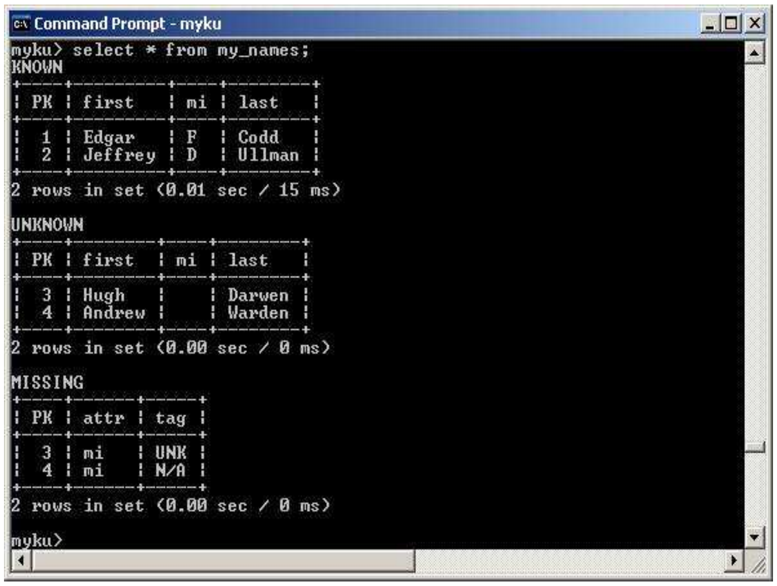
```
myku> select * from my_names;
KNOWN
+----+-------+----+------+
| PK | first | mi | last |
+----+-------+----+------+
|  1 | Edgar | F  | Codd |
|  2 | Chris | J  | Date |
+----+-------+----+------+
2 rows in set (0.00 sec / 0 ms)

UNKNOWN
+----+--------+----+--------+
| PK | first  | mi | last   |
+----+--------+----+--------+
|  3 | Hugh   |    | Darwen |
|  4 | Andrew |    | Warden |
+----+--------+----+--------+
2 rows in set (0.01 sec / 15 ms)

MISSING
+----+------+-----+
| PK | attr | tag |
+----+------+-----+
|  3 | mi   | UNK |
|  4 | mi   | N/A |
+----+------+-----+
2 rows in set (0.00 sec / 0 ms)

myku> _
```

Figure 7: Case 1 (a) my_names relvar



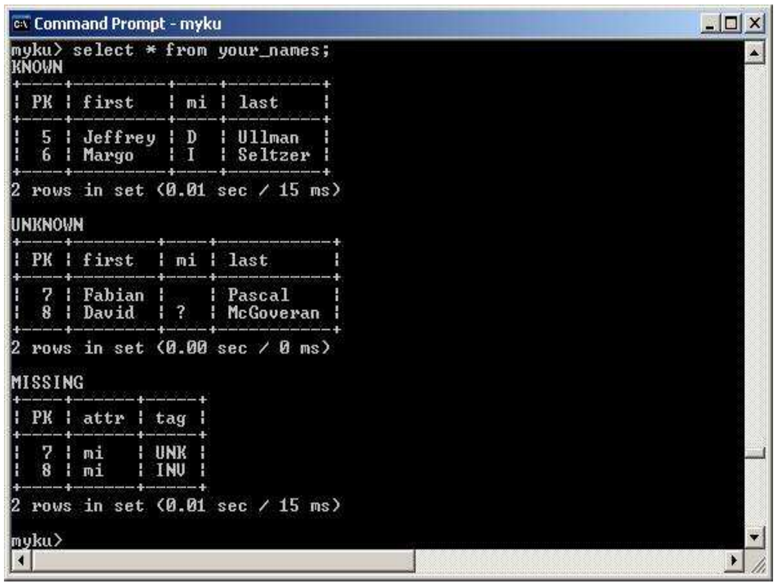
```
myku> select * from your_names;
KNOWN
+----+---------+----+---------+
| PK | first   | mi | last    |
+----+---------+----+---------+
|  5 | Jeffrey | D  | Ullman  |
|  6 | Margo   | I  | Seltzer |
+----+---------+----+---------+
2 rows in set (0.00 sec / 0 ms)

UNKNOWN
+----+--------+----+-----------+
| PK | first  | mi | last      |
+----+--------+----+-----------+
|  7 | Fabian |    | Pascal    |
|  8 | David  | ?  | McGoveran |
+----+--------+----+-----------+
2 rows in set (0.00 sec / 0 ms)

MISSING
+----+------+-----+
| PK | attr | tag |
+----+------+-----+
|  7 | mi   | UNK |
|  8 | mi   | INV |
+----+------+-----+
2 rows in set (0.01 sec / 15 ms)

myku> _
```

Figure 8: Case 1 (b) your_names relvar



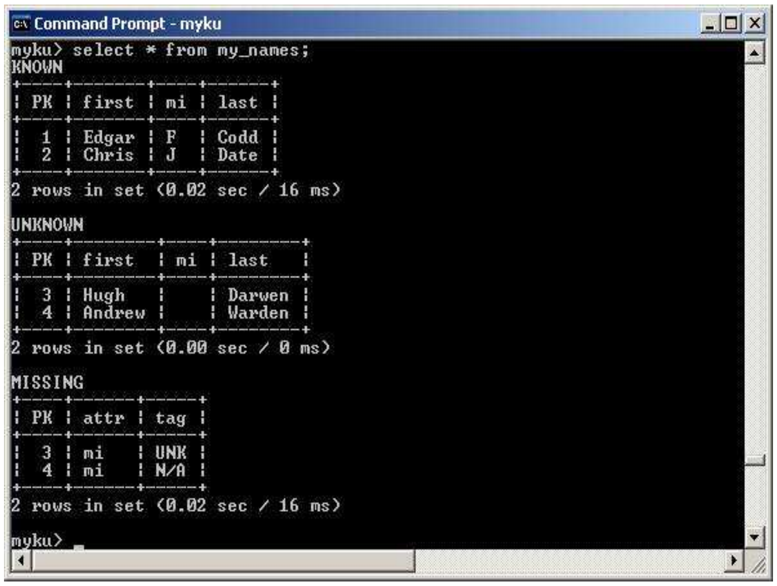
```
myku> select * from my_names;
KNOWN
+----+---------+----+--------+
| PK | first   | mi | last   |
+----+---------+----+--------+
|  1 | Edgar   | F  | Codd   |
|  2 | Jeffrey | D  | Ullman |
+----+---------+----+--------+
2 rows in set (0.01 sec / 15 ms)

UNKNOWN
+----+--------+----+--------+
| PK | first  | mi | last   |
+----+--------+----+--------+
|  3 | Hugh   |    | Darwen |
|  4 | Andrew |    | Warden |
+----+--------+----+--------+
2 rows in set (0.00 sec / 0 ms)

MISSING
+----+------+-----+
| PK | attr | tag |
+----+------+-----+
|  3 | mi   | UNK |
|  4 | mi   | N/A |
+----+------+-----+
2 rows in set (0.00 sec / 0 ms)

myku>
```

Figure 9: Case 2 (a) my_names relvar



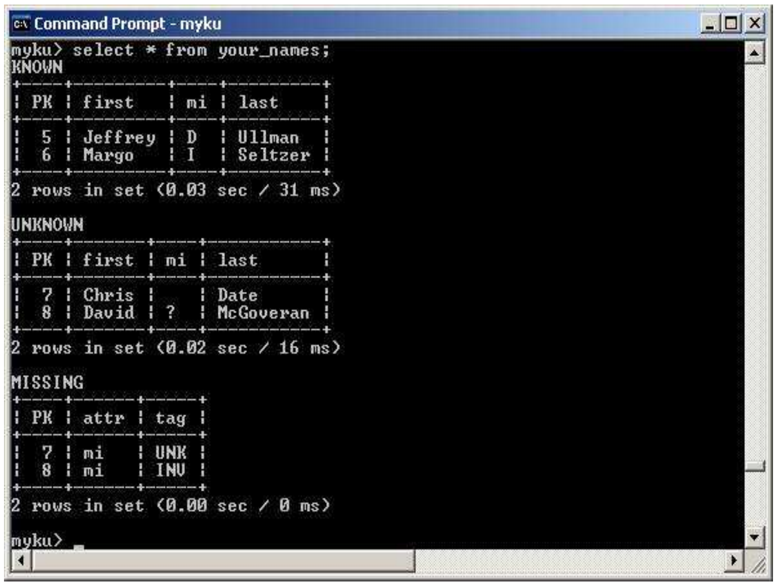
```
myku> select * from your_names;
KNOWN
+----+---------+----+---------+
| PK | first   | mi | last    |
+----+---------+----+---------+
|  5 | Jeffrey | D  | Ullman  |
|  6 | Margo   | I  | Seltzer |
+----+---------+----+---------+
2 rows in set (0.01 sec / 15 ms)

UNKNOWN
+----+--------+----+-----------+
| PK | first  | mi | last      |
+----+--------+----+-----------+
|  7 | Fabian |    | Pascal    |
|  8 | David  | ?  | McGoveran |
+----+--------+----+-----------+
2 rows in set (0.00 sec / 0 ms)

MISSING
+----+------+-----+
| PK | attr | tag |
+----+------+-----+
|  7 | mi   | UNK |
|  8 | mi   | INV |
+----+------+-----+
2 rows in set (0.01 sec / 15 ms)

myku>
```

Figure 10: Case 2 (b) your_names relvar



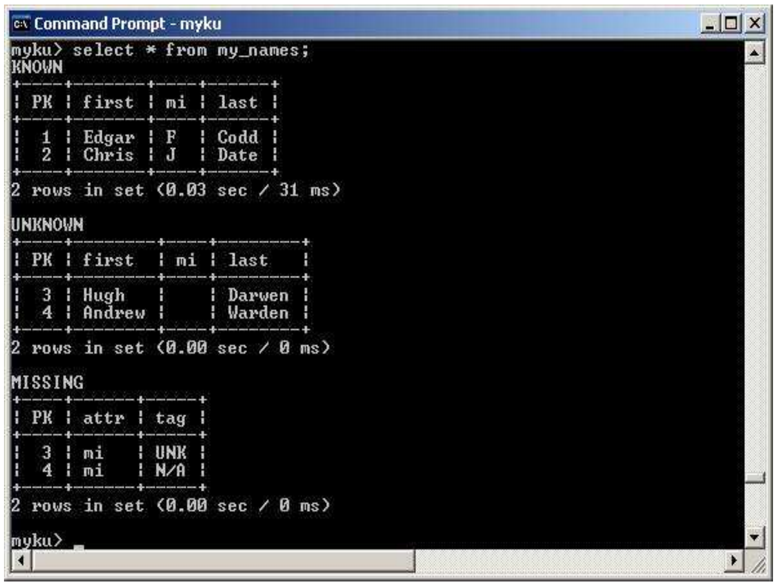
```
myku> select * from my_names;
KNOWN
+----+--------+----+------+
| PK | first  | mi | last |
+----+--------+----+------+
|  1 | Edgar  | F  | Codd |
|  2 | Chris  | J  | Date |
+----+--------+----+------+
2 rows in set (0.02 sec / 16 ms)

UNKNOWN
+----+---------+----+--------+
| PK | first   | mi | last   |
+----+---------+----+--------+
|  3 | Hugh    |    | Darwen |
|  4 | Andrew  |    | Warden |
+----+---------+----+--------+
2 rows in set (0.00 sec / 0 ms)

MISSING
+----+------+-----+
| PK | attr | tag |
+----+------+-----+
|  3 | mi   | UNK |
|  4 | mi   | N/A |
+----+------+-----+
2 rows in set (0.02 sec / 16 ms)

myku> _
```

Figure 11: Case 3 (a) my_names relvar



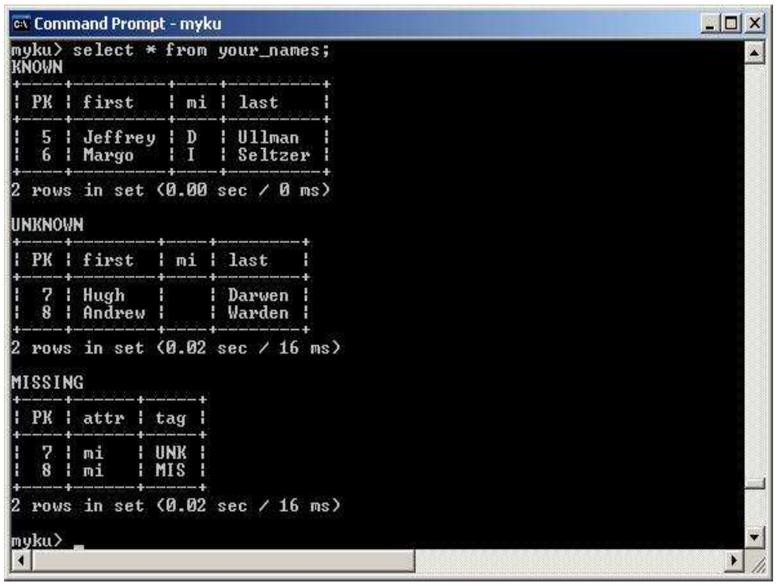
```
myku> select * from your_names;
KNOWN
+----+---------+----+---------+
| PK | first   | mi | last    |
+----+---------+----+---------+
|  5 | Jeffrey | D  | Ullman  |
|  6 | Margo   | I  | Seltzer |
+----+---------+----+---------+
2 rows in set (0.03 sec / 31 ms)

UNKNOWN
+----+-------+----+-----------+
| PK | first | mi | last      |
+----+-------+----+-----------+
|  7 | Chris |    | Date      |
|  8 | David | ?  | McGoveran |
+----+-------+----+-----------+
2 rows in set (0.02 sec / 16 ms)

MISSING
+----+------+-----+
| PK | attr | tag |
+----+------+-----+
|  7 | mi   | UNK |
|  8 | mi   | INV |
+----+------+-----+
2 rows in set (0.00 sec / 0 ms)

myku> _
```

Figure 12: Case 3 (b) your_names relvar



```
myku> select * from my_names;
KNOWN
+----+-------+----+------+
| PK | first | mi | last |
+----+-------+----+------+
|  1 | Edgar | F  | Codd |
|  2 | Chris | J  | Date |
+----+-------+----+------+
2 rows in set (0.03 sec / 31 ms)

UNKNOWN
+----+--------+----+--------+
| PK | first  | mi | last   |
+----+--------+----+--------+
|  3 | Hugh   |    | Darwen |
|  4 | Andrew |    | Warden |
+----+--------+----+--------+
2 rows in set (0.00 sec / 0 ms)

MISSING
+----+------+-----+
| PK | attr | tag |
+----+------+-----+
|  3 | mi   | UNK |
|  4 | mi   | N/A |
+----+------+-----+
2 rows in set (0.00 sec / 0 ms)

myku> _
```

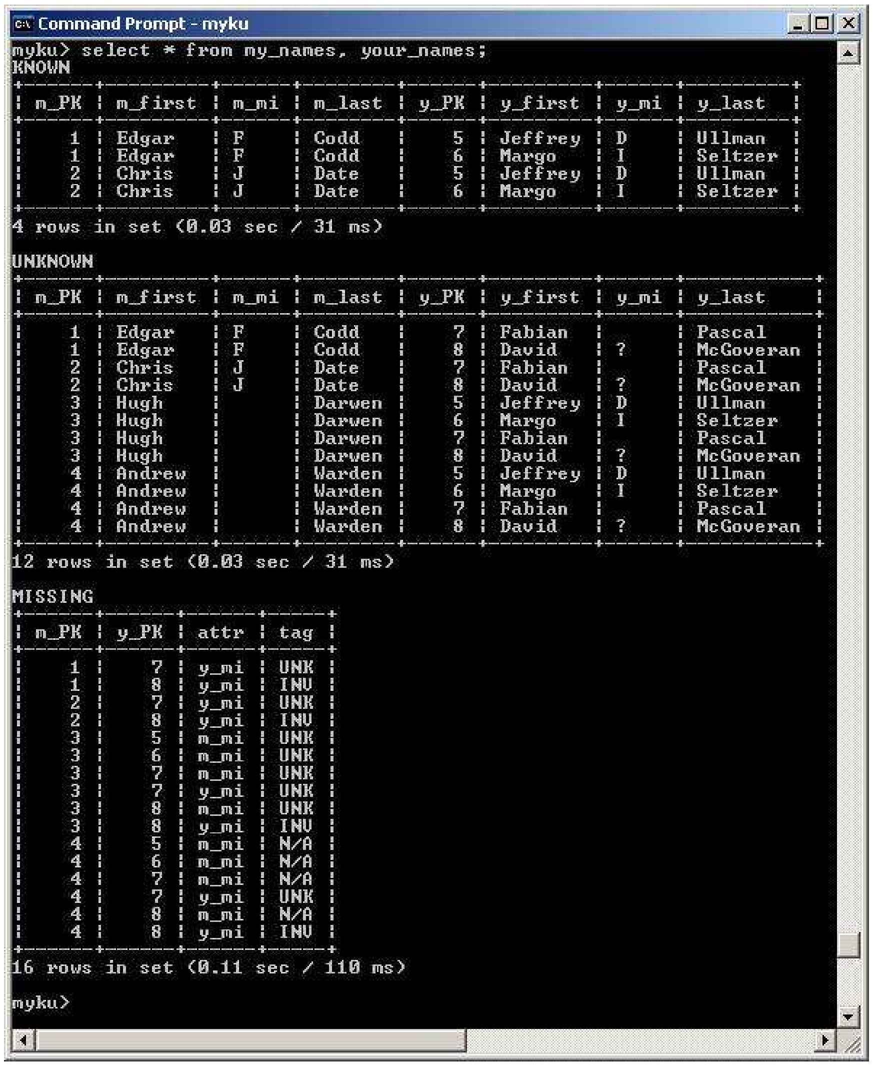
Figure 13: Case 4 (a) my_names relvar



```
myku> select * from your_names;
KNOWN
+----+---------+----+---------+
| PK | first   | mi | last    |
+----+---------+----+---------+
|  5 | Jeffrey | D  | Ullman  |
|  6 | Margo   | I  | Seltzer |
+----+---------+----+---------+
2 rows in set (0.00 sec / 0 ms)

UNKNOWN
+----+--------+----+--------+
| PK | first  | mi | last   |
+----+--------+----+--------+
|  7 | Hugh   |    | Darwen |
|  8 | Andrew |    | Warden |
+----+--------+----+--------+
2 rows in set (0.02 sec / 16 ms)

MISSING
+----+------+-----+
| PK | attr | tag |
+----+------+-----+
|  7 | mi   | UNK |
|  8 | mi   | MIS |
+----+------+-----+
2 rows in set (0.02 sec / 16 ms)

myku> _
```

Figure 14: Case 4 (b) your_names relvar

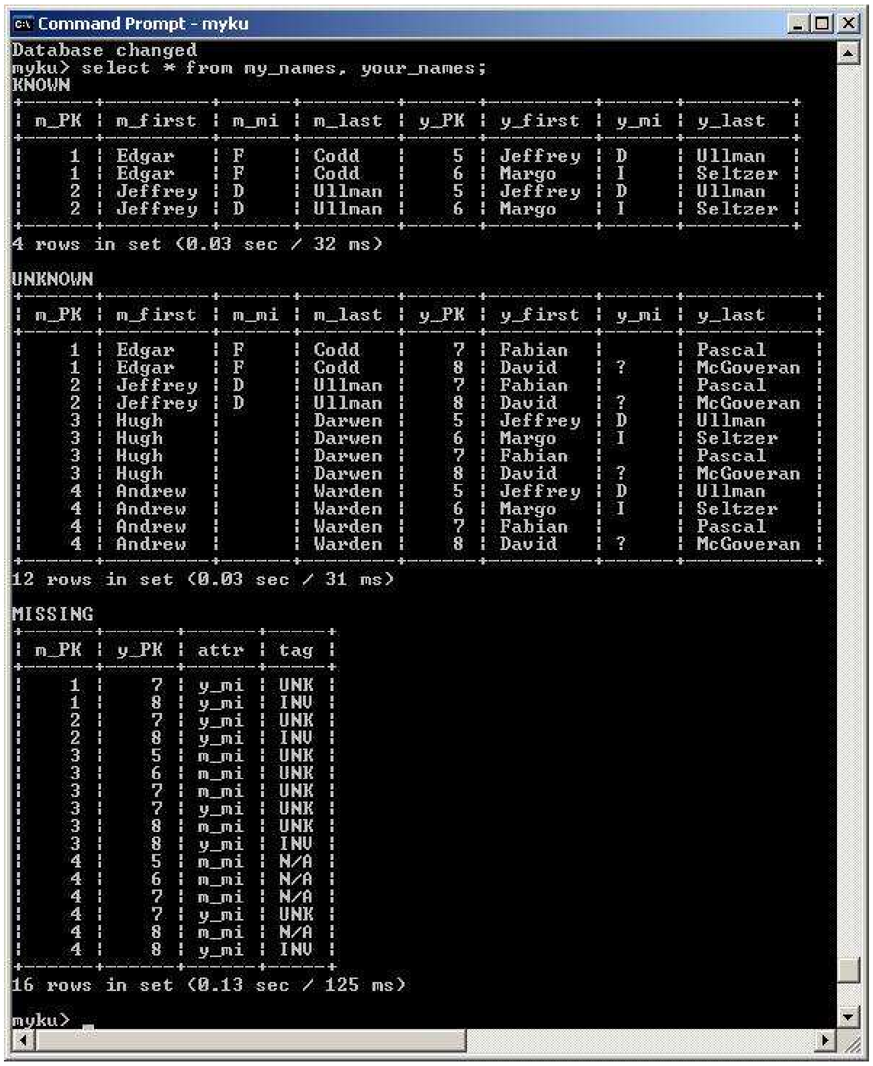
7.3.2 Cartesian Product



```
myku> select * from my_names, your_names;
KNOWN
+------+---------+------+--------+------+---------+------+---------+
| m_PK | m_first | m_mi | m_last | y_PK | y_first | y_mi | y_last  |
+------+---------+------+--------+------+---------+------+---------+
|    1 | Edgar   | F    | Codd   |    5 | Jeffrey | D    | Ullman  |
|    1 | Edgar   | F    | Codd   |    6 | Margo   | I    | Seltzer |
|    2 | Chris   | J    | Date   |    5 | Jeffrey | D    | Ullman  |
|    2 | Chris   | J    | Date   |    6 | Margo   | I    | Seltzer |
+------+---------+------+--------+------+---------+------+---------+
4 rows in set (0.03 sec / 31 ms)

UNKNOWN
+------+---------+------+--------+------+---------+------+-----------+
| m_PK | m_first | m_mi | m_last | y_PK | y_first | y_mi | y_last    |
+------+---------+------+--------+------+---------+------+-----------+
|    1 | Edgar   | F    | Codd   |    7 | Fabian  |      | Pascal    |
|    1 | Edgar   | F    | Codd   |    8 | David   | ?    | McGoveran |
|    2 | Chris   | J    | Date   |    7 | Fabian  |      | Pascal    |
|    2 | Chris   | J    | Date   |    8 | David   | ?    | McGoveran |
|    3 | Hugh    |      | Darwen |    5 | Jeffrey | D    | Ullman    |
|    3 | Hugh    |      | Darwen |    6 | Margo   | I    | Seltzer   |
|    3 | Hugh    |      | Darwen |    7 | Fabian  |      | Pascal    |
|    3 | Hugh    |      | Darwen |    8 | David   | ?    | McGoveran |
|    4 | Andrew  |      | Warden |    5 | Jeffrey | D    | Ullman    |
|    4 | Andrew  |      | Warden |    6 | Margo   | I    | Seltzer   |
|    4 | Andrew  |      | Warden |    7 | Fabian  |      | Pascal    |
|    4 | Andrew  |      | Warden |    8 | David   | ?    | McGoveran |
+------+---------+------+--------+------+---------+------+-----------+
12 rows in set (0.03 sec / 31 ms)

MISSING
+------+------+------+-----+
| m_PK | y_PK | attr | tag |
+------+------+------+-----+
|    1 |    7 | y_mi | UNK |
|    1 |    8 | y_mi | INV |
|    2 |    7 | y_mi | UNK |
|    2 |    8 | y_mi | INV |
|    3 |    5 | m_mi | UNK |
|    3 |    6 | m_mi | UNK |
|    3 |    7 | m_mi | UNK |
|    3 |    7 | y_mi | UNK |
|    3 |    8 | m_mi | UNK |
|    3 |    8 | y_mi | INV |
|    4 |    5 | m_mi | N/A |
|    4 |    6 | m_mi | N/A |
|    4 |    7 | m_mi | N/A |
|    4 |    7 | y_mi | UNK |
|    4 |    8 | m_mi | N/A |
|    4 |    8 | y_mi | INV |
+------+------+------+-----+
16 rows in set (0.11 sec / 110 ms)

myku>
```

Figure 15: Case 1 product of my_names and your_names

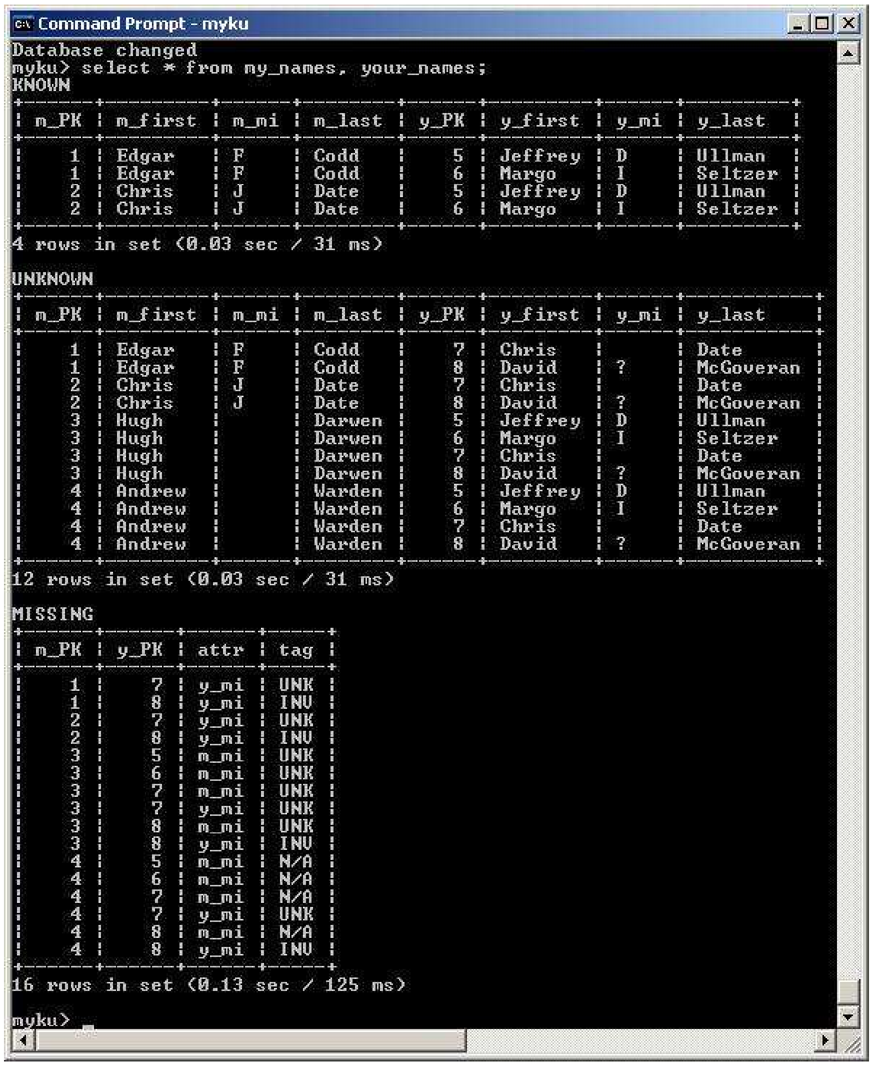
The Cartesian product for case 1 shown in Figure 15 is the correct result for Tables 14, 15, and 16.



```
Database changed
myku> select * from my_names, your_names;
KNOWN
+------+---------+------+--------+------+---------+------+---------+
| m_PK | m_first | m_mi | m_last | y_PK | y_first | y_mi | y_last  |
+------+---------+------+--------+------+---------+------+---------+
|    1 | Edgar   | F    | Codd   |    5 | Jeffrey | D    | Ullman  |
|    1 | Edgar   | F    | Codd   |    6 | Margo   | I    | Seltzer |
|    2 | Jeffrey | D    | Ullman |    5 | Jeffrey | D    | Ullman  |
|    2 | Jeffrey | D    | Ullman |    6 | Margo   | I    | Seltzer |
+------+---------+------+--------+------+---------+------+---------+
4 rows in set (0.03 sec / 32 ms)

UNKNOWN
+------+---------+------+--------+------+---------+------+-----------+
| m_PK | m_first | m_mi | m_last | y_PK | y_first | y_mi | y_last    |
+------+---------+------+--------+------+---------+------+-----------+
|    1 | Edgar   | F    | Codd   |    7 | Fabian  |      | Pascal    |
|    1 | Edgar   | F    | Codd   |    8 | David   | ?    | McGoveran |
|    2 | Jeffrey | D    | Ullman |    7 | Fabian  |      | Pascal    |
|    2 | Jeffrey | D    | Ullman |    8 | David   | ?    | McGoveran |
|    3 | Hugh    |      | Darwen |    5 | Jeffrey | D    | Ullman    |
|    3 | Hugh    |      | Darwen |    6 | Margo   | I    | Seltzer   |
|    3 | Hugh    |      | Darwen |    7 | Fabian  |      | Pascal    |
|    3 | Hugh    |      | Darwen |    8 | David   | ?    | McGoveran |
|    4 | Andrew  |      | Warden |    5 | Jeffrey | D    | Ullman    |
|    4 | Andrew  |      | Warden |    6 | Margo   | I    | Seltzer   |
|    4 | Andrew  |      | Warden |    7 | Fabian  |      | Pascal    |
|    4 | Andrew  |      | Warden |    8 | David   | ?    | McGoveran |
+------+---------+------+--------+------+---------+------+-----------+
12 rows in set (0.03 sec / 31 ms)

MISSING
+------+------+------+-----+
| m_PK | y_PK | attr | tag |
+------+------+------+-----+
|    1 |    7 | y_mi | UNK |
|    1 |    8 | y_mi | INV |
|    2 |    7 | y_mi | UNK |
|    2 |    8 | y_mi | INV |
|    3 |    5 | m_mi | UNK |
|    3 |    6 | m_mi | UNK |
|    3 |    7 | m_mi | UNK |
|    3 |    7 | y_mi | UNK |
|    3 |    8 | m_mi | UNK |
|    3 |    8 | y_mi | INV |
|    4 |    5 | m_mi | N/A |
|    4 |    6 | m_mi | N/A |
|    4 |    7 | m_mi | N/A |
|    4 |    7 | y_mi | UNK |
|    4 |    8 | m_mi | N/A |
|    4 |    8 | y_mi | INV |
+------+------+------+-----+
16 rows in set (0.13 sec / 125 ms)

myku>
```

Figure 16: Case 2 product of my_names and your_names

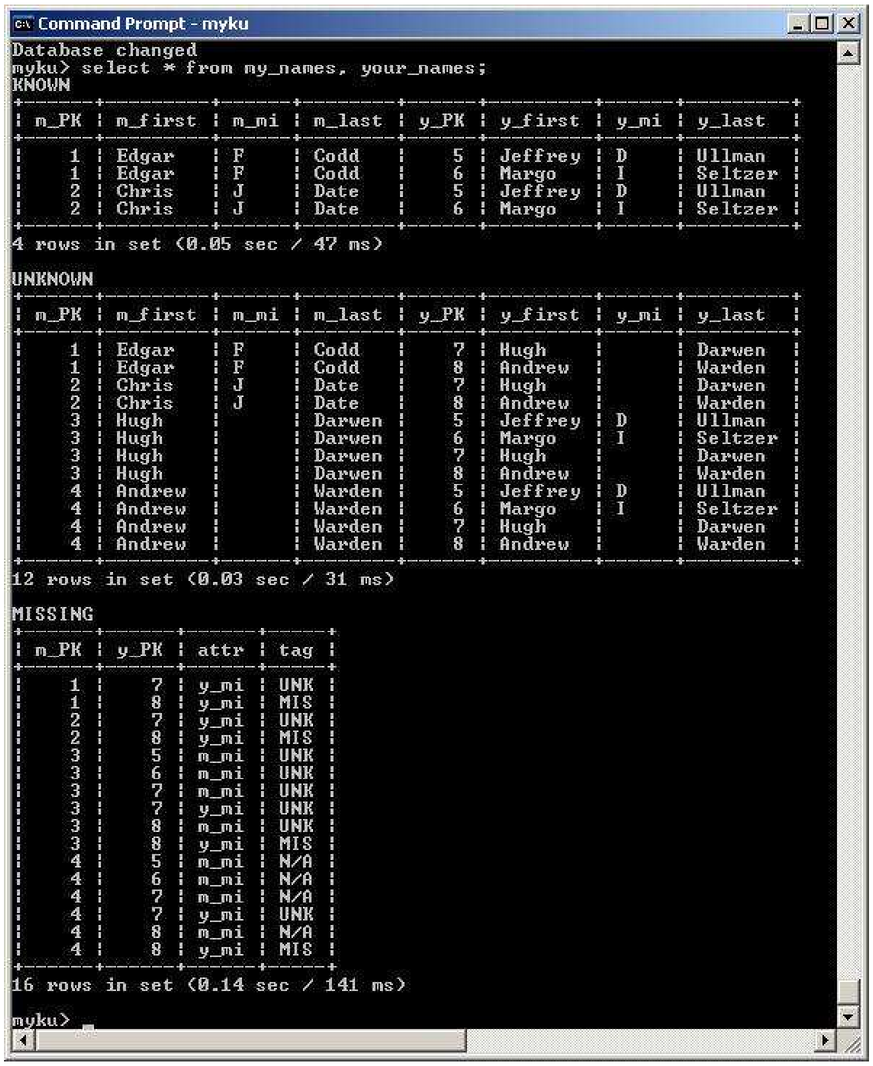
The Cartesian product for case 2 shown in Figure 16 is the correct result for Tables 17, 18, and 19.



```
Database changed
myku> select * from my_names, your_names;
KNOWN
+------+---------+------+--------+------+---------+------+---------+
| m_PK | m_first | m_mi | m_last | y_PK | y_first | y_mi | y_last  |
+------+---------+------+--------+------+---------+------+---------+
|    1 | Edgar   | F    | Codd   |    5 | Jeffrey | D    | Ullman  |
|    1 | Edgar   | F    | Codd   |    6 | Margo   | I    | Seltzer |
|    2 | Chris   | J    | Date   |    5 | Jeffrey | D    | Ullman  |
|    2 | Chris   | J    | Date   |    6 | Margo   | I    | Seltzer |
+------+---------+------+--------+------+---------+------+---------+
4 rows in set (0.03 sec / 31 ms)

UNKNOWN
+------+---------+------+--------+------+---------+------+-----------+
| m_PK | m_first | m_mi | m_last | y_PK | y_first | y_mi | y_last    |
+------+---------+------+--------+------+---------+------+-----------+
|    1 | Edgar   | F    | Codd   |    7 | Chris   |      | Date      |
|    1 | Edgar   | F    | Codd   |    8 | David   | ?    | McGoveran |
|    2 | Chris   | J    | Date   |    7 | Chris   |      | Date      |
|    2 | Chris   | J    | Date   |    8 | David   | ?    | McGoveran |
|    3 | Hugh    |      | Darwen |    5 | Jeffrey | D    | Ullman    |
|    3 | Hugh    |      | Darwen |    6 | Margo   | I    | Seltzer   |
|    3 | Hugh    |      | Darwen |    7 | Chris   |      | Date      |
|    3 | Hugh    |      | Darwen |    8 | David   | ?    | McGoveran |
|    4 | Andrew  |      | Warden |    5 | Jeffrey | D    | Ullman    |
|    4 | Andrew  |      | Warden |    6 | Margo   | I    | Seltzer   |
|    4 | Andrew  |      | Warden |    7 | Chris   |      | Date      |
|    4 | Andrew  |      | Warden |    8 | David   | ?    | McGoveran |
+------+---------+------+--------+------+---------+------+-----------+
12 rows in set (0.03 sec / 31 ms)

MISSING
+------+------+------+-----+
| m_PK | y_PK | attr | tag |
+------+------+------+-----+
|    1 |    7 | y_mi | UNK |
|    1 |    8 | y_mi | INV |
|    2 |    7 | y_mi | UNK |
|    2 |    8 | y_mi | INV |
|    3 |    5 | m_mi | UNK |
|    3 |    6 | m_mi | UNK |
|    3 |    7 | m_mi | UNK |
|    3 |    7 | y_mi | UNK |
|    3 |    8 | m_mi | UNK |
|    3 |    8 | y_mi | INV |
|    4 |    5 | m_mi | N/A |
|    4 |    6 | m_mi | N/A |
|    4 |    7 | m_mi | N/A |
|    4 |    7 | y_mi | UNK |
|    4 |    8 | m_mi | N/A |
|    4 |    8 | y_mi | INV |
+------+------+------+-----+
16 rows in set (0.13 sec / 125 ms)

myku> _
```

Figure 17: Case 3 product of my_names and your_names

The Cartesian product for case 3 shown in Figure 17 is the correct result for Tables 20, 21, and 22.



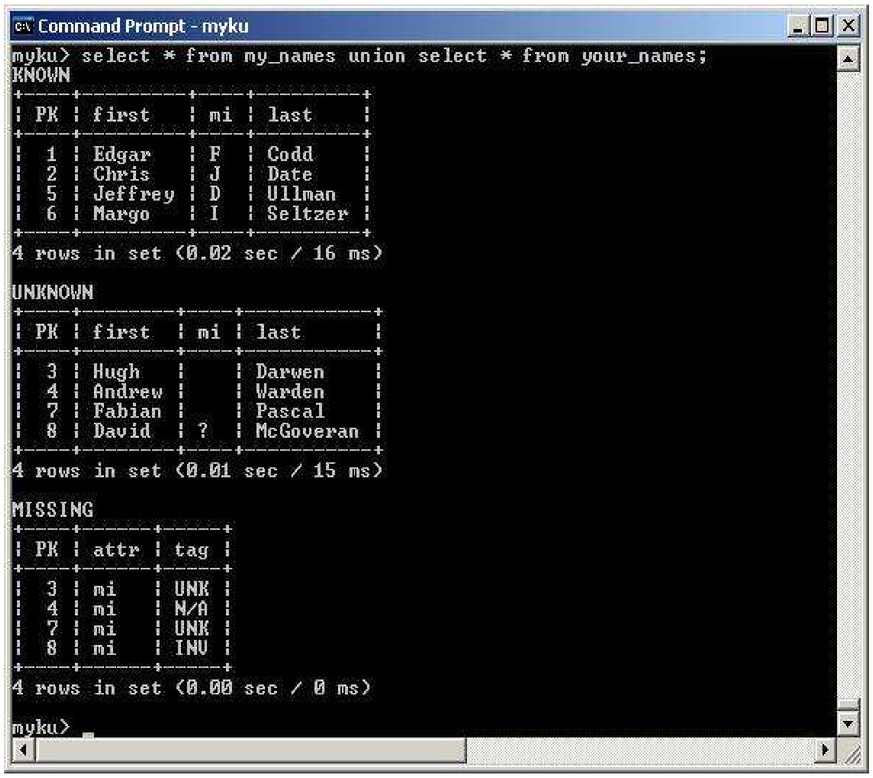
```
Database changed
myku> select * from my_names, your_names;
KNOWN
+------+---------+------+--------+------+---------+------+---------+
| m_PK | m_first | m_mi | m_last | y_PK | y_first | y_mi | y_last  |
+------+---------+------+--------+------+---------+------+---------+
|    1 | Edgar   | F    | Codd   |    5 | Jeffrey | D    | Ullman  |
|    1 | Edgar   | F    | Codd   |    6 | Margo   | I    | Seltzer |
|    2 | Chris   | J    | Date   |    5 | Jeffrey | D    | Ullman  |
|    2 | Chris   | J    | Date   |    6 | Margo   | I    | Seltzer |
+------+---------+------+--------+------+---------+------+---------+
4 rows in set (0.05 sec / 47 ms)

UNKNOWN
+------+---------+------+--------+------+---------+------+---------+
| m_PK | m_first | m_mi | m_last | y_PK | y_first | y_mi | y_last  |
+------+---------+------+--------+------+---------+------+---------+
|    1 | Edgar   | F    | Codd   |    7 | Hugh    |      | Darwen  |
|    1 | Edgar   | F    | Codd   |    8 | Andrew  |      | Warden  |
|    2 | Chris   | J    | Date   |    7 | Hugh    |      | Darwen  |
|    2 | Chris   | J    | Date   |    8 | Andrew  |      | Warden  |
|    3 | Hugh    |      | Darwen |    5 | Jeffrey | D    | Ullman  |
|    3 | Hugh    |      | Darwen |    6 | Margo   | I    | Seltzer |
|    3 | Hugh    |      | Darwen |    7 | Hugh    |      | Darwen  |
|    3 | Hugh    |      | Darwen |    8 | Andrew  |      | Warden  |
|    4 | Andrew  |      | Warden |    5 | Jeffrey | D    | Ullman  |
|    4 | Andrew  |      | Warden |    6 | Margo   | I    | Seltzer |
|    4 | Andrew  |      | Warden |    7 | Hugh    |      | Darwen  |
|    4 | Andrew  |      | Warden |    8 | Andrew  |      | Warden  |
+------+---------+------+--------+------+---------+------+---------+
12 rows in set (0.03 sec / 31 ms)

MISSING
+------+------+------+-----+
| m_PK | y_PK | attr | tag |
+------+------+------+-----+
|    1 |    7 | y_mi | UNK |
|    1 |    8 | y_mi | MIS |
|    2 |    7 | y_mi | UNK |
|    2 |    8 | y_mi | MIS |
|    3 |    5 | m_mi | UNK |
|    3 |    6 | m_mi | UNK |
|    3 |    7 | m_mi | UNK |
|    3 |    7 | y_mi | UNK |
|    3 |    8 | m_mi | UNK |
|    3 |    8 | y_mi | MIS |
|    4 |    5 | m_mi | N/A |
|    4 |    6 | m_mi | N/A |
|    4 |    7 | m_mi | N/A |
|    4 |    7 | y_mi | UNK |
|    4 |    8 | m_mi | N/A |
|    4 |    8 | y_mi | MIS |
+------+------+------+-----+
16 rows in set (0.14 sec / 141 ms)

myku> _
```

Figure 18: Case 4 product of my_names and your_names

The Cartesian product for case 4 shown in Figure 18 is correct the result for Tables 25, 24, and 25.

7.3.3 Set Union

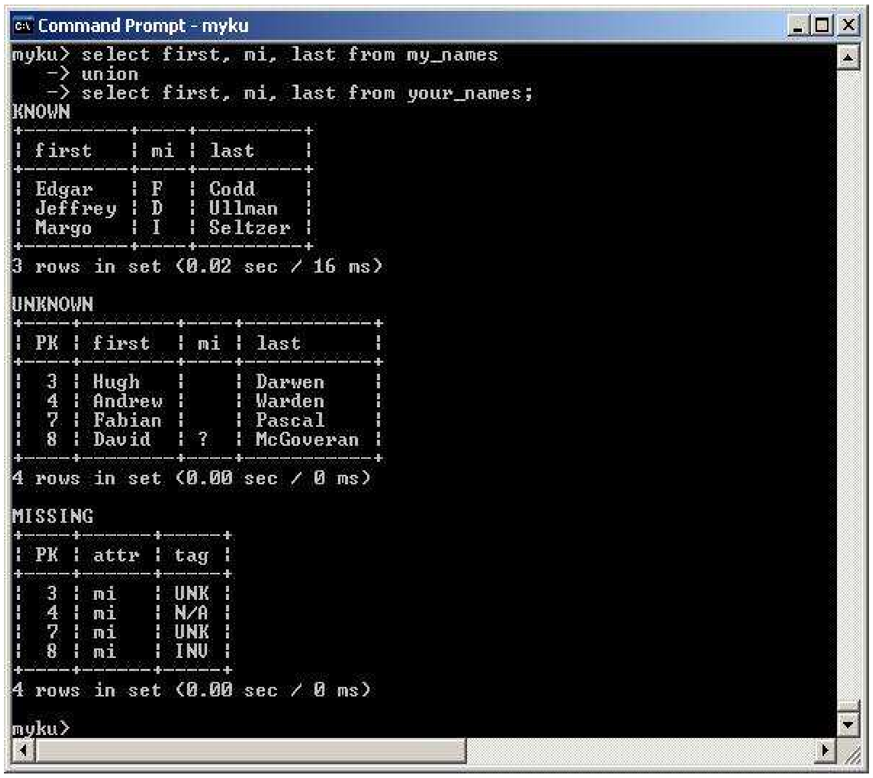
MyKU is verified correct for set union using the first three test cases. The fourth test case fails to correctly remove duplicate rows. The solution to this problem is described with Figure 22.



```
myku> select * from my_names union select * from your_names;
KNOWN
+----+---------+----+---------+
| PK | first   | mi | last    |
+----+---------+----+---------+
|  1 | Edgar   | F  | Codd    |
|  2 | Chris   | J  | Date    |
|  5 | Jeffrey | D  | Ullman  |
|  6 | Margo   | I  | Seltzer |
+----+---------+----+---------+
4 rows in set (0.02 sec / 16 ms)

UNKNOWN
+----+--------+----+-----------+
| PK | first  | mi | last      |
+----+--------+----+-----------+
|  3 | Hugh   |    | Darwen    |
|  4 | Andrew |    | Warden    |
|  7 | Fabian |    | Pascal    |
|  8 | David  | ?  | McGoveran |
+----+--------+----+-----------+
4 rows in set (0.01 sec / 15 ms)

MISSING
+----+------+-----+
| PK | attr | tag |
+----+------+-----+
|  3 | mi   | UNK |
|  4 | mi   | N/A |
|  7 | mi   | UNK |
|  8 | mi   | INV |
+----+------+-----+
4 rows in set (0.00 sec / 0 ms)

myku> _
```

Figure 19: Case 1 set union of my_names and your_names

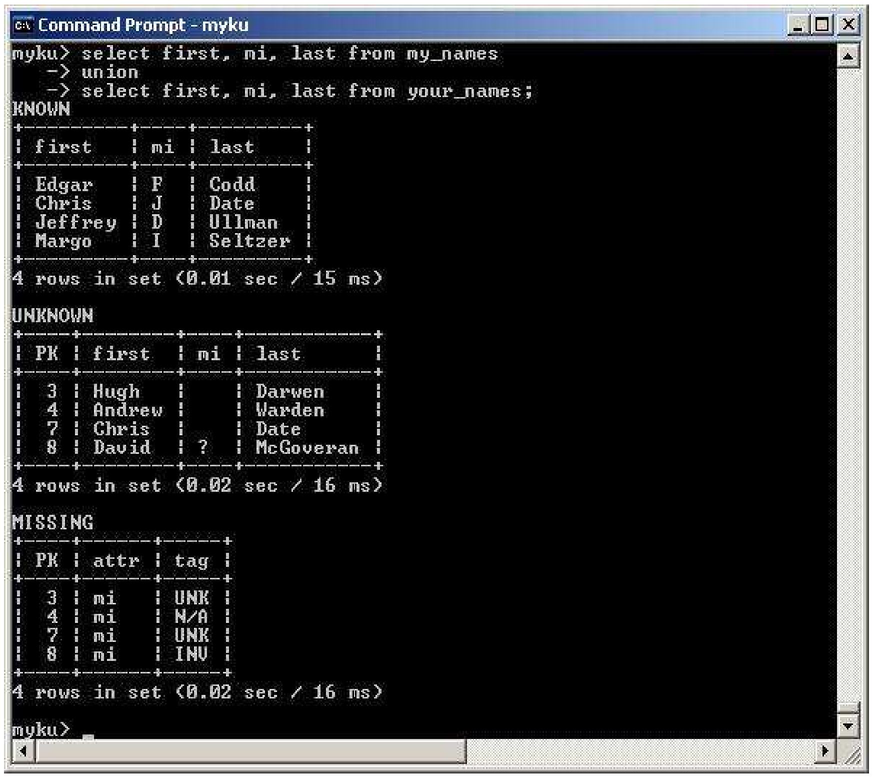
The set union for case 1 shown in Figure 19 is the correct result for Table 26.



```
myku> select first, mi, last from my_names
    -> union
    -> select first, mi, last from your_names;
KNOWN
+---------+----+----------+
| first   | mi | last     |
+---------+----+----------+
| Edgar   | F  | Codd     |
| Jeffrey | D  | Ullman   |
| Margo   | I  | Seltzer  |
+---------+----+----------+
3 rows in set (0.02 sec / 16 ms)

UNKNOWN
+----+--------+----+-----------+
| PK | first  | mi | last      |
+----+--------+----+-----------+
|  3 | Hugh   |    | Darwen    |
|  4 | Andrew |    | Warden    |
|  7 | Fabian |    | Pascal    |
|  8 | David  | ?  | McGoveran |
+----+--------+----+-----------+
4 rows in set (0.00 sec / 0 ms)

MISSING
+----+------+-----+
| PK | attr | tag |
+----+------+-----+
|  3 | mi   | UNK |
|  4 | mi   | N/A |
|  7 | mi   | UNK |
|  8 | mi   | INV |
+----+------+-----+
4 rows in set (0.00 sec / 0 ms)

myku>
```

Figure 20: Case 2 set union of my_names and your_names

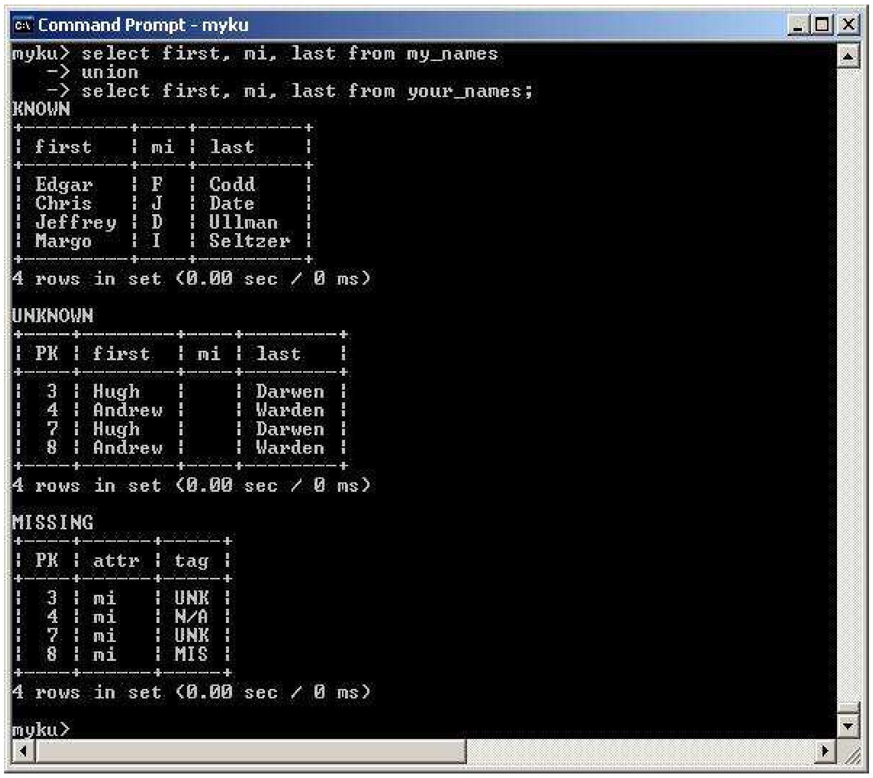
The set union for case 2 shown in Figure 20 is the correct result for Table 27.


Command Prompt - myku

```
myku> select first, mi, last from my_names
   -> union
   -> select first, mi, last from your_names;
KNOWN
+---------+----+---------+
| first   | mi | last    |
+---------+----+---------+
| Edgar   | F  | Codd    |
| Chris   | J  | Date    |
| Jeffrey | D  | Ullman  |
| Margo   | I  | Seltzer |
+---------+----+---------+
4 rows in set (0.01 sec / 15 ms)

UNKNOWN
+----+--------+----+-----------+
| PK | first  | mi | last      |
+----+--------+----+-----------+
|  3 | Hugh   |    | Darwen    |
|  4 | Andrew |    | Warden    |
|  7 | Chris  |    | Date      |
|  8 | David  | ?  | McGoveran |
+----+--------+----+-----------+
4 rows in set (0.02 sec / 16 ms)

MISSING
+----+------+-----+
| PK | attr | tag |
+----+------+-----+
|  3 | mi   | UNK |
|  4 | mi   | N/A |
|  7 | mi   | UNK |
|  8 | mi   | INV |
+----+------+-----+
4 rows in set (0.02 sec / 16 ms)

myku> _
```

Figure 21: Case 3 set union of my_names and your_names

The set union for case 3 shown in Figure 21 is the correct result for Table 28.



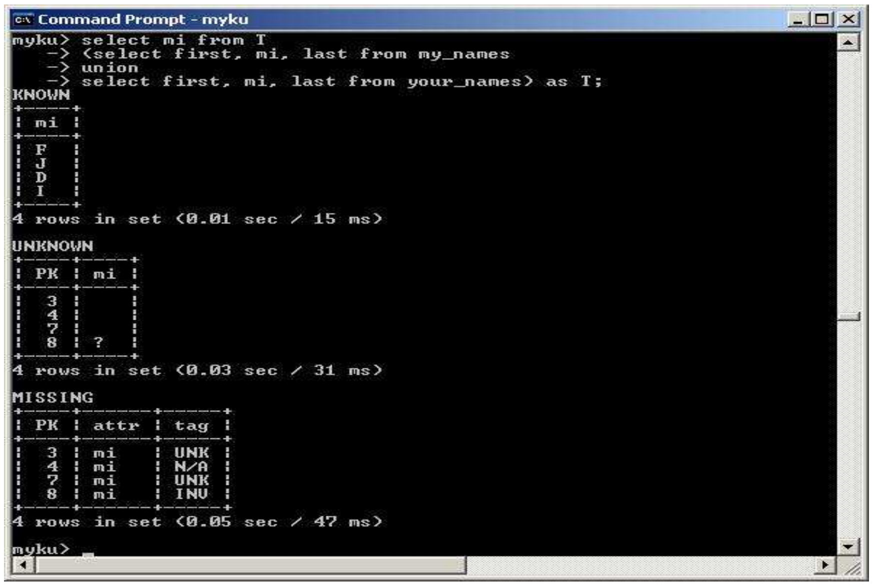
```
myku> select first, mi, last from my_names
    -> union
    -> select first, mi, last from your_names;
KNOWN
+---------+----+---------+
| first   | mi | last    |
+---------+----+---------+
| Edgar   | F  | Codd    |
| Chris   | J  | Date    |
| Jeffrey | D  | Ullman  |
| Margo   | I  | Seltzer |
+---------+----+---------+
4 rows in set (0.00 sec / 0 ms)

UNKNOWN
+----+--------+----+--------+
| PK | first  | mi | last   |
+----+--------+----+--------+
|  3 | Hugh   |    | Darwen |
|  4 | Andrew |    | Warden |
|  7 | Hugh   |    | Darwen |
|  8 | Andrew |    | Warden |
+----+--------+----+--------+
4 rows in set (0.00 sec / 0 ms)

MISSING
+----+------+-----+
| PK | attr | tag |
+----+------+-----+
|  3 | mi   | UNK |
|  4 | mi   | N/A |
|  7 | mi   | UNK |
|  8 | mi   | MIS |
+----+------+-----+
4 rows in set (0.00 sec / 0 ms)

myku>
```

Figure 22: Case 4 set union of my_names and your_names

The set union for case 4 shown in Figure 22 is not the correct result for Table 29. MyKU failed to correctly create the set union using first, mi, and last columns because it must have a key to connect the _UNKNOWN rows to _MISSING rows. To ensure that the key is available, the query rewrite logic adds the key to any projection of unknown and missing if it is not included. In this case “Hugh Darwen” is not identified as a duplicated row because each row has a unique key. Correcting this problem requires the fix for the duplicate removal problem described in section 7.2.3.

7.3.4 Project

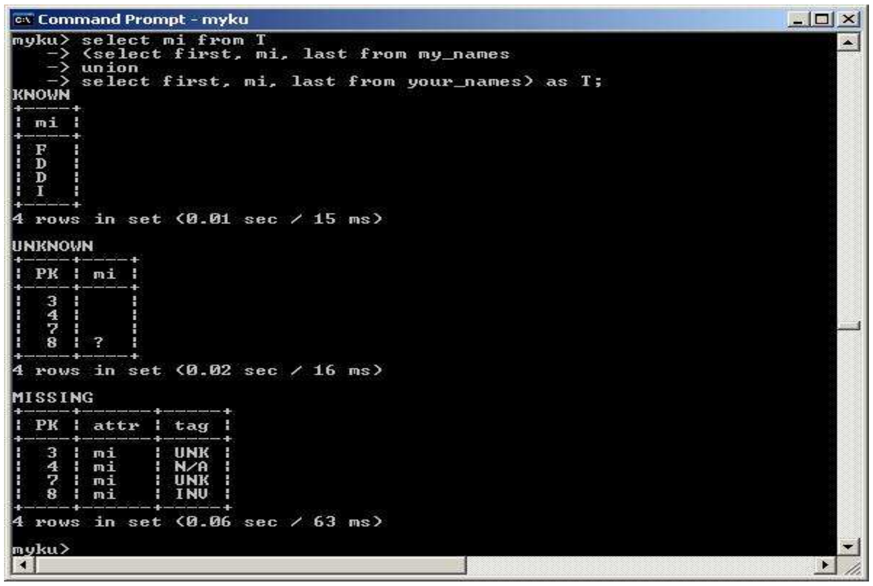
The test cases for projection rely on set union as an intermediate result. The rows in the derived union of tables are created before columns are projected and duplicate rows are not removed from the results. MyKU fails to correctly project attributes from an intermediate set union for the reasons described in sections 7.2.2 and 7.2.3. Additional explanation of the duplicate removal problem for set union can be found with Figure 22.



```
myku> select mi from T
    -> (select first, mi, last from my_names
    -> union
    -> select first, mi, last from your_names) as T;
KNOWN
+----+
| mi |
+----+
| F  |
| J  |
| D  |
| I  |
+----+
4 rows in set (0.01 sec / 15 ms)
UNKNOWN
+----+----+
| PK | mi |
+----+----+
|  3 |    |
|  4 |    |
|  7 |    |
|  8 | ?  |
+----+----+
4 rows in set (0.03 sec / 31 ms)
MISSING
+----+------+-----+
| PK | attr | tag |
+----+------+-----+
|  3 | mi   | UNK |
|  4 | mi   | N/A |
|  7 | mi   | UNK |
|  8 | mi   | INV |
+----+------+-----+
4 rows in set (0.05 sec / 47 ms)

myku> _
```

Figure 23: Case 1 project from my_names union your_names

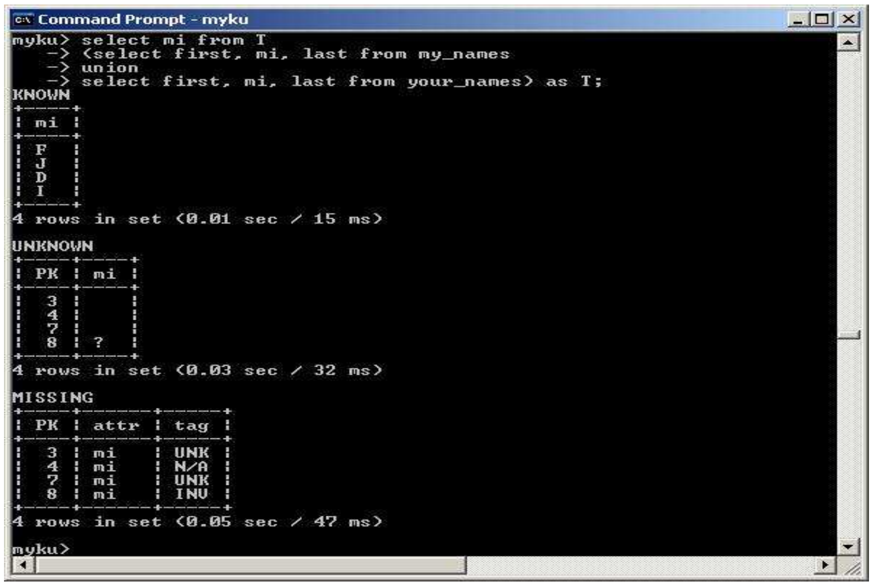
The projection of attributes from my_names for case 1 shown in Figure 23 is not the correct result for Table 30. MyKU did not remove the duplicate rows from tables RESULT_UNKNOWN and RESULT_MISSING where PK is equal to 7.



```
myku> select mi from T
    -> (select first, mi, last from my_names
    -> union
    -> select first, mi, last from your_names) as T;
KNOWN
+----+
| mi |
+----+
| F  |
| D  |
| D  |
| I  |
+----+
4 rows in set (0.01 sec / 15 ms)

UNKNOWN
+----+----+
| PK | mi |
+----+----+
|  3 |    |
|  4 |    |
|  7 |    |
|  8 | ?  |
+----+----+
4 rows in set (0.02 sec / 16 ms)

MISSING
+----+------+-----+
| PK | attr | tag |
+----+------+-----+
|  3 | mi   | UNK |
|  4 | mi   | N/A |
|  7 | mi   | UNK |
|  8 | mi   | INV |
+----+------+-----+
4 rows in set (0.06 sec / 63 ms)

myku>
```

Figure 24: Case 2 project from my_names union your_names

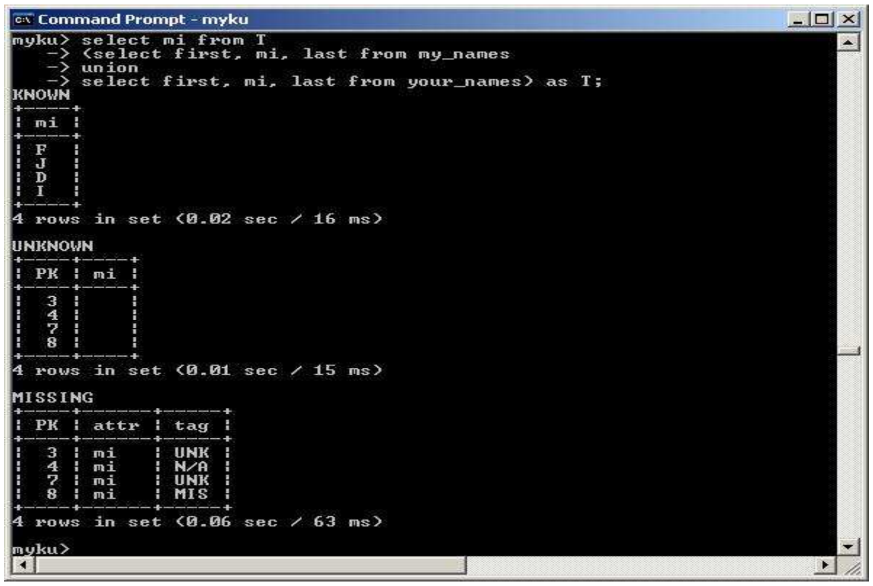
The projection of attributes from my_names for case 2 shown in Figure 24 is not the correct result for Table 31. MyKU did not remove the duplicate rows from tables RESULT_KNOWN where mi is ’D’ or RESULT_UNKNOWN and RESULT_MISSING where PK is equal to 7.



```
myku> select mi from T
    -> (select first, mi, last from my_names
    -> union
    -> select first, mi, last from your_names) as T;
KNOWN
+----+
| mi |
+----+
| F  |
| J  |
| D  |
| I  |
+----+
4 rows in set (0.01 sec / 15 ms)

UNKNOWN
+----+----+
| PK | mi |
+----+----+
|  3 |    |
|  4 |    |
|  7 |    |
|  8 | ?  |
+----+----+
4 rows in set (0.03 sec / 32 ms)

MISSING
+----+------+-----+
| PK | attr | tag |
+----+------+-----+
|  3 | mi   | UNK |
|  4 | mi   | N/A |
|  7 | mi   | UNK |
|  8 | mi   | INV |
+----+------+-----+
4 rows in set (0.05 sec / 47 ms)

myku>
```

Figure 25: Case 3 project from my_names union your_names

The projection of attributes from my_names for case 3 shown in Figure 25 is not the correct result for Table 32. MyKU did not remove the duplicate rows from tables RESULT_UNKNOWN and RESULT_MISSING where PK is equal to 7.



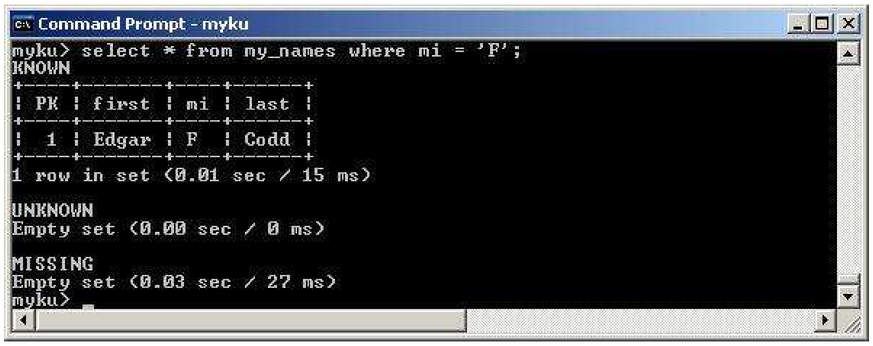
```
myku> select mi from T
    -> (select first, mi, last from my_names
    -> union
    -> select first, mi, last from your_names) as T;
KNOWN
+----+
| mi |
+----+
| F  |
| J  |
| D  |
| I  |
+----+
4 rows in set (0.02 sec / 16 ms)

UNKNOWN
+----+----+
| PK | mi |
+----+----+
|  3 |    |
|  4 |    |
|  7 |    |
|  8 |    |
+----+----+
4 rows in set (0.01 sec / 15 ms)

MISSING
+----+------+-----+
| PK | attr | tag |
+----+------+-----+
|  3 | mi   | UNK |
|  4 | mi   | N/A |
|  7 | mi   | UNK |
|  8 | mi   | MIS |
+----+------+-----+
4 rows in set (0.06 sec / 63 ms)

myku>
```

Figure 26: Case 4 project from my_names union your_names

The projection of attributes from my_names for case 4 shown in Figure 26 is not the correct result for Table 33. MyKU did not remove the duplicate rows from tables RESULT_UNKNOWN and RESULT_MISSING where PK is equal to 7.

7.3.5 Restrict

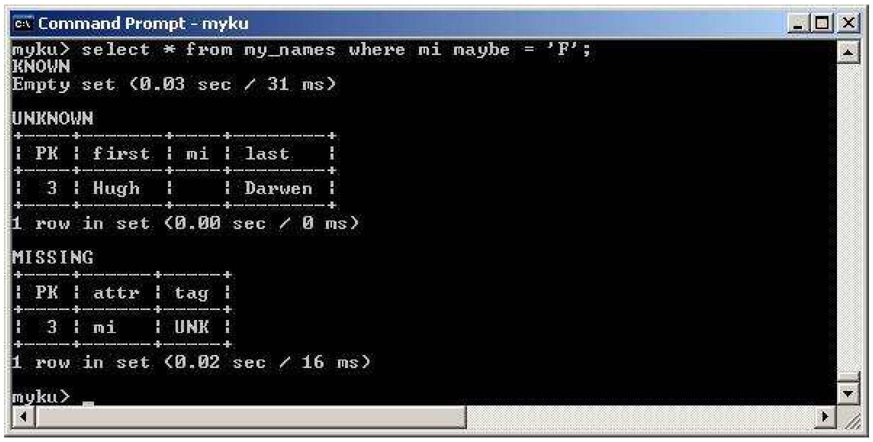
MyKU is verified correct for row restriction for test cases one and three. MyKU failed on test case two in which the restriction was to be applied to each table in a derived KNOWN/UNKNOWN relvar. MyKU failed on test case four which required the standard SQL rewritten from extended SQL to include a subquery on the _KNOWN and _UNKNOWN tables.



```
myku> select * from my_names where mi = 'F';
KNOWN
+----+-------+----+------+
| PK | first | mi | last |
+----+-------+----+------+
|  1 | Edgar | F  | Codd |
+----+-------+----+------+
1 row in set (0.01 sec / 15 ms)

UNKNOWN
Empty set (0.00 sec / 0 ms)

MISSING
Empty set (0.03 sec / 27 ms)
myku> _
```

Figure 27: Case 1 restrict of middle initial 'F'

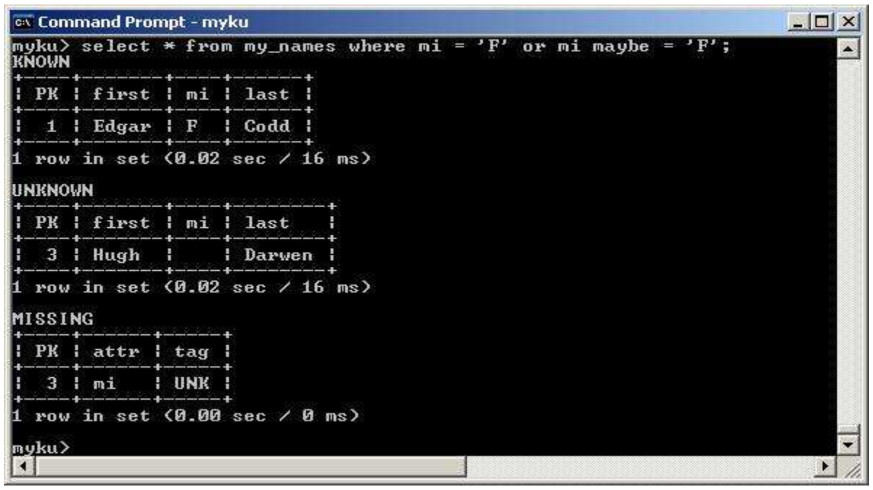
The restriction shown for case 1 in Figure 27 is the correct result for Table 38.



```
myku> select * from my_names where mi maybe = 'F';
KNOWN
Empty set (0.03 sec / 31 ms)

UNKNOWN
+----+-------+----+--------+
| PK | first | mi | last   |
+----+-------+----+--------+
|  3 | Hugh  |    | Darwen |
+----+-------+----+--------+
1 row in set (0.00 sec / 0 ms)

MISSING
+----+------+-----+
| PK | attr | tag |
+----+------+-----+
|  3 | mi   | UNK |
+----+------+-----+
1 row in set (0.02 sec / 16 ms)

myku> _
```

Figure 28: Case 1 restrict of middle initial maybe 'F'

The restriction for case 1 shown in Figure 28 is an example of the MAYBE modifier in the KNOWN/UNKNOWN model matching applicable, but missing middle initials. This is the correct result from chapter 6 for test case 1 in Table 39.

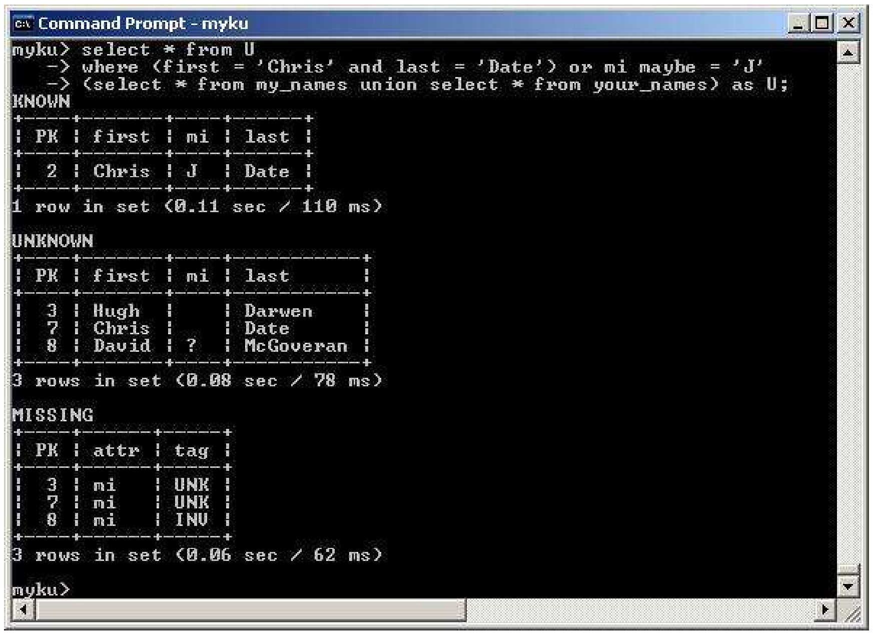


```
myku> select * from my_names where mi = 'F' or mi maybe = 'F';
KNOWN
+----+-------+----+------+
| PK | first | mi | last |
+----+-------+----+------+
|  1 | Edgar | F  | Codd |
+----+-------+----+------+
1 row in set (0.02 sec / 16 ms)

UNKNOWN
+----+-------+----+--------+
| PK | first | mi | last   |
+----+-------+----+--------+
|  3 | Hugh  |    | Darwen |
+----+-------+----+--------+
1 row in set (0.02 sec / 16 ms)

MISSING
+----+------+-----+
| PK | attr | tag |
+----+------+-----+
|  3 | mi   | UNK |
+----+------+-----+
1 row in set (0.00 sec / 0 ms)

myku>
```

Figure 29: Case 1 restrict of middle initial ’F’ or maybe ’F’

The restriction for case 1 shown in Figure 29 is an example of query that combines exact matches and maybe-matches in the KNOWN/UNKNOWN model. This is the correct result from chapter 6 for test case 1 in Table 40.

MyKU failed to correctly restrict the Cartesian product of my_names and your_names to rows where middle initials in the product are equal as shown for test case 2 in Tables 41, 42, and 43. In this case the match criteria must be applied to each component of a derived intermediate result relvar. While SQL can be written to do this specific query, MyKU does not support the necessary recursive query rewrite capability to create a correct result. The fix for the recursive querying (subqueries) (see section 7.2.2) is planned for future work in section 10.1.1 and requires implementing the KNOWN/UNKNOWN model at a lower level within the DBMS.



```
myku> select * from U
   -> where (first = 'Chris' and last = 'Date') or mi maybe = 'J'
   -> (select * from my_names union select * from your_names) as U;
KNOWN
+----+-------+----+------+
| PK | first | mi | last |
+----+-------+----+------+
|  2 | Chris | J  | Date |
+----+-------+----+------+
1 row in set (0.11 sec / 110 ms)

UNKNOWN
+----+-------+----+-----------+
| PK | first | mi | last      |
+----+-------+----+-----------+
|  3 | Hugh  |    | Darwen    |
|  7 | Chris |    | Date      |
|  8 | David | ?  | McGoveran |
+----+-------+----+-----------+
3 rows in set (0.08 sec / 78 ms)

MISSING
+----+------+-----+
| PK | attr | tag |
+----+------+-----+
|  3 | mi   | UNK |
|  7 | mi   | UNK |
|  8 | mi   | INV |
+----+------+-----+
3 rows in set (0.06 sec / 62 ms)

myku>
```

Figure 30: Case 3 restrict of my_names union your_names on equal and maybe

The restriction shown for case 3 in Figure 30 is the correct result for Table 44. The query select a row by known first and last names or those whose middle initial may be 'J'.

CHAPTER 8 Feasibility Study

The feasibility study evaluates user perception and understanding of missing data and its representation. First, standard SQL and the KNOWN/UNKNOWN SQL extensions were introduced and explained in a tutorial using hands-on examples (see Appendices I and J). Next, the users executed a series of database queries from a script (see Appendix K). This script recorded query results and answers to questions about the results. The final script questions solicited participant opinions about the clarity of the KNOWN/UNKNOWN model and alternative results presentations. The goal was to evaluate the KNOWN/UNKNOWN model's ability to represent missing data in a meaningful way that makes it understandable and solves the problem of providing adequate information about missing data.

The Virginia Commonwealth University (VCU) Institution Review Board (IRB) evaluated the study proposal and determined it to be exempt from federal regulations requiring documented informed consent. Participation was optional, risks and benefits were clear, and participants were allowed to leave the study when they chose.

8.1 Participant recruitment

Study participants were offered an SQL tutorial and a chance to participate in a research study. Recruitment efforts included a brief presentation to undergraduate computer science and information system classes where flyers were distributed promoting SQL training for those willing to assist with database research (see Appendix H). Flyers were posted on bulletin boards in the School of Engineering and the Business School. Fifty-five students voluntarily attended one of three SQL tutorials and thirty-six participated in this study.

There was a two-fold purpose for the SQL tutorial. First to attract feasibility study participants. Secondly to ensure sufficient proficiency with SQL to successfully participate in the study.

8.2 Tutorial and study

The topics of the standard SQL tutorial are described in Appendix I. Essential SQL query operations needed answer questions were covered. The KNOWN/UNKNOWN model and its extended SQL are described in Appendix J. The tutorial materials include example queries presented with explanation, tutorial database tables, and sample exercises. While the tutorial and the feasibility study use the same database, the tutorial examples and the study script were different. Participants emailed the completed script documents to the researcher.

8.3 Study results

Study results were collected from thirty-six scripts (see Appendix K) returned by email. Fifteen script documents were either blank or essentially incomplete. Twenty-one scripts were essentially complete and were used to document the study. This section summarizes the responses to nine questions/query requests that could be answered by the following methods:

- A text answer typed by the user.
- Marking a **Yes**, **No**, or **Unsure**.
- Entering a value on a scale of **1** (strong negative) to **5** (strong affirmative).
- The result of a user written query, cut from the client screen and pasted into the script document.
- Commenting about one of the questions or answers in the script.

The test database (see Appendix L) has a table *emp* of employees in which missing data is marked using null and a table *person* of the same employees (people) which uses the KNOWN/UNKNOWN model. The first three questions are related to missing data represented using nulls and the last six refer to the KNOWN/UNKOWN model. The *emp* table and the *person* relation variable are shown in the script with missing data represented by blanks. The tables in the database use the representation being evaluated, either null or _MISSING.

8.3.1 Nulls

Tables that use null and do not have the KNOWN/UNKNOWN _MISSING table to represent missing data values are supported transparently by the MyKU client program. In the feasibility study the *emp* table was used for this section of the study.

1. Which columns are missing a data value?

The purpose of this question was to develop familiarity with the script data by asking participants to examine the *emp* table on paper and/or in the database. Thirteen of the twenty-one answered this question correctly. The columns *first*, *mi*, and *dob* had rows with blanks in the printed table and NULL in the database. Two answered incorrectly, one identified two columns and another listed four. Six did not enter an answer.

2. Using emp *and* IS NULL *find employees whose middle initial is missing.*

The goal of this question was to verify that participants could write a basic SQL query to find rows with nulls. The notion that null is not a value and cannot be matched using comparison operators that depend on values was covered in the tutorial. Proof of success was a cut and paste of the results into the script document.

Nineteen of of twenty-one participants created the correct results. One selected all rows in *emp* rather than only those with nulls in the *mi* column. Another typed that "It gives me an error."

3. Do you know why these middle initials are missing? How do you know?

The tutorial on missing data indicated that nulls mean a value is missing without an identified reason. The purpose of this question was to determine if the user would either state the obvious or go beyond what is known and make an assumption.

All participants showed an understanding of null and answered this pair of questions with a meaningful answer. A representative list follows:

- "No"
- "No. Information not known"
- "The 'mi' column is NULL"
- "I don't know why"
- One answered that from the *emp* table it is not known why the middle initial is missing except that they are not there and that he or she did not know how to make the MISSING table appear.

Part *3.a.* of this question asked for an opinion about the participant's expectations for nulls on a scale from 1 (no) to 5 (yes). Thirteen answered that nulls met expectations as the strongest *yes* with 5. Two answered as a strong *yes* with 4. There was one 3 and three 2 suggesting that null only met or barely met expectations. Two did not answer this question.

8.3.2 KNOWN/UKNOWN _MISSING data tags

The *person* relation variable includes person_MISSING which is supported by the MyKU client program. To validate the model study participants need to use the extended SQL, write queries that can match missing data, and interpret the results. In the feasibility study the *person* relvar tables were used for this section of the study.

4. Using person *find people whose middle initial is 'F'*

This question started participants using a standard SQL query that works for both nulls and the KNOWN/UNKNOWN model.

One of the twenty-one participants did not enter results or a comment for questions (4), but the other twenty were able to write a correct query and pasted correct results into the script document.

5. Using person *and MAYBE find people whose middle initial may be 'F'*

This question asked participants to write an extended SQL query that searched the _UNKNOWN and _MISSING tables for individuals who have a middle initial that is not known, but not for those who do not have a middle name or initial. This distinction has a role in question (7).

The purpose of this question was to write a KNOWN/UNKNOWN query equivalent to the query for NULL in question (2).

Twenty participants were able to write this query and pasted correct results into the script document.

6. Combine queries (4) and (5) to match middle initials that are or may be 'F'

This question asked participants to write an extended SQL query that combined data from the representation of complete information in the _KNOWN table with data from the incomplete information in the _UNKNOWN table using what is known about the incomplete information from the _MISSING table.

The purpose of this question was to encourage an understanding of how complete and incomplete information are separated in the KNOWN/UNKNOWN model before asking the participants to make judgments and offer opinions.

Twenty participants were able to write this query and pasted correct results into the script document.

7. Why does the question (2) query return more rows than the question (5) query?

The goal of this question was to provoke thought about the query results from the KNOWN/UNKNOWN model compared to the same basic query results from a table using nulls. This is not a simple question with an obvious answer. The number of rows in each query was the same, but the rows were different. While 95% of participants had answered the previous questions and pasted the data needed to answer this question, only two answers were completely accurate. A list of representative answers follows:

- "Not sure"

- “I do not know”
- “They are equal”
- “I do not know, they seem to be the same.”
- “My 2 queries have the same number of rows (6)”
- “I got the same number, but I am guessing that one would be longer than the other because the attribute value '?' is added to the group that has the null attributes.”
- “While they have the same number of rows, the rows appear to be different. EKEY 24 is not listed in query (2) and the mi is labeled ?. In query(2), Andrew is listed twice as EKEY 14 and 26, in query (5), only once.”
- “False, they have the same number of rows (but different rows).”

An evaluation of this question, its goals, and the success of the best answers is included in the summary of this chapter (see section 8.4).

8. Compare three different representations of equivalent results for one query.

The goal of this question was to determine the adequacy of the KNOWN/UNKNOWN representation of missing data using the _UNKNOWN and _MISSING relations. The script database is sufficiently small that it fits on a single page making it clear what is missing and what is not, but the intent of asking for an evaluation of three models is to see how well different models fit the participant's knowledge framework.

The query results are for "All persons for whom the middle initial is missing." Each of the following three representations described below are shown in the script document in Appendix K. The study participants were asked to rate how easy it is to understand the data in the results on a scale where 1 is low and 5 is high clarity. An invitation for comments was included.

Part 8.a. *Model A*

Model A is the KNOWN/UNKNOWN representation of missing data using the _UNKNOWN and _MISSING relations. This representation is fully compliant with the relational model. The presentation format is identical with the two relations represented in the database.

1. Lowest clarity (1 participant rating)

2. Low clarity (3 participant ratings)

    - "The Question mark doesn't belong in this data set. It should be a null area"

    - "Having 2 different data sets makes it a little more difficult to interpret the data."

3. Clear (7 participant ratings)

- “It’s easy to read on inspection, but confusion might stem from table headers in ”UNKNOWN” becoming attribute fields in ”MISSING””
- “It’s easier to know why it’s missing”

4. High clarity (6 participant ratings)

   - “...[Model] A with the two tables is more descriptive as in what is missing and what is tagged. Has more than one way to find [an explanation]”
   - “The top table is showing all of the people who have unknown middle initials. The bottom is showing why it is missing whether it is unknown, N/A, etc. To me, it is pretty easy to understand.”

5. Highest clarity (4 participant ratings)

   - “I think it’s extremely easy to understand. It’s clear enough why the results came out the way they did (mi either missing or having a question mark), and as long as you have some sort of reference to understand the tags (though they’re still relatively clear) it seems anyone could figure out what data is missing and for what reason.”
   - “Model A had two tables but the tables seemed to go together a little bit better.” [than the other cases]
   - “It is pretty intuitive how the comparison between tables works; having ekey”

Part 8.b. *Model B*

Model B is the _UNKNOWN relation with the tag from the _MISSING relation projected as a column next to the column for which data is missing. The table in this representation is compatible with the relational model, but is not union compatible with the _KNOWN relation. This means that it is an acceptable query result, but it does not comply with the KNOWN/UNKNOWN model for missing data.

1. Lowest clarity (no participant ratings)

2. Low clarity (no participant ratings)

3. Clear (8 participant ratings)

   - "Easier because you're showing the column that explains what classification the missing mi falls under."

   - "[Model] B is better because the information is within in one table which is more convenient."

4. High clarity (8 participant ratings)

   - "Better than model A"

   - "It is better than [Model] A because it puts the information into one table."

- “I think it’s easier to understand just because it puts all the information on one table instead of splitting it into two.”
- “I feel like Model B is the clearest of them all because it shows the relationship of the mi and why they’re missing.”
- “This model is easier to understand because the information for why it is missing is included right here on the table.”
- “Better because it is more compact, you can’t just look to the right on your corresponding row.”
- “... Having both the mi and the mi_TAG is helpful”
- “Better, they are right next to each other so its easier to see why”

5. Highest clarity (5 participant ratings)

   - “Yes, yes, because it’s cleaner and reduces unnecessary data”
   - “Better, it presents the explanation of the missing mi in a more visually accessible way.”
   - “Better than Model A. Model B provides the attribute to the missing data within the table, making it easier to read.”
   - “I like model B the best. Model C makes me think that UNK, N/A, or INV are the actual middle names. Also, in cases like ekey 24, I think I’d like to know what the invalid input is.”

Part 8.c. *Model C*

Model C is the _UNKNOWN relation with the tag from the _MISSING relation projected in place of the column of missing data. This representation replaces a missing data value of some domain type with a string of three characters which may be an acceptable presentation for a query result, but if these tuples were in the same relation as complete information, the *mi* domain would have to support multiple data types (i.e. a single character and a string of three characters). The relational model requires that all values in a domain be of the same type.

1. Lowest clarity (4 participant ratings)

    - “Mi intuitively would signify middle initial, but instead you had symbols in as the attributes implying some kind of code.”
    - “Worse since this replaces the mi with the type of unknown it is. Thus it would be confusing when comparing to something with an actual mi.”

2. Low clarity (1 participant rating)

    - “Worse than model A and B”

3. Clear (3 participant ratings)

    - “Better because that ’?’ is gone and the mi just displays the status of the missing mi.”

- "Worse, does not specify attributes missing which makes person have to do more reference"

4. High clarity (5 participant ratings)

5. Highest clarity (8 participant ratings)

   - "This is by far my favorite; I like the fact that it tells you why the data is missing in the very spot the data should be."
   - "It is better than both Model A and B. The table already includes the values such as "Unknown," etc in those fields. We only have have to look at one column."
   - "It's better than both because the data is compactly organized and the inclusion of the mi field seems almost useless in this case."
   - "About the same, collapsing the two columns together offers about the same readability."

9. Participant opinions requested as "Observations:"

This part of the study solicited the opinion of the participant as an SQL user based on all previous knowledge and experience as well as participation in the tutorial and feasibility study.

Question *9.a.* asked if missing data tags provided more information than nulls.

- **Yes** (19 participant responses)

- **No** (1 participant response)

- **Unsure** (1 participant response)

Question *9.b.* asked if the participant saw a benefit to representing missing data with tags in a DBMS.

- **Yes** (19 participant response)

- **No** (1 participant response)

- **Unsure** (1 participant response)

Question *9.c.* asked the participant how intuitive he or she found the MAYBE modifier.

1. Lowest intuitiveness (no participant ratings)

2. Low intuitiveness (1 participant rating)

3. Clear (6 participant ratings)

4. High intuitiveness (8 participant ratings)

5. Highest intuitiveness (6 participant ratings)

8.4 Feasibility study summary

The goal of this study was to determine if the KNOWN/UNKNOWN model is able to represent the available information about missing data values in a way that is understandable to users and meets their needs.

8.4.1 Context

There is not sufficient data from the feasibility study to reach statistically significant conclusions, but study participant responses suggest reasonable observations. The study allows comparisons between the experiences of novice SQL users with the representation of missing data using the standard SQL null and the KNOWN/ UNKNOWN model. These comparisons are significant.

8.4.2 Observations about nulls

Study participants quickly grasped the idea that a null represents a missing data value in a table row and column. A majority of participants were able to correctly answer questions about null, write queries to match null, express that it cannot be known why a data value marked by null is missing, and accept that standard SQL cannot be used to query why data is missing. Four participants did indicate that null did not meet expectations. One participant tried to find the data he or she knew would be in the _MISSING table of the KNOWN/UNKNOWN model, the *emp* table could not provide this information.

8.4.3 Observations about KNOWN/UNKNOWN

Study participants grasped the idea that the metadata in the _MISSING table identified which data values were missing and explained why the data was missing. Ninety-five percent were able to incrementally write correct queries using the MAYBE modifier a short time after a brief introduction. Most had opinions about the best presentation for incomplete information metadata that varied from a preference for short, concise reports in one table to a preference for complete and accurate information about what was missing and why, even if it was more difficult to interpret.

8.4.4 Analysis of missing data using metadata

Two of the twenty-one participants were able to determine the correct answer to a complex and confusing question expressed in a simple form.
Question (7):
"Why does the question (2) query return more rows than the question (5) query?"
To answer this question, study participants needed to analyze the differences between a query for nulls from table *emp* and a query for attributes that MAYBE equal an unknown value from relvar *person.* Using the KNOWN/UNKNOWN model an attribute that may be one value may be any value, if it is unknown, but its attribute is applicable.

There were two good answers, but the best answer to question (7):

"False, they have the same number of rows (but different rows)."

Question (7) is an indirect question. It is a false statement because query results for questions (2) and (5) have the same number of rows, but they are not the same rows. The real question is, if the results for questions (2) and (5) do not represent the same information, why not?

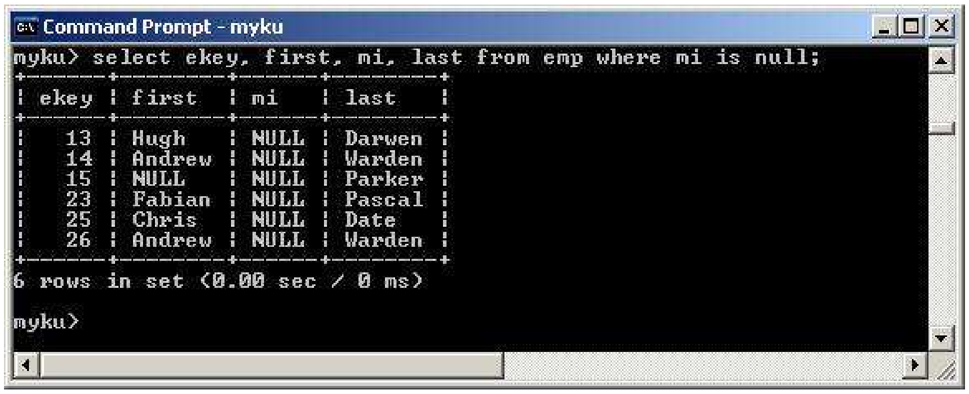
The study participant who entered the best answer also included a comment at the end of the script document describing his or her efforts to investigate the difference between two fundamentally similar queries for missing data. This individual intuitively knew that investigation was necessary, knew where to start, and tried the following:

```
In question 2, I couldn't figure out how to get the reasons
for being null to show up.  I tried the queries:
  select EKEY, FIRST, MI, LAST from EMP where MI maybe IS NULL;
  select EKEY, FIRST, MI, LAST from EMP where MI maybe = NULL;
  select EKEY, FIRST, MI, LAST from EMP where MI maybe = NULL;
To no avail.
```

A tutorial on investigation techniques for the KNOWN/UNKNOWN metadata would train both participants who gave correct answers to question (7) how to fully explain why their answers were correct. An example of how this would be done follows below.

An analysis of question (2)

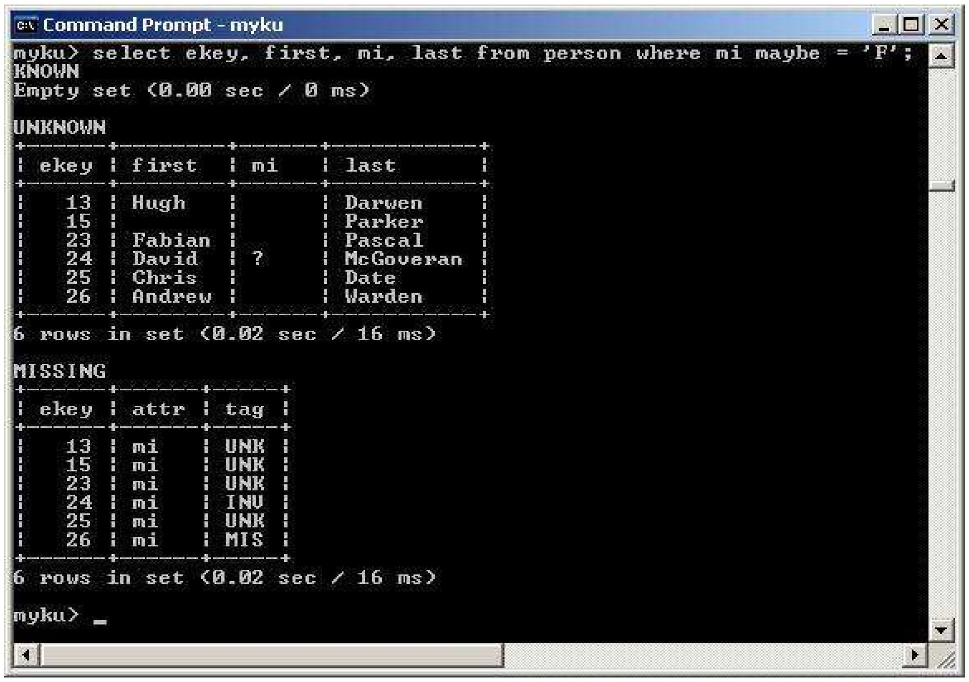
The query from question (2) shown in figure 31 has six rows and includes "Andrew Warden" twice, but "David McGoveran" who has an invalid middle initial (i.e. a question mark '?' instead of a null) is not included.



```
myku> select ekey, first, mi, last from emp where mi is null;
+------+--------+------+--------+
| ekey | first  | mi   | last   |
+------+--------+------+--------+
|   13 | Hugh   | NULL | Darwen |
|   14 | Andrew | NULL | Warden |
|   15 | NULL   | NULL | Parker |
|   23 | Fabian | NULL | Pascal |
|   25 | Chris  | NULL | Date   |
|   26 | Andrew | NULL | Warden |
+------+--------+------+--------+
6 rows in set (0.00 sec / 0 ms)

myku>
```

Figure 31: Query results for feasibility study question 2

An analysis of question (5)



```
myku> select ekey, first, mi, last from person where mi maybe = 'F';
KNOWN
Empty set (0.00 sec / 0 ms)

UNKNOWN
+------+--------+------+-----------+
| ekey | first  | mi   | last      |
+------+--------+------+-----------+
|   13 | Hugh   |      | Darwen    |
|   15 |        |      | Parker    |
|   23 | Fabian |      | Pascal    |
|   24 | David  | ?    | McGoveran |
|   25 | Chris  |      | Date      |
|   26 | Andrew |      | Warden    |
+------+--------+------+-----------+
6 rows in set (0.02 sec / 16 ms)

MISSING
+------+------+-----+
| ekey | attr | tag |
+------+------+-----+
|   13 | mi   | UNK |
|   15 | mi   | UNK |
|   23 | mi   | UNK |
|   24 | mi   | INV |
|   25 | mi   | UNK |
|   26 | mi   | MIS |
+------+------+-----+
6 rows in set (0.02 sec / 16 ms)

myku> _
```

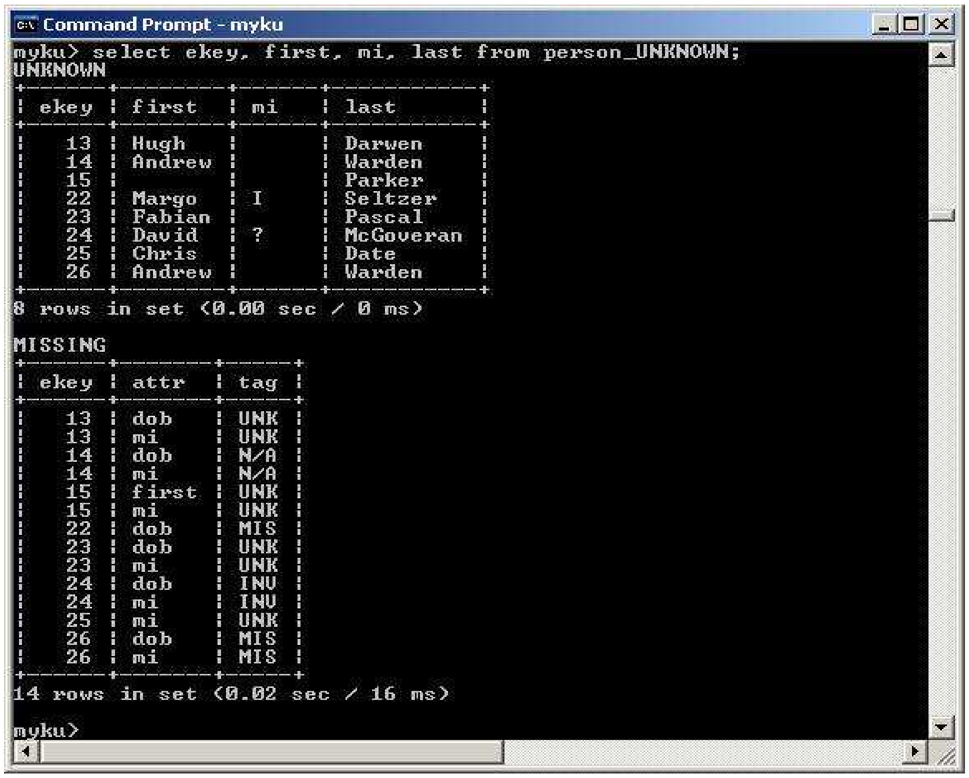
Figure 32: Query results for feasibility study question 5

The query from question (5) shown in figure 32 also has six rows and includes

“David McGoveran” whose middle initial is tagged as an invalid data value for an attribute that is applicable to the row, but “Andrew Warden” appears once with a middle initial tagged as withheld at input. An awareness that null is not a value and a careful comparison of the two results sets can explain why “David McGoveran” does not appear in the answer to question (2). The question mark is not a middle initial and was probably used as a sentinel value during data entry to indicate that the middle initial was missing. This is an example of information lost between data collection and data reporting in a way that is confusing.

An analysis of question (7)

Investigation as shown in figure 33 is necessary to determine why “Andrew Warden” appears twice in the question (2) query and only once in the question (5) query. A query for ekey, first, mi, last columns for all rows in the person_UNKNOW table shows results with both instances of “Andrew Warden” and conflicting reasons for the missing middle initial. This is also confusing, but it includes enough information to suggest a resolution to the problem (i.e. each row represents the same individual, ”Andrew Warden” was entered into the database twice, he does not have a middle initial for one of two possible reasons, and when it is determined what his middle initial is or why he has no middle initial, the problem can be corrected).



```
myku> select ekey, first, mi, last from person_UNKNOWN;
UNKNOWN
+------+--------+------+------------+
| ekey | first  | mi   | last       |
+------+--------+------+------------+
|   13 | Hugh   |      | Darwen     |
|   14 | Andrew |      | Warden     |
|   15 |        |      | Parker     |
|   22 | Margo  | I    | Seltzer    |
|   23 | Fabian |      | Pascal     |
|   24 | David  | ?    | McGoveran  |
|   25 | Chris  |      | Date       |
|   26 | Andrew |      | Warden     |
+------+--------+------+------------+
8 rows in set (0.00 sec / 0 ms)

MISSING
+------+-------+-----+
| ekey | attr  | tag |
+------+-------+-----+
|   13 | dob   | UNK |
|   13 | mi    | UNK |
|   14 | dob   | N/A |
|   14 | mi    | N/A |
|   15 | first | UNK |
|   15 | mi    | UNK |
|   22 | dob   | MIS |
|   23 | dob   | UNK |
|   23 | mi    | UNK |
|   24 | dob   | INV |
|   24 | mi    | INV |
|   25 | mi    | UNK |
|   26 | dob   | MIS |
|   26 | mi    | MIS |
+------+-------+-----+
14 rows in set (0.02 sec / 16 ms)

myku>
```

Figure 33: Query results to investigate feasibility study question 7

8.4.5 Study conclusion

The ability of study participants to learn SQL querying in less than an hour and use it as needed to answer questions about missing data in standard SQL tables using nulls and in KNOWN/UNKNOWN relvars using extended SQL and metadata indicates that the participants were capable and qualified to compare the two models for missing data and offer opinions.

Ninety-five percent were able to work with missing data and answer questions (see section 8.3.2). Twenty of the twenty-one participants who completed the study script used the KNOWN/UNKNOWN model to write correct queries for questions (4), (5), and (6) and answered question (9.c) indicating that the maybe modifier was clearly intuitive. Nineteen of the twenty-one participants answered "yes" to questions (9.a) agreeing "tags" provide more information than "nulls" and (9.b) acknowledging the representation of missing data with "tags" to be beneficial.

Ten percent were able to see the subtle difference between nulls and metadata. Two of the twenty-one participants who completed the study script correctly identified the issue raised by question (7) and attempted to determine why matching nulls in the *emp* table and matching maybe-tuples in the *person* relvar retrieved rows representing different people (see section 8.4.4). The study suggests that a majority of participants found the KNOWN/UNKNOWN model sufficiently intuitive to understand it and use it while a small group showed the potential to use its represent ion of missing data to resolve complex problems related to incomplete information.

CHAPTER 9 Conclusion

If a database permits missing data, it should be supported and its representation must be natural for the user. This requires a coherent implementation in the DBMS with a clear demarcation between the system's responsibilities and the user's responsibilities.

The relational database model in this research allows complete information and incomplete information to exist in the same relation variable, but places missing data values in a separate category. Metadata that can explain what is missing and why, is included in the system catalog and in the relvar. It is possible to write queries for exact matches, maybe-matches or both. Additionally, users can query the MISSING table to learn more about the missing data.

9.1 Metrics

One of the problems with using null to represent missing data values is that users become aware of nulls in a database when a query result is confusing. SQL nulls are known to give incorrect results [68, p.510]. Statistics that compare complete information to incomplete information in relational databases and query results are not available. [1] Most databases are missing some values [11, p.24] [18, p.314], but the

[1]An extensive search of the Internet and literature found matches to search terms such as relational DBMS statistics, frequency of query, missing or incomplete data, and null, but no study of or reference to complete information contrasted with missing data.

focus for database usage is queries written to match known information and answer questions. To take advantage of a better representation for incomplete databases, a method to investigate the impact of missing data on query results is needed. In addition to the main purpose of the KNOWN/UNKNOWN model, it can also provide missing data metrics that show how much information is missing from a relation variable. These metrics are calculated using SQL and are a first step in investigating missing data's impact on results.

A relvar's lack of completeness can be measured as value in the range [0, 1] using equation (2) where function `Card()` returns the cardinality of a relation. The smaller this value, the more complete the specified relvar.

$$\text{Relvar\_incompleteness } (R) = \frac{\texttt{Card}(R\text{\_UNKNOWN})}{\texttt{Card}(R\text{\_KNOWN}) \; + \; \texttt{Card}(R\text{\_UNKNOWN})} \tag{2}$$

A measurement of how many data values may be missing from the relvar ranges from a minimum of zero to a maximum of all possible missing attributes in the _UNKNOWN relation. The range of this measurement depends on the function `Degree`() to return the number of attributes in a tuple or a tuple's key. The smallest degree of a relation is one because relations require keys, but no key may have a missing value therefore the minimum number of attributes that can have a missing value is zero. The minimum and maximum number of possible missing data values in a tuple are defined in equation (3).

$$\text{Tuple metrics}: \begin{cases} \text{min\_tuple}(R) = & 0 \\ \text{max\_tuple}(R) = & \texttt{Degree}(R) \; - \; \texttt{Degree}(R\text{\_KEY}) \end{cases} \tag{3}$$

The actual current number of missing data values in a relation variable is the cardinality of the _MISSING relation. This measurement and the range from the minimum of zero to the maximum possible number of missing values in a relvar are defined by the functions in equation (4).

$$\text{Relvar metrics}: \begin{cases} \text{min\_relvar}(R) = & \texttt{min\_tuple}(R) \\ \text{cur\_relvar}(R) = & \texttt{Card}(R\text{\_MISSING}) \\ \text{max\_relvar}(R) = & \texttt{max\_tuple}(R) \; \times \; \texttt{Card}(R\text{\_UNKNOWN}) \end{cases} \tag{4}$$

The ratio of the number of tuples in the missing relation to the total number of tuples in the known and unknown relations is the average number of missing data values for each tuple in the relvar as shown in equation (5). This measures a level of how incomplete a relvar is at any moment. A value of zero indicates that the relvar is complete. A value above zero indicates that data is missing. This measurement's upper bound is the `max_tuple`(R). As the value of this metric approaches its upper bound, the more incomplete the relvar. The smaller this metric is the better.

$$\text{Tuple\_avg }(R)\begin{cases} 0 & \text{when } \texttt{Card}(R\text{\_UNKNOWN}) = 0 \\ = & \dfrac{\texttt{Card}(R\text{\_MISSING})}{(\texttt{Card}(R\text{\_KNOWN}) + \texttt{Card}(R\text{\_UNKNOWN}))} \end{cases} \tag{5}$$

Use of these metrics allows the user to be aware of missing data's possible impact on query answers.

9.2 Advantages of KNOWN/UNKNOWN model

The KNOWN/UNKNOWN model is an improved representation for missing data that complies with the relational model.

9.2.1 New model avoids problems of null

In the SQL standard, null is an indicator rather than a value. A query for IS NULL can find attributes that do not have a data value, but there is no interpretation beyond *missing data*.

In the KNOWN/UNKNOWN model an attribute missing data may contain an invalid value or no value, but metadata about what is missing and why is stored as data values in relations.

The KNOWN/UNKNOWN model supports searching for attributes that do not have a data value by category (class and type). Class explains the relationship (applicable, invalid, non-applicable or unknowable) of the missing data to the item the known data would describe. Types explain why the data is missing (unknown, not yet known, withheld by user, not applicable, or removed for reason).

9.2.2 Metadata available to user and DBMS

More information about missing data is available at data entry than can be represented in other missing data models. A data entry application able to capture information about missing data can use the KNOWN/UNKNOWN model to store this information as metadata. This metadata is a component of each relational database that implements the KNOWN/UNKNOWN model.

The DBMS uses the metadata to make processing decisions related to missing data.

For database users, the advantage of using metadata is two-fold. First, complete information about missing data allows the user to interpret query results and reach a meaningful conclusion or determine why a conclusion is not possible. Second, by knowing all that was known at data entry, the user may determine how to seek the missing data and update the database.

9.2.3 Backward compatibility with nulls

The KNOWN/UNKNOWN model is backward compatible with the SQL standard for missing data using nulls and 3-valued logic. Legacy databases, queries, and applications that rely on null can remain in use during the transition from the SQL standard to the KNOWN/UNKNOWN model.

If there are no metadata tables, the DBMS processes missing data using nulls returning a single table in the result set. There is a need for a technique that allows database users to select between models when there is a choice (see chapter 10). While the choice between null and metadata is always available, its purpose is to provide a straightforward migration path from the SQL standard to the KNOWN/ UNKNOWN model.

9.2.4 Database maintenance and application development

The primary purpose of metadata is to increase the information content of query results, but metadata can be used to evaluate the state of a database.

Queries that combine exact matches and maybe-matches can be used to find duplicates inserted using generated keys instead of domain keys. If an item is added to the database before all data values are available, it must be updated later. When an update is inserted as if it is new data, there is now a maybe-duplicate item in the database. The KNOWN/UNKNOWN metadata can be used to find this kind of error.

If an attribute is no longer applicable to any database item, it may be feasible to delete the attribute from the database. Being aware that a query creates result sets with SQL generated missing values allows the user to develop a different query for the same information or modify the presentation of the result set to reduce confusion.

The well-defined metadata component of the KNOWN/UNKNOWN model supports database development during application design. Documented categories for missing data and the capability for expansion encourages planning for missing data.

If missing data is needed in the application, the use cases developed during project planning contribute to application validation and may be used in regression testing.

9.2.5 Database metrics for missing data

There is not enough information about missing data in the SQL standard null representation to support database metrics. The capability to measure missing data by category provides a method to check database completeness and offers addition research opportunities.

9.3 Summary

The KNOWN/UNKNOWN model provides an improved representation for data values that are missing from a relational database. This representation maps to a presentation that is intuitive to use and supports the interpretation of missing data (see section 8.4.5). The separate categories for complete and incomplete data model the real world in a way that is consistent with the relational model and the closed-world assumption. The metadata in this model helps explain the incomplete information. Understanding what is missing and why improves data administration and can facilitate database update by identifying data values that need to be retrieved from the real world.

CHAPTER 10 Future Work

This research identifies an improved representation for missing data in the relational model, but it is only a start towards a useful DBMS implementation.

10.1 Complete implementation of model in MySQL

The MySQL embedded server was initially used to implement a query only version of the KNOWN/UNKNOWN model and not a complete DBMS. Implementing the full model in the server requires changes to MySQL's threading structure and its subsystem. To do this requires a detailed knowledge of MySQL's internal data structures and functions.

MyKU has served as a prototype which will be replaced by a complete solution. Brooks [9, p.116] suggests programmers should expect to throw away the first version of any system. A plan for a complete implementation of the KNOWN/UNKNOWN model begins with an evaluation of what works in the MyKU client and query rewrite components, what failed in the initial MyKU implementation (i.e. access to intermediate results and problems with duplicate removal), and what is needed in a full implementation. The second version of the system will build from MyKU using step-wise refinement towards a complete integrated model in the MySQL DBMS. As each MyKU refinement is developed, the MySQL internal data structures, processes, and functions will be investigated and documented.

10.1.1 Intermediate results

The steps required to create a KNOWN/UNKNOWN result using an integrated known and unknown table (see section 7.1.1) indicate the need for intermediate result tables. These table may be temporary, but must persist during both iterative and recursive processing by complex queries that project from a union or a product and for queries that rely on subqueries. In some cases more than one subquery result may be needed requiring more than one stage for an intermediate result.

This approach is a refinement towards an implementation in the MySQL DBMS. If it is feasible to create multiple temporary tables using SQL written by MyKU, an implementation of a KNOWN/UNKNOW relvar in MySQL is also feasible.

10.1.2 Duplicate removal

While SQL does not require tables to have a key, the KNOWN/UNKNOWN model requires a primary key for each table and it depends on the key to connect rows with missing data values to the _MISSING table. The current approach of MyKU forces the primary key into any projection of rows with unknown values represented in the _MISSING table to maintain the connection between unknown and missing tables. This makes duplicate removal a problem.

The presence of unique keys in MyKU's intermediate results means all rows are unique and not duplicated, it is necessary to manage an intermediate result using a different approach. A projection of a join without keys from _UNKNOWN and _MISSING into a temporary intermediate result table would eliminate duplicates in the intermediate results. The pseudo-key could then be added when the result relvar is projected from this intermediate result. Another way to do this is to project only specified columns into a temporary table in which all columns are part of the key and use the duplicate key constraint to eliminate duplicate rows as they are inserted.

Using one of these approaches is a refinement towards an implementation in the MySQL DBMS. If it is feasible to create multiple temporary tables using SQL written by MyKU, an implementation in MySQL using its method for duplicate removal is also feasible.

10.1.3 MyKU client

The MyKU program's client components accept SQL statements from the user, send SQL statements to the scanner/parser, send SQL statements to the MySQL DBMS, and present either a KNOWN/UNKNOWN relvar or single relation result to the user. Each of these components is needed in an improved MyKU and as part of a DBMS version of the KNOWN/UNKNOWN model.

Query parser

The abstract syntax tree representation of SQL statements in MyKU is created once and may be used many times. After using the AST, its allocated memory is freed before the next statement is parsed. During initial development of MyKU it was useful to visit the AST nodes to determine how queries could be rewritten. Now the AST is only visited once and storing only what is needed as it is parsed is a more efficient approach. This is the method used by MySQL. Replacing the AST in MyKU is another step toward moving the KNOWN/UNKNOWN model into the DBMS.

Information schema

Information from the database schema is needed to expand MyKU's ability to manage intermediate results and rewrite queries. In the next iteration of MyKU schema data will be read using SQL and stored in MyKU data structures. Backward compatibility with nulls should be implemented using the schema and a first step is to determine if a table has a _MISSING relation or not. This is a step toward moving the KNOWN/UNKNOWN model into the DBMS where direct access to the schema is possible.

10.1.4 MySQL DBMS

The complete solution to the missing data problem implementing the KNOWN/ UNKNOWN model in MySQL uses the integrated known and unknown table design (see section 7.1.1). This design option marks attributes missing data values using the null bit and metadata from a corresponding _MISSING table. This is the second version of the KNOWN/UNKNOWN model.

Lexical scanner

MySQL uses a hand-written scanner which will have to be modified to add tokens needed for the MAYBE operator and missing data tags for table definitions.

Statement parser

MySQL uses a parser generated by Bison [28] and its SQL grammar will require changes to add the KNOWN/UNKNOWN extensions to SQL. Some of these changes can be taken from the grammar used in the MyKU embedded server subset, but MySQL's grammar is significantly more complex. Adding the KNOWN/UNKNOWN model to the MySQL thread structure during statement parsing is a necessary step.

Backward compatibility

A modification to allow standard SQL nulls to represent missing data or the _MISSING table in a way that makes sense to the user is needed. The current version evaluates a query and if it includes the IS NULL string or if the table is identified as using nulls only, a single query is passed to MySQL. What is needed is a check in MySQL for the existence of the _MISSING table and a check for a preferred mode of either NULLS or KNOWN/UNKNOWN. Using a mode switch for queries supports selection of a missing data representation in a way that allows both missing data models to coexist.

10.1.5 Set difference

MySQL does not provide set difference. If this can be implemented for the KNOWN/ UNKNOWN model, it will have to be done in the MySQL server. A workaround for set difference using a test for EXIST/NOT EXIST with a subquery is feasible when support for recursive queries (subquerying) is implemented in MySQL server for the KNOWN/UNKNOWN model.

10.2 Query analyzer to identify tautologies

The query parser can use resolution to reduce complex queries to a simpler logic. Removing complexity and representing a query as predicate that is a conjunction or a disjunction is a method that can identify some tautologies. Some research will be required to determine if this is feasible. Codd asserted that this problem is unsolvable in a general way using predicate logic [14, p.64].

10.3 Modifications identified from acceptance study

10.3.1 Presentation

A significant number of feasibility study participants expressed a preference for query results presented in a single table with missing data tags either next to or in place of the missing data column (see section 8.3.2). While not compliant with the relational model or the KNOWN/UNKNOWN model, it is feasible to include these alternate presentations as query options in a database client program. This feature is part of the user application, but if it is used often the DBMS could support it as an option.

10.3.2 MAYBE operator

Querying for maybe-tuples using the MyKU implementation requires that the comparison operators include a MAYBE modifier. This modifier indicates missing data matches by class and type (i.e. tag) using metadata in the _MISSING table. Feasibility study participants often incorrectly used MAYBE as an operator. This makes sense intuitively. There is no difference between the concept of missing data values being MAYBE equal and being MAYBE not equal. Changing MAYBE to a comparison operator is feasible and does not change the KNOWN/UNKNOWN model.

10.3.3 Query missing data by tag

Feasibility study participants learned to search for null using the IS NULL and attempted to search for KNOWN/UNKNOWN missing data using IS UNK or IS N/A as match criteria. This kind of match criteria is possible, but is constrained by flexible support for missing data types (see section 10.4).

10.4 Missing data types and metadata

The KNOWN/UNKNOWN model is designed to allow modifications to the types of missing data in the metadata. It is necessary to allow the range of missing data types to be established when tables are created or altered. It should be possible to allow, restrict, or expand missing data types as required by database design.

Bibliography

Bibliography

[1] Serge Abiteboul and Nicole Bidoit. Non first normal form relations to represent heirarchically organized data. In *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, (Waterloo, Apr.1984), pages 191–200, New York, NY, USA, 1984. ACM.

[2] Actian. *Ingres 10.0 SP1 SQL Reference Guide.* Actian Corporation, Redwood City, CA, 2012.

[3] ANSI SPARC/DBMS Study Group. Reference model for dbms standardization (interim report). *ACM SIGMOD Record*, 7(2):1–140, 1975.

[4] Daniel Barbara, Hector Garcia-Molina, and Darl Porter. The management of probabilistic data. *IEEE Transactions on Knowledge and Data Engineering*, 4:487–502, 1992.

[5] Lynn Beighley. *Head First SQL.* O'Reilly, Sebastopol, CA, 2007.

[6] Charles A. Bell. *Expert MySQL.* Apress, Berkeley, CA, 2007.

[7] Joachim Biskup. A foundation of Codd's relational maybe-operations. *ACM Transactions on Database Systems*, 8:608–636, 1983.

[8] Robert Bosak, Richard F. Clippinger, Carey Dobbs, Roy Goldfinger, Renee B. Jasper, William Keating, George Kendrick, and Jean E. Sammet. An Information Algebra: Phase 1 Report—Language Structure Group of the CODASYL Development Committee. *Communications of the ACM*, 5:190–204, 1962.

[9] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, Boston, MA, 1995. 2nd Edition (Anniversary Edition).

[10] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[11] E. F. Codd. Understanding Relations (Installment #7). *ACM SIGMOD Record*, 7(3-4):23–28, 1975.

[12] E. F. Codd. Extending the Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems*, 4(4):397–434, 1979.

[13] E. F. Codd. Relational Database: A Practical Foundation for Productivity. *Communications of the ACM*, 25(2):109–117, 1982.

[14] E. F. Codd. Missing Information (Applicable and Inapplicable) in Relational Databases. *ACM SIGMOD Record*, 15(4):53–78, 1986.

[15] E. F. Codd. *The Relational Model for Database Management: Version 2.* Addison-Wesley, Reading, MA, 1990.

[16] Hugh Darwen. How To Handle Missing Information Without Using NULL. First presented at Warwick University 2003 updated 2006, September 2006. http://www.dcs.warwick.ac.uk/ hugh/TTM/Missing-info-without-nulls.pdf.

[17] S. K. Das. Modal logics in the theory of relational databases. *Information Processing Letters*, 57(1):1–7, 1996.

[18] C. J. Date. *'Null Values in Database Management' in Relational Database: Selected Writings.* Addison-Wesley, Reading, MA, 1986.

[19] C. J. Date. *'NOT Is Not "Not"!'(Notes on Three-Valued Logic and Related Matters) in Relational Database Writings 1985-1989.* Addison-Wesley, Reading, MA, 1990.

[20] C. J. Date. *'Notes Toward a Reconstituted Definition of the Relational Model Version 1 (RM/V1)' in Relational Database Writings 1989-1991.* Addison-Wesley, Reading, MA, 1992.

[21] C. J. Date. *'Oh No Not Nulls Again' in Relational Database Writings 1989-1991.* Addison-Wesley, Reading, MA, 1992.

[22] C. J. Date. *'The Default Values Approach to Missing Information' in Relational Database Writings 1989-1991.* Addison-Wesley, Reading, MA, 1992.

[23] C. J. Date. *An Introduction to Database Systems.* Pearson Education, Boston, MA, 2004. 8th edition.

[24] C. J. Date and Hugh Darwen. *Databases, Types, and the Relational Model: The Third Manifesto.* Addison-Wesley, Reading, MA, 2007. 3rd edition.

[25] Guy de Tre, Rita de Caluwe, and Henri Prade. Null values in fuzzy databases. *Journal of Intelligent Information Systems*, 30(2):93–114, 2008.

[26] Debabrata Dey and Sumit Sarkar. A probabilistic relational model and algebra. *ACM Transactions on Database Systems*, 21:339–369, 1996.

[27] Debabrata Dey and Sumit Sarkar. A probabilistic relational model and algebra. *IEEE Transactions on Knowledge and Data Engineering*, 14:485–497, 2002.

[28] Charles Donnelly and Richard Stallman. *GNU Bison version 2.4.1 (Manual).* Free Software Foundation, Boston, MA, 2008.

[29] Didier Dubois and Henri Prade. Necessity Measures and the Resolution Principle. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-17(3):474–478, 1987.

[30] Paul DuBois. *MySQL (Developer's Library).* Addison-Wesley, Reading, MA, 2008. 4th edition (first printing).

[31] Norbert Fuhr. A probabilistic framework for vague queries and imprecise information in databases. In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, pages 696–707, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[32] G. H. Gessert. Handling Missing Data by Using Stored Truth Values. *ACM SIGMOND Record*, 20(3):30–42, 1991.

[33] John Grant. A Non-Truth-Functional 3-Valued Logic. *Mathematics Magazine*, 47(4):221–223, 1974.

[34] John Grant. Null Values in a Relational Data Base. *Information Processing Letters*, 6(5):156–157, 1977.

[35] John Grant. Incomplete information in a relational database. *Annales Societalis Mathematicae Polonae*, 3(3):363–378, 1980.

[36] Jan L. Harrington. *Relational Database Design Clearly Explained.* Morgan Kaufmann, San Francisco, CA, 2002. 2nd edition.

[37] John R. Levine. *flex & bison.* O'Reilly, Sebastopol, CA, 2009.

[38] Witold Lipski, Jr. On Semantic Issues Connected with Incomplete Information Databases. *ACM Transactions on Database Systems*, 4(3):262–296, 1979.

[39] Ken-Chih Liu and Rajshekhar Sunderraman. A Generalized Relational Model for Indefinite and Maybe Information. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):65–77, 1991.

[40] Zongmin Ma. *Fuzzy Database Modeling of Imprecise and Uncertain Engineering Informations.* Springer, Berlin, 2006.

[41] Daniel McNeill and Paul Freiberger. *Fuzzy Logic.* Simon & Schuster, New York, NY, 1993.

[42] J. M. Medina, M. A. Vila, J. C. Cubero, and O. Pons. Towards the implementation of a generalized fuzzy relational database model. *Fuzzy Set and Systems*, 75:273–289, 1995.

[43] Microsoft. *Microsoft Office Access 2003.* Microsoft Corporation, Redmond, WA, 2003. Office Help System.

[44] Microsoft. *Transact-SQL Reference.* Microsoft Corporation, Redmond, WA, 2012. http://msdn.microsoft.com.

[45] Marion R. Morrissett and Lorraine M. Parker. Handling missing data in relational databases: A survey. Submitted for publication, unpublished manuscript.

[46] Marion R. Morrissett and Lorraine M. Parker. Implementation of a new model for missing data in relational databases. In preparation, unpublished manuscript.

[47] Marion R. Morrissett and Lorraine M. Parker. A new model for missing data in relational databases. Submitted for publication, unpublished manuscript.

[48] Marion R. Morrissett, Larry R. Williams, Jr., and Lorraine M. Parker. A survey of fuzzy data with respect to its storage and retrieval. Submitted for publication, unpublished manuscript.

[49] Amihai Motro. Sources of uncertaintiy, imprecision, and inconsistency in information systems. In Amihai Motro and Philippe Smets, editors, *Uncertainty Management in Information Systems from Needs to Solutions.* Kluwer Academic Publishers, Boston, MA, 1996.

[50] T. William Olle. *The Codasyl Approach to Data Base Management.* John Wiley & Sons, Chichester, UK, 1978.

[51] Oracle. ask Tom. Internet, January 2012. http://asktom.oracle.com.

[52] Oracle. *MySQL 5.1 Reference Manual.* Oracle Corporation, Redwood Shores, CA, 2012.

[53] Oracle. *Oracle Database SQL Reference 11g Release 2.* Oracle Corporation, Redwood City, CA, 2012.

[54] Fabian Pascal. The Final Null in the Coffin: A Proposed Relational Solution to Missing Data. *Internet*, pages 1–29, January 2011. http://www.dbdebunk.com/publications.html.

[55] Vern Paxson. *Flex, version 2.5 (Manual).* University of California, Berkeley, CA, 1995.

[56] Frederick E. Petry. *Fuzzy Databases Principles and Applications.* Kluwer Academic Publishers, Boston, MA, 1996.

[57] PostgreSQL Global Development Group. *PostgresSQL 9.2.3 Documentation.* PostgreSQL Global Development Group, Internet Community, 2013.

[58] Raymond Reiter. On Clossed World Databases. In H. Gallaire and J. Minker, editors, *Logic and Databases.* Plenum Press, New York, NY, 1978.

[59] Raymond Reiter. Data bases: A logical perspective. *ACM SIGMOD Record*, 11(2):174–176, 1980.

[60] Raymond Reiter. A sound and sometimes complete query evaluation algorithm for relational databases with null values. *Journal of the ACM*, 33(2):349–370, 1986.

[61] Mark A. Roth, Henry F. Korth, and Abraham Silberschatz. Extended Algebra and Calculus for Nested Relational Databases. *ACM Transactions on Database Systems*, 13(4):389–417, 1988.

[62] Fernando Sanez-Perez. *Datalog Educational System V2.4 User's Manual.* Universidad Complutense de Madrid, Madrid, Spain, 2011.

[63] Margo Seltzer. Beyond Relational Databases. *Communications of the ACM*, 51(7):52–58, 2008.

[64] Technical Committee Group NCITS H2. *ANSI/ISO/IEC 9075-1:1999 (SQL/Framework).* ANSI/ISO/IEC International Standard (IS), Washington, DC, 1999.

[65] Technical Committee Group NCITS H2. *ANSI/ISO/IEC 9075-2:1999 (SQL/Foundation).* ANSI/ISO/IEC International Standard (IS), Washington, DC, 1999.

[66] Jeffrey D. Ullman. *Principles of database and Knowledge-Base Systems Volume I.* Computer Science Press, Rockville, MD, 1988.

[67] Yannis Vassiliou. *'Null values in data base management a denotational semantics approach' in Proceedings of the 1979 ACM SIGMOD international conference on Management of data.* ACM, New York, NY, 1979. 163-169.

[68] Andrew Warden. *'Into the Unknown' in Relational Database Writings 1985-1989.* Addison-Wesley, Reading, MA, 1990.

[69] Eugene Wong. A Statistical Approach to Incomplete Information in Database Systems. *ACM Transactions on Database Systems*, 7(3):470–488, 1982.

[70] Li Yan Yuan and Ding-An Chiang. A sound and complete query evaluation algorithm for relational databases with null values. In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, SIGMOD '88, pages 74–81, New York, NY, USA, 1988. ACM.

[71] L. A. Zadeh. Fuzzy Sets. *Information and Control*, 8:338–353, 1965.

[72] L. A. Zadeh. Fuzzy Sets as a Basis for a Theory of Possibility. *Fuzzy Sets and Systems*, 100(Supplement):9–34, 1999. Reprinted from Fuzzy Sets and Systems 1 (1978) 3-28.

[73] Lotfi A. Zadeh. Position Paper: Toward extended fuzzy logic–A first step. *Fuzzy Sets and Systems*, 160(21):3175–3181, 2009.

Appendix A **KNOWN/UNKNOWN model using set notation**

The relational model of data is based on set theory and first-order predicate logic [23, pp.67-68]. Relational databases are time-varying collections of relations composed of tuples of attribues containing data values taken from a universe of discourse *U*. The time-varying nature of this model supports addition, deletion, and modification as necessary to improve and maintain an accurate representation of the real world. [10]

A domain of data values as shown in equation (6) is defined from the power set of *U*. A specific domain holds values of a single data type and represents all possible values that an attribute of this domain may take on.

$$\mathit{Domain}\ D = \{d | d \in D, D \in \mathcal{P}\left(U\right)\} \tag{6}$$

The active domain as shown in equation (7) is a subset of the domain and represents the actual values present in a domain instance of the database [10, p.380].

$$\mathit{Active\ Domain}\ D_{\text{ACTIVE}} \subseteq \mathit{Domain}\ D \tag{7}$$

An attribute is an ordered triple represented by a name, a domain variable that takes its values from a domain and has a data type that may be constrained to a range of values as shown in equation (8).

$$Attribute\ A = \{name \mid string,\ \{\ domain \mid Domain,\ type \mid (number, string)\ \}\} \quad (8)$$

A relation has two parts. The first is the heading, which is a set of attribute triples as shown in equation (9). The heading is a predicate that ranges over the values of its attribute's domains. The predicate is a truth-bearer that may be evaluated using a truth-valued or a non-truth-valued function.

$$Heading = \{A_1, A_2, A_3, \ldots, A_n\} \quad (9)$$

The second part of the relation is its body, which is a set of tuples defined as a subset of the Cartesian products of the attribute domains in the heading, shown in equation (10).

$$Body \subseteq \{D_1 \times D_2 \times D_3 \times \ldots, D_n\} \quad (10)$$

Each tuple is a set of attribute data values as shown in equation (11) and is a proposition that is true or possibly true within the closed world of the database. A tuple of attribute values not present in the database is a false proposition. A query which returns an empty results relation, shows no data that can indicate a true proposition.

$$Tuple\ t \in Body \quad (11)$$

Queries are defined by a set of values paired with an attribute variable from the relation heading. Each query is a proposition that may be true in the closed world of the database. Queries are matched with tuples in the relation body. Query evaluation relies on a function to map query search values to data values in the database. Each tuple in the query results set is a true proposition about the database's knowledge and representation of a real-world item.

Membership of an attribute value d in a domain D is determined by the domain's characteristic function which returns a truth value from {0,1}. A *0* indicates that the element is not in the domain. A *1* indicates that the element is a member of the domain.

$$\mathrm{Char}_D(d) : D \to \{0, 1\} \tag{12}$$

An attribute that does not allow missing data can use a truth-valued function as shown in equation (13) for its truthbearer. An attribute value d that is a member of the domain and in the active domain is true. An attribute value d that is not a member of the domain or not in the active domain is false. This truth-valued function is based on 2VL truth tables and the closed-world assumption.

$$\text{Truth-valued}(d) = \begin{cases} T & \mathrm{Char}_{\mathrm{D}}(d) = 1 \land d \in D_{\text{ACTIVE}} \\ F & \mathrm{Char}_{\mathrm{D}}(d) = 0 \lor d \notin D_{\text{ACTIVE}} \end{cases} \tag{13}$$

An attribute that allows missing data must use a more complex truthbearer. A non-truth-valued function derived from 3VL truth table (see figure 34) and heuristics determines if a data value is true, maybe-true, or false. In addition to T and F, a truth value of M indicates that the missing data value is a MAYBE member of the domain. Until the data value is known, it cannot be determined with certainty that it is a member of the active domain. The non-truth-valued function shown in equation (14) uses metadata to determine maybe-matches and to decide when inapplicable and unknowable attributes are false matches. For the case of complete information, the non-truth-valued function evaluates the data value using the truth-valued function.

AND	t	m	f
t	t	m	f
m	m	m	f
f	f	f	f

OR	t	m	f
t	t	t	t
m	t	m	m
f	t	m	f

NOT	
t	f
m	m
f	t

$MAYBE$	
t	f
m	t
f	f

Figure 34: 3-valued logic truth tables {*true, maybe, false*}

$$\text{Non-Truth-Valued}(d) = \begin{cases} M & \textit{applicable and value missing} \\ M & \textit{applicable and value invalid} \\ F & \textit{attribute inapplicable} \\ F & \textit{value unknowable} \\ \text{Truth-Valued}(d) & \textit{applicable and value known} \end{cases} \tag{14}$$

The KNOWN/UNKNOWN model as shown in equation (15) eliminates the null by replacing it with metadata. Metadata related to the entire database is stored in the system catalog, but there is also metadata closely associated with a relation that allows missing data. A relation variable (relvar) which allows missing data uses three relations to represent complete and incomplete information. The KNOWN relation stores only tuples with complete information. The UNKNOWN relation stores tuples that are missing data values and may be matches. Information about what data is missing from the UNKNOWN relation and why it is missing is stored in the MISSING relation.

$$\textit{Relvar}\ \textsc{r} = \{\textsc{r\_known} \mid \textit{table},\ \{\textsc{r\_unknown} \mid \textit{table},\ \textsc{r\_missing} \mid \textit{table}\}\} \quad (15)$$

Appendix B **Requirements for software**

The user needs two capabilities defined in the requirements. The first is the ability to write database queries and understand the results when data values may be missing. The second is an awareness of missing data with the understanding that the databases must be updated when data becomes available.

Tasks to support these capabilities include table creation, alteration, and deletion; row insertion, update, and deletion; and operations necessary to query data tables and metadata tables. These operations must be consistent when data values are missing and when they are not. The requirements are capabilities that extend MySQL to support a complete implementation of the KNOWN/UNKNOWN model for missing data.

B.1 Capabilities

B.1.1 Data Definition

1. **Column Definition**

   DBMS shall allow a column to be designated missing data capable.

   This capability corresponds to MySQL command to allow nulls.

   (a) This capability should be allowed for any non-key column in a table.

   (b) This capability shall be allowed at or after table creation.

(c) The MISSING data types to be allowed shall be specified.

(d) Supported MISSING data types shall be from the database's system catalog of missing data types.

2. **Column Definition**

   DBMS shall allow a column to be designated missing data incapable.

   This capability corresponds to MySQL command to disallow nulls.

   The user will be warned of values that default to zero or the empty string.

   (a) This capability should be allowed for any column in a table.

   (b) This capability shall be allowed at or after table creation.

   (c) Row columns that hold missing data values shall be set to zero or the empty string and the user shall be warned.

   (d) Metadata shall be removed from the the MISSING table.

   (e) Information about the missing data shall be lost.

   (f) Key attributes are missing data incapable by default.

3. **Table Definition**

   DBMS shall create tables that support missing data.

   This capability corresponds to MySQL command to create a table that allows nulls.

(a) This shall be the case for base tables created as a KNOWN/UNKNOWN relvar.

(b) The MISSING data types supported shall be those available in the database schema.

4. **Table Definition**

   DBMS shall not create tables that allow key columns to be missing data capable.

   This capability equates with MySQL command to identify a table key that does not allow nulls.

   (a) MySQL does not allow nulls in columns that are part of a table key.

   (b) The work around is to fabricate a unique key using AUTO_INCREMENT.

   (c) It is not feasible to allow unknown key values.

5. **Table Definition**

   DBMS should allow columns with missing data to be indexed.

   This capability corresponds to MySQL command to index a column with nulls enabled.

   (a) If indexed or not, a column with nulls can match using "is null."

   (b) A corresponding capability shall match using "is unknown" or a similar method defined for the KNOWN/UNKNOWN model.

   (c) A metadata table shall exist in the database schema and include a reference to for each MISSING data column.

6. **Table Deletion**

   DBMS shall allow tables that support KNOWN/UNKNOWN missing data to be deleted.

   This capability corresponds to MySQL command to drop a table that allows nulls.

   (a) If tables can be created, the capability to drop a table is necessary to correct some table definition errors.

   (b) All references to missing data values shall be removed from the schema.

7. **Table Modification**

   DBMS shall support all categories of missing data including those not yet identified.

   This capability corresponds to MySQL's ability to alter tables that allow nulls.

   (a) All components of a table definition should be modifiable.

   (b) MISSING data types shall be part of the table definition.

   (c) The database schema of missing data types is the source missing data categories.

   (d) The database schema of missing data types shall be alterable.

B.1.2 Data manipulation

1. **Data Insert**

   DBMS shall allow a row that with KNOWN/UNKNOWN missing data to be inserted into a table.

   This capability corresponds to MySQL's ability to insert a row containing a null.

   (a) This capability shall have a method to indicate that a column of a row is to hold a missing data value.

   (b) This method shall support a way to indicate a MISSING data type for a missing data value.

   (c) Each missing data type shall be a supported MISSING data type enabled for the column.

2. **Data Update**

   DBMS shall allow a MISSING data type to replace a data value in a table row and column.

   This capability corresponds to MySQL's ability to replace a data value with a null.

   (a) The updated column shall be missing data capable.

   (b) This capability shall have a method to indicate that a column of a row is to hold a missing data value.

(c) This method shall support a way to indicate a MISSING data type for a missing data value.

(d) Each missing data type shall be a supported MISSING data type enabled for the table column.

3. **Data Update**

   DBMS shall allow a valid data value to replace MISSING data type in a table row and column.

   This capability corresponds to MySQL's ability to replace a null with a data value.

   (a) This capability is the case where missing data value is now known.

   (b) The known data value must be of the data type assigned to the table column being updated.

4. **Data Query**

   DBMS shall support EXACT matches for queries using known data values.

   Corresponds to MySQL's match criteria in a SELECT/WHERE statement.

   (a) A known data value in the query shall be compared to a known data value stored in a table column.

   (b) The query data type to match shall be the data type assigned to the table column being searched.

(c) An EXACT match shall be evaluated in the context of a query's comparison operator(s)

$(<, \leq, =, \neq, \geq, >)$.

(d) A query shall be simple, complex, or compound.

5. **Data Query**

   DBMS shall support MAYBE matches for queries using known data values.

   Corresponds to MySQL's match criteria in a SELECT/WHERE statement.

   (a) A known data value in the query shall be compared to an UNKNOWN missing data value in a table column.

   (b) The query data type to match shall be the data type assigned to the table column being searched.

   (c) A MAYBE match shall be evaluated in the context of a query's comparison operator(s) $(<, \leq, =, \neq, \geq, >)$.

   (d) A comparison operator should use the term "MAYBE" as a modifier in its syntax

   (i.e. "Smith" MAYBE = customer.key).

   (e) If the table column holds an UNKNOWN missing data value that could be updated to an exact data value that evaluates to true, the match shall be true. Otherwise it shall be false.

(f) A query shall be simple, complex, or compound.

6. **Data Query**

   DBMS shall support EXACT matches for queries using MISSING data types in place of data values.

   Corresponds to MySQL's IS NULL match criteria in a SELECT/WHERE statement.

   (a) The missing data value of the query search criteria shall be represented by an UNKNOWN missing data type.

   (b) The query shall have a method to indicate the UNKNOWN missing data type to match.

   (c) An UNKNOWN missing data value in a query shall be a search of the metadata in the MISSING table.

   (d) This capability shall include a help option aliased by a Data Administration SHOW command.

7. **Data Query**

   DBMS shall support MAYBE matches for queries using MISSING data types in place of data values.

   Corresponds to MySQL's use of a SELECT statement without criteria for matching.

   (a) This capability is the case of selecting all rows in a table (i.e. SELECT statement without the WHERE option).

8. **Data Query**

   DBMS shall have a method of indicating a MISSING data type in a query result table.

   Corresponds to MySQL's representation of nulls as "NULL" in a results set to indicate missing data values.

   (a) MISSING data types shall be clearly indicated and identified in query result tables.

   (b) The MISSING data type for missing data created by an SQL operator such as in an outer join shall meet this requirement.

B.1.3 Database administration

1. **Database Administration**

   Determine if UNKNOWN missing data is allowed.

   The database user must know this to determine how to use the DBMS.

   (a) The default missing data representation for MySQL shall be nulls.

   (b) A database system variable shall indicate if UNKNOWN missing data is to be used in place of nulls.

   (c) This capability should be implemented as a Data Administration SHOW command.

2. **Database Administration**

   Determine if data values are missing.

   The database user must know this to determine how to use the DBMS.

   (a) If data is not missing, the database shall function as if all data values are exact and known.

   (b) If data values are missing, the database user shall consider missing data as a possibility.

   (c) The capability to determine if data is missing should be implemented using SQL.

3. **Database Administration**

   Identify MISSING data types permitted in a table column.

   The database user must know this to determine how to use the DBMS.

   (a) MISSING data types are stored as metadata in the database schema and a relvar MISSING table.

   (b) Metadata shall be queried using SQL.

   (c) This capability should be implemented using SQL.

B.1.4 Utility statements

1. **Utility statement**

   Help commands shall include information about KNOWN/UNKNOWN/ MISSING data tables.

   Corresponds to MySQL's provision of help for SQL statements.

   (a) SQL statements that are modified.

   (b) Tables added to the database schema.

B.2 Constraints

Constraints are non-functional requirements that may be derived from a user requirement, but often originate from a domain or system requirement.

1. **System Usability Constraint**

   Missing data should be easy to understand.

   Missing data is not well understood and a new representation must be explained. The existing implementation of missing data as nulls and 3VL in SQL creates confusing results.

   (a) An experienced MySQL user shall be able to search and update a database after a 30 minute tutorial.

   (b) The tutorial shall be available in a printable electronic format.

   (c) If feasible, the tutorial shall be supplemented by a help system.

2. **Database Administration Constraint**

   DBMS shall create a table that is missing data capable and allows missing data by default.

   The SQL Standard and MySQL represent missing data values using nulls. Nulls are allowed by default for non-key columns.

   (a) Database default is to support missing data using nulls.

   (b) Database shall allow KNOWN/UNKNOWN missing data representation to be enabled and once enable will be used in place of nulls.

   Existence of the _MISSING table indicates KNOWN/UNKNOWN enabled.

   (c) If a database is set to enable KNOWN/UNKNOWN missing data representation will support null only if “NULL” or “IS NULL” is explicitly used in a query.

3. **Table Definition Constraint**

   A relation key shall not include a missing data capable attribute..

   - Rationale: Table keys may not be null. A requirement that a key be missing data capable may be motivated by the belief that a key equates to an index.
   - Specification:
   - Source: RM/V1

4. **Index Definition Constraint**

   Missing data indexing shall use metadata.

   Columns that allows null and contain null can be indexed using MySQL. An implementation that is compatible with MySQL must allow missing data capable attributes to be indexed.

   (a) It is not possible to match a data value that is not present.

   (b) If a missing data capable column is indexed,rows and columns holding missing data values shall be found using metadata.

   (c) This method should be used to simulate an index for UNKNOWN table using the MISSING data table.

Appendix C **MyKU client component source code**

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "my_global.h"
#include "mysql.h"
#include "myku.CLI.h"
#include "myku.AST.h"
#include "myku.SEL.h"
#include "debug.h"

/* embedded server handle MYSQL and setup (ini and db location */
MYSQL *dbs;
static char *server_options[] =
{"mysql-MYKU", "--defaults-file=/myku-database/my.ini"};
int num_elements=sizeof(server_options) / sizeof(char *);
static char *server_groups[] =
{"server", "embedded", "client", NULL};

#define QUERYLENGTH 1024
char statement[QUERYLENGTH];
unsigned long query_timer;

/* parser interface handle */
YY_BUFFER_STATE yy_buffer_handle;

int empty_set_flag;

int main(void)
{

   /* Initialize the server and set server options */
   mysql_library_init(num_elements, server_options, server_groups);
   /*
    * The following call enables debugging programmatically.
    * mysql_debug("d:t:i:O,/mysql-myku/client.trace");
    * Debugging can be enabled in the [client] section of the my.ini file;
    * if not enabled, mysql_dbug_print() commands do not log.
    */
   /* Connect to embedded server. */
   dbs = srv_connect("case1");
   /* set input source print name in bison parser */
   filename = "user query";
   /* print myku info and enter query response loop */
   cli_splash(dbs);
   /* get and execute query statments until "exit" is entered */
   while (cli_get_query(statement))
   {
      maybe_flag = FALSE;
      anull_flag = FALSE;
      select_flag = FALSE;
      select_nodata_flag = FALSE;
      mysql_dbug_print(statement);
      yy_buffer_handle = yy_scan_string(statement);
      if(yyparse() == 0)
      {
#ifdef BISON_DEBUG
         printf("SQL: parsed\n");
#endif
      }
      else
      {
#ifdef BISON_DEBUG
         printf("SQL: parser failed\n");
#endif
      }
      yy_delete_buffer(yy_buffer_handle);
#ifdef QUERY_DEBUG
      if(maybe_flag) printf("SQL: MAYBE present\n");
      if(anull_flag) printf("SQL: NULL present\n");
      if(select_flag) printf("SQL: SELECT statement\n");
      if(select_nodata_flag) printf("SQL: SELECT NODATA\n");
#endif
      if((anull_flag) || (!select_flag) || (select_nodata_flag))
      {
      if(srv_get_result(dbs, statement, &query_timer))
         cli_put_result(dbs, statement, query_timer);
      }
      else
      {
         if(strlen(query_known) > 0)
            if(srv_get_result(dbs, query_known, &query_timer))
            {
               printf("KNOWN\n");
               cli_put_result(dbs, query_known, query_timer);
            }

         if(strlen(query_unknown) > 0)
            if(srv_get_result(dbs, query_unknown, &query_timer))
            {
               printf("UNKNOWN\n");
               cli_put_result(dbs, query_unknown, query_timer);
            }

         if(strlen(query_missing) > 0)
         {
            if (empty_set_flag == TRUE)
            {
               printf("MISSING\n");
```

```
                printf("Empty set (0.03 sec / 27 ms)\n");
                empty_set_flag = FALSE;

            }
            else
            {
                if(srv_get_result(dbs, query_missing, &query_timer))
                {
                    printf("MISSING\n");
                    cli_put_result(dbs, query_missing, query_timer);
                }
            }
        }
        else
        {
            printf("MISSING\n");
            printf("Empty set (0.01 sec / 19 ms)\n");
        }
    }

  }
  /* close the server connection and tell server to shutdown. */
  srv_disconnect(dbs);
  mysql_library_end();
  exit(EXIT_SUCCESS);
}
MYSQL *
srv_connect(const char *dbname)
{
  /*
   * allocate a mysql database server (dbs) handle;
   * use and return the pointer to this handle
   */
  MYSQL *dbs = mysql_init(NULL);
  if (!dbs)
     srv_terminate(dbs, "mysql_init failed: no memory");
  /*
   * The client and server use separate group names. This is critical.
   * The server does not accept the client's options, and vice versa.
   */
  mysql_options(dbs, MYSQL_READ_DEFAULT_GROUP, "embedded");
  mysql_options(dbs, MYSQL_OPT_USE_EMBEDDED_CONNECTION, NULL);
  /*
   * use the mysql database server handle to
   * connect to an instance of mysql running on a host;
   * the embedded server libmysqld.dll in this case
   */
  if(!mysql_real_connect(dbs, NULL, NULL, NULL, dbname, 0, NULL, 0))
     srv_terminate(dbs, "mysql_real_connect failed: %s", mysql_error(dbs));

  return dbs;
}
```

```
void
srv_disconnect(MYSQL *dbs)
{
   mysql_close(dbs);
}
static void
srv_terminate(MYSQL *dbs, char *fmt, ...)
{
   va_list ap;
   va_start(ap, fmt);
   vfprintf(stdout, fmt, ap);
   va_end(ap);
   (void)putc('\n', stdout);
   if (dbs)
      srv_disconnect(dbs);
   exit(EXIT_FAILURE);
}
void
cli_splash(MYSQL *dbs)
{
   printf("MyKU is a DBMS built on MySQL using ");
   printf("the KNOWN/UNKNOWN model of missing data.\n");
   printf("Server version: %s Source distribution.\n",
           mysql_get_server_info(dbs));
   printf("Type '\\c' to clear current input statement.\n");
   printf("Commands end with a semi-colon ';'\n\n");
}
int
cli_get_query(char* statement)
{
   char stmt[QUERYLENGTH] = {'\0'};
   char buf[QUERYLENGTH];
   char *ptr, *comment, *newline;

   yycolumn = 1;
   fputs("myku> ", stdout);
   while (fgets(buf, sizeof buf, stdin) != NULL)
   {
      for(ptr = buf; isspace(*ptr); ptr++);
      if(_strnicmp(ptr, "exit", 4) == 0 || _strnicmp(ptr, "quit", 4) == 0)
      {
         fputs("Bye\n\n", stdout);
         return FALSE;
      }
      /* isspace spc, tab, nl */
      if (strlen(ptr) == 0 && strlen(stmt) == 0)
      {
         fputs("myku> ", stdout);
         continue;
      }
      if (strstr(ptr, "\\c"))
      {
         *stmt = '\0';
         fputs("myku> ", stdout);
         continue;
      }
      comment = strchr(ptr, '#');
      if (comment != NULL)
         memset(comment, '\0', strlen(comment));

      newline = strchr(ptr, '\n');
      if (newline != NULL)
         *newline = ' ';

      strcat(stmt, ptr);
      if (strchr(ptr, ';') == NULL)
         fputs("   -> ", stdout);
      else
         break;
   }
   strcpy(statement, stmt);
   return TRUE;
}
void
cli_help(void)
{
}
int
srv_get_result(MYSQL *dbs, const char *stmt, unsigned long *timer)
{
   DWORD TimerMS = GetTickCount();
   if (mysql_query(dbs, stmt) == 0)
   {
      TimerMS = GetTickCount() - TimerMS;
      *timer = (unsigned long) TimerMS;
      return TRUE;
   }
   else
   {
      printf("ERROR %d (%s): %s\n", mysql_errno(dbs),
              mysql_sqlstate(dbs), mysql_error(dbs));
      printf("QUERY or Operation: %s\n", stmt);
      return FALSE;
   }
}
void
cli_put_result(MYSQL *dbs, const char *query, unsigned long timer)
{
   MYSQL_RES   *result;
   MYSQL_ROW    row;
   MYSQL_FIELD *field;
   unsigned long num_rows;
   int num_fields;
   unsigned int i, col_len;

   empty_set_flag = FALSE;
```

```
/* mysql_store_result() = 0 if malloc exceeds available memory and fails */
result = mysql_store_result(dbs);
if(result)
{
   num_rows = (unsigned long) mysql_num_rows(result);
   if(num_rows > 0)
   {
      /* determine column display widths */
      num_fields = mysql_num_fields(result);
      mysql_field_seek (result, 0);
      for(i = 0; i < mysql_num_fields (result); i++)
      {
         field = mysql_fetch_field (result);
         col_len = (unsigned int) strlen(field->name);
         if (col_len < field->max_length)
            col_len = field->max_length;
         if (col_len < 4 && !IS_NOT_NULL(field->flags))
            col_len = 4; /* 4 = length of the word NULL */
         field->max_length = col_len; /* reset column info */
      }
      cli_put_dash_line(result);
      fputc ('|', stdout);
      mysql_field_seek(result, 0);
      for(i = 0; i < mysql_num_fields(result); i++)
      {
         field = mysql_fetch_field(result);
         printf(" %-*s |", field->max_length, field->name);
      }
      fputc('\n', stdout);
      cli_put_dash_line (result);
      while((row = mysql_fetch_row(result)) != NULL)
      {
         mysql_field_seek (result, 0);
         fputc('|', stdout);
         for (i = 0; i < mysql_num_fields(result); i++)
         {
            field = mysql_fetch_field(result);
            if(row[i] == NULL)
               printf (" %-*s |", field->max_length, "NULL");
            else
            if(IS_NUM(field->type))
               printf(" %*s |", field->max_length, row[i]);
            else
               printf(" %-*s |", field->max_length, row[i]);
         }
         fputc('\n', stdout);
      }
      cli_put_dash_line(result);
        printf("%s in set (%.2f sec / %lu ms)\n",
                cli_get_row_str(num_rows),
                (float) timer / 1000, timer);
   }
   else
```

```
      {
         empty_set_flag = TRUE;
         printf("Empty set (%.2f sec / %lu ms)\n", (float) timer / 1000, timer);
      }
      fputc('\n', stdout);
      mysql_free_result(result);
   }
   else
   {
      if(mysql_field_count(dbs) == 0)
         if(_strnicmp(query, "use", 3) == 0)
         {
            const char *ptr;
            for(ptr = query+3; isspace(*ptr); ptr++);
            printf("Database changed\n");
         }
         else
         {
            num_rows = (unsigned long) mysql_affected_rows(dbs);
              printf("Query OK, %s affected (%.2f sec / %lu ms)\n",
                      cli_get_row_str(num_rows),
                      (float) timer / 1000, timer);
         }
      else
         srv_terminate(dbs, "mysql_store_result failed: %s [%s]",
                        mysql_error(dbs), query);
   }
   return;
}
void
cli_put_dash_line (MYSQL_RES *result)
{
   MYSQL_FIELD *field;
   unsigned int i, j;
   mysql_field_seek(result, 0);

   fputc('+', stdout);
   for (i = 0; i < mysql_num_fields(result); i++)
   {
      field = mysql_fetch_field(result);
      for (j = 0; j < field->max_length + 2; j++)
         fputc('-', stdout);
      fputc('+', stdout);
   }
   fputc('\n', stdout);
}
char*
cli_get_row_str(unsigned long num_rows)
{
   static char buffer[16];
   num_rows > 1 ? sprintf(buffer, "%lu rows", num_rows) : sprintf(buffer, "1 row");
   return buffer;
}
```

Appendix D **MyKU select query rewrite source code**

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include "myku.AST.h"
#include "myku.SEL.h"
#include "myku.SYS.h"
#include "debug.h"

char sel[256] = {'\0'};
char frm[256] = {'\0'};
char whr[256] = {'\0'};

unsigned int ftab1_known = FALSE;
unsigned int ftab1_unknown = FALSE;
unsigned int ftab1_missing = FALSE;
unsigned int join_flag = FALSE;
unsigned int ftab2_known = FALSE;
unsigned int ftab2_unknown = FALSE;
unsigned int ftab2_missing = FALSE;

char tab1_queried[64] = {'\0'};
char tab1_alias[64] = {'\0'};
char tab1_root[32] = "notab";
char tab1_key[16] = "nokey";
char tab1_known[64];
char tab1_unknown[64];
char tab1_missing[64];

char tab2_queried[64] = {'\0'};
char tab2_alias[64] = {'\0'};
char tab2_root[32] = "notab";
char tab2_key[16] = "nokey";
char tab2_known[64];
char tab2_unknown[64];
char tab2_missing[64];

char whr_attr[32] = "noatt";
char whr_expr[32] = {'\0'} ;
int maybe_done = FALSE;

char query[256] = {'\0'};
char query_known[256] = {'\0'};
char query_unknown[256] = {'\0'};
char query_missing[256] = {'\0'};
char union_known[256] = {'\0'};
char union_unknown[256] = {'\0'};
char union_missing[256] = {'\0'};

char miss_query[256];
char unk_tag[48] = "and tag in ('UNK', 'NYE', 'UND', 'INV', 'MIS')";
char not_tag[48] = "and tag in ('N/A', 'REM', 'NUL')";
char alt_missing[128];

/*
 * helper functions some of which are to get information
 * from MySQL information_schema tables in next release
 */
char *
primary_key(char *tbl)
{
   return tab1_key;
}
void
build_select_expression(struct node *n)
{
   int type;
   struct sel_expr *se;
   struct item *it;
    struct expr *ex;
   char s[32] = {'\0'};

   if (n == NULL) return;

   type = n->nodetype;
   switch(type) {
   case AST_SEL_EXPR:
      se = (struct sel_expr *)n;
      build_select_expression(se->expr);
      if (strlen(se-> alias) > 0)
      {
         strcat(sel, " as ");
         strcat(sel, se->alias);
      }
   break;
   case AST_ITEM:
      it = (struct item *)n;
      build_select_expression(it->content);
      if(it->next)
         strcat(sel, ", ");
      else
         strcat(sel, " ");
      build_select_expression(it->next);
   break;
   case AST_EXPRESSION:
      ex = (struct expr *)n;
      switch (ex->datatype) {
```

```
      case VAL_NUM:
         sprintf(s, "%d", ex->intval);
         strcat(sel, s);
      break;
      case VAL_FLOAT:
         sprintf(s, "%g", ex->floatval);
         strcat(sel, s);
      break;
      case VAL_STR:
         sprintf(s, "%s", ex->strval);
         strcat(sel, s);
      break;
      }
   break;
     default: printf("exception: build_select_expression node %d\n", type);
   }
}

void
build_from_references(struct node *n)
{
   int type;
   struct tab_ref *tr;
   struct item *it;

   if (n == NULL) return;

   type = n->nodetype;
   switch(type) {
   case AST_ITEM:
      it = (struct item *)n;
      build_from_references(it->content);
      if(it->next)
      /* join = true */
      {
         join_flag = TRUE;
      }
      build_from_references(it->next);
   break;

   case AST_TAB_REF:
      tr = (struct tab_ref *)n;
      if (join_flag)
         strcpy(tab2_queried, tr->name);
      else
         strcpy(tab1_queried, tr->name);
      if(strlen(tr->alias) > 0)
      {
         if (join_flag)
            strcpy(tab2_alias, tr->alias);
         else
            strcpy(tab1_alias, tr->alias);
         strcat(frm, tr->alias);
      }
      /* check to see if there is one or two tables (i.e. a JOIN */
      if (join_flag)
         tab2_table_names();
      else
         tab1_table_names();
   break;
     default: printf("exception: build_from_table_references node %d\n", type);
   }
}

void
build_where_expression(struct node *n)
{
   int type;
   struct oper *op;
   struct expr *ex;
   char wrk[128];

   if (n == NULL) return;

   type = n->nodetype;
   switch(type) {
   case AST_OPERATOR:
      op = (struct oper *)n;
      if(strcmp(op->op, "(") == 0)
      {
         /* expression of an operator tree node enclosed in parentheses */
         strcat(whr, "(");
         build_where_expression(op->left);
         build_where_expression(op->right);
         strcat(whr, ")");
      }
      else
      {
         if(op->noway)
            strcat(whr, " NOT ");
         if(op->maybe)
         {
            /* MAYBE operator that must use the left expression attribute
               to find missing data in _MISSING */
              build_where_expression(op->left);
            strcpy(whr_attr, whr_expr);
            sprintf(wrk, "%s in (select %s from %s where attr = '%s' %s)",
                    tab1_key, tab1_key, tab1_missing, whr_attr, unk_tag);
            strcat(whr, wrk);
            sprintf(miss_query, "select * from %s where attr = '%s' %s",
                    tab1_missing, whr_attr, unk_tag);
            maybe_done = TRUE;
         }
         else
         {
            /* standard operator must be converted
```

```
                  from prefix to infix notation */
               build_where_expression(op->left);
               strcat(whr, whr_expr);
               strcpy(whr_attr, whr_expr);
               whr_expr[0] = '\0';
               strcat(whr, op->op);
               if(op->paren)
                  strcat(whr, "(");
               build_where_expression(op->right);
               if (!maybe_done)
                  strcat(whr, whr_expr);
               sprintf(alt_missing,
                  "select * from %s where %s in (select %s from %s where %s = %s);",
                  tab1_missing, primary_key(tab1_missing),
                  primary_key(tab1_unknown),
                  tab1_unknown, whr_attr, whr_expr);
               whr_expr[0] = '\0';
               if(op->paren)
                  strcat(whr, ")");
            }
         }
      break;
      case AST_EXPRESSION:
         ex = (struct expr *)n;
         switch (ex->datatype) {
         case VAL_NUM:
            sprintf(whr_expr, "%d", ex->intval);
         break;
         case VAL_FLOAT:
            sprintf(whr_expr, "%g", ex->floatval);
         break;
         case VAL_STR:
            sprintf(whr_expr, "%s", ex->strval);
         break;
         }
      break;
        default: printf("exception: build_where_expression node %d\n", type);
      }
   }

   void
   tab1_table_names()
   {
      char *first;
      char *last;

      ftab1_known = FALSE;
      ftab1_unknown = FALSE;
      ftab1_missing = FALSE;

      first = strchr(tab1_queried, '_');
      last = strrchr(tab1_queried, '_');
      if (last)
```

```
{
   if (stricmp(last, "_known") == 0)
   {
      ftab1_known = TRUE;
      strncopy(tab1_root, tab1_queried, last - tab1_queried);
   }
   else
   if (stricmp(last, "_unknown") == 0)
   {
      ftab1_unknown = TRUE;
      ftab1_missing = TRUE;
      strncopy(tab1_root, tab1_queried, last - tab1_queried);
   }
   else
   if (stricmp(last, "_missing") == 0)
   {
      ftab1_missing = TRUE;
      strncopy(tab1_root, tab1_queried, last - tab1_queried);
   }
   else
   {
      if (first == last) /* only one underscore in table name */
      {
         ftab1_known = TRUE;
         ftab1_unknown = TRUE;
         ftab1_missing = TRUE;
         strcpy(tab1_root, tab1_queried);
      }
      else /* error underscore in table name but not myku */
      {
         printf("More than one underscore in name, ");
         printf("but not a MyKU table type\n");
      }
   }
}
else /* no underscores in table name; must be MyKU root */
{
   ftab1_known = TRUE;
   ftab1_unknown = TRUE;
   ftab1_missing = TRUE;
   strcpy(tab1_root, tab1_queried);
}

if((stricmp(tab1_queried, "emp") == 0) ||
   (stricmp(tab1_queried, "assign") == 0))
    anull_flag = TRUE;

strcpy(tab1_known, strundstr(tab1_root, "KNOWN"));
strcpy(tab1_unknown, strundstr(tab1_root, "UNKNOWN"));
strcpy(tab1_missing, strundstr(tab1_root, "MISSING"));

if((stricmp(tab1_root, "T") == 0) ||
   (stricmp(tab1_root, "U") == 0))
```

```
        strcpy(tab1_key, "PK");
    if((stricmp(tab1_root, "my_names") == 0) ||
       (stricmp(tab1_root, "your_names") == 0))
        strcpy(tab1_key, "PK");
    if((stricmp(tab1_root, "person") == 0)  ||
       (stricmp(tab1_root, "emp") == 0))
        strcpy(tab1_key, "ekey");
}

void
tab2_table_names()
{
   char *first;
   char *last;

   ftab2_known = FALSE;
   ftab2_unknown = FALSE;
   ftab2_missing = FALSE;

   first = strchr(tab2_queried, '_');
   last = strrchr(tab2_queried, '_');
   if (last)
   {
      if (stricmp(last, "_known") == 0)
      {
         ftab2_known = TRUE;
         strncopy(tab2_root, tab2_queried, last - tab2_queried);
      }
      else
      if (stricmp(last, "_unknown") == 0)
      {
         ftab2_unknown = TRUE;
         ftab2_missing = TRUE;
         strncopy(tab2_root, tab2_queried, last - tab2_queried);
      }
      else
      if (stricmp(last, "_missing") == 0)
      {
         ftab2_missing = TRUE;
         strncopy(tab2_root, tab2_queried, last - tab2_queried);
      }
      else
      {
         if (first == last) /* only one underscore in table name */
         {
            ftab2_known = TRUE;
            ftab2_unknown = TRUE;
            ftab2_missing = TRUE;
            strcpy(tab2_root, tab2_queried);
         }
         else /* error underscore in table name but not myku */
         /* error ? under line in table name but not myku */
         {
            printf("More than one underscore in name, ");
            printf("but not a MyKU table type\n");
         }
      }
   }
   else /* no underscores in table name; must be a MyKU root */
   {
      ftab2_known = TRUE;
      ftab2_unknown = TRUE;
      ftab2_missing = TRUE;
      strcpy(tab2_root, tab2_queried);
   }

   strcpy(tab2_known, strundstr(tab2_root, "KNOWN"));
   strcpy(tab2_unknown, strundstr(tab2_root, "UNKNOWN"));
   strcpy(tab2_missing, strundstr(tab2_root, "MISSING"));

   if((stricmp(tab2_root, "T") == 0) ||
      (stricmp(tab2_root, "U") == 0))
       strcpy(tab1_key, "PK");
   if((stricmp(tab2_root, "my_names") == 0) ||
      (stricmp(tab2_root, "your_names") == 0))
       strcpy(tab2_key, "PK");
   if((stricmp(tab2_root, "person") == 0)  ||
      (stricmp(tab2_root, "emp") == 0))
       strcpy(tab2_key, "ekey");
}
void
build_select_queries()
{
   /*
    * rewrite extended SQL query into 3 standard SQL queries
    * for known, unknown, and missing tables
    */
   struct node *n;
   struct select *s;
   struct sel_expr *se;
   struct tab_ref *tr;
   struct oper *op;

   /* values for creating known, unknown and missing select statements in AST */
   n = root;
   s = (struct select *) n;
   se = (struct sel_expr *) s->select_expr_list;
   tr = (struct tab_ref *) s->from_tab_ref_list;
   op = (struct oper *) s->where_expr_tree;

   if (union_flag)
   {
      if (strlen(query_known) > 0)
         strcpy(union_known, query_known);
      if (strlen(query_unknown) > 0)
         strcpy(union_unknown, query_unknown);
      if (strlen(query_missing) > 0)
         strcpy(union_missing, query_missing);
   }

   /* clear known/unknown query strings before building new ones */
   strclear(query_known);
   strclear(query_unknown);
   strclear(query_missing);
   strclear(miss_query);
   strclear(alt_missing);

   strclear(sel);
   build_select_expression((struct node *) se);

   join_flag = FALSE;
   strclear(tab1_queried);
   strclear(tab1_alias);
   strclear(tab2_queried);
   strclear(tab2_alias);
   build_from_references((struct node *) tr);
#ifdef QUERY_DEBUG
   printf("tab1 (queried) is %s \n", tab1_queried);
   printf("tab1 (aliased) is %s \n", tab1_alias);
   printf("tab1 (root)    is %s \n", tab1_root);
   printf("tab1 (key)     is %s \n", tab1_key);
   printf("tab1 (known)   is %s \n", tab1_known);
   printf("tab1 (unknown) is %s \n", tab1_unknown);
   printf("tab1 (missing) is %s \n", tab1_missing);

   printf("%s join tab2 with tab1 \n", join_flag ? "Do" : "Do NOT");
   if (join_flag) {
   printf("tab2 (joined)  is %s \n", tab2_queried);
   printf("tab2 (aliased) is %s \n", tab2_alias);
   printf("tab2 (root)    is %s \n", tab2_root);
   printf("tab2 (key)     is %s \n", tab2_key);
   printf("tab2 (known)   is %s \n", tab2_known);
   printf("tab2 (unknown) is %s \n", tab2_unknown);
   printf("tab2 (missing) is %s \n", tab2_missing);
   }
#endif

   /* op is a pointer to AST list of optional where expressions */
   strclear(whr);
   maybe_done = FALSE;
   if(op) strcpy(whr, " where ");
   build_where_expression((struct node *) op);

   if (union_flag)
      select_union();
   else
   if (join_flag)
      select_product();
   else
```

```
      select_restrict();

#ifdef QUERY_DEBUG
   printf("\nMyKU user statement: %s\n\n", statement);
   printf("  Query KNOWN (trn): \n%s\n\n", query_known);
   printf("Query UNKNOWN (trn): \n%s\n\n", query_unknown);
   printf("Query MISSING (trn): \n%s\n\n", query_missing);
   printf("     SubQuery: %s\n", miss_query);
   printf(" Alt SubQuery: %s\n", alt_missing);
#endif
}

void
select_product()
{

   strcpy(query_known,   "select * from product_known;");
   strcpy(query_unknown, "select * from product_unknown;");
   strcpy(query_missing, "select * from product_missing;");

}
void
select_restrict()
{
   if (ftab1_known)
   {
      strcpy(query_known, "select ");
      strcat(query_known, sel);
      sprintf(frm, "from %s", tab1_known);
      strcat(query_known, frm);
      strcat(query_known, whr);
      strcat(query_known, ";");
   }
   if (ftab1_unknown)
   {
      /* _unknown requires a key in select to connected with _missing;
         check and add if needed */
      if (!(stristr(sel, tab1_key) || stristr(sel, "*")))
         sprintf(query_unknown, "select %s, ", tab1_key);
      else
         strcpy(query_unknown, "select ");
      strcat(query_unknown, sel);
      sprintf(frm, "from %s", tab1_unknown);
      strcat(query_unknown, frm);
      strcat(query_unknown, whr);
      strcat(query_unknown, ";");
      if (strlen(miss_query) > 0)
      {
         strcpy(query_missing, miss_query);
         strcat(query_missing, ";");
      }
      else
      if (strlen(whr) == 0)
      {
         sprintf(query_missing, "select * from %s;", tab1_missing);
      }
      else
      if (strlen(alt_missing) > 0)
      {
         sprintf(query_missing, alt_missing);
      }
   }
   else
   if (ftab1_missing)
   {
      strcpy(query_missing, statement);
   }
}
void
select_union()
{
   char *p;

   select_restrict();

   if(strlen(query_known) > 0)
      if(strlen(union_known) > 0)
      {
         p = strchr(union_known, ';');
         if (p) *p = '\0';
         strcat(union_known, " union ");
         strcat(union_known, query_known);
         strcpy(query_known, union_known);
      }
      else
      {   /* query_known is the union */ }
   else
      /* union_known is the union */
      if(strlen(union_known) > 0)
         strcpy(query_known, union_known);

   if (strlen(query_unknown) > 0)
      if(strlen(union_unknown) > 0)
      {
         p = strchr(union_unknown, ';');
         if (p) *p = '\0';
         strcat(union_unknown, " union ");
         strcat(union_unknown, query_unknown);
         strcpy(query_unknown, union_unknown);
      }
      else
      {   /* query_known is the union */ }
   else
      /* union_known is the union */
      if(strlen(union_unknown) > 0)
         strcpy(query_unknown, union_unknown);
   if (stristr(union_missing, "union"))
      strcpy(query_missing, union_missing);
   else
   if (strlen(query_missing) > 0)
      if(strlen(union_missing) > 0)
      {
         p = strchr(union_missing, ';');
         if (p) *p = '\0';
         strcat(union_missing, " union ");
         strcat(union_missing, query_missing);
         strcpy(query_missing, union_missing);
      }
      else
      {   /* query_known is the union */ }
   else
      /* union_known is the union */
      if(strlen(union_missing) > 0)
         strcpy(query_missing, union_missing);

   strclear(union_known);
   strclear(union_unknown);
   strclear(union_missing);
}
```

Appendix E **Standard SQL for derived my_names relvar**

```
use case1;

create table my_names (PK smallint unsigned not null auto_increment,
                       first char(12) not null,
                       mi  char(1) not null,
                       last char(12) not null,
                       primary key (PK));

insert into my_names values (1, 'Edgar',  'F', 'Codd');
insert into my_names values (2, 'Chris',  'J', 'Date');
insert into my_names values (3, 'Hugh',   ' ', 'Darwen');
insert into my_names values (4, 'Andrew', ' ', 'Warden');


create table my_names_MISSING (PK smallint unsigned not null,
                               attr char(5) not null,
                               tag  char(3) not null,
                               primary key (PK));

insert into my_names_MISSING values(3, 'mi', 'UNK');
insert into my_names_MISSING values(4, 'mi', 'N/A');


create table my_names_KNOWN like my_names;
insert into my_names_KNOWN
      (select * from my_names
                where pk not in (select pk from my_names_missing));


create table my_names_UNKNOWN like my_names;
insert into my_names_UNKNOWN
      (select * from my_names
                where pk in (select pk from my_names_missing));
```

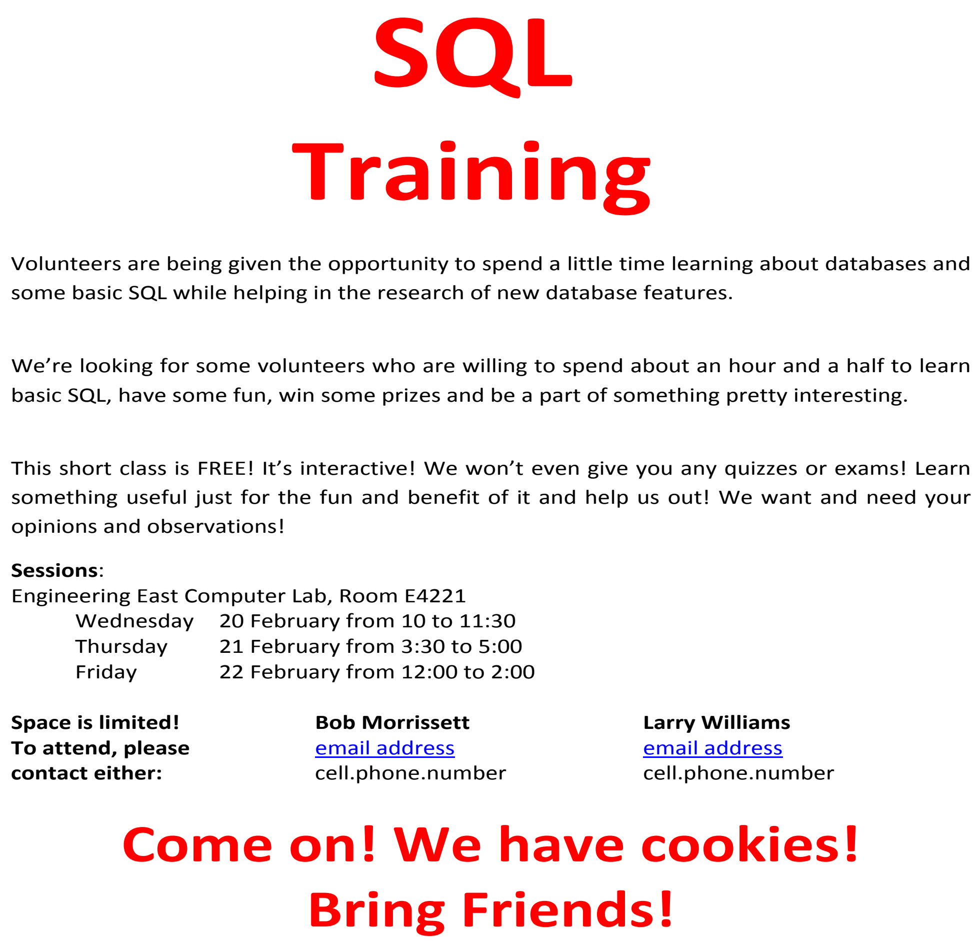
Appendix F **Standard SQL to define a product view**

```
use case1;

drop view derived_known;
# Cartesian product view for known
create view product_known as
select m.PK as m_PK, m.first as m_first, m.mi as m_mi, m.last as m_last,
       y.PK as y_PK, y.first as y_first, y.mi as y_mi, y.last as y_last
from my_names as m, your_names as y
where m.PK not in (select PK from my_names_missing)
  and y.PK not in (select PK from your_names_missing)
order by m_PK, y_PK;


drop view derived_unknown;
# Cartesian product view for unknown
create view product_unknown as
select m.PK as m_PK, m.first as m_first, m.mi as m_mi, m.last as m_last,
       y.PK as y_PK, y.first as y_first, y.mi as y_mi, y.last as y_last
from my_names as m, your_names as y
where m.PK in (select PK from my_names_missing)
   or y.PK in (select PK from your_names_missing)
order by m_PK, y_PK;


drop view missing_keys;
# view creates keys for product_missing of Cartesian product
create view product_missing_keys as
select m_PK, y_PK from product_unknown
 where m_PK in (select PK from my_names_missing)
    or y_PK in (select PK from your_names_missing);


drop view derived_missing;
# Cartesian product missing uses product_missing_keys in view of product_missing
create view product_missing as
select m.PK as m_PK, y_PK, concat('m_',attr) as attr, tag
  from my_names_missing as m, product_missing_keys
 where m.PK = m_PK
union
select m_PK, y.PK as y_PK, concat('y_',attr) as attr, tag
  from your_names_missing as y, product_missing_keys
 where y.PK = y_PK
order by m_PK, y_PK, attr;
```

Appendix G **Standard SQL to define a union view**

```
use case1;

drop view T_known;
# create view of my_names_known and your_names_known as T_known
create view T_known as
select * from my_names_known union select * from your_names_known;

drop view T_unknown;
# create view of my_names_unknown and your_names_unknown as T_unknown
create view T_unknown as
select * from my_names_unknown union select * from your_names_unknown;

drop view T_missing;
# create view of my_names_missing and your_names_unknown as T_missing
create view T_missing as
select * from my_names_missing where pk in (select pk from T_unknown)
union
select * from your_names_missing where pk in (select pk from T_unknown);
```

Appendix H **Study Recruitment Flyer**



SQL

Training

Volunteers are being given the opportunity to spend a little time learning about databases and some basic SQL while helping in the research of new database features.

We're looking for some volunteers who are willing to spend about an hour and a half to learn basic SQL, have some fun, win some prizes and be a part of something pretty interesting.

This short class is FREE! It's interactive! We won't even give you any quizzes or exams! Learn something useful just for the fun and benefit of it and help us out! We want and need your opinions and observations!

Sessions:
Engineering East Computer Lab, Room E4221

Wednesday	20 February from 10 to 11:30
Thursday	21 February from 3:30 to 5:00
Friday	22 February from 12:00 to 2:00

Space is limited!	**Bob Morrissett**	**Larry Williams**
To attend, please	email address	email address
contact either:	cell.phone.number	cell.phone.number

Come on! We have cookies!
Bring Friends!

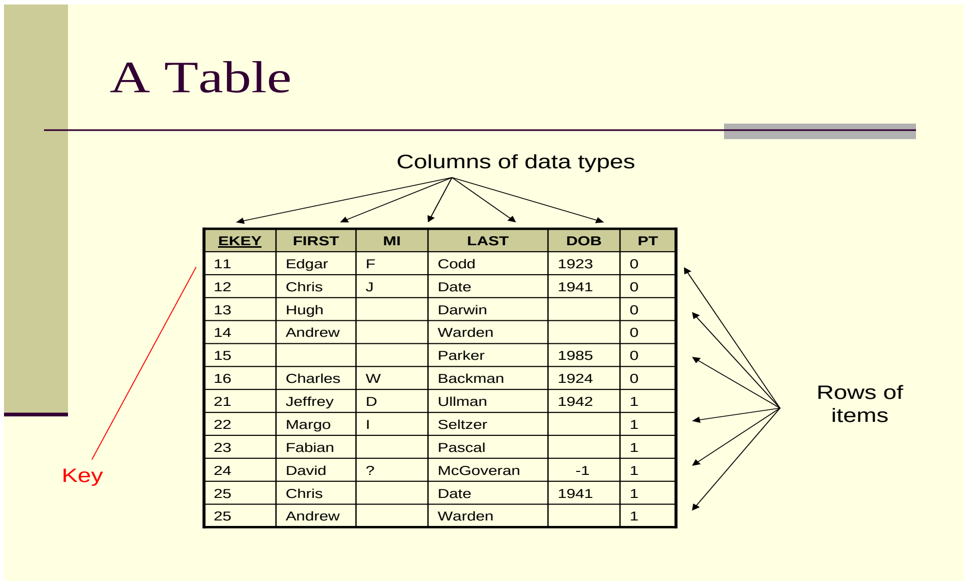
Figure 35: Feasibility Study Recruitment Flyer

Appendix I **SQL Tutorial**

An SQL tutorial introduced basic database query to undergraduate volunteers who were interested in SQL and willing to participate in a feasibility study for two research features added to MySQL. Each of these features were described after the tutorial with its hands-on SQL query experience. The tutorial focus was on SQL rather than the relational model and referred to relations as “tables,” domains as “columns,” and tuples as “rows,” with data values contained in “attributes.”

I.1 Database concepts

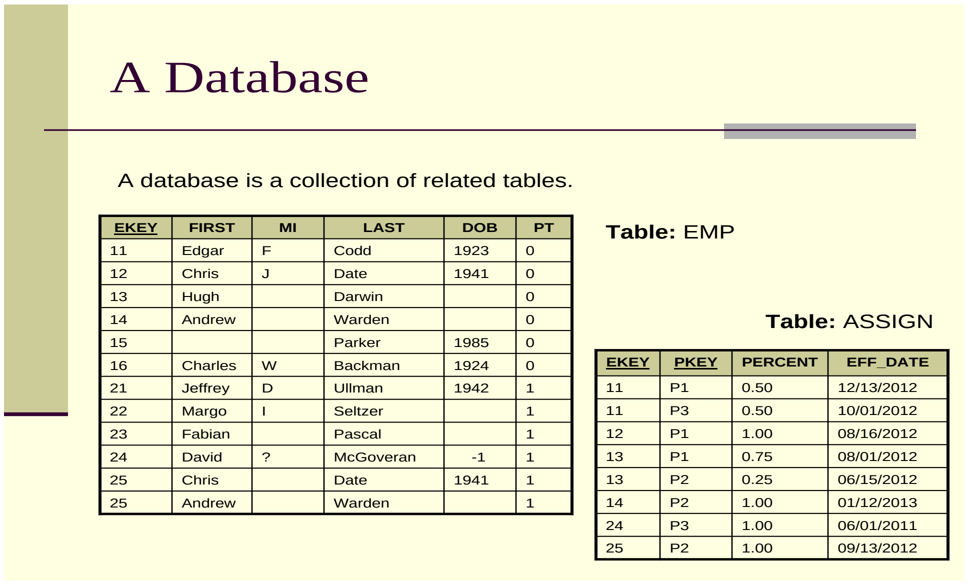
The tutorial described relational databases as collections of related tables. A table is a set of rows composed of named columns that contain data values in variables (attributes) at the intersection of each column and row. While all columns of the same data type can be used to define a relationship between tables, columns used as table keys with data values unique for a row are often used in this role.



A Table

<u>EKEY</u>	FIRST	MI	LAST	DOB	PT
11	Edgar	F	Codd	1923	0
12	Chris	J	Date	1941	0
13	Hugh		Darwin		0
14	Andrew		Warden		0
15			Parker	1985	0
16	Charles	W	Backman	1924	0
21	Jeffrey	D	Ullman	1942	1
22	Margo	I	Seltzer		1
23	Fabian		Pascal		1
24	David	?	McGoveran	-1	1
25	Chris		Date	1941	1
25	Andrew		Warden		1

Figure 36: SQL Tutorial table definition



A Database

A database is a collection of related tables.

Table: EMP

<u>EKEY</u>	FIRST	MI	LAST	DOB	PT
11	Edgar	F	Codd	1923	0
12	Chris	J	Date	1941	0
13	Hugh		Darwin		0
14	Andrew		Warden		0
15			Parker	1985	0
16	Charles	W	Backman	1924	0
21	Jeffrey	D	Ullman	1942	1
22	Margo	I	Seltzer		1
23	Fabian		Pascal		1
24	David	?	McGoveran	-1	1
25	Chris		Date	1941	1
25	Andrew		Warden		1

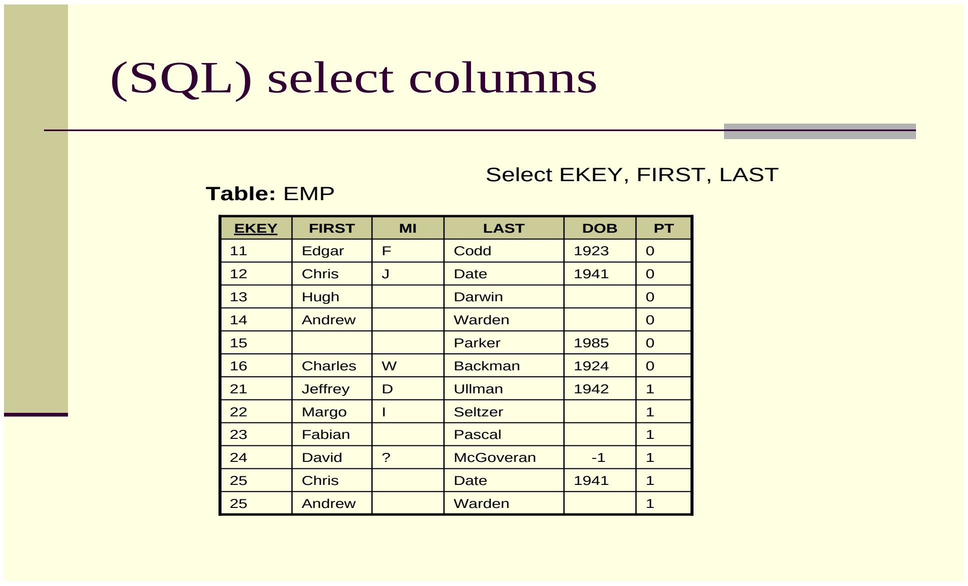
Table: ASSIGN

<u>EKEY</u>	<u>PKEY</u>	PERCENT	EFF_DATE
11	P1	0.50	12/13/2012
11	P3	0.50	10/01/2012
12	P1	1.00	08/16/2012
13	P1	0.75	08/01/2012
13	P2	0.25	06/15/2012
14	P2	1.00	01/12/2013
24	P3	1.00	06/01/2011
25	P2	1.00	09/13/2012

Figure 37: SQL Tutorial database definition

I.2 Select data from a table

A basic SQL query was developed incrementally to select (project) columns from a table (restricted) where an attribute in zero or more rows matches a value. Participants were shown how to start a MySQL server and client and encouraged to enter the SQL as it was developed.



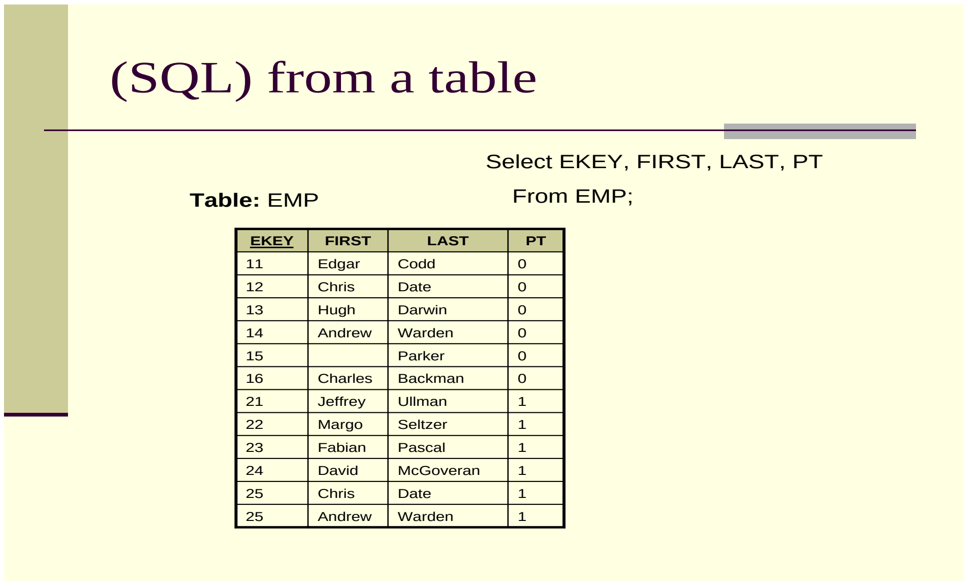
(SQL) select columns

Select EKEY, FIRST, LAST

Table: EMP

EKEY	FIRST	MI	LAST	DOB	PT
11	Edgar	F	Codd	1923	0
12	Chris	J	Date	1941	0
13	Hugh		Darwin		0
14	Andrew		Warden		0
15			Parker	1985	0
16	Charles	W	Backman	1924	0
21	Jeffrey	D	Ullman	1942	1
22	Margo	I	Seltzer		1
23	Fabian		Pascal		1
24	David	?	McGoveran	-1	1
25	Chris		Date	1941	1
25	Andrew		Warden		1

Figure 38: SQL Tutorial column specification



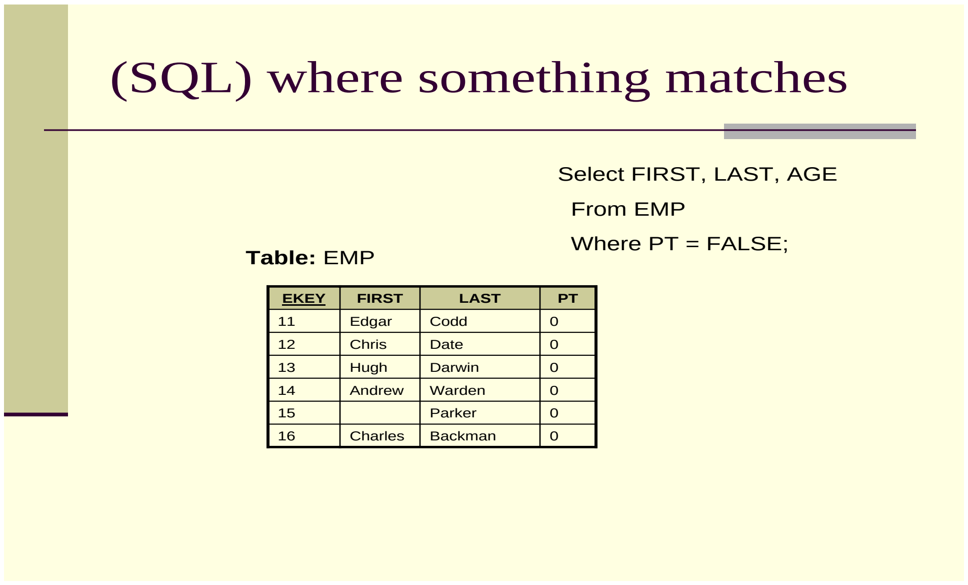
(SQL) from a table

```
Select EKEY, FIRST, LAST, PT
  From EMP;
```

Table: EMP

EKEY	FIRST	LAST	PT
11	Edgar	Codd	0
12	Chris	Date	0
13	Hugh	Darwin	0
14	Andrew	Warden	0
15		Parker	0
16	Charles	Backman	0
21	Jeffrey	Ullman	1
22	Margo	Seltzer	1
23	Fabian	Pascal	1
24	David	McGoveran	1
25	Chris	Date	1
25	Andrew	Warden	1

Figure 39: SQL Tutorial table specification



(SQL) where something matches

```
Select FIRST, LAST, AGE
  From EMP
  Where PT = FALSE;
```

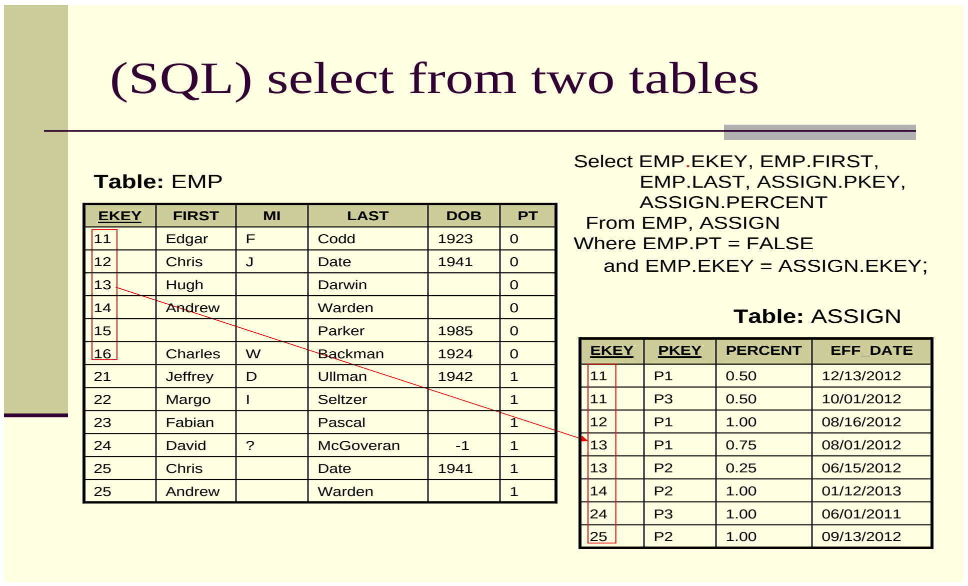
Table: EMP

EKEY	FIRST	LAST	PT
11	Edgar	Codd	0
12	Chris	Date	0
13	Hugh	Darwin	0
14	Andrew	Warden	0
15		Parker	0
16	Charles	Backman	0

Figure 40: SQL Tutorial row match criteria

I.3 Join connects two tables

A more complex query was developed to create a new table that used the connection (join) between two tables where table keys are equal.



(SQL) select from two tables

Table: EMP

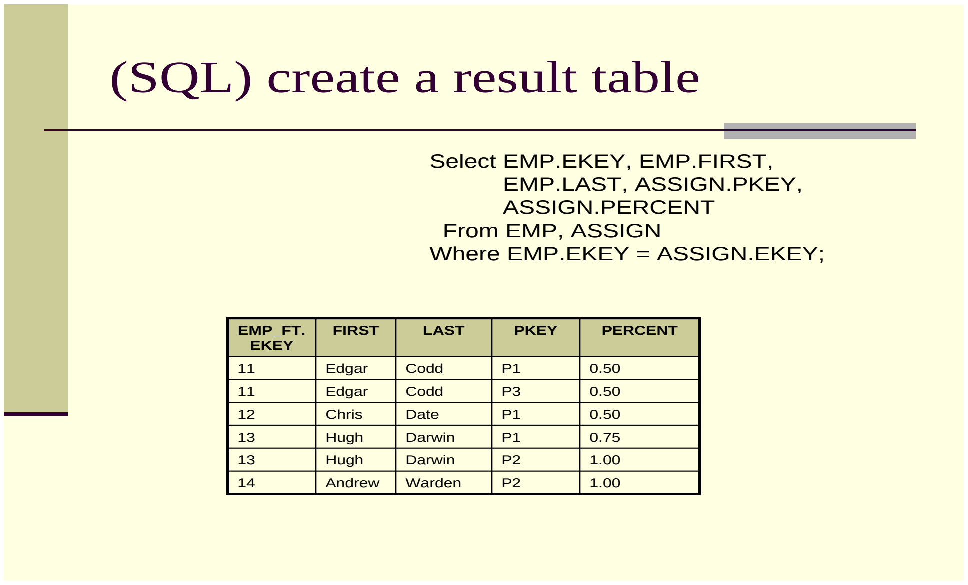
EKEY	FIRST	MI	LAST	DOB	PT
11	Edgar	F	Codd	1923	0
12	Chris	J	Date	1941	0
13	Hugh		Darwin		0
14	Andrew		Warden		0
15			Parker	1985	0
16	Charles	W	Backman	1924	0
21	Jeffrey	D	Ullman	1942	1
22	Margo	I	Seltzer		1
23	Fabian		Pascal		1
24	David	?	McGoveran	-1	1
25	Chris		Date	1941	1
25	Andrew		Warden		1

```
Select EMP.EKEY, EMP.FIRST,
       EMP.LAST, ASSIGN.PKEY,
       ASSIGN.PERCENT
 From EMP, ASSIGN
Where EMP.PT = FALSE
   and EMP.EKEY = ASSIGN.EKEY;
```

Table: ASSIGN

EKEY	PKEY	PERCENT	EFF_DATE
11	P1	0.50	12/13/2012
11	P3	0.50	10/01/2012
12	P1	1.00	08/16/2012
13	P1	0.75	08/01/2012
13	P2	0.25	06/15/2012
14	P2	1.00	01/12/2013
24	P3	1.00	06/01/2011
25	P2	1.00	09/13/2012

Figure 41: SQL Tutorial joining two tables



(SQL) create a result table

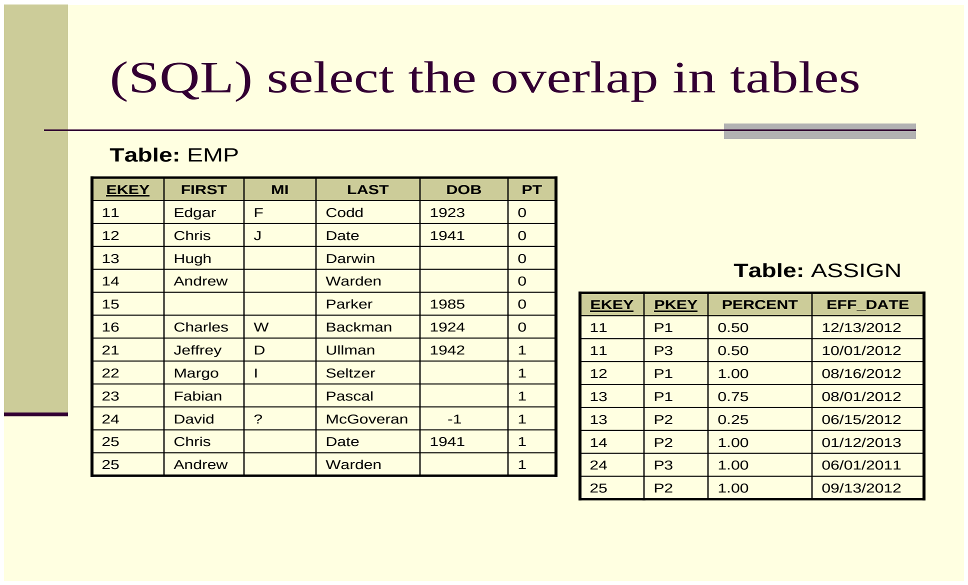
```
Select EMP.EKEY, EMP.FIRST,
           EMP.LAST, ASSIGN.PKEY,
           ASSIGN.PERCENT
  From EMP, ASSIGN
Where EMP.EKEY = ASSIGN.EKEY;
```

EMP_FT. EKEY	FIRST	LAST	PKEY	PERCENT
11	Edgar	Codd	P1	0.50
11	Edgar	Codd	P3	0.50
12	Chris	Date	P1	0.50
13	Hugh	Darwin	P1	0.75
13	Hugh	Darwin	P2	1.00
14	Andrew	Warden	P2	1.00

Figure 42: SQL Tutorial query results

I.4 Subquery intersects two tables

The concept of an inner query nested within an outer query as match criteria was presented.



(SQL) select the overlap in tables

Table: EMP

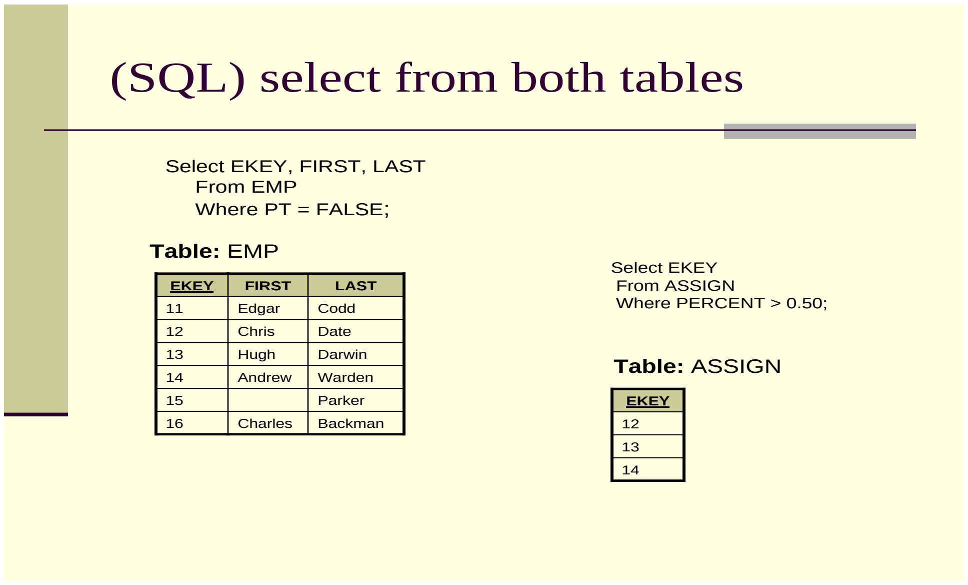
EKEY	FIRST	MI	LAST	DOB	PT
11	Edgar	F	Codd	1923	0
12	Chris	J	Date	1941	0
13	Hugh		Darwin		0
14	Andrew		Warden		0
15			Parker	1985	0
16	Charles	W	Backman	1924	0
21	Jeffrey	D	Ullman	1942	1
22	Margo	I	Seltzer		1
23	Fabian		Pascal		1
24	David	?	McGoveran	-1	1
25	Chris		Date	1941	1
25	Andrew		Warden		1

Table: ASSIGN

EKEY	PKEY	PERCENT	EFF_DATE
11	P1	0.50	12/13/2012
11	P3	0.50	10/01/2012
12	P1	1.00	08/16/2012
13	P1	0.75	08/01/2012
13	P2	0.25	06/15/2012
14	P2	1.00	01/12/2013
24	P3	1.00	06/01/2011
25	P2	1.00	09/13/2012

Figure 43: SQL Tutorial subquery definition

Using two different tables with keys of the same data type (domain) an SQL query that selects a table subset from the first table was shown with another SQL query that creates a table of key values selected from the second table. The second query is then nested in the first which matches table keys to those in the results of the nested query.



(SQL) select from both tables

```
Select EKEY, FIRST, LAST
    From EMP
    Where PT = FALSE;
```

Table: EMP

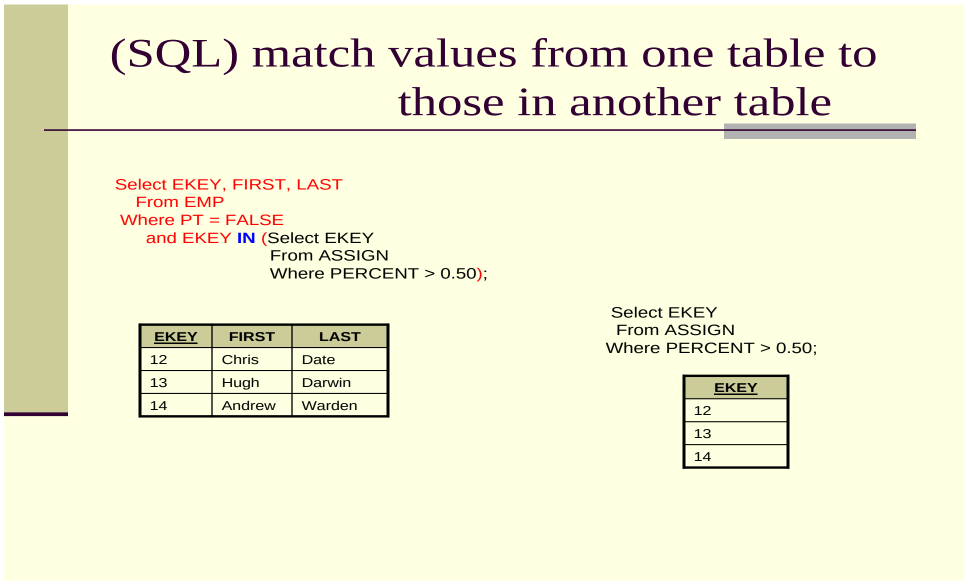
EKEY	FIRST	LAST
11	Edgar	Codd
12	Chris	Date
13	Hugh	Darwin
14	Andrew	Warden
15		Parker
16	Charles	Backman

```
Select EKEY
 From ASSIGN
 Where PERCENT > 0.50;
```

Table: ASSIGN

EKEY
12
13
14

Figure 44: SQL Tutorial query and subquery



(SQL) match values from one table to those in another table

```
Select EKEY, FIRST, LAST
   From EMP
 Where PT = FALSE
    and EKEY IN (Select EKEY
                    From ASSIGN
                    Where PERCENT > 0.50);
```

EKEY	FIRST	LAST
12	Chris	Date
13	Hugh	Darwin
14	Andrew	Warden

```
Select EKEY
 From ASSIGN
Where PERCENT > 0.50;
```

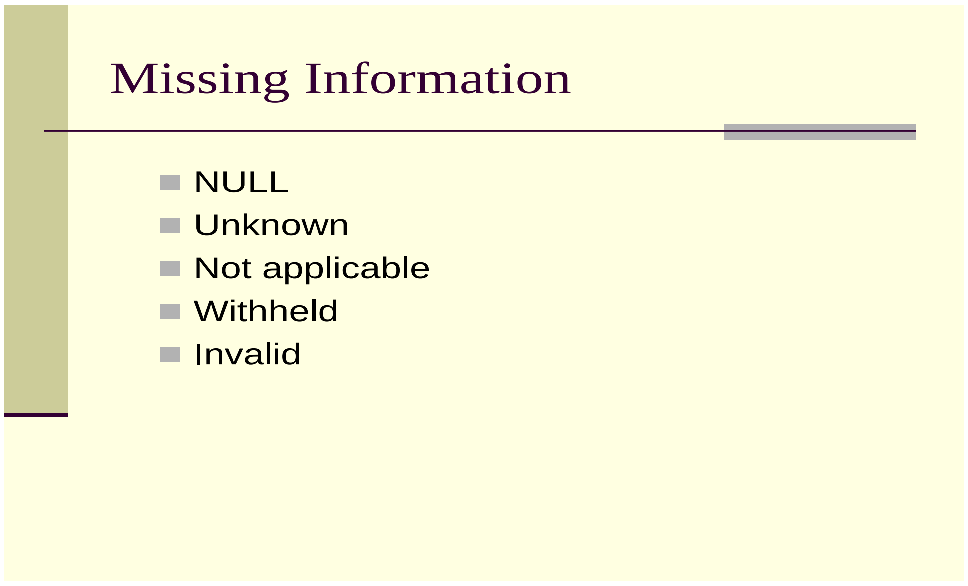
EKEY
12
13
14

Figure 45: SQL Tutorial nesting the subquery

Appendix J **MyKU Tutorial**

The feasibility study for the MySQL embedded server client program MyKU that implements the KNOWN/UNKNOWN model required an introduction to representations for missing data values. The presentation described issues related to adding data to relational databases that use SQL.

J.1 Missing data values



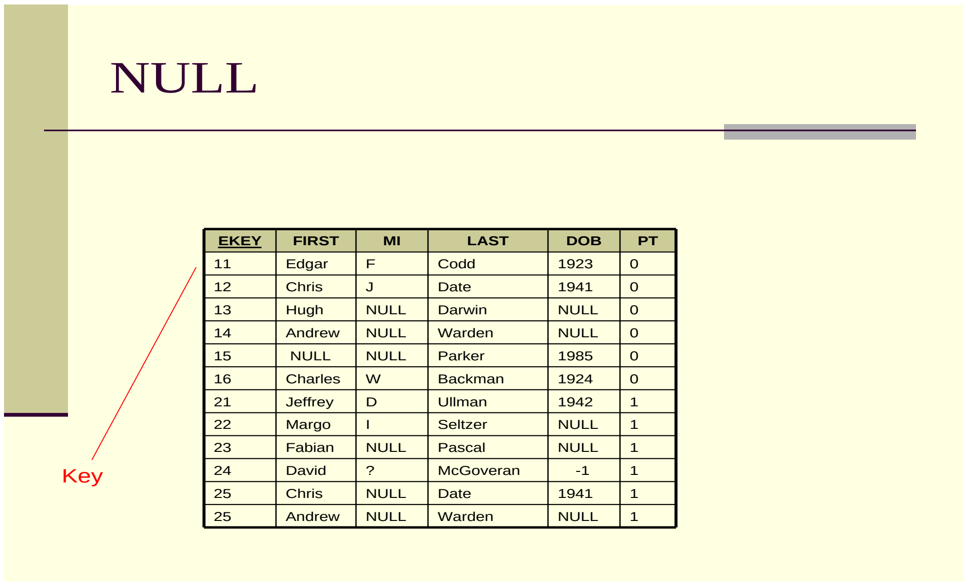
Missing Information

- NULL
- Unknown
- Not applicable
- Withheld
- Invalid

Figure 46: MyKU Tutorial Missing Data

A presentation about data that may be missing, why it can be missing, and how its interpretation depends why it is missing followed the SQL tutorial.

J.2 Null



NULL

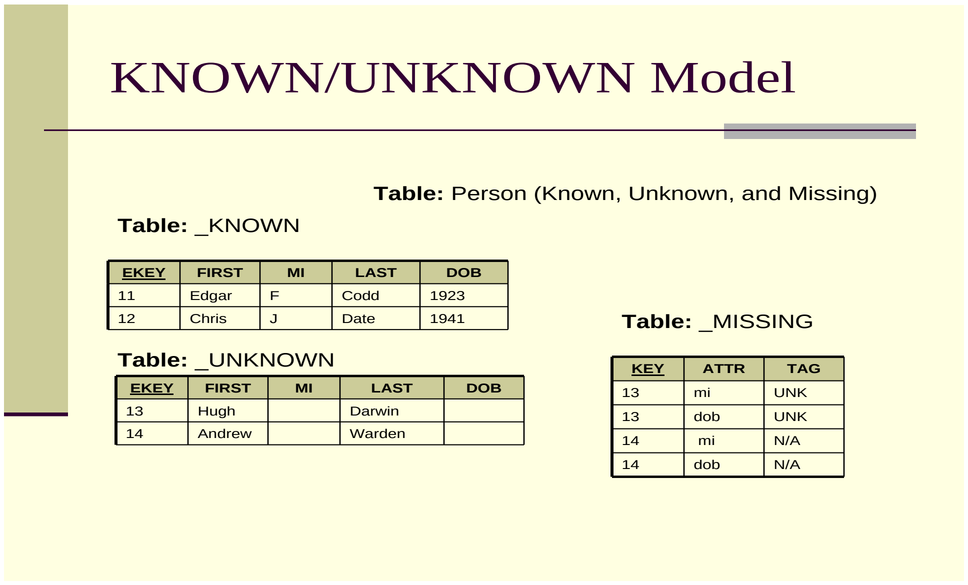
EKEY	FIRST	MI	LAST	DOB	PT
11	Edgar	F	Codd	1923	0
12	Chris	J	Date	1941	0
13	Hugh	NULL	Darwin	NULL	0
14	Andrew	NULL	Warden	NULL	0
15	NULL	NULL	Parker	1985	0
16	Charles	W	Backman	1924	0
21	Jeffrey	D	Ullman	1942	1
22	Margo	I	Seltzer	NULL	1
23	Fabian	NULL	Pascal	NULL	1
24	David	?	McGoveran	-1	1
25	Chris	NULL	Date	1941	1
25	Andrew	NULL	Warden	NULL	1

Figure 47: MyKU Tutorial NULL

The use of NULL to represent missing data as a missing value placeholder and not a value was explained. It was made clear that because null is not a value, one null is not equal to another. The standard SQL IS NULL match criteria was described.

J.3 KNOWN/UNKNOWN

This section of the presentation introduced the idea of a relation variable that stores complete information in one table, incomplete information in another, and information about what was missing and why in a third table. Using an example from the *person* relvar of the feasibility study, each of the three tables was explained with emphasis on the relationship between the _UNKNOWN and the _MISSING tables.



KNOWN/UNKNOWN Model

Table: Person (Known, Unknown, and Missing)

Table: _KNOWN

EKEY	FIRST	MI	LAST	DOB
11	Edgar	F	Codd	1923
12	Chris	J	Date	1941

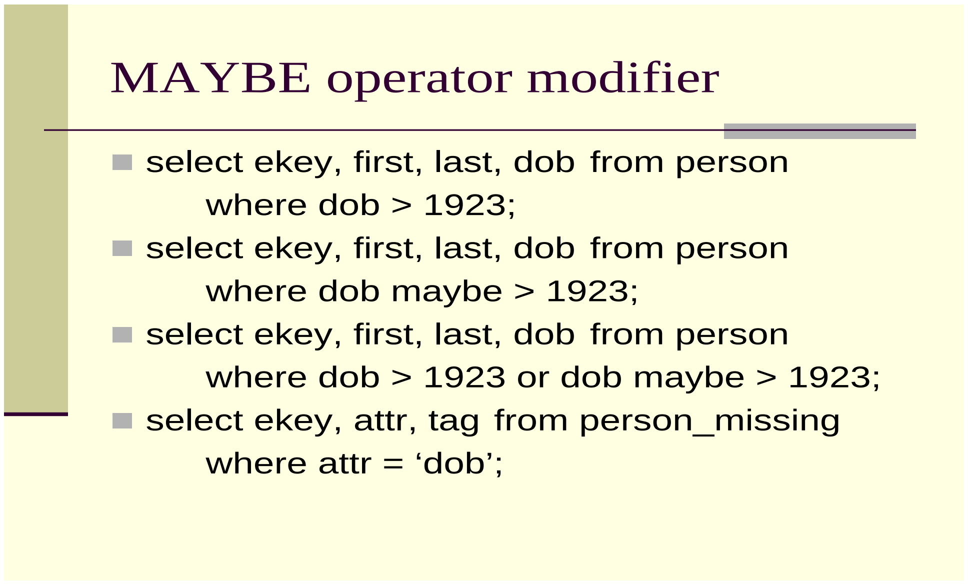
Table: _UNKNOWN

EKEY	FIRST	MI	LAST	DOB
13	Hugh		Darwin	
14	Andrew		Warden	

Table: _MISSING

KEY	ATTR	TAG
13	mi	UNK
13	dob	UNK
14	mi	N/A
14	dob	N/A

Figure 48: MyKU Tutorial KNOWN/UNKNOWN relvar



MAYBE operator modifier

- select ekey, first, last, dob from person
  where dob > 1923;
- select ekey, first, last, dob from person
  where dob maybe > 1923;
- select ekey, first, last, dob from person
  where dob > 1923 or dob maybe > 1923;
- select ekey, attr, tag from person_missing
  where attr = ‘dob’;

Figure 49: MyKU Tutorial MAYBE match operator

Appendix K **Feasibility Study Script**

Test Script

When answering a question, please consider 1 being 'low' or 'not very much' and 5 being 'high' or 'a great deal'. If you have any questions, please do not hesitate to ask.

Have fun!

Section 1.

If you have any questions, please do not hesitate to ask. The tables shown below are for your reference. The first section asks you to use this table of employees that represents missing values using NULL. A NULL is shown by the value being blank.

1. Table **emp** has six columns. Which columns are missing a data value?

2. Using table **emp** and **IS NULL**, write a query in SQL that chooses the columns ekey, first, mi, and last for the employees whose middle initial is missing. Cut and paste your query and the result of running it below.

3. Do you know why these middle initials are missing?

   How do you know?

Table 51: emp

ekey	first	mi	last	dob	pt
11	Edgar	F	Codd	1923	0
12	Chris	J	Date	1941	0
13	Hugh	NULL	Darwen	NULL	0
14	Andrew	NULL	Warden	NULL	0
15	NULL	NULL	Parker	1985	0
16	Charles	W	Bachman	1924	0
21	Jeffrey	D	Ullman	1942	1
22	Margo	I	Seltzer	NULL	1
23	Fabian	NULL	Pascal	NULL	1
24	David	?	McGoveran	-1	1
25	Chris	NULL	Date	1941	1
26	Andrew	NULL	Warden	NULL	1

3.a. Did the result you received meet with your expectations?

1 is a strong no, 5 is a strong yes ____.

This second section of the script asks you to use the following tables. The metadata is in a table named **md**, which explains the meaning of the tags.

Table 52: md (metadata)

tab	**name**	**meaning**
UNK	Applicable	property applicable - value unknown
NYE	Applicable	property applicable - value does not yet exist
UND	Invalid	property applicable - value is undefined
INV	Invalid	property applicable - value input is invalid
MIS	Invalid	property applicable - value withheld at input
N/A	Inapplicable	property not applicable to this item
REM	Unknowable	value declared unknowable; withheld or removed
NIL	Unknowable	value result from SQL operation is empty set

Person data can be queried using table **person** or each part of **person** can be queried separately using **person_KNOWN**, **person_UNKNOWN**, and **person_MISSING**.

Table **person_KNOWN** contains only complete information.

Table 53: person_KNOWN

ekey	first	mi	last	dob	pt
11	Edgar	F	Codd	1923	0
12	Chris	J	Date	1941	0
16	Charles	W	Bachman	1924	0
21	Jeffrey	D	Ullman	1942	1

Table **person_UNKNOWN** contain rows that are missing a data value.

Table 54: person_UNKNOWN

ekey	first	mi	last	dob	pt
13	Hugh		Darwen		0
14	Andrew		Warden		0
15			Parker	1985	0
22	Margo	I	Seltzer		1
23	Fabian		Pascal		1
24	David	?	McGoveran	-1	1
25	Chris		Date	1941	1
26	Andrew		Warden		1

Table **person_MISSING** explains why the data in **person_UNKNOWN** is missing.

Table 55: person_MISSING

ekey	attr	tag
13	dob	UNK
13	mi	UNK
14	dob	N/A
14	mi	N/A
15	first	UNK
15	mi	UNK
22	dob	MIS
23	dob	UNK
23	mi	UNK
24	dob	INV
24	mi	INV
25	mi	UNK
26	dob	MIS
26	mi	MIS

4. Use table person and write a query to choose ekey, first, mi, and last columns for persons whose middle initial is 'F'.

   Cut and paste your query and result below.

5. Use MAYBE and table person to write a query that chooses ekey, first, mi, and last columns for persons whose middle initial may be 'F'.

   Cut and paste your query and result below

6. Write a query that combines the queries in (4) and (5) above to create a result set where middle initial is equal to or may be equal to 'F'?

   Cut and paste your query and result below.

7. Refer to the result sets from query (2) and query (5) above.

Why does query (2) have more rows than query (5)?

8. The following query results all display the same information:

   "All persons for whom the middle initial is missing."

   How easy is each result to understand?

8.a. **Model A**

Table 56: UNKNOWN

ekey	first	mi	last
13	Hugh		Darwen
14	Andrew		Warden
15			Parker
23	Fabian		Pascal
24	David	?	McGoveran
25	Chris		Date
26	Andrew		Warden

Table 57: MISSING

ekey	attr	tag
13	mi	UNK
14	mi	N/A
15	mi	UNK
23	mi	UNK
24	mi	INV
25	mi	UNK
26	mi	MIS

8.a. Is it easy to understand this data?

1 is low and 5 is high clarity. ____

Comments:

8.b. **Model B**

Table 58: UNKNOWN

ekey	first	mi	mi_TAG	last
13	Hugh		UNK	Darwen
14	Andrew		N/A	Warden
15			UNK	Parker
23	Fabian		UNK	Pascal
24	David	?	INV	McGoveran
25	Chris		UNK	Date
26	Andrew		MIS	Warden

8.b Is it easy to understand this data?

1 is low and 5 is high clarity. ____

Comments:

8.c. **Model C**

8.c Is it easy to understand this data?

1 is low and 5 is high clarity. ____

Comments:

Table 59: UNKNOWN

ekey	first	mi	last
13	Hugh	UNK	Darwen
14	Andrew	N/A	Warden
15		UNK	Parker
23	Fabian	UNK	Pascal
24	David	INV	McGoveran
25	Chris	UNK	Date
26	Andrew	MIS	Warden

9. **Observations:**

Please answer the following questions either **Yes, No,** or **Unsure** on a scale with **1** being **low** and **5** being **high**.

9.a Did the missing data tags provide more information than NULL?

Yes ____ No ____ Unsure ____

9.b Do you see a benefit to representing missing data with tags (UNK, N/A, INV, etc) in database systems?

Yes ____ No ____ Unsure ____

9.c How intuitive is the MAYBE modifier for writing queries when data values are missing?

With **1** being **low** and **5** being **high** _____

<u>PLEASE SAVE THIS FILE</u>

Thank you very much for your help!

Appendix L **Database for Tutorial and Study**

Table 60: EMP

ekey	first	mi	last	dob	pt
11	Edgar	F	Codd	1923	0
12	Chris	J	Date	1941	0
13	Hugh	NULL	Darwen	NULL	0
14	Andrew	NULL	Warden	NULL	0
15	NULL	NULL	Parker	1985	0
16	Charles	W	Bachman	1924	0
21	Jeffrey	D	Ullman	1942	1
22	Margo	I	Seltzer	NULL	1
23	Fabian	NULL	Pascal	NULL	1
24	David	?	McGoveran	-1	1
25	Chris	NULL	Date	1941	1
26	Andrew	NULL	Warden	NULL	1

Table 61: ASSIGN

ekey	pkey	percent	eff_date
11	P1	0.50	12/13/2012
11	P3	0.50	10/01/2012
12	P1	1.00	08/16/2012
13	P1	0.75	08/01/2012
13	P2	0.25	06/15/2012
14	P2	1.00	10/12/2012
24	P3	1.00	06/01/2012
25	P2	1.00	09/13/2012

Table 62: person_KNOWN

ekey	first	mi	last	dob	pt
11	Edgar	F	Codd	1923	0
12	Chris	J	Date	1941	0
16	Charles	W	Bachman	1924	0
21	Jeffrey	D	Ullman	1942	1

Table 63: person_UNKNOWN

ekey	first	mi	last	dob	pt
13	Hugh		Darwen	NULL	0
14	Andrew		Warden	NULL	0
15			Parker	1985	0
22	Margo	I	Seltzer		1
23	Fabian		Pascal		1
24	David	?	McGoveran	-1	1
25	Chris		Date	1941	1
26	Andrew		Warden		1

Table 64: person_MISSING

ekey	attr	tag
13	dob	UNK
13	mi	UNK
14	dob	N/A
14	mi	N/A
15	first	UNK
15	mi	UNK
22	dob	MIS
23	dob	UNK
23	mi	UNK
24	dob	INV
24	mi	INV
25	mi	UNK
26	dob	MIS
26	mi	MIS

VITA

Marion Roberts Morrissett was born on February 3, 1950, in Roanoke, Virginia, and is an American citizen. He graduated from Patrick Henry High School, Roanoke, Virginia in 1968. He studyied history and received his Bachelor of Arts from the University of Virginia, Charlottesville, Virginia in 1972. He subsequently worked in Rare Books at Alderman Library of the University of Virginia. After learning to program IBM mainframe computers using the COBOL programming language, he worked for the Federal Reserve Bank of Richmond for twenty-eight years as a programmer, network administrator, and network engineer. He received his Mathematical Sciences Certificate in Computer Science in 1987 and his Master of Science in 1994 from Virginia Commonwealth University.