Performance Evaluation of $3^{rd}$ Normal Form Decompositions

by

Ben Mathew

Bachelor of Science
Computer Science
Florida Institute of Technology
2016

A thesis submitted to
Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Melbourne, Florida
May, 2019
The author grants permission to make single copies
We the undersigned committee hereby recommend that the attached
document be accepted as fulfilling in part the requirements for the degree of
Master of Science in Computer Science.

“Performance Evaluation of 3\textsuperscript{rd} Normal Form Decompositions”
a thesis by Ben Mathew

Marius C. Silaghi, Ph.D.
Associate Professor, Computer Sciences
Major Advisor

Khaled Shloub, Ph.D.
Assistant Professor, Computer Sciences
Committee Member

Stephen Wood, Ph.D.
Associate Professor, Ocean Engineering
Committee Member

Philip J. Bernhardt, Ph.D
Associate Professor and Department Head,
Department of Computer Engineering and Sciences
ABSTRACT

Performance Evaluation of 3\textsuperscript{rd} Normal Form Decompositions

by

Ben Mathew

Thesis Advisor: Marius C. Silaghi, Ph.D.

When a relational database is chosen, normalization theory is a set of guidelines that may lead to efficient database designs. Thus, normalization of tables in a database is a common process used for the analysis of relational databases. Sufficient normalization of databases aims to decompose existing relational tables in order to minimize database redundancy while preserving dependencies between attributes. It also facilitates correct insertion, deletion, and modification of data in the database. Given an un-normalized relational database, a redesign with no data redundancy and which is guaranteed to preserve dependencies is not always achievable. Previous studies have shown that it is not always possible to decompose databases into relations complying with the Boyce Codd Normal Form (which eliminates many of the simple redundancies), such as to guarantee the preservation of functional dependencies. Due to this fact, the immediately weaker normalization concept that guarantees the preservation of functional dependencies, the 3\textsuperscript{rd} Normal Form, is generally regarded as an industry standard for many types of database applications. Normalization does
not always reduce the size of a given database and frequently increases the modification and retrieval time of a given transaction. Decrease in performance stems from the fact that decomposition causes some queries to reconstruct original relations by joining multiple tables. The complexity of the joins between tables depends on the types of involved attributes and the integrity constraints imposed on those attributes.

The effect of normalization on the size of the database depends on the data stored in it. Since there is no guarantee on the effect in terms of size of the database, database designers need to always predict the impact each normalization step may have on the overall database. Thus, database designers always need to select good trade-offs between memory consumed by the database, presence of challenging database maintenance anomalies, and speed at which one can query data from the database. For the above reason, performance minded database designers tend to de-normalize relations that are accessed frequently. The table decomposition process sometimes involves reducing the set of functional dependencies to a minimalistic set which is known as canonical cover.

The main focus of this thesis is to examine the query performance of distinct versions of the same database where all these versions in the 3rd Normal Form as they are all derived using the same canonical cover. The databases represent the same relations with distinct schemas but are populated with the same data. This allows us to evaluate the impact of normalization approaches on the database space requirements. In order to evaluate the performance of each version of the database we
run the same queries on each of them. Here we define the performance of a database as the time taken by the database to execute the query.
## Contents

1 Introduction ................................................. 1
   1.1 Performance Evaluation Criteria .................. 3
   1.2 Hypothesis ........................................... 4

2 Background .................................................. 5
   2.1 Relational Model ..................................... 6
   2.2 Entity Relationship Model ......................... 7
   2.3 Database Design and Normalization Process ....... 9
   2.4 Query Optimization Techniques Enabled .......... 17

3 Problem Description ........................................ 20

4 Algorithm .................................................... 26
   4.0.1 Example ........................................... 28

5 Space and Time - Based Approach to Database Design .... 30
   5.1 Size of The Database ................................ 31
5.2 Time Taken by a Query on a Database

6 Experiments and Results
   6.1 Introduction
   6.2 Experimental Setup
   6.3 Experiment
      6.3.1 Version-1
      6.3.2 Version-2
      6.3.3 Version-3
   6.4 Queries
      6.4.1 Discussion
   6.5 Results
      6.5.1 Examination of Execution Time of Queries
      6.5.2 Examination of Memory Consumed by Databases
   6.6 Analysis

7 Applications
   7.1 University Database
      7.1.1 Data Requirements and Constraints
      7.1.2 Entities
      7.1.3 Entity Relationship Diagram
      7.1.4 Relational Schema
      7.1.5 Functional Dependencies
8 Contributions and Conclusions 83
List of Figures

2.1 Sample Relationship Set ........................................ 9

3.1 E R diagram for Voting System ................................. 22

6.1 Summary of Execution Time for given Query ............... 55

6.2 Real Execution Time of Query ................................. 56

6.3 SYS Execution Time ............................................. 57

6.4 User Execution Time ............................................. 58

6.5 Memory Requirement for Each Version ....................... 59

6.6 Graphical representation of Memory Requirement .......... 59
# List of Tables

2.1 Un-normalized database system ........................................... 11
2.2 1st Normal Form Normalized database system .......................... 12
2.3 2nd Normal Form Normalized database system .......................... 13
2.4 2nd Normal Form Normalized database system .......................... 14
2.5 3\textsuperscript{rd} Normal Form Normalized database system ............... 15
2.6 3\textsuperscript{rd} Normal Form Normalized database system ............... 15
2.7 3\textsuperscript{rd} Normal Form Normalized database system ............... 16
Acknowledgements

I am deeply grateful to everyone that helped me bring this work into existence. I would like to thank my parents for their unconditional support in all my endeavors. I would also like to thank my friends and professors of the computer science department at Florida Institute of Technology for all the support and guidance throughout the development of this Master’s thesis. I would like to thank Dr. William Shoaff for all his help as an academic advisor and mentor during my graduate and undergraduate study at Florida Institute of Technology. I would also like to thank Dr. Marius C. Silaghi for his constant support and guidance during this endeavor.
Dedication

This work is dedicated to which it belongs to, the Supreme Power, that made way for this work’s existence.
Chapter 1

Introduction

Data has become an important strategic resource for many organizations, including industry, government, and even academia. The simplest data management is based on file systems. The complexity of data management and data manipulation techniques increased in recent times. Trade-offs between time and memory performance are difficult to predict. It is often found that the performance of these database systems can be improved by changing the logical schema according to which data is represented and stored. Storage mechanisms have to be evaluated both theoretically as well as experimentally. At the start of a typical System Development Life cycle (SDLC) data analysts first establish the requirements of the application in a correct and consistent manner [12]. Some of the most widely used designing techniques include the Entity - Relationship Model which was originally proposed by [5] and an Object-Oriented Model (e.g. [4]). However, in a broader sense, the depiction of interconnectedness of several entities can be traced back to ancient Greece with the works of Aristotle,
Socrates, and Plato. Any of the above modeling techniques can be used to transform data into relations at the logical design phase. However, the resulting relations may contain patterns with undesirable anomalies and therefore need to be normalized. Such anomalies may consist in repetition of data, inability to represent some data, and even loss of data. In order to reduce the occurrence of such anomalies in the database, the schema can be normalized.

Normalization of databases refers to the process of regrouping attributes into relations in ways that can guarantee the avoidance of certain types of anomalies. This technique was introduced by Edgar F. Codd [8]. The process of reorganizing relations according to normal form concepts has been investigated by many researchers since Codd initiated the subject in 1971. The normal forms represent standardized sets of criteria that ensure specific anomaly avoidance. The most well known normalization criteria include the First Normal Form (1NF), the Second Normal Form (2NF), the Third Normal Form (3NF), the Fourth Normal Form (4NF), and the Fifth Normal Form. The process of normalization of a database is typically done in stages i.e. data is first normalized to the First Normal From. If it already is in the first normal from then it is normalized to the Second Normal Form, and so on. Such a systematic process is well documented in a number of different sources (e.g. Kent, 1983, Dutka and Hanson 1989 [15, 18]).

Although relations of a database in Higher Normal forms may rid data of redundancies and anomalies they may incur further maintenance costs or even degrade
system performance. Hence the database designer has to constantly consider the
above trade-offs when normalizing the various relation of the database. An over-
all cost/benefit analysis is therefore a useful guideline that is more likely to lead
to better database design. literature in the above research area has shown that re-
duced anomalies, storage requirements and transaction response times are three of
the major parameters used for such an analysis. Tables in databases that are overly
de-normalized can be advantageous when the attributes in them are static in nature.
Queries from such databases tend to have lesser execution time due to the limited
number of table joins required to fetch the data. On the other hand, a table that
is overly normalized requires multiple table joins to get an attribute from the table.
This hinders query performance and can be noticed in databases systems that are
large in complexity.

1.1 Performance Evaluation Criteria

In this work we focus on the following criteria for evaluating the performance of
a logical database design (aka. schema) for managing a given body of data (i.e.,
database instance benchmark):

- Actual machine usage which includes the persistent storage needed for the con-
tents of the database instance

- Amount of time taken by a benchmark query to be executed [2].
Other criteria, including ease of maintenance and security, are not covered in this effort. From the perspective of ease of maintenance, all our databases are in 3rd Normal Form, inheriting the corresponding resistance to update, deletion, and insertion anomalies.

1.2 Hypothesis

It is common practice to recommend decomposition of databases to the 3rd Normal Form, which is an industry standard for relational databases. However this decomposition is not unique, as it is easily shown using several major domains of application for relational databases, exemplified later in this work.

Our hypothesis is that there can be large differences, between the possible 3rd Normal Form decompositions of a database, in what concerns their performance with respect to the aforementioned performance metrics on relevant benchmarks, and that a methodology can be developed for their evaluation.

This prompts a need for more careful consideration of the possible normalization outcomes, and their evaluation on benchmarks as close to the target application as possible.
Chapter 2

Background

Digital databases are used for storing a wide range of data. Examples can be found in the fields of Medicine, Banking, and even Academia. Databases form an essential part of every enterprise today. Some types of data are specific to given enterprises while others are common to categories of enterprises. A data model can be seen as a combination of three components namely a collection of data structure types which are the building blocks of any database system, a set of inference rules which can be applied to any valid instance of data that is listed in the model, and a collection of general integrity rules which implicitly or explicitly define the states that the given database may go through [6]. The most popular data model that is used by many corporations today is the Relational Model, proposed by Edgar F. Codd, that structures and manages data in a form that is consistent with first order logic [7].

The reason why it has achieved such a primal position in the design process is due to its simplicity and ease of use which enables database designers and programmers
to design and implement well-functioning databases.

The task of creating an efficient database application is complex in nature. It involves design of a database schema (which is defined as the structure of a database annotated as a formal language or in a pictorial representation), design of programs that fetch the data from the tables in a given database system, and design of a security protocol that controls the ease of access to the data stored in a database.

When designing small applications, it may be feasible for a database designer who understands the requirements to directly decide on what tables need to be created, and on the attributes and constraints that should be imposed in a relation. However, such a direct approach may prove to be error prone when dealing with real-life, large applications, as they tend to have a greater deal of complexity in them. Hence using the design process outlined in the relational model is helpful.

2.1 Relational Model

In the early development stages of the field of Database Systems, in order to appropriately represent the data three models were proposed, namely the network model, the relational model, and the entity set model. The Network model provided a more natural view of the data by separating data entities and the relationships between these entities.

The relational model was introduced by Edgar F. Codd in 1970 [8]. Entity sets used to be store in separate files, also called *tables*, and the relational model allowed
any file to relate to another file by means of a common field. The provided analysis showed that the data size obtained with the relational model for the accompanying data was reduced when compared to the network model [16]. The relational model allows for changes to be made to the physical structure of the database without affecting the system’s ability or performance to access or modify data, providing a basis for a number of important research efforts in the area of database systems, some of which includes:

- The theory of data and relationship constraints

- The development of a database access language which is known as the structured query language (SQL)

- The development and a major proportion of the common commercial database systems.

### 2.2 Entity Relationship Model

The Entity Relationship Model was a model that was proposed by Chen in 1976 [5]. It has most of the advantages that have been seen in the aforementioned relations. The entity relationship model adopts a more natural representation of the real world which consists of entities and relationships. It was developed as a means to facilitate database design by mapping each aspect of the set to the three main notions, namely entity sets, relationship sets, and attributes.
Entity Set

An entity is an ontological element that is distinguishable from all other objects in the real world. Each entity has a set of properties and values of the properties that uniquely identify the entity. An entity set is a set of entities that share the same type and have the same properties or attributes. An entity in the above model is analogous to a table in the relational model. For example, one can model real world people like Employee, Student, Professor as entities. An entity is pictorially represented by means of a square box in an Entity Relationship Diagram.

Attribute

An attribute is a descriptive property that is possessed by each member of the entity set. If in the above case we have the entity Employee then we have employeeID, employeeName, designated as attributes. They are pictorially represented by means of an oval in a Entity Relationship Diagram.

Relationship

A relationship is an association between a set of entities. A relationship set is a set of relationships of the same type. For example, as mentioned above, we can have a relationship set that associates students with professors. It is pictorially represented by means of a Diamond figure in a Entity Relationship Diagram.

With the use of the above terms we can model real world concepts like Students, Professors and the relationship between them as in the Figure 2.1. In this diagram,
Professor is an entity with attributes ProfessorName, ProfessorID, Department. In a similar fashion Student is an entity with attributes StudentName, address and StudentID. Each of these entities are associated by “advise” which is a relationship.

2.3 Database Design and Normalization Process

According to the Relational Model, the process of database design begins with gathering the requirements that the database must satisfy. This is achieved by interacting with the end-user of the database system in order to figure out his/her needs. The outcome of this phase is a requirement document that outlines all the criteria that the system must comply with [11].

From those requirements data designers must identify all key entities and relationships between the entities by means of an appropriate Entity Relationship Diagram. A fully developed conceptual model indicates all the functional requirements of the enterprise and avoid the major pitfalls that are common in designing a database schema and redundancy. A good and complete relational model would represent ev-
ery aspect of the requirements in a manner that is easily maintainable, as defined by any of the various available standards.

Database designers use a process called normalization to reduce redundancy and enhance maintainability. Normalization was introduced by Edgar F. Codd [7] as a guideline to transform relations in a database such that they conform with any of a set of given standards for avoiding redundancy. The extent to which a table is to be normalized is dependent on desired trade-offs concerning other properties. It is generally desired for the transformation of each given relation to be lossless in nature and for the preservation of the relationship between different attributes to be guaranteed.

A decomposition is lossless in nature if the intersection of attributes from the decomposed tables functionally determine at least one of relations. This can be mathematically represented as follows [7]:

If $R_1$ and $R_2$ are the decomposed relations of the relations $R$ then the decompositions are said to be lossless in nature if $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$

Next we define the main standards that were identified by past research on Database Normalization.

1st Normal Form  A given relation is said to be in the first normal form if all its attributes are atomic in nature. An attribute in a relation is said to be atomic when attributes are not composite and consist of no more than one semantically valid unit. If the above conditions hold then the relation is said to be in 1st Normal form. In order
to better understand the above definition the following example may be considered.

**Example:** In order to better explain the normalization process let us use a Commercial database system for a Movie rental store. The database system stores all the information of all the customers that rented a movie form the store. The database system thus stores the Full name of the customer, Physical Address, Movies Rented, Salutations and the category of the movie. A un-normalized version of the database is presented below

<table>
<thead>
<tr>
<th>Membership-ID</th>
<th>SalutationID</th>
<th>Salutations</th>
<th>Full-Name</th>
<th>Physical Address</th>
<th>Movies Rented</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Mr.</td>
<td>Ben Mathew</td>
<td>3151 South Babcock Street</td>
<td>Clash of Titans, Pirates of the Caribbean</td>
<td>Action</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Mrs.</td>
<td>Jessica Edwards</td>
<td>3152 South</td>
<td>Daddy’s Little Girl, X-men</td>
<td>Romance</td>
</tr>
</tbody>
</table>

A version of a database that is not in the 1\text{st} Normal Form is seen in Table 2.1. This table is not in the First Normal Form because the **Movies Rented** can contain multiple values. For example the first row of the table includes both values “Clash of Titans” and “Pirates of the Caribbean ” under **Movies Rented** In order to convert the above database into the First Normal Form we have to split the rows. A possible solution is shown in Tables 2.2.

2\text{nd Normal Form} The 2\text{nd} Normal Form is an extension of the 1\text{st} Normal Form. In addition to requirement that each attribute in the table should be atomic, the above normal form stipulates that proper subsets of a candidate key do not functionally determine non-key attributes [7, 9].
In order to better understand the implications of the 2\textsuperscript{nd} Normal Form, we introduce the notion of Keys and Functional dependencies which are more generally considered as constraints on a database to ensure that it satisfies certain properties. Tables or relations that satisfy all such constraints are know as legal constraints.

Given a relation $R$, a set of attributes $X$ in $R$ is said to functionally determine another set of attributes $Y$, also in $R$, if, and only if, each $X$ value in $R$ is associated with precisely one $Y$ value in $R$. $R$ is then said to satisfy a functional dependency [18].

We can use functional dependencies in two ways:

- To test a given relations, to see whether they would be considered “legal” under a given set of functional dependencies. If a relation would be legal under a set of Functional dependencies $F$ then we say that $r$ satisfies $F$.

- To specify the constraints $F$ on the set of legal relations. Thus one shall only be concerned with relations that satisfy those given constraints. Thus we say that a set of functional dependencies $F$ hold on a set of relations $R$.

For example, if we were to consider the example we used for the 1st Normal and add the restriction that a customer can rent more than one movie we see that the same

<table>
<thead>
<tr>
<th>Membership-ID</th>
<th>SalutationID</th>
<th>Salutations</th>
<th>Full-Name</th>
<th>Physical Address</th>
<th>Movies Rented</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Mr.</td>
<td>Ben Mathew</td>
<td>3151 South Babcock Street</td>
<td>Clash of Titans</td>
<td>Action</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Mr</td>
<td>Ben Mathew</td>
<td>3151 South Babcock Street</td>
<td>Pirates of the Caribbean</td>
<td>Action</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Mrs</td>
<td>Jessica Edwards</td>
<td>3152 South</td>
<td>Daddy's Little Girl</td>
<td>Romance</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Mrs</td>
<td>Jessica Edwards</td>
<td>3152 South</td>
<td>X-men</td>
<td>Romance</td>
</tr>
</tbody>
</table>
customer appears in more than one row. as seen in 2.3. Hence the candidate keys in this case are Membership-ID and Full-Name and the non-prime attributes are Physical Address, Movies Rented, SalutationID, Salutations, and Category.

We see that the database displayed in 2.3 adheres to the 1st Normal Form as each attribute is atomic in nature. However it does not adhere to the 2nd Normal Form as the non-prime attribute is dependent on the candidate keys alone which is a proper subset of the candidate key. This violates the preconditions for a table to be in the 2nd Normal Form. In order to remedy the above situation we break the above table into two tables as shown in figure 2.3, 2.4. The decomposed tables satisfies the conditions of the 2nd Normal Form. Additionally it should be noted that the decompositions are lossless in nature and the functional dependencies are preserved.

<table>
<thead>
<tr>
<th>Membership-ID</th>
<th>Full-Name</th>
<th>Physical Address</th>
<th>Movies Rented</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ben Mathew</td>
<td>3151 South Babcock Street</td>
<td>Clash of Titans</td>
<td>Action</td>
</tr>
<tr>
<td>1</td>
<td>Ben Mathew</td>
<td>3151 South Babcock Street</td>
<td>Pirates of the Caribbean</td>
<td>Action</td>
</tr>
<tr>
<td>2</td>
<td>Jessica Edwards</td>
<td>3152 South</td>
<td>Daddy's Little Girl</td>
<td>Romance</td>
</tr>
<tr>
<td>2</td>
<td>Jessica Edwards</td>
<td>3152 South</td>
<td>X-men</td>
<td>Romance</td>
</tr>
</tbody>
</table>

3rd Normal Form A relation is said to be 3rd Normal Form if it adheres to all the requirements set forth by the 2nd and 1st Normal Form. In addition, it should also have the property that all of its non-key attributes depend solely on the primary key and nothing else. The above requirement removes the redundancy that is caused by
attributes which are dependent on the non-key attribute.

The process of converting a table into the 3\textsuperscript{rd} Normal Form is represented by means of an algorithm that starts off by computing the minimal cover of the functional dependencies and grouping key and non-key attributes [21]. Relations that are in the 3\textsuperscript{rd} Normal Form are guaranteed to have lossless decomposition and the functional dependencies are guaranteed to be functionally preserving. For example if we were to refer to the schema used as an example of the 2nd Normal Form which is a database that stores attributes of a customer that borrows movies from a video store . in 2.3 the attribute \textbf{Salutations} is functionally dependent on the \textbf{SalutationID} . The attribute \textbf{SalutationID} is functional dependent on the attribute set \textbf{Member-ID} and \textbf{Full-Name}. Hence the attribute Salutations which is a non-prime attribute is dependent on the super key \textbf{Member-ID} and \textbf{Full-Name}. The above scenario does not comply with the 3\textsuperscript{rd} Normal Form . We can resolve the above redundancy by decomposing the table 2.3 into tables. When we perform such a decomposition step we arrive at relations 2.5, 2.6 and 2.7.
Table 2.5: 3\textsuperscript{rd} Normal Form Normalized database system

<table>
<thead>
<tr>
<th>Membership-ID</th>
<th>Full-Name</th>
<th>Physical Address</th>
<th>Movies Rented</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ben Mathew</td>
<td>3151 South Babcock Street</td>
<td>Clash of Titans</td>
<td>Action</td>
</tr>
<tr>
<td>1</td>
<td>Ben Mathew</td>
<td>3151 South Babcock Street</td>
<td>Pirates of the Caribbean</td>
<td>Action</td>
</tr>
<tr>
<td>2</td>
<td>Jessica Edwards</td>
<td>3152 South</td>
<td>Daddy’s Little Girl</td>
<td>Romance</td>
</tr>
<tr>
<td>2</td>
<td>Jessica Edwards</td>
<td>3152 South</td>
<td>X-men</td>
<td>Romance</td>
</tr>
</tbody>
</table>

Table 2.6: 3\textsuperscript{rd} Normal Form Normalized database system

<table>
<thead>
<tr>
<th>Membership-ID</th>
<th>Full-Name</th>
<th>SalutationID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ben Mathew</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Jessica Edwards</td>
<td>2</td>
</tr>
</tbody>
</table>

**Boyce Codd Normal Form** A relation is said to be in Boyce Codd Normal Form if it adheres to all the requirements set forth in the 1\textsuperscript{st}, 2\textsuperscript{nd}, and 3\textsuperscript{rd} Normal Forms. In addition to the above requirement it is an extension of the 3\textsuperscript{rd} Normal Form which stipulates that even the attributes in a candidate key should only non-trivially depend on the candidate keys. If the above conditions is satisfied then the relation is said to be in Boyce Codd Normal Form. The above form is not a popular choice of normalization standard for databases. This is due to the fact that although decompositions of this form are satisfied to have lossless decompositions, they do not guarantee the preservation of functional dependencies. For example if we were to take the relational schemas 2.3, 2.3, 2.3 that served as an example for decomposition of the 3rd Normal
<table>
<thead>
<tr>
<th>SalutationID</th>
<th>Salutations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mr.</td>
</tr>
<tr>
<td>2</td>
<td>Mrs.</td>
</tr>
</tbody>
</table>

Table 2.7: 3rd Normal Form Normalized database system

Form. We can list out the functional dependencies as follows:

- Membership-ID, Full-Name → Physical Address, Movies Rented, Category
- Membership-ID, Full-Name → SalutationID
- SalutationID → Salutations

We can see in the above tables that Membership-ID, Full-Name and SalutationID are candidate keys in the respective relations. Thus each functional dependency in the above set of functional dependencies are trivial in nature the attributes Membership-ID, Full-Name, and SalutationID are superkeys in the above relation Thus the relations 2.5, 2.6 and 2.7 are in adherence to the Boyce Codd Normal Form and no further normalization is necessary.
Higher Normal Form  There are higher Normal Forms that help us understand and remove complex types of redundancies. For example, the 4th Normal Form removes redundancy that might arise from Multivariate dependencies. Another example is the 5th Normal Form which removes redundancy that arises from join dependencies.

2.4 Query Optimization Techniques Enabled

For a given SQL statement there can be a countable infinite number of ways of performing the task specified in the query, depending on the complexity of the statement and the underlying database schema that is used. It is the job of the query planner to select an algorithm that provides a response to a given query with minimum disk input-output and CPU overhead [10].

The performance with which a query can be executed on a database system depends largely on the Query optimization techniques that were used to gather the information required. Such gathering of information must be done by performing the prerequisite join operations in an efficient manner. If the Data Management Systems lacked performance, handling, selecting and/or updating operations must be viewed as a major challenge on the part of such systems. By enabling various query optimization techniques, such challenges can be addressed. These techniques range from indexing and logical transformations of the input query to the optimization of the access paths and storage data on the file system level [1].

We have chosen a SQLite database system for the purposes of our experiments
and we are mostly interested in queries that have complex join operations. In this regard the current SQLite version (version 3.23.1) that is used for the purposes of the experiment employs loop joins. This means that joins are implemented as nested loops, and may be reordered to achieve greater performance.

The default order for nested loops in a join operation starts with the left-most table in the FROM clause of a query, and the right-most table forms the inner loop. However this is not the case in SQLite. It will nest of the loops in an order that is completely different from the above if it considers that above ordering will help it select a better index.

Also the optimizations strategy that is used is based on the nature of the JOIN. INNER JOIN can be freely reordered as the ordering of the tables does not make any difference in the result of the join operation [13]. However LEFT OUTER JOINs are neither commutative nor associate in nature, hence they will not be reordered by the optimizer. However in the event that there is a query that has a INNER JOIN to the left and right of a OUTER JOIN, this might be reordered if the optimizer thinks it to be advantageous. In all other scenarios OUTER JOINS are evaluated in the order that they occur [3].

Lastly a special case which is of importance to us is the manner in which the query optimizer deals with CROSS JOIN operations. The cross join operation is by nature commutative in nature and hence reordering the tables will not have much result on the final result set. However SQLite chooses never to reorder the tables
in the event of a CROSS JOIN. Instead it provides a mechanism by which the programmer can force SQLite to select a particular loop nesting order.

In all the above cases whether it is a INNER JOIN, OUTER JOIN or CROSS JOIN, SQLite uses an efficient algorithm that runs in polynomial time. Due to the above fact SQLite is able to execute queries that involve a large number of joins in a matter of milliseconds.

Join reordering is automatic and usually works well enough such that a database programmer doesn’t need to consider it but if we are to gather information of the execution time of query it is a noteworthy factor to consider.
Chapter 3

Problem Description

The Time Database Design Problem is a pair $\langle R, A \rangle$. We make the assumption that we are given a pair of relation and functional dependencies that are in the third normal form. We remind that a database is said to be in the Third Normal Form if every non-prime attribute of $R$ is non-transitively dependent on every key of the relation (see Section 2.3).

It may be noted that, the problem statement can be formalized as follows. We wish to find a decomposition of $R$ into smaller relations such that, when in each of these new relations the functional dependencies that prevailed in $R$ are preserved, then by merging the relations we can recover the original relation $R$ with its functional dependencies. Also we wish that the set of new relations occupy the least amount of space and queries that run on them can be executed in an efficient and speedy fashion.
Discussion  Also, it should be noted that the functional dependencies must still be preserved in the decomposed relation. It is interesting to note that the resultant table that is received after the decomposition process is not unique. For example, the decomposed relations that is formed after the decomposition process in the 3rd Normal Form can vary depending on the manner in which the functional dependencies are analyzed. Hence a database designer can generate multiple such decompositions but always takes the one that gives an optimal solution in terms of space and time taken to execute a query.

From a theoretical perspective it is possible to have a distinct 3rd Normal Form for all possible permutations of functional dependencies. However the number of such permutations in the general case is exponential in nature and cost of generating those permutations strongly prohibits using such an approach. Thus we need to explore another approach by which we can analyze decompositions that are in the above Normal Form.

Example  We can model the problem of a committee of politicians debating over a network over a set of policies that needs to be implemented for the proper functioning of a government. The data handled can be managed by the set of relations given below:

\[
\text{Peer(peer_ID, GID_key, peer_address_ID, peer_instance_ID, peer_org_ID)}
\]

\[
\text{Peer_address(peer_address_ID, peer_ID, instance, type, domain, agent_version)}
\]
Peer_instance(peer_instance_ID, peer_ID, peer_instance, branch, version, plugin_info)

Peer_org(peer_org_ID, peer_ID, organization_ID, served, last_sync_date)

Figure 3.1: E R diagram for Voting System
The Entity Relationship Diagram for the above schema are shown in Figure 3.1.

The functional dependencies that hold true for the above database system can be described as follows:

\[ \text{peer\_ID, GID\_key} \rightarrow \text{peer\_address\_ID, peer\_instance\_ID, peer\_org\_ID} \]

\[ \text{peer\_address\_ID} \rightarrow \text{peer\_ID, instance, type, domain, agent\_version} \]

\[ \text{peer\_instance\_ID} \rightarrow \text{peer\_ID, peer\_instance, branch, version, plugin\_info, last\_sync\_date} \]

\[ \text{peer\_org\_ID} \rightarrow \text{peer\_ID, organization\_ID, served, last\_sync\_date} \]

This database can be reduced to a set of relations in the 3\textsuperscript{rd} Normal Form using the standard decomposition procedure [21]. Using this decomposition algorithm we will get a set of relations \( R \) as given below:

**Peer**

\[
\begin{align*}
\text{peer\_ID}, \\
\text{GID\_key}, \\
\text{peer\_address\_ID}, \\
\text{peer\_instance\_ID}, \\
\text{peer\_org\_ID}
\end{align*}
\]

**Peer\_Address**

\[
\begin{align*}
\text{peer\_address\_ID}, \\
\text{peer\_ID}, \\
\text{instance},
\end{align*}
\]
domain,
agent_version

**Peer_instance**

peer_instance_ID,
peer_ID,
instance,
type,
domain,
agent_version

**Peer_org**

peer_org_ID,
peer_JD,
organisation_ID,
served,
last_sync_date

The problem as stated earlier in the chapter is to find a decomposition-reorganization of the relations that minimizes the overall space requirements of the database. In order to achieve that, we might consider joining two or more tables into a single table. One can advance towards this goal in the above schema by combining Peer and Peer_org by a join operation. This gives the following set of relations:
Peer

peer_ID,
GID_key,
peer_address_ID,
peer_instance_ID,
orGANisation_ID
served,
last_sync_date

Peer_address

peer_address_ID,
peer_ID,
instance,
type,
domain,
agent_version
Chapter 4

Algorithm

For most applications the Boyce Codd Normal Form represents a powerful normalization standard. But in certain circumstances the above normal form does not always guarantee that the functional dependencies are preserved in the decomposed schemas. When such a scenario arises a database designer can resort to the 3rd Normal Form which is weaker compared to the above but always considered to be a lossless decomposition and preserves the functional dependencies. Hence for the above reasons the 3rd Normal Form decomposition is considered to be an industry standard Normal Form for all schemas of a given database. The algorithm to compute the 3rd Normal Form decompositions is shown in Figure 1:

Given a set of functional dependencies, the 3rd Normal Form first computes the canonical cover of the above set which is the set of minimalistic functional dependencies that are devoid of redundancies and extraneous attributes. An attribute in a functional dependency is said to be extraneous in nature if the absence of the specific
let $F_c$ be a canonical cover for $F$;

i := 0;

foreach functional dependency $\alpha \rightarrow \beta$ in $F_c$ do

i := i + 1;

$R_i := \alpha\beta$;

if none of the schemas $R_j$, $j = 1, 2, \ldots, i$ contains a candidate key for $R$ then

i := i + 1;

$R_i :=$ any candidate key for $R$;

Until no more $R_j$'s can be deleted do

if any schema $R_j$ is contained in another schema $R_k$ then

$R_j := R_i$;

i := i - 1;

return $(R_1, R_2, \ldots R_i)$;

Algorithm 1: Algorithm for Normalization to 3NF [20]
attribute does not change the closure of the set of functional dependencies.

As relevant in the above algorithm, decomposition of the original table into sub tables is achieved by taking a set of functional dependencies and extracting one schema at a time. This is a unique feature of the above algorithm when compared to other algorithms used to normalize schemas.

Another feature that is seen in the decomposed schemas obtained by using the above algorithm is that the decompositions are not unique in nature. This is prevalent due to the fact that a set of functional dependencies can have more than one canonical cover. Furthermore it is observed that the decomposed tables that are obtained by using the above algorithm can vary depending on the order in which the functional dependencies are considered. Thus in this thesis we adopt a performance evaluation perspective to decide which of the decompositions is most apt to be considered as the final version of the database. This is achieved by considering the trade-offs that are listed earlier. Lastly it is noteworthy that the above algorithm which decomposed a given scheme into the 3\textsuperscript{rd} Normal Form can be implemented in polynomial time.

4.0.1 Example

If we were to consider the example of a voting database system that was introduced in the previous section. The functional dependencies that constraints such a database system can be listed as follows:

- peer\_ID, GID key \rightarrow peer\_address\_ID, peer\_instance\_ID, peer\_org\_ID
We prove that the above set of functional dependencies are minimal in nature and hence is the canonical cover of the set of functional dependencies in subsequent chapters of this thesis. On applying the above mentioned algorithm to the above canonical cover we get the following set of relations which are listed below:

- Peer (peer\_ID, GID\_key, peer\_address\_ID, peer\_instance\_ID, peer\_org\_ID)
- Peer\_address(peer\_address\_ID, peer\_ID, instance, type, domain, agent\_version)
- Peer\_instance(peer\_instance\_ID, peer\_ID, peer\_instance, branch, version, plugin\_info, last\_sync\_date)
- Peer\_org(peer\_org\_ID, peer\_ID, organization\_ID, served, last\_sync\_date)
Chapter 5

Space and Time - Based Approach to Database Design

In the previous section we discussed in detail the algorithm to decompose a relation in the 3\textsuperscript{rd} Normal Form. Also certain cases may arise in the above algorithm where the canonical cover used as input for the above algorithm may not be unique. In such cases we can obtain different decompositions, all of which are in the 3\textsuperscript{rd} Normal Form. All such versions represent lossless decompositions and a guarantee of the algorithm is that the functional dependencies in the decompositions are functionally preserving in nature. An ideal candidate database version will require the least amount of memory for it’s storage and would return results back from a given query in the least amount of time. Hence in the above scenario we need an evaluation standard to determine which of the candidate versions should be picked to represent the schema of the database.
system that a database designer might want to design. The candidate version that would be deemed as optimal would adhere to the above guidelines mentioned above. In this section we address the approach of improving Database Design techniques based on the amount of space the entire database would take and the time taken to execute a particular query on the version of a database that we have taken into consideration. Below we have listed a set of considerations taken while making the choice of version of a particular database. Database designers benefit by understanding the trade-offs that criteria listed below may raise while choosing the correct version of a database.

- Size of the database
- Time taken by a query to execute on a database
- Query optimization techniques enabled

5.1 Size of The Database

The input to the problem is a database that is populated with ample data. The amount of data in the database used in the experiment is representative of the final application domain. In order to conduct the experiment that is outlined we use the SQLITE database system. The above choice is purely based on the simplicity of the documentation and use of the database. In order to compute the size of a given database we use two methods. The first one is by reading the size of the database file
containing a given database, as the whole data that is present in SQLite database systems is stored in the above file. This is adopted in the above experiment to analyze the space requirements of a given database system. The main reason being that in file based database systems like SQLite, the entirety of the data is available on the same client and is placed in a single file. The size of the file on the client side is an exact estimation of the amount of memory that is consumed by the database system, including the history of the data manipulation. A noticeable drawback for using the above method is that it is hard to know how scalable a given database system is. It is hard to predict the behavior of a given database system when it is made to scale to a larger dataset than what is carried out during this experiment [17].

5.2 Time Taken by a Query on a Database

The time taken for a query to complete execution on a given database system depends on the ease with which the item that is being queried is accessible. When the data to be queried is present in one table with applicable indexes then the execution can be fast but when the data is distributed across various tables then join operations must be performed on the tables where the data resides. In the worst case the size of the resultant set from a given join operation is the product of the number of rows in all the tables that are being considered in the join operation. When such a join operation is performed any missed opportunity to filter the result set or any other inefficiencies in the query could lead to an operation that does not complete in a time
period that is considered to be practical in nature.

For the purposes of our experiment we select a set of queries with an amount of join operations that expect running times that allow a database designer enough insight to differentiate between the various decomposition versions of a database in the 3\textsuperscript{rd} Normal Form.
Chapter 6

Experiments and Results

6.1 Introduction

In this chapter we discuss the abnormalities measured when the same query is tested on different versions of the same database, as resulting from different decomposition processes. A list of experiments is designed to help determine the best version of a database that should be chosen based on the trade-offs of memory and time taken for the given query.

6.2 Experimental Setup

We describe in-depth the details of the experiment that was conducted to prove above hypothesis in 1.2. In order to carry out the above experiment we used a 2.2 gigahertz Intel Core i7 Process which has a 16GB Ram and 250 gigabyte memory.
The databases that were used in the experiment were constructed using the SQLite database management system. Other client-server SQL databases are created with the main intention of implementing a shared repository of enterprise data. In order to achieve the above goal they emphasize scalability, concurrency, centralization and control. However SQLite strives to provide local data storage for individual applications and devices. Hence SQLite emphasizes on economy, performance, reliability. However the trade-off of using SQLite is that it cannot hold data that grows far beyond the size of the single disk. Additionally if we have to consider the databases to be shared across various clients it is more advisable to choose some other database system as SQLite only supports one writer at a time per database file. But in most cases, a write transaction only takes milliseconds and so multiple writers can simply take turns. SQLite will handle more write concurrency than many people suspect [14]. Nevertheless, when a system architect considers to build a database system for a client-server architecture which deals with a long running server process that coordinate access among the clients, SQLite is not the first choice of a database system. Also it should be noted that SQLite can only support databases up to 140 terabytes in size. For the purposes of this thesis since we are not going to be dealing with a client-service architecture and the size of data that we are going to be experimenting on is small enough to fit on a single hard disk, we can safely say that SQLite is a good choice of a database system as it requires no configuration or maintenance. Thus it can be used swiftly to provide experimental data that proves
our given hypothesis.

6.3 Experiment

The reported experiment is based on the database schema of DebateDecide [19] introduced in Chapter 3. A series of queries has been applied to the three candidate versions of the database. Three versions based on distinct 3NF decompositions are introduced and analyzed below. The individual schemas of each table can be listed below:
6.3.1 Version-1

The Entity Relationship Diagram of the schemas in Version-1 is illustrated below:

**Description**  Version-1 represents a version of the database that does not violate the constraints imposed by the 3\(^{rd}\) Normal Form. It is obtained by applying the 3\(^{rd}\) Normal Form Synthesis Algorithm to the set of functional dependencies that form the canonical cover, as introduced in Chapter 3:

**Relevant set of Functional Dependencies**

The functional dependencies that are applicable to the above version can be listed below: peer\_ID, GID\_key $\rightarrow$ peer\_address\_ID, peer\_instance\_ID, peer\_org\_ID
Proof of Canonical Cover

The canonical cover of the functional dependencies are a minimal set of functional dependencies that does not have any extraneous attributes or redundant functional dependencies in the set of functional dependencies described above. Therefore in order to prove a given set is a minimal cover we need above properties which can be proved as follows

1. **Extraneous Attributes**: An attribute is known to be extraneous if we can remove it from the set of functional dependencies and not have any change in the closure of the attributes of the functional dependency. Thus on computing the closure of each attribute in the functional dependency we have:

   - \( peer\_ID^+ = \{peer\_ID\} \)
   - \( GID\_key^+ = \{GID\_key\} \)
   - \( peer\_address\_ID^+ = \{peer\_address\_ID, peer\_ID, instance, type, \)
     \( domain, agent\_version\} \)
   - \( peer\_instance\_ID^+ = \{peer\_instance\_ID, peer\_ID, peer\_instance, \)
     \( branch, version, plugin\_info, last\_sync\_date\} \)
• peer.org.ID\(^+\) = \{peer.org.ID, peer.ID, organisation.ID, served, last_sync.date\}

• Attributes on the right hand side of the functional dependency such as instance, type, domain, agent.version, peer.instance, branch, version, plugin.info, last_sync.date, organization.ID, served, last_sync.date are not extraneous in nature as they cannot be inferred from the above functional dependencies,

None of the attributes in the functional dependency are extraneous in nature.

2. **Redundant Functional dependencies**: A functional dependency is known to redundant in nature if can be derived from the other functional dependencies in the given set. Thus one inspection of the set of functional dependencies that are applicable to version 1 it can be clearly seen and trivially proved that none of the functional dependencies are redundant in nature.

**Relation Set formed on Application of 3\(^{rd}\) Synthesis Algorithm**

When we apply the 3rd Normal Form synthesis algorithm that is described in the algorithm in chapter 3 we get the set of relations that are a guaranteed to be the 3\(^{rd}\) Normal Form. The relation set is as follows:

• Peer (peer.ID, GID_key, peer_address.ID, peer_instance.ID, peer.org.ID)

• Peer_address (peer_address_ID, peer.ID, instance, type, domain, agent_version)
• Peer_instance(peer_instance_ID, peer_ID, peer_instance, branch, version, plugin_info, last_sync_date)

• Peer_org(peer_org_ID, peer_ID, organization_ID, served, last_sync_date)

Proof of 3\textsuperscript{rd} Normal Form

A relation is said to be in the 3\textsuperscript{rd} Normal Form if for every non-trivial Functional dependency X → A, where X is a superkey or A is part of some key for R. Thus If we wish to prove that the relations that were generated by the 3\textsuperscript{rd} synthesis algorithm in the previous subsection adheres the condition set by the 3\textsuperscript{rd} Normal Form we can do so by making the following observation

1. The candidate keys in the relations peer, peer_address peer_instance and peer_org are peer_ID, peer_address_ID, peer_instance_ID, and peer_org_ID respectively.

2. The attributes in the left hand side of each functional dependency is entirely composed of the candidate key of each table.

Conclusion: The above relations are adherent to the conditions laid out by the 3\textsuperscript{rd} Normal form.
6.3.2 Version-2

The Entity Relationship Diagram of the schemas in Version-2 is illustrated below:

Description

Version-2 represents a version of the database that does not violate the constraints imposed by the 3\(^{rd}\) Normal Form. It is sufficiently normalized to ensure an optimal time taken is taken to execute any query on the database. Version-2 is obtained from Version-1 with the following modification:

1. A join operation is performed between tables `peer.address` and `peer`.

The `peer` table in Version-2 contains all the attributes that existed in the `peer.address` and `peer` table in Version-1.
This modification is theoretically expected to have a lesser query execution time when compared to the version-1. This is due to lesser number of join operations that is needed to fetch the data from such a database system.

**Relevant Functional Dependencies**

The functional dependencies that are relevant to the above version can be listed below:

- peer_ID, GID_Key → instance, type, domain, agent_version, tcp_port, udp_port, address, certified, priority, my_last_connection

- peer_instance_ID → peer_ID, peer_instance, branch, version, plugin_info, last_sync_date, last_reset, last_contact_date, objects_synchronized, signature_date, signature, created_locally.

- peer_org_ID → peer_ID, organization_ID, served, last_sync_date

**Proof of Canonical Cover**

The canonical cover of the functional dependencies are a minimal set of functional dependencies that does not have any extraneous attributes or redundant functional dependencies in the set of functional dependencies described above. Therefore in order to prove a given set is a minimal cover we need to prove the above properties which can be proved as follows

1. **Extraneous Attributes**: An attribute is known to be extraneous if we can remove it from the set of functional dependencies and not have any change in
the closure of the attributes of the functional dependency. Thus on computing the closure of each attribute in the functional dependency we have:

- \( peer\_ID^+ = \{ peer\_ID \} \)

- \( GID\_key^+ = \{ GID\_key \} \)

- \( peer\_instance\_ID^+ = \{ peer\_instance\_ID, peer\_ID, peer\_instance, branch, version, plugin\_info, last\_sync\_date \} \)

- \( peer\_org\_ID^+ = \{ peer\_org\_ID, peer\_ID, organisation\_ID, served, last\_sync\_date \} \)

attributes in the right hand side of the functional dependency such as: instance, type, domain, agent_version, tcp_port, udp_port, address, certified, priority, my_last_connection, peer_instance, branch, version, plugin_info, last_sync_date, last_reset, last_contact_date, objects_synchronized, signature_date, signature, created_locally, organisation_ID, served, last_sync_date are not extraneous in nature as they cannot be inferred any of the functional dependency in the set of functional dependencies.

None of the attributes on the left hand side in the functional dependency are extraneous in nature.

2. **Redundant Functional dependencies**: A functional dependency is known
to redundant in nature if can be derived from the other functional dependencies in the given set. Thus we proved that none of the above functional dependencies are redundant in nature.

**Relation Set formed on Application of 3\textsuperscript{rd} Synthesis Algorithm**

When we apply the 3\textsuperscript{rd} Normal Form synthesis algorithm that is described in 1 we get the set of relations that are a guaranteed to be the 3\textsuperscript{rd} Normal Form. The relation set is as follows:

- Peer(\texttt{peer\_ID}, \texttt{GID\_Key}, \texttt{instance}, \texttt{type}, \texttt{domain}, \texttt{agent\_version}, \texttt{tcp\_port}, \texttt{udp\_port}, \texttt{address}, \texttt{certified}, \texttt{priority}, \texttt{my\_last\_connection})

- peer\_instance(\texttt{peer\_instance\_ID} \texttt{peer\_ID}, \texttt{peer\_instance}, \texttt{branch}, \texttt{version}, \texttt{plugin\_info}, \texttt{last\_sync\_date}, \texttt{last\_reset}, \texttt{last\_contact\_date}, \texttt{objects\_synchronized}, \texttt{signature\_date}, \texttt{signature}, \texttt{created\_locally})

- peer\_org(\texttt{peer\_org\_ID} \texttt{peer\_ID}, \texttt{organization\_ID}, \texttt{served}, \texttt{last\_sync\_date})

**Proof of 3\textsuperscript{rd} Normal Form**

A relation is said to be the 3\textsuperscript{rd} Normal Form if for every non-trivial Functional dependency X \(\rightarrow\) A, where X is a superkey or A is part of some key for R. Thus if we wish to prove that the relations that were generated by the 3\textsuperscript{rd} synthesis algorithm in the previous subsection adheres the condition set by the 3\textsuperscript{rd} Normal Form we can do so by making the following observation
1. The candidate keys in the relations peer, peer_instance and peer_org are peer_ID, GID_Key, peer_instance_ID, and peer_org_ID respectively.

2. The attributes in the left hand side of each functional dependency is entirely composed of the candidate key of each table.

**Conclusion:** The above relations are adherent to the conditions laid out by the 3rd Normal form.
6.3.3 Version-3

The Entity Relationship Diagram of the schemas in Version-3 is illustrated below:

Description

The attributes in the original tables of Version-1 are now divided into the various tables of Version-3 which implies that it would take a longer time for it to perform join operations and execute queries on it. It would also ideally consume the largest amount of space compared to all the other versions. Version-3 is obtained from
Version-2 and Version-1 with the following modifications:

1. Starting from Version-2 tables the **prime** and **non prime** attributes are separated into two individual tables. An attribute is said to be **prime** if it forms a candidate key of a relation, conversely all the other attributes that are not part of a candidate key are called non prime attributes. Thus by applying the above change we get the following new tables

   - **peer_org_non_key** where **peer_org_ID** and **organization ID** forms the set of candidate keys.
   - **peer_org_key** where **peer_or_ID** forms the primary key.
   - **peer_address_non_key** where **peer_address ID** forms the primary key.
   - **peer_address_key** where **peer_address ID** forms the primary key.
   - **peer_instance_non_key** where **peer_instance ID** and **peer_instance** forms the set of candidate key.
   - **peer_instance_key** where **peer_address ID** and **global peer ID** forms the set of candidate keys.

**Relevant Functional Dependencies**

The functional dependencies that are applicable to version-3 can be listed below:

- **peer_ID, GID_key → peer_address_ID**
- **peer_ID, GID_key → peer_instance_ID**, 

47
• peer.ID, GID_key → peer.org.ID

• peer.address.ID → peer.ID

• peer.address.ID → instance, type, domain, agent.version, tcp_port, udp_port, address, certified, priority, my_last_connection, arrival_date

• peer.instance.ID → peer.ID

• peer.instance.ID → peer.instance, branch, version, plugin_info, last_sync_date, last_reset, last_contact_date, objects_synchronized, signature_date, signature(created locally).

• peer.org.ID → peer.ID

• peer.org.ID → organization.ID, served, last_sync_date

Proof of Canonical Cover

The canonical cover of the functional dependencies are a minimal set of functional dependencies that does not have any extraneous attributes or redundant functional dependencies in the set of functional dependencies described above. Therefore in order to prove a given set is a minimal cover we need to prove the above properties which can be proved as follows

1. Extraneous Attributes: An attribute is known to be extraneous if we can remove it from the set of functional dependencies and not have any change in
the closure of the attributes of the functional dependency. Thus on computing
the closure of each attribute in the functional dependency we have:

- **peer_ID**\(^+\) = \{peer_ID\}
- **GID_key**\(^+\) = \{GID_key\}
- **peer_address_ID**\(^+\) = \{peer_address_ID, peer_ID, instance, type, domain, agent_version\}
- **peer_instance_ID**\(^+\) = \{peer_instance_ID, peer_ID, peer_instance, branch, version, plugin_info, last_sync_date\}
- **peer_org_ID**\(^+\) = \{peer_org_ID, peer_ID, organisation_ID, served, last_sync_date\}

attributes in the right hand side of the functional dependency such as
instance, type, domain, agent_version, tcp_port, udp_port, address, certified,
priority, my_last_connection, peer_instance, branch, version, plugin_info
last_sync_date, last_reset, last_contact_date, objects_synchronized
, signature_date, signature, created_locally, organization_ID, served, last_sync_date
are not extraneous in nature as they cannot be inferred any of the func-
tional dependency in the set of functional dependencies.

None of the attributes in the functional dependency are extraneous in nature.

2. **Redundant Functional Dependencies**: Thus on inspection of the set of
functional dependencies that are applicable to Version 3 none of the functional
dependencies are redundant in nature.

**Relation Set formed on Application of 3\textsuperscript{rd} Synthesis Algorithm**

When we apply the 3\textsuperscript{rd} Normal Form synthesis algorithm that is described in 1 we get the set of relations that are a guaranteed to be the 3\textsuperscript{rd} Normal Form. The relation set is as follows:

- \texttt{peer\_key(\texttt{peer\_ID, GID\_key,peer\_address\_ID})}
- \texttt{peer\_non\_key\_instance(\texttt{peer\_ID, GID\_key,peer\_instance\_ID})}
- \texttt{peer\_non\_key\_org(\texttt{peer\_ID, GID\_key, peer\_org\_ID})}
- \texttt{peer\_address\_key (\texttt{peer\_address\_ID, peer\_ID})}
- \texttt{peer\_address\_non\_key(\texttt{peer\_address\_ID, instance, type, domain, agent\_version, tcp\_port, udp\_port, address, certified, priority, my\_last\_connection,arrival\_date})}
- \texttt{peer\_instance\_key(\texttt{peer\_instance\_ID , peer\_ID})}
- \texttt{peer\_instance\_non\_key (\texttt{peer\_instance\_ID, peer\_instance, branch, version, plugin\_info, last\_sync\_date,last\_reset, last\_contact\_date, objects\_synchronized,signature\_date, signature,created\_locally}).}
- \texttt{peer\_org\_key(\texttt{peer\_org\_ID , peer\_ID})}
- \texttt{peer\_org\_non\_key(\texttt{peer\_org\_ID, organization\_ID, served, last\_sync\_date})}
Proof of $3^{rd}$ Normal Form

A relation is said to be the $3^{rd}$ Normal Form if for every non-trivial Functional dependency $X \rightarrow A$, where $X$ is a superkey or $A$ is part of some key for $R$. Thus if we wish to prove that the relations that were generated by the $3^{rd}$ synthesis algorithm in the previous subsection adheres to the condition set by the $3^{rd}$ Normal Form, we can do so by making the following observation

1. The candidate keys in the relations peer_key and peer_non_key are attributes in the set peer_ID, GID_key.

2. The candidate keys in the tables peer_address_key and peer_address_non_key is peer_address_ID.

3. The candidate keys in the tables peer_instance_key and peer_instance_non_key is peer_instance_ID.

4. The candidate keys in the table peer_org_key and peer_org_non_key is peer_org_ID.

5. The attributes in the left hand side of each functional dependency is entirely composed of the candidate key of each table. Also since the above relations are derived from the Cannonical Cover, it is proved to be in third normal form.
6.4 Queries

The queries used to conduct the following experiments are listed below.

1. \textbf{SELECT} peer\_address.peer\_ID, 
   peer\_address.address, 
   peer\_address.branch, peer\_instance.signature 
\textbf{FROM} peer\_instance, peer\_address 
\textbf{WHERE INNER JOIN} peer\_instance 
on peer\_address.peer\_ID == peer\_instance.peer\_ID

2. \textbf{SELECT} peer.peer\_ID, peer.address, peer.branch, peer\_instance.signature 
\textbf{FROM} peer, peer\_instance 
\textbf{WHERE} peer.peer\_instance\_ID == peer\_instance.peer\_instance\_ID

3. \textbf{SELECT} peer\_address.peer\_ID, peer\_address.address, 
   peer\_address.branch, peer\_instance.signature 
\textbf{FROM} peer\_address\_non\_key , peer\_address\_key, peer\_instance\_non\_key, 
   peer\_address 
\textbf{WHERE} peer\_address.peer\_address\_ID == peer\_address\_non\_key.peer\_address\_ID 
\textbf{AND} peer\_address\_non\_key.peer\_address\_ID== peer\_address\_key.peer\_address\_ID 
\textbf{AND} peer\_org.peer\_ID== peer\_instance.peer\_ID
AND peer_instance.peer_instance_ID == peer_instance_non_key.peer_instance_ID
AND peer_instance_non_key.peer_instance_ID == peer_instance_key.peer_instance_ID

6.4.1 Discussion

In each of the above queries we try to fetch the attributes peer_ID of each peer, address of each peer, branch of each peer and, the signature of each peer. The queries were selected to fetch the above attributes from each of the versions. The queries were selected in such a manner that will enable us to study the difference in query execution time in each of the versions of the database.

1. In Query-1 we fetch the attributes from the respective tables by performing a INNER JOIN with the peer_address and peer_instance table.

2. In Query-2 we fetch the attributes from the respective tables by performing a NATURAL JOIN with the respective primary keys of peer and peer_instance namely peer_instance_ID of each table.

This version of the query is expected to execute faster of Version 2 than query-1. This is mainly due to the fact that version-2 is got by a join operation between the tables peer_address and peer. This join operation causes all the attributes in peer_address and peer to be in the same table namely peer. This property that is persistent in the above version allows this query to be faster compared to version-1.
3. In Query-3 we fetch the attributes from the respective tables by performing a

**NATURAL JOIN** with the respective primary keys of the following tables.

- **peer.address** where the primary key is peer.address.ID.
- **peer.address.non_key** where the primary key is peer.address.ID.
- **peer.address.key** where the primary key is peer.address.ID.
- **peer.org** where the primary key is peer.ID.
- **peer.instance** where the primary key is peer.ID.
- **peer.instance.non_key** where the primary key is peer.instance.ID.
- **peer.instance.key** where the primary key is peer.instance.ID.

Version-3 is developed from Version-2 by separating the prime and non-prime attributes. This separation of attributes in the various tables increases the tables that is developed in this version. This in turn increases the number of join operations that is needed to query the results in the Query-3. Thus Query-3 is expected to have the highest execution time among all the queries.

### 6.5 Results

The queries in the previous section are executed on the data from Section 3. The data and queries were selected in such a manner which would produce a pronounced difference in execution times of the queries when it is executed the different versions.
of the database. The execution time of a query in the SQLite database comprises three different execution times: User, Sys, and Real-Time.

- **User**: This is the time used by the central processing unit to execute a given query in the user space which in this case is the database itself.

- **Sys**: This is the time taken to execute the code at the kernel level.

- **Real-Time**: This is the actual time that is needed by the processor to perform the relevant join operations and fetch the data that is essential to the query.

Hence using the above metrics we can execute the queries on the database and the results of which are summarized below.

### 6.5.1 Examination of Execution Time of Queries

On running the queries we obtain results that can be summarized in the table below:

<table>
<thead>
<tr>
<th>Version</th>
<th>sys</th>
<th>Real-Time</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version-1</td>
<td>0.128872</td>
<td>0.468</td>
<td>0.080593</td>
</tr>
<tr>
<td>Version-2</td>
<td>0.032322</td>
<td>0.122</td>
<td>0.055922</td>
</tr>
<tr>
<td>Version-3</td>
<td>0.032663</td>
<td>0.126</td>
<td>0.058529</td>
</tr>
</tbody>
</table>

Figure 6.1: Summary of Execution Time for given Query
Figure 6.2: Real Execution Time of Query
Figure 6.3: SYS Execution Time
Figure 6.4: User Execution Time
6.5.2 Examination of Memory Consumed by Databases

We now measure the memory consumed by each version of a given database which will be used to draw inferences about each version of the database.
6.6 Analysis

From the results obtained from the experiments, we see that with respect to the time consumed by the CPU to execute a given query, Version-2 and Version-3 take the least amount of time but when we consider the memory consumed by each of the databases we see that Version-3 takes far more memory to store the entire database than Version-2. Thus using the framework that we developed throughout this thesis one should choose Version-2 as a appropriate version of the database system as it takes less amount of time to execute a given query and consumes the least amount of space amongst the other versions of the database.

The above notion can be generalized for all decompositions that are received by the 3rd Normal form Synthesis Algorithm. The result of the above algorithm is not uniquely defined since a given set of functional dependencies can have more than one canonical cover (set of minimal functional dependencies). Furthermore in some cases the result of the algorithm depends on the manner in which functional dependencies are considered.

This performance metric can therefore be used in general to analyze all decompositions that are formed by the 3rd normal form synthesis algorithm mentioned in Algorithm 4. Furthermore based on the comparison of the memory requirements of each decomposition and the time taken to execute a given query we can differentiate the trade-offs offered by each version. Thus helping us to pick the optimal version to represent the attributes in the database.
Chapter 7

Applications

Normalization is a fundamental idea in any database design. Thus such a canonical cover with multiple decompositions can be applied to numerous areas such as Medicine, Banking, Health-care, and corporate entities. Thus it is essential for any database designer to be aware of the trade off a particular design may have.

- University Database System
- Medical Database System
- Library Database System
- Banking Database System
7.1 University Database

The above schema used in the experiment can be applied to a University Database System. When we look into the data requirements listed below we can find that one can develop a database system using the above mechanisms.

7.1.1 Data Requirements and Constraints

- Each Student at the university is apart of a specific department
- Each Student has a single instructor as an advisor at the university
- Each Instructor has many courses that he is supposed to teach
- Each Student can attend multiple courses at the university

7.1.2 Entities

- Student
- Instructor
- Department
- Courses
7.1.3 Entity Relationship Diagram

![Entity Relationship Diagram]

7.1.4 Relational Schema

```
Student
- Student_ID
- Name
- Department_Name
- Department_ID
- Advisor_ID
- Course_ID

Department
- Department_ID
- Department_Name
- Building_Name

Instructor
- Instructor_ID
- Department_Name
- Salary

Courses
- Course_ID
- Dept_ID
- Instructor_ID
- Title
- Credit
```

63
7.1.5 Functional Dependencies

StudentID, DepartmentName, DeptID → DeptID, advisorID, CourseID

DeptID → DepartmentName, DeptID, buildingName

advisorID → DepartmentName, DeptID, salary

CourseID → DepartmentName, DeptID, Credit, title

7.1.6 Alternate Decompositions

As discussed in the previous Chapters, decomposing a table on the 3rd Normal Form can lead to multiple versions, depending on how the functional dependencies are taken. The alternate versions for the above University Database System can be listed below:

Version-1

The schemas for the above version can be listed as follows:

Student(StudentID, DepartmentName, DeptID, AdvisorID, CourseID)

Department(DeptID, DepartmentName, BuildingName)

Advisor(AdvisorID, DepartmentName, DeptID, salary)

Course(CourseID, DepartmentName, DeptID, credit, title)
Version-2

The schemas can be listed as follows:

Student(StudentID, DepartmentName, DeptID, AdvisorID, buildingName)

DepartmentInfo(DeptID, DepartmentName, salary)

Advisor(AdvisorID, DepartmentName, DeptID, credit, title)

Version-3

The schemas for the above version can be listed as follows:

Student(StudentID, DepartmentName, DeptID, courseID)

StudentKey(StudentID, DepartmentName, DeptID,)

StudentNonKey(StudentID, DepartmentName, DeptID, AdvisorID)

CourseKey(CourseID, DeptID, DepartmentName)

CourseNonKey(CourseID, Salary)

DeptNonKey(DeptID, Salary)

DeptKey(DeptID, DepartmentName)

AdvisorKey(AdvisorID, DeptID, DepartmentName)

AdvisorNonKey(AdvisorID, credit, title)
7.1.7 Normal Form

It can be easily verified that the schema obtained by the above tables satisfies the prevalent conditions of the 3\textsuperscript{rd} Normal Form. This can be done by considering every non trivial functional dependencies whose right hand side is is a single attribute eg X \rightarrow A and checking whether X is a superkey. (This can be done by computing the attribute closure of the attribute. If the attribute closure contains all the attributes of relation). If the above condition fails then we check the right hand side of the functional dependency and verify whether or not it is part of the key of the relation. Both of these conditions hold true for the above set of functional dependency. The above set of relations are in the 3\textsuperscript{rd} Normal form.
7.2 Medical Database System

Using the set of functional dependencies as a substructure we can build a Database System that can be used for the medical domain. We assume the following data constraints that are followed by the system.

7.2.1 Data Requirements and Constraints

- Each Patient has a Doctor that he consults in event of a disease
- Each Patient can have multiple disease
- Each Doctor may prescribe multiple diseases for a given patient
- Each Disease may have more than one medication prescribed as a cure.

7.2.2 Entities

- Patient
- Disease
- Doctor
- Medication
7.2.3 Entity- Relationship Diagram
7.2.4 Relational Schema

7.2.5 Functional Dependencies

PatientID, HospitalName → DiseaseID, DoctorID, MedID

DiseaseID → HospitalName, Symptoms

DoctorID → DoctorName, HospitalName, speciality, Salary

MedID → Dosage, cost, HospitalName

7.2.6 Alternate Decompositions

As discussed in the previous Chapters, decomposing a table on the 3rd Normal Form can lead to multiple versions, depending on how the functional dependencies are taken. The alternate versions for the above Medical Database System can be listed below:
Version-1

The schemas for version-1 can be shown as follows:

Patient(\text{PatientID}, \text{HospitalName}, \text{DiseaseID}, \text{DoctorID}, \text{MedID})

Medication(\text{MedID}, \text{Dosage}, \text{Cost}, \text{HospitalName})

Disease(\text{DiseaseID}, \text{PatientID}, \text{HospitalName})

Doctor(\text{DoctorID}, \text{HospitalName}, \text{DoctorName}, \text{Speciality}, \text{Salary})

Version-2

The schemas for version-2 can be shown as follows:

Patient (\text{PatientID}, \text{HospitalName}, \text{DiseaseID}, \text{DoctorID}, \text{Dosage}, \text{Cost})

Disease(\text{DiseaseID}, \text{HospitalName}, \text{Symptoms})

Doctor(\text{DoctorID}, \text{HospitalName}, \text{DoctorName}, \text{speciality}, \text{Salary})

Version-3

The Schemas for version-3 can be shown as follows:

PatientKey(\text{PatientID}, \text{HospitalName}, \text{MedID})

PatientNonKey(\text{PatientID}, \text{HospitalName}, \text{DiseaseID})

DoctorKey(\text{PatientID}, \text{HospitalName}, \text{DoctorID})

DoctorNonKey(\text{DoctorID}, \text{DoctorName}, \text{speciality}, \text{Salary})

MedicationKey(\text{MedID}, \text{HospitalName})
MedicationNonKey(\text{MedID}, \text{Symptoms})

DiseaseNonKey(\text{DiseaseID}, \text{Symptoms})

DiseaseKey(\text{DiseaseID}, \text{HospitalName})

\subsection*{7.2.7 Normal Form}

It can be easily verified that the schema obtained by the above tables satisfies the prevalent conditions of the 3\textsuperscript{rd} Normal Form. This can easily be done by considering every non trivial functional dependencies whose right hand side is is a single attribute eg \(X \rightarrow A\) and checking whether \(X\) is a superkey. (This can be easily done by computing the attribute closure of the attribute. If the attribute closure contains all the attributes of relation). If the above condition fails then we check the right hand side of the functional dependency and verify whether or not it is part of the key of the relation. It is evident that both of these conditions hold true for the above set of functional dependency Thus it is trivial to prove that the above set of relations are in the 3\textsuperscript{rd} Normal form.
7.3 Library Database System

Using the functional dependencies used in this thesis we can build a database system that can be used to store the information of the books that are being loaned to students in a library. In order to build the above database system we assume the following data constraints

7.3.1 Data Constraints and Requirement

- Each Member of the library can borrow as many books as he wants
- Each book that is present is the library has one Author
- Each Author has one publisher that publishes the book for him

Entities

- Book
- Member
- Author
- Publisher
7.3.2 Entity Relationship Diagram

7.3.3 Database Schema
7.3.4 Functional Dependencies

BookID, libraryName → MemberID, AuthorID, publisherID

MemberID → MemberType, contactNo, address, joiningDate, libraryName

AuthorID → AuthorName, address, joiningDate, libraryName

publisherID → publisherName, address, email, libraryName

7.3.5 Alternate Decompositions

As discussed in the previous Chapters, decomposing a table on the 3\textsuperscript{rd} Normal Form can lead to multiple versions, depending on how the functional dependencies are taken. The alternate versions for the above Library Database System can be listed below:

Version-1

The schemas for version-1 can be shown as follows:

Book(\textbf{BookID}, libraryName, MemberID, AuthorID, publisherID)

Publisher(\textbf{PublisherID}, libraryName,publisherName, address,email)

Member(\textbf{MemberID},libraryName,memberType,ContactNo,address,joiningDate)

Author(\textbf{AuthorID},libraryName,AuthorName,address,joiningDate)
**Version-2**

The schemas for version-2 can be shown as follows:

Book(\texttt{BookID},libraryName, MemberID,AuthorID,publisherName,address,email)

Member(\texttt{MemberID},libraryName,memberType,ContactNo,address,joiningDate)

Author(\texttt{AuthorID},libraryName,AuthorName,address,joiningDate)

**Version-3**

The schemas for version-3 can be shown as follows:

BookPublisher(\texttt{BookID},libraryName, publisherID)

BookMember(\texttt{BookID},libraryName, MemberID)

BookMember(\texttt{BookID},libraryName, AuthorID)

PublisherKey(\texttt{PublisherID}, libraryName)

PublisherNonKey(\texttt{PublisherID}, memberType,ContactNo,address,joiningDate)

MemberKey(\texttt{MemberID}, libraryName)

MemberNonKey(\texttt{MemberID}, memberType, ContactNo, address, joiningDate)

AuthorKey(\texttt{AuthorID}, LibraryName)

AuthorNonKey(\texttt{AuthorID}, AuthorName,address,joiningDate)
7.3.6 Normal Form

It can be verified that the schema obtained by the above tables satisfies the prevalent conditions of the $3^{rd}$ Normal Form. This can be done by considering every non-trivial functional dependencies whose right hand side is a single attribute e.g. $X \rightarrow A$ and checking whether $X$ is a superkey. (This can be done by computing the attribute closure of the attribute. If the attribute closure contains all the attributes of relation). If the above condition fails then we check the right hand side of the functional dependency and verify whether or not it is part of the key of the relation. Both of these conditions hold true for the above set of functional dependency. The above set of relations are in the $3^{rd}$ Normal form.
7.4 Banking Database System

Using the functional dependencies used in this thesis we can build a database system that can be used to store the information of a bank. In order to build the above database system we assume the following data constraints.

7.4.1 Data Constraints and Requirements

- Each Customer has a account at a designated Branch

- Each Customer can have many saving Accounts at the designated branch

- Each Customer can have many loan Accounts at the designated Branch

7.4.2 Entities

- Customer

- Saving Account

- Loan Account

- Branch
7.4.3 Entity Relationship Diagram
### 7.4.4 Database Schema

![Database Schema Diagram]

### 7.4.5 Functional Dependencies

CustomerID, CustomerName, BranchName $\rightarrow$ AccountNumber, LoanNumber, BranchNumber

AccountNumber $\rightarrow$ BranchName, balance, type

LoanNumber $\rightarrow$ BranchName, Amount, type

BranchNumber $\rightarrow$ BranchName, address, ContactInfo
7.4.6 Alternate Decompositions

As discussed in the previous Chapters, decomposing a table on the 3\textsuperscript{rd} Normal Form can lead to multiple versions, depending on how the functional dependencies are taken. The alternate versions for the above Library Database System can be listed below:

**Version-1**

The schemas for version-1 can be shown as follows:

Customer(\textbf{CustomerID}, CustomerName, BranchName, AccountNumber, LoanNumber, BranchNumber)

SavingAccount(\textbf{AccountNumber}, BranchName, balance, type)

LoanAccount(\textbf{LoanNumber}, BranchName, amount, type)

Branch(\textbf{BranchNumber}, BranchName address, ContactInfo)

**Version-2**

The schemas for version-2 can be shown as follows:

Customer(\textbf{CustomerId}, CustomerName, BranchName, AccountNumber, LoanNumber, address, ContactInfo)

Account(\textbf{AccountNumber}, BranchName, balance, type)

Branch(\textbf{BranchNumber}, BranchName, address, ContactInfo)
Version-3

The schemas for version-3 can be shown as follows:

CustomerBranchNumber(\textbf{CustomerID}, \textbf{CustomerName}, \textbf{BranchNumber})

CustomerAccountNumber(\textbf{CustomerID}, \textbf{CustomerName}, \textbf{AccountNumber})

CustomerLoanNumber(\textbf{CustomerID}, \textbf{CustomerName}, \textbf{LoanNumber})

BranchKey(\textbf{BranchNumber}, \textbf{BranchName})

BranchNonKey(\textbf{BranchNumber}, \textbf{amount}, \textbf{type})

AccountKey(\textbf{AccountNumber}, \textbf{BranchName})

AccountNonKey(\textbf{AccountNumber}, \textbf{balance}, \textbf{type})

BranchAddress(\textbf{BranchNumber}, \textbf{address}, \textbf{contactInfo})

7.4.7 Normal Form

It can be verified that the schema obtained by the above tables satisfies the prevalent conditions of the $3^{rd}$ Normal Form. This can be done by considering every non trivial functional dependencies whose right hand side is is a single attribute eg X $\rightarrow$ A and checking whether X is a superkey. (This can be done by computing the attribute closure of the attribute. If the attribute closure contains all the attributes of relation). If the above condition fails then we check the right hand side of the functional dependency and verify whether or not it is part of the key of the relation.

It is evident that both of these conditions hold true for the above set of functional
dependency. The above set of relations are in the 3rd Normal form.

\section*{7.5 Conclusion}

The examples analyzed in this chapter demonstrate that for real applications there may exist multiple decompositions for a relation in the 3rd normal form depending on how the functional dependencies are considered. There is a growing need to evaluate such decompositions based on space and time taken to execute a query on such databases. Based on such evaluations a database designer needs to consider the inherent trade-offs contained in such a system. The work presented above focuses on the evaluation of such schemas based on the physical memory consumed by the schema and the time taken by a query to join and return the information needed from the database.
Chapter 8

Contributions and Conclusions

In this thesis we have shown that since the canonical cover of a given database system is not unique and depends on how the functional dependencies are taken on the basis that we can arrive at different versions of a given database system with significant difference in performance. Thus there is a need for a framework to analyze and differentiate the various versions of a given database system depending on the amount of memory that is consumed by the database system and the time taken to execute a set of queries on a given database system. With the help of the above parameters and at the same time by taking into account the trade offs that are presented by either of them an appropriate version can be chosen from the different versions of the database.

This work focuses on evaluation of such a sample database system from the perspective of the memory consumed by it and the time taken for the database system to return the results of a set queries that were executed on it. Thus it is further
concluded due to the emergence of such cases of ambiguity arising from 3rd Normal Form decompositions, there exists a general need for a more concrete algorithm. If these ideas are taken into consideration properly it will help database designers to compute the decompositions and figure out the one decomposition that would better model the requirements that are needed.
Bibliography


