A Genetic Programming Approach To Normalise Databases

Mini Dissertation by

Donovan Casper van Wyk
9604075 2

submitted in partial fulfilment of the requirements for the degree

MAGISTER IN INFORMATION TECHNOLOGY

in the

SCHOOL OF INFORMATION TECHNOLOGY

of the

FACULTY OF ENGINEERING, BUILT ENVIRONMENT AND INFORMATION TECHNOLOGY

Supervisor: Prof A Engelbrecht

May 2003 - 3nd Draft

Acknowledgements

This dissertation would not have been possible without the endless encouragement and support from:

- My parents who took pride in my successes.
- Prof. Andries Engelbrecht, my supervisor, who guided me through this research, only after showing me the beauty of Artificial Intelligence in my preceeding degree.
- Terence Royeppen. Although we only worked together for a few months; he knew I was capable of more and he made sure to remind me of that constantly.
- Dimension Data South Africa for letting me take the time needed to complete this entire degree.

Contents

List of Figures

List of Tables

A Genetic Programming Approach To Normalise Databases

Almost every commercial application is running off a database somewhere, somehow. Fast, efficient and correct databases adhere to the laws of normalisation. What happens when a database designer, application designer or support engineer has to make alterations to some aspect of the database; particularly the tables that make up the database? Making a change and hoping for the best is one alternative, the other is to use a tool to correct database designs in place.

This dissertation takes the first step in the development of such a tool. Programmatically correcting database tables, knowing nothing about the original intentions of the design, is ultimately the goal of this dissertation. Procedural computer programs that normalise database tables have been developed with the assumption that some basic information about the database in question is available. An evolutionary computation approach to the problem will negate the requirement for meta-data about the database in question.

By creating a Genetic Program that normalises database relations it is shown that relations can be normalised, correctly, without prior knowledge. It is also shown that such a program is simple to implement, test and understand.

Chapter 1

Introduction

At the heart of every major corporation lies a database system: a collection of data and data manipulation programs stored on a server somewhere. Users at all levels constantly query database systems, making sure changes in the real world are reflected in the database.

Before committing to a database technology a corporate has to decide what the system will be used for. An investment in an operational database system is wasted if all they want is to analyse data. Similarly, an analytical database system is ill suited to real-time transaction processing. The choice affects every aspect of the database system design, implementation and usage.

An analytical database system is very different from an operational database system in both design and usage. Where analytical databases are used to store rarely queried historical data, operational databases are used to store and maintain constantly queried current data.

Good relational database design takes time, effort and knowledge. It takes time to capture high-level business requirements and translate them into low-level database tables. It takes some effort to extract rules the business wants to apply to its data, especially when the business develops its rules on the fly. The database designer has to have enough knowledge to know how to create meaningful tables and assign unique keys. Designing a meaningful table means, removing duplicate field names; removing cryptic field names; making sure that related tables have primary/foreign key pairs; not overfilling tables with meaningless fields; and most importantly making the database design simple and flexible enough for anybody in future to come in and make necessary adjustments. The resulting relational database should be normalised adequately, implemented in a robust database management system and well integrated into client applications.

1.1 Dissertation Focus

A relational database can start its life as a perfectly normalised entity obeying all the rules of good design and implementation. Over time, as business needs change, the database will also change. Fields will be added, removed or misused possibly ignoring the principles of normalisation. Fields added to the wrong tables will increase query complexity, thus increase query execution time. The haphazard removal of fields will not only destroy critical data it will also impact queries causing unsuspected errors in client application programs. Misuse of fields not only breaks the fundamental premise of databases, i.e. the right data in the right field, but also leads to user confusion and makes further changes to the database very difficult.

Customisation of a database is not only limited to the modification of fields; new tables could be required too. The entire database management system could be changed, either upgraded to a later version or down graded to a cheaper alternative. Data in the database has to remain trustworthy through out the change process. The introduction of a new table should not require data to have to be duplicated in both the new table and some obscure existing table (or field in a table) deep inside the original design. In the case of moving data from existing tables to new tables, care needs to be taken that records are not repeated in their new tables and the original records are removed. Similarly, when the database is migrated to a newer or different database management system, the migration process must ensure that records remain atomic.

Data in the database will begin to degrade if the rules of normalisation are not followed. Redundancy will be introduced, relations (tables) will no longer conform to the definition of a relation, and joins between tables will yield incorrect results. How can the corporate hope to survive and grow if its data looses value over time?

This dissertation presents a method for correcting a relational database regardless of what the initial state of the database was using the **data itself** to guide the "search" for an optimal design. Relational database design principles are adhered to in this "search" ensuring that the resulting database is adequately normalised for the data it holds.

The goals of this dissertation are to show that:

- Relational databases can be normalised programmatically.
- Genetic Programming can be used to normalise relational databases.
- A genetic strategy is simpler to implement than a procedural strategy.

This dissertation develops and tests a genetic programming approach to programmatically normalise

databases. Research scope has been limited to proving that a database table can be normalised up to the $2NF$ (Second Normal Form). Suggestions for $3NF$ (Third Normal Form) normalisation are given.

The research approach chosen was that of "Constructive Research" where technical development was undertaken to prove the hypothesis. Technical development meant writing a computer program and observing its outputs given various inputs. The computer program written in this case is an Evolutionary Algorithm. Genetic Algorithms, Genetic Programs, Evolutionary Programs and Evolutionary Strategies are all examples of Evolutionary Algorithms. Inputs to this Genetic Program take the form of database relations (including their data). Outputs take the form of graphs, chromosome trees and "Entity Relationship Diagrams".

1.2 Organisation of this Dissertation

The plan for this dissertation is as follows:

- Chapter 2 delves into the background of relational database design. Fundamental database principles are introduced. Along with an exploration into Genetic Programming, related work is presented.
- Chapter 3 explores the creation of a Genetic Program to automatically normalise relations.
- Chapter 4 presents the results of the resultant experimental program.
- Chapter 5 offers a conclusion and suggests future work.

Chapter 2

Background and Literature Study

Before delving into the depths of database normalisation with Genetic Programs, tools need to be mastered. In the case of this dissertation the reader needs to become familiar with the theory behind databases, particularly relational databases. The reader then needs to familiarise him/herself with the history and theory of Evolutionary Computation.

The purpose of this chapter is to introduce the tools required to use evolutionary computation principles to normalise relational databases. Starting off, a brief history of database management is given. Then a look at relational databases is taken after which normalisation theory is presented. Finally, a very brief discussion of pertinent concepts within Entity Relationship Modelling is given.

Artificial Intelligence is given due treatment. Followed by Evolutionary Computation and finally Genetic Programming. Before moving on to the next chapter, related work is presented in the area of programmatically normalising databases. All this serves as preparation for the journey ahead in the following chapter: "GDNA Algorithm Discovery" in which the "Genetic Database Normalisation Algorithm is developed and explored.

2.1 Database Management

2.1.1 History of Database management

Database management evolved from humble beginnings. Initially there was nothing. An application program would define, manage and use its own files. Application files could include any data in any format. Typically, files would be seen as a collection of records containing logically related data. Instantly a scenario develops where each individual application maintains its own file set. Sure, files across many applications may contain similar, if not the same (redundant), data. Yet each application would have its own file access routines. At best some applications may use the same library functions but for the most

part, data and program were one and the same.

Such a tight coupling between data and application is not practical. The biggest and most obvious concerns are cost and time related. How long will it take to modify a single file based application? How long will it take for several different file based applications, or data sets, to be modified? What happens when each application was custom written by different vendors? It seems like a "File System" is not going to be an adequate solution for a modern day corporate.

Possible solutions take the form of "Hierarchical" and "Network" systems. Here data are organised in tree like structures. An application program would navigate through the data hierarchy or network to find the records of interest. Unfortunately this navigation would be clearly visible in application source code. Although these systems are a great improvement over a flat file system, they are still lacking data independence. A user of both types of systems would have to know the structure of the file hierarchy or network if he/she wanted to do anything meaningful with the data. The major advantage of hierarchical/network systems over file systems is a reduction of data redundancy. Where file base applications maintained their own data, hierarchical/network based applications could all share the same data. Possibly using the same libraries thus becoming simpler and cheaper to maintain. Still, the tight application-data coupling is not ideal.

Wouldn't it be excellent if there was a total separation of application and data? The data should manage itself and applications should query "data management systems" for any data they wanted. That way an abstraction can be created. The programmer's world would end at the database management system (DBMS) interface. Applications would become simpler to develop and maintain. Removing all the data redundancy would also improve application and DBMS throughput. Development and maintenance could become so simple it could allow a corporate to maintain only a small pool of programmers for application/database maintenance once contractors had left.

Dr E.F. Codd may have been thinking along these lines when he developed the "Relational Model" [17]. He wanted to solve the problems with file and network systems: namely the lack of distinction between data and application. Using relational algebra, he developed a system proven on mathematical principles. Although developed in 1970, the first commercial applications made their appearance in the early 1980s. By the early 1990s most database systems sold were relational databases. Not to say that the relational model was shelved for a decade. Dr Codd and his associates at IBM developed the first relational database system dubbed "System R" [23] during that decade.

The relational model is not without its flaws. Relational database systems are heavily reliant on simple data types: integers, variable character strings, and binary data. Data has changed throughout the years: now there is audio data, video data, images, work flows and other non-simple data types.

Still, the common denominator is the *bit* and that is what is used, but the responsibility of converting binary data falls to application programs. Similarly, conceptual models need to be maintained within applications. The DBMS will know about relations and relationships and will enforce those but it still doesn't know what relations mean and how they should be used. This is the role of the conceptual model and unfortunately, it resides within application programs. Clearly there is no true distinction between data and application.

Database evolution did not stop at the relational model. The object-oriented paradigm [44] has also been applied to database theory resulting in Object-oriented data models. These models support complex mechanisms allowing the reuse of data and processes through inheritance, the building of complex objects and the identification of objects independently of their values [71].

The use of database systems has been extended towards data mining and data warehousing [53]; using their own interpretation of the relational model. Currently these are still niche applications of database systems - mostly due to investments made in relational technology. Still, they are gaining ground slowly.

2.1.2 Relational Databases

This dissertation is primarily focused on the relational model. Particularly one aspect of relational theory: "normalisation", the process used, by database designers, to eliminate data redundancy. Normalisation theory will now be explored since several key concepts feature later in this dissertation.

A relation schema consists of a set of relations where each relation is defined by its attributes and some semantic constraints. At the lowest level a relation is made up of attributes (columns) and tuples (rows), the number of which varies over time. Semantic constraints can be expressed as functional dependencies (FD)[18]. Given a relation R, a non-empty attribute set B is functionally dependant on a non-empty attribute set A (denoted $f : A \rightarrow B$) if, and only if, each value of A has associated with it precisely one value of B.

Consider the following "Employee" relation: Employee

In this example, each employee's number is associated with precisely one name ($EmployeeNum \rightarrow Name$), likewise $EmployeeNum \rightarrow IDNumber$, $EmployeeNum \rightarrow Job$, $EmployeeNum \rightarrow DNumber$ and $EmployeeNum \rightarrow DName$. $EmployeeNum$ is called the "key" of relation "Employee" because $EmployeeNum$ functionally determines all other (nonkey) attributes. Another dependency is $Job \rightarrow Name$, assuming that each job can only be done by one employee. This dependency causes some redundancy in the "Employee" relation (the same job can be done by several employees). It is also responsible for some database manipulation anomalies:

(**EmployeeNum**	**Name**,	**IDNumber**,	**Job**,	**DNumber**,	**DName**)
1	Jacob	12345789	Programmer	54	IT
2	Edwin	98765321	Programmer	54	IT
3	Gordon	45679123	Consultant	12	HR
4	Graham	89234567	Manager	12	HR
5	Henry	56789123	Analyst	54	IT

Table 2.1: Employee(EmployeeNum, Name, IDNumber, Job, DNumber, DName)

- How would users distinguish employees who share the same name, job and department if they do not have access to the $IDNumber$?
- It is not possible to create a job function without creating an employee to do that job.
- Several tuples in the relation are affected whenever a Job title changes.
- Removing the last employee doing a particular job will eliminate all traces that such a job existed.

The "key" ($EmployeeNum$) uniquely identifies tuples in "Employee". Normally one attribute (or combination of attributes) of a relation has values that uniquely identify each tuple. This attribute (or combination of attributes) is called the "primary key". If the domain of an attribute is simple (non-redundant) then the primary key is non-redundant.

A common requirement is for tuples of a relation to cross-reference other tuples of the same or different relations. Keys provide a user-oriented means of expressing such cross-references. This attribute (or combination of attributes) of relation R is called a "foreign key" if it is not the primary key of R but its elements are values of the primary key of some relation S (S and R could be the same relation) [17].

The "keys" concept is the basis for the definition of the "normal forms". Normalisation of a relation eliminates the types of database manipulation anomalies seen earlier. Initially Codd defined three normal forms [20] abbreviated $1NF$, $2NF$ and $3NF$. Later a stronger definition of $3NF$ was proposed by Boyce and Codd [21]. This normal form is known as the Boyce-Codd normal form ($BCNF$). These four normal forms are based on the correlation between the primary key of a relation and its attributes. The fourth normal form ($4NF$), based on multi-valued dependencies, and the fifth normal form ($5NF$), based on join dependencies, was later proposed. The five normal forms are defined as

- A relation is said to be in first normal form ($1NF$) if it has non-redundant tuples and attributes [17].

- For a relation to be in second normal form ($2NF$), it firstly has to be in $1NF$. Furthermore, all nonkey attributes are dependant on the entire key [18, 19].
- Relations in third normal form ($3NF$) are in $2NF$ and all transitive dependencies are eliminated. In other words, no attribute of a relation is dependant on any other attribute of that relation other than the primary key [18, 19].
- A relation is in Boyce-Codd Normal Form (BCNF) [21] if all its nontrivial dependencies has a left hand side that contains a key of relation R [15].
- Relations in fourth normal form ($4NF$) are so if they are firstly in $3NF$ and tuples should not contain two or more independent multi-valued facts about an entity [26].
- Fifth normal form ($5NF$) deals with cases where information can be reconstructed from smaller pieces of information that can be maintained with less redundancy [25].

2.1.3 Normalisation

Using the normalisation rules defined previously it is now possible to demonstrate the normalisation process. Since this dissertation is limited to $3NF$, the following examples, and rest of this dissertation will focus on normalisation up to $3NF$. Take the "Employee" relation presented in table 2.1. Clearly it is not normalised.

To put $Employee(\underline{EmployeeNum}, Name, IDNumber, Job, DNumber, DName)$ in $1NF$ we need to remove all repeating groups. Job, $DNumber$ and $DName$ are repeating groups and need to be removed. Removing any attribute to a foreign relation requires that a "foreign key" be placed in the affected relation, and a primary key is chosen/created in the foreign relation [17]. Doing so results in the following relations:

- $Employee(\underline{EmployeeNum}, Name, IDNumber, \textit{Job\_key}, \textit{DNumber})$
- $Job(\underline{Job\_key}, JobTitle)$
- $Department(\underline{DNumber}, DName)$

To test whether or not these new relations are in $2NF$ we evaluate each relation for its $1NF$ status. Then we would test that each attribute is functionally dependant on the primary key. The primary key is made up of one attribute only, therefore all nonkey attributes are functionally dependent only on that key attribute. These relations are all in $2NF$ already.

Are these relations in $3NF$? Yes. For each relation we test for $2NF$ then test each attribute to ensure that it is not dependant on any other attribute except the key attribute. In this case that is so.

Normalisation seems a simple enough process. The previous example would suggest that all one does is to apply a few tests to one's relation schemas. These tests could easily be programmed and automatic normalisation would become a standard feature on all database management systems. Unfortunately this illusion is shattered when a more complex example is tackled.

Consider the following "Customer" relation: Customer

(**FName**	**LName**,	**IDNumber**,	**Cell**,	**Street**,	**City**,	**Province**,	**ProvAbbrev**)
Able	Jones	565557	0721355892	First	Randburg	Gauteng	GP
Danie	Niewoudt	298519	0851565843	Second	Randburg	Gauteng	GP
Terence	Royeppen	156720	0852663343	First	Randburg	Gauteng	GP

Table 2.2: Customer(FName, LName, IDNumber, Cell, Street, City, Province, ProvAbbrev)

The first thing to notice is the primary key. It is a composite of $FName$, $LName$ and $IDNumber$ (as before, primary keys are underlined). As a primary key it is adequate until new customers are added that are missing one of these fields. After a number of such records are added, a situation will arise where a customer cannot be added because his $FName$, $LNAME$ and empty $IDNumber$ matches an existing customer with the same $FName$, $LNAME$ and empty $IDNumber$. Using the government issued $IDNumber$ field exclusively as the primary key will not work either because there is no guarantee that deification numbers are unique.

$1NF$ transformation is done by removing all repeating groups. In the case of this relation the offending attributes are $Street$, $City$, $Province$, $ProvAbbrev$. Move these attributes to a new relation called $Address$ and assign it a primary key, the resulting $1NF$ relations are:

- $Customer(\underline{FName}, \underline{LName}, \underline{IDNumber}, Cell, $*Address_key*$)$
- $Address(\underline{Address\_key}, Street, City, Province, ProvAbbrev)$

As for a $2NF$ transformation: since all resulting relations are in $1NF$ already, we only need to test each attribute for a dependency on the entire key. $Address$ has a single attribute primary key therefore all remaining attributes are only dependant on that key. As for the "Customer" relation, we need to test the attribute $Cell$ for dependence on the entire key and not only on any one of the keys making up the primary key. From the data set presented, it is clear that $Cell$ can be obtained from querying $IDNumber$ since $IDNumber$ is non-redundant. Furthermore, customers could have more than one $Cell$ number. By

moving $Cell$ into its own relation, future problems are avoided when a customer gains another contact number.

The best $2NF$ transformation for this problem is to introduce a new (single attribute) primary key, although we could exclude $IDNumber$ from the current primary key. Either way is correct. The results of a $2NF$ transformation are as follows:

- $Customer(\underline{Customer\_key}, FName, LName, IDNumber,$ **Cell_key**, **Address_key**$)$
- $Contact(\underline{Cell\_key}, Cell)$
- $Address(\underline{Address\_key}, Street, City, Province, ProvAbbrev)$

A $3NF$ transformation can now take place on these $2NF$ relations. Here a test for transitive dependencies is done. "Customer" has no transitive dependencies but "Address" has one. $ProvAbbrev$ is transitively dependant on $Province$. This means whenever a change is made to either $Province$ or $ProvAbbrev$, the other needs to be changed too, all because the abbreviation for province is derived from the name of the province. To eliminate this dependency we move $Province$ and $ProvinceAbbrev$ into a new relation. The resulting $3NF$ relation set becomes:

- $Customer(\underline{Customer\_key}, FName, LName, IDNumber,$ **Cell_key**, **Address_key**$)$
- $Contact(\underline{Cell\_key}, Cell)$
- $Address(\underline{Address\_key}, Street, City,$ **ProvAbbrev**$)$
- $Province(\underline{ProvAbbrev}, Province)$

Suddenly, normalisation becomes a more complex process, simply because of semantics. Humans are best suited to decipher the meanings and context of attributes within relations. Programming a normalisation process is not a matter of implementing a few standard rules.

2.1.4 Entity Relationship Diagrams

Normalisation is a low level operation applied to the physical data model. On a higher level, the conceptual level, one uses the Entity Relationship Model [16] to design databases. Entity Relationship Models, like Relational Models, are based on relation theory. Unlike Relational Models, Entity Relationship Models incorporate semantic information about the real world.

In the Entity Relationship Model the world is viewed in terms of entities, relationships and roles. An "entity" is a distinctly identifiable object like a person, part, department etc. A "relationship" is the association amongst entities for example, a person *works in* a department. The function an entity performs

in the relationship is called its "role". Entity Relationship Models separate entities from relationships making it simpler to identify functional dependencies amongst data.

Entity Relationship Modelling includes a diagrammatic technique for modelling databases. Visually, a rectangular box represents an entity. A relationship is presented as a diamond shaped box. Entities are connected to relationships with simple lines on which roles (mappings) are printed.

Three mappings exist in the Entity Relationship Model: $(1 : n)$,$(m : n)$and$(1 : 1)$. A $1 : n$ mapping is seen as a "one to many" relationship. The entity on the left (1) side of this mapping is associated with 1 or many of the entities on the right side. In other words one person entity could have n number of address entities. Similarly a $m : n$ (many to many) mapping indicates that many entities on the left can be associated with many entities on the right. For example, customers belong to many banks and banks have many customers. The simple case of a $1 : 1$ (one to one) mapping means that the entity to the left can be associated with one entity to the right, for example, one person can only have one gender, one identification number, one blood type.
As an example of an entity-relationship diagram, consider the problem:

- A company employs n number of employees;
- (in case of an accident) each employee can only receive a certain type of blood;
- employees could have m number of addresses;
- if employees are related, chances are they have the same address.

The resulting Entity-Relationship-Diagram depicted in figure 2.1 is produced.

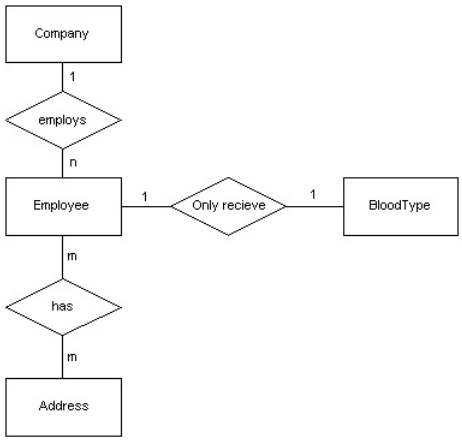

Figure 2.1: Simple Entity-Relationship diagram

From figure 2.1 the distinction between entities and relationships is clear. Lines joining entities to entities via relationships are labelled with their mappings. There exists a $1 : n$ relationship between company and employee (a company employs many employees). Employee shares a $1 : 1$ relationship with Blood type (an employee can only receive one type of blood). Address and Employee participate in a $m : m$ relationship (an employee can have many addresses and addresses can be shared by many employees).
Using the Entity Relationship Model to design data models can be done in four steps [16]:

- Identify entities and relationships;
- Identify semantic (mapping) information within relations;
- Define value sets and attributes;
- Organise data into entity/relationship relations and decide on primary keys.

2.2 Artificial Intelligence

The combination of Relational Database Theory and Artificial Intelligence presented in this dissertation also requires an exploration of Artificial Intelligence. A history of Artificial Intelligence will be presented, after which focus will be placed on the branch of Artificial Intelligence called Genetic Programming.

2.2.1 Artificial Intelligence (AI)

Artificial Intelligence (AI) enjoys a long established history. Early AI history includes two notable events. The first is the development of the "Turing Test" [62] and the second is the "Dartmouth Conference on Artificial Intelligence".

The Turing Test was introduced by Alan M. Turing as "the imitation game". Three people, a man (A), woman (B) and an interrogator (C) (in a separate room) would play. C would have to determine which of the other two players is male or female. At some point B would be replaced with a computer. If C cannot tell the computer and B apart then the computer can be considered intelligent. The Turing Test is meant to determine if a computer program has intelligence.

In 1956 John McCarthy coined the term "artificial intelligence" at the Dartmouth Conference - the first AI conference. AI is the term used to describe a branch of computer science that is concerned with the automation of intelligent behaviour [43]. Since then AI has diversified into several distinct paradigms. Each based on its own interpretation of what AI is.

The earliest application of AI is study of "Artificial Neural Networks" (ANNs). Initially proposed by Rosenblatt in the 1950's, the "Perceptron" is a model of a biological neuron with learning ability [57]. ANNs function similar to the way the human brain does; neurons receive inputs and fire off other attached neurons until some intelligent action takes place. Towards the end of the 1960s Minskey and Papert [46] studied the capability of the simple perceptron concluding that its performance was limited in relation to its computational complexity.

"Expert Systems" were soon to follow. An expert system is an attempt to capture the knowledge and reasoning ability of a human expert as a set of rules and expert data. Dr E. A. Feigenbaum, after a decade long collaboration with chemists, geneticists and computer scientists, presented the first rule-based expert system: DENDRAL [52]. DENDRAL had become a significant tool for molecular structure analysis, being used in both academic and industrial research labs. DENDRAL proposes plausible candidate structures for new or unknown chemical compounds.

At around the same time (1965) Professor L. A. Zadeh developed Fuzzy Logic [72]. Fuzzy logic is a representation scheme for uncertain or vague notions. It is a multi-valued logic that allows more human-like reasoning in machines. Fuzzy logic resolves intermediate categories between notions such as true/false, wet/dry, and busy/available used in Boolean Logic. In Fuzzy logic everything is true or false to certain degree.

The development of EC can be traced back to biologist Alex Fraser [28, 29]. Fraser simulated the evolution of 15-bit binary string generations and calculated the percentage of individuals with acceptable phenotypes with successive generations. Since then, John Holland developed "Genetic Algorithms" [33]. Ingo Rechenberg developed "Evolutionary Strategies" [54], David Fogel developed "Evolutionary Programming" [27] and John Koza is credited with the development of "Genetic Programming" [36].

2.2.2 Evolutionary Computation

Evolutionary Computation (EC) was inspired by nature. In nature, organisms in an environment are constantly adapting to better exploit their environment. This is Darwin's theory of evolution in action, first recognised in his book "On the origin of species by means of natural selection" published in 1859 [24]. Computer science uses this principle of "survival of the fittest" to great effect as a heuristic search method. Given a number of candidate solutions, a few standard operations and an evaluation function, a model closely resembling natural evolution is created to guide searches towards optimal candidates. EC is aimed at finding most optimal solutions in a problem space littered with sub optimal options. Having said this, its vital to note that EC has found successful application within the realm of

- Classification [34],

- Optimisation [35],
- Control [50] and
- Scheduling problems [40].

More importantly, EC has found commercial application ranging from industrial optimisation and design [11, 51], neural network design [56], management and finance [47], artificial life [13] to communication networks [38] and the list goes on [8, 9, 58].

In EC jargon, a candidate solution is known as an "individual". The collective noun used to describe a pool of individuals is "population". The encoding of an individual's solution is called its "genome". The solution's representation, while undergoing modification, is known as the individual's "genotype". The way the solution behaves when tested in the problem environment is known as the individual's "phenotype". "Breeding" is what happens when individuals are modified to produce new individuals. During testing each individual is graded to indicate how well it performs. This grade is called the individual's "fitness". When a population is replaced completely by children, the new population is known as the next "generation". The process of finding an optimal solution is known as "evolving a solution".

General EC Algorithm

```
Initialise a Population P of size N
Set best = null
Do:
    For each individual p_i in P
        Measure Fitness of p_i
        If fitness of p_i is optimal: Halt and return p_i
        Else if p_i is fitter than best: best = p_i
    Breed new population P' size N from P
Repeat:
Return best
```

Figure 2.2: General EC Algorithm

In figure 2.2 an initial population of a given size, N, is randomly created. The fitness of each individual is measured and the best individuals are selected to breed. A new population is created to replace the original population. This process is repeated until the ideal individual is discovered or some other termination

condition is met, for example, when the maximum number of generations is exceeded. The result of the algorithm is an individual with the best possible fitness value.
As simple as this algorithm sounds, various aspects of it needs clarification:

Evaluating Fitness

A fitness function is used to measure the fitness of individuals. The fitness function reflects the objective, the problem to be optimised, and quantifies how close an individual is the optimum solution. Selection of suitable individuals for later breeding can be done in a number of ways:
Random Selection: Parents are chosen totally at random. Each individual has an equal chance of producing offspring, regardless of its fitness.

Fitness-Proportional Selection: An individual is selected based on normalised fitness values. The more fit an individual is, the better the chance that the individual will take part in reproduction.

Ranked Selection: Each individual is ranked by its fitness and this ranking is then used to determine selection probability. In linear ranking [31, 32] individuals are first sorted from worst to best. Each individual is then selected with a probability, i, based on some linear function of its sorted rank, e.g.

$$||P|| = \frac{1}{||P||}(2 - c + (2c - 2)\frac{i - 1}{||P|| - 1})$$

Where $||P||$ is the size of the Population P, and $1 < c < 2$ is the *selection bias*: the higher the value of c, the more the system will favour better individuals. The best individual is selected with probability $\frac{c}{||P||}$, and the worst individual is selected with probability $\frac{2-c}{||P||}$.

Tournament Selection: A pool of individuals are picked at random from the population. The individual with the highest fitness in the pool is then selected [12]. The larger the size of the pool, the more directed this method is towards picking highly fit individuals but the less diversity of the new population. If the size of the pool is 1, individuals are selected totally at random.

Elitism: A few individuals from the current population are selected to take part in the next generation. These individuals are copied directly to the next population to ensure that the overall fitness does not decrease.

Breeding: Two breeding schemes exist, namely

- "Mutation", which randomly manipulates the structure of a single parent to create a single offspring. The objective of mutation is to introduce more genetic material into a population, thereby increasing genetic diversity. For evolutionary programming (EP), L. Fogel et. al [39] suggest that mutation should be the only breeding mechanism.
- "Crossover", where parents are selected to produce offspring. Offspring are generated by combining genetic material from both parents.

2.2.3 Evolutionary Computing Paradigms

EC has become an umbrella term describing various sub-disciplines, including "Evolutionary Strategies" (ES), "Evolutionary Programming" (EP), "Genetic Algorithms" (GA), "Genetic Programs" (GP), "Coevolution" and "Cultural Evolution". Each of these paradigms is based on the model of EC presented previously with their own twists on the theme. The remainder of this section will focus on each of these paradigms.

Evolutionary Strategies

ESs are based on the principle that evolution itself is an optimisation process. The objective of an ES is then to optimise the evolution process itself through the optimisation of the strategy parameters that control the evolutionary process, in parallel with the genetic evolution of individuals.

Ingo Rechenberg and H.P Schwefel wanted to use natural evolution as a basis for the development of robust algorithms for parameter optimisation problems [54]. This approach, called "Evolutionary Strategies"(ES), initially used a population of only two members [60]: a parent (represented by a real-valued vector) and one descendant created by adding normally distributed random numbers to the parent. The parent and descendant are evaluated and the better of the two becomes the ancestor for the next generation. A mutation operation is applied, after which a selection operator is applied to select the parent for the next generation.

The first multimember ES [60] replaced the single parent used previously with a population of parents. Any number of parents in the population could participate in the generation of one descendant. Key to this enhancement is the introduction of a crossover operator. All parents in the population have the same mating probability and resultant offspring are products of crossover.

Evolutionary Programming

The main focus of "Evolutionary Programming" (EP) is to evolve the behavioural traits of individuals as represented in the phenotypes of the individuals. No genetic evolution is considered. This is achieved by using only a mutation operator. The crossover operator, which exchanges genetic material between individuals, is not implemented. The first application of EP viewed individuals as finite state machines (FSM). Offspring machines are created by randomly mutating parent machines. Each machine, parent and offspring, are assigned a payoff; the selection operator will only select the best machines for the next generation based on the calculated payoff.

Genetic Algorithms

Genetic evolution is the focus of GAs. The first GAs lacked the cornerstone of ES and EP, mutation operations. GAs also had well defined chromosome representations unlike ES and EP where almost any representation was adequate, as long as a mutation function could be created to manipulate individuals meaningfully. In GAs, individuals are usually represented as bit strings and crossover is the primary breeding mechanism. Several variations on the original GA theme have been developed over time, to include the usage of mutation operators, more elaborate selection operators and different chromosome representation schemes.

Genetic Programs

"Genetic Programs" [36] differ from GAs in that a tree is used to represent the individual, instead of a bit string encoded representation. Each tree is seen as an executable program that is executed and measured. As with Genetic Algorithms, crossover is favoured more than mutation.

GP is used exclusively in this dissertation. Section 2.2.4 discusses GP in more detail.

Coevolution

Coevolution (CoE) is the competitive or complementary interaction between agents, as well as between agents and their physical environment. Typical examples of coevolution in nature are the predator-prey concept (in competition) and parasites (symbiotic). In competitive CoE this interplay is simulated with two or more populations competing against each other. The best individual would be the one who evaded destruction by the competing population. Consequently fitness is not measured in terms of how well a problem is solved, but how well the individual performs in a hostile environment. In this example of "predator-prey" evolution, highly complex populations develop as each population tries to survive the

pressures of the environment in which it exists.

Two populations may work together instead of compete with each other. In this symbiosis, success in one population will strengthen the other population. Fitness of an individual, again in relation to the other population, can be measured in terms of what contribution it has made to the overall success of co-operating populations.

Cultural Evolution

Natural evolution is a slow process. "Cultural Evolution Algorithms" (CEA) improve on natural evolution by allowing changes in a population's "culture" to direct that population's development [55]. For example, when a radical new car design is introduced, how long does it take for other manufactures to copy that design? Car bumpers are a prime example. In the 1970s, they were nothing but metal strips and now in the 2000s bumpers are huge cosmetic affairs whose primary purpose has almost been forgotten. In terms of EC, culture is data that influences the behaviour of all individuals within a population. CEAs therefore have to model the population and belief space (the term used to describe the populations' cultural information). The belief space is then used to guide individuals away from undesirable areas of the search space. The belief space and population are evolved together. Changes in the belief space influence the development of the population. Changes in the population affect the belief space. This complementary interaction between population and belief is what makes CEAs quicker than the other types of ECs.

2.2.4 Genetic Programs

From what has just been said, a general model for evolutionary computing can be derived. Generate a population, evaluate its fitness, reproduce and select a new population. Repeat until convergence and select the best individual as the solution. This algorithm describes exactly what the typical steps a GP would follow.

Precise details regarding GP have been illusive until now. The best way to demonstrate the power and beauty of a GP is to use an example. In a fictitious game show, a mathematical challenge is presented. One of the two contestants picks six numbers randomly from a range 'high' or 'low'. High numbers are all those numbers greater than 100 (inclusive) and all low numbers are less than 100 (exclusive). The presenter randomly generates a 'target' number. The contestants use the numbers chosen to create a function whose result is the 'target' number. If the target number was 598 and the chosen numbers are 500,100,100,75,5 and 2, a valid solution to the problem could be $(100 \times 5) + (100 - 2) = 598$. There is only one rule: "not all the numbers need to be used".

In a genetic program, "function trees" are created to represent the candidate functions. The fitness function would evaluate the difference between the individuals' result and the 'target' number. The closer this fitness is to zero, the better the individual is. An initial population of two individuals is randomly created as illustrated in figure 2.3.

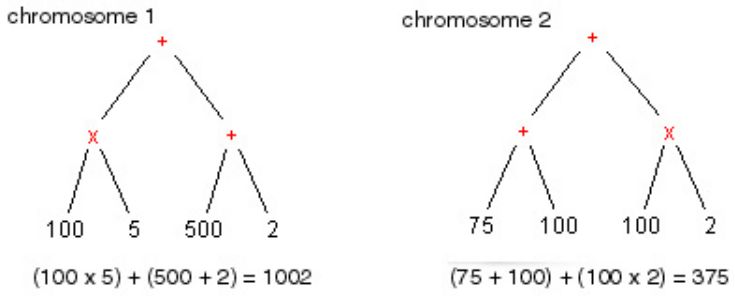

Figure 2.3: Initial GP Population

Each chromosome in figure 2.3 is a function tree made up of terminal and non-terminal symbols. Terminal symbols are the leaf nodes in the tree. A terminal symbol can only be an integer in this example. Non-terminal symbols are found that the branches within each tree. The alphabet of a non-terminal symbol is $(+, -, \times, /)$.

The fitness function does an inorder traversal of each chromosome to construct an equation. Each equation is evaluated and its result is compared to the optimal result. Chromosome 1 in figure 2.3 generates the equation

$$(100 \times 5) + (500 + 2) = 1002 \tag{2.1}$$

and chromosome 2 in figure 2.3 generates the equation

$$(75 + 100) + (100 \times 2) = 375 \tag{2.2}$$

The optimal solution to this problem is 598 and both equations 2.1 and 2.2 do not result in 598. At this point a 'one-point' crossover takes place illustrated in figure 2.4. The highlighted nodes in chromosomes 1 and 2 are selected for crossover. The resulting chromosome 1 represents the equation

$$(100 \times 5) + (100 \times 2) = 700 \tag{2.3}$$

and the resulting chromosome represents the equation

$$(75 + 100) + (500 + 2) = 677 \tag{2.4}$$

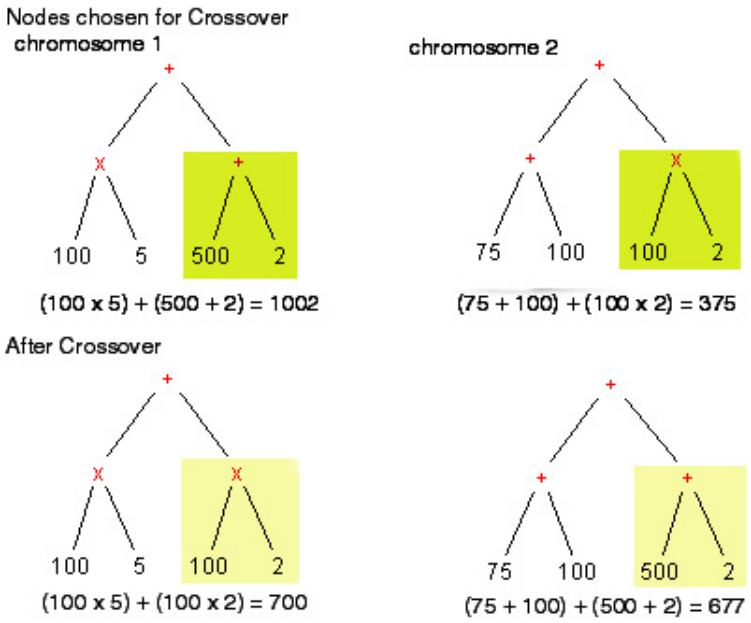

Figure 2.4: GP Crossover

The result of the crossover is then mutated at a certain (usually small) probability. The mutation operation is defined as 'randomly replace an operator'. In figure 2.5, only the left, highlighted, chromosome was selected for mutation. In this case the mutation function randomly replaced the highlighted $\times$ symbol with a $-$ symbol. After mutation chromosome 1 represents the equation

$$(100 \times 5) + (100 - 2) = 598 \tag{2.5}$$

The results of the crossover and mutation operators will become the next generation. We can see, by equation 2.5, that the fitness function was satisfied in the first generation after a single crossover and a single mutation operation (one generation). The highlighted candidate will be presented as the solution.

As an illustration, the example shows that candidates are encoded as trees (binary trees specifically); the trademark of genetic programs. The root of each sub tree is an operator. A pre-order traversal is performed to 'generate' the candidate function. This function is evaluated and its result denotes its fitness. When a candidate function evaluates to the required optimum fitness (598 in this case) the evolutionary process is halted and that candidate is presented as the 'optimal solution'.

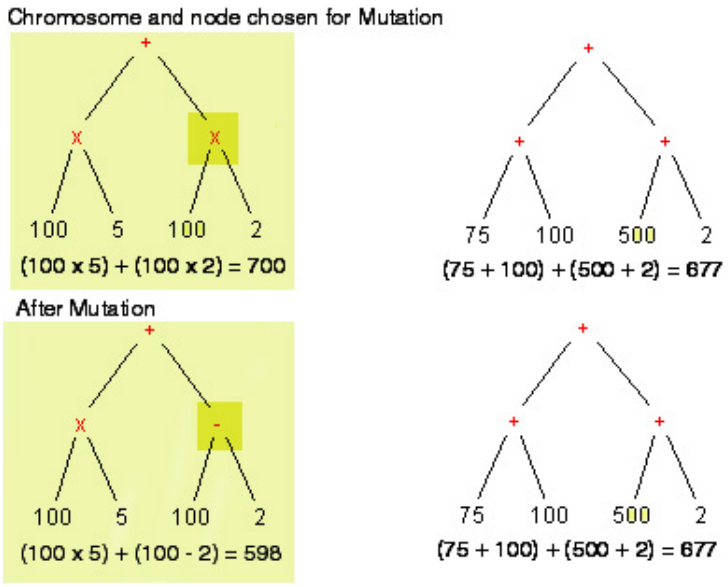

Figure 2.5: GP Mutation Operation

2.3 Related Work

Programmatically normalising databases has not enjoyed much research. The next section describes an attempt to use Prolog to normalise databases. After which, a discussion of how EC had been applied to databases previously.

2.3.1 Logical approach to Normalisation

The combination of EC and databases has not enjoyed much research activity. S. Ceri and G. Gottlob present a system for normalising relations using Prolog [15]. Here, a logical approach to normalisation is presented. Bernstein's algorithm for decomposition into $3NF$ [10], Lucchesi and Osborne's algorithm for finding all keys of a relation [42] and Tsou and Fischer's algorithm for the decomposition of relations into Boyce-Codd normal form (BCNF) [61] are the normalisation specific algorithms.

Decomposing relations into $3NF$ using Bernstein's algorithm occurs in five steps. The first step is to find a minimal cover H of the input functional dependencies using all attributes. "Minimal cover" is the term used to describe a set of functional dependencies that have, for every functional dependency, a right side that is a single attribute. No left side of any functional dependency has superfluous attributes and no functional dependency in the cover can be determined from the other functional dependencies.

The second step is to partition the set of dependencies H into groups so that all dependencies in a

group have identical left sides. Then, as a third step, two groups H_i and H_j with left sides X and Y are created. X and Y are called "equivalent keys" when the dependencies $X \rightarrow Y$ and $Y \rightarrow X$ hold. The fourth step is to build a particular cover H that includes all equivalent keys. Finally the $3NF$ relations are constructed.

A user of this Prolog system would specify a relation schema and a set of functional dependencies. The user would then apply the algorithms, implemented as rules, in any order they wish. If applied correctly, the outcome of this Prolog system is normalised relation schema.

A relation could have zero to many possible keys. Lucchesi and Osborne [42] have developed two algorithms for finding keys of a relation. Given a relation schema and a set of functional dependencies a key can be found using the first algorithm. The second algorithm will find all keys of a relation given at least one key.

The first algorithm progressively removes attributes from the given relation schema. If the removal of an attribute does not violate the rule that the key's closure must be equal to the entire relation schema, then that attribute was not a key.

The second algorithm builds, for each key K and functional dependency, left functionally dependent of right, $L \rightarrow R$, a set of attributes $S = L \cup (K - R)$. If no parts of an existing key (given or derived) fall outside of S, S will become a super key containing all possible key attributes. Superfluous attributes are then removed resulting in a key S. Lucchesi and Osborne proved that this method eventually finds all possible keys [42].

Tsou and Fischer's [61] algorithm for normalising relation schemas to BCNF is a two-condition test. If the schema of relation R has two attributes then R is in BCNF. If the schema S of R has more than two attributes and for any pair of distinct attributes A and B of R, if A does not belong to the closure of $S - [A, B]$, then R is in BCNF.

2.3.2 EC applied to databases

Not much could be found in the use of EC in databases, specifically with regard to normalisation. Other application of EC to databases are briefly described in this section.

In the technical report by W. B. Langdon entitled "Genetic Programming and Databases" [41], Langdon describes some of the research in applying EC to databases. In particular, Langdon cites research into Data Mining and Query Optimisation.

In a report by D. Wickert of Microsoft, research into physical database design with GAs is presented [70]. This research focuses on five areas of physical database design: data type selection, data placement, index creation, determining the type of index to be used, and relational database management system

internal configurations. This report is focused on index creation only and presents nominal results.

In a series of articles [63, 64, 65, 66, 67, 68, 69, 48, 49] by Dr. Patrick van Bommel *et al.*, the logical design of databases using GAs is presented.

Research most closely resembling what is presented in this dissertation is the articles entitled "Genetic Algorithms for Optimal Logical Database Design" by P. van Bommel, C. B. Lucasius and Th. P. van der Weide[48] and "Experiences with EDO: an Evolutionary Database Optimizer" by P. van Bommel[65]. These articles, the last ones in the series of articles, summarise all previous work done before [63, 64, 66, 67, 68, 69, 49]. They are most relevant because they outline the implementation details of a GA to optimise database design, just as this dissertation outlines the implementation details of a Genetic Database Normalisation Algorithm.

The goal of these articles is to present a method to transform conceptual models to efficient internal models. A conceptual model of a database describes what data is going to reside in the database, not how the data will reside. An ERD described in section 2.1.4, or in the case of [48], an Object Data Model, can be used to represent a conceptual model. An efficient internal model can be based on the relational model, network model or hierarchical model (see section 2.1.1).

There are a number of ways a conceptual model can be interpreted. P. van Bommel *et al.* limit the variety by specifying database storage requirements up front. Then proceed to estimate the time needed for queries on the database. The fitness function used in the article takes the expected storage and expected average access times into account when evaluating individuals. The objective is therefore to design an optima schema by minimizing storage requirements and access times.

2.4 Summary

This chapter introduced the two primary areas of computer science the rest of this dissertation is based on, namely database normalisation theory and evolutionary computing. Previous work done in the field of programmatically normalising databases is also highlighted. The next chapter combines Relational Databases and GP in order to develop a model to normalise relational databases programmatically.

Chapter 3

Genetic Database Normalisation Algorithm

Conceptually, the creation of a Genetic Program is straightforward. Normalising database relation schemas is reasonably simple too. Using Genetic Programming to Normalise relation schemas without any prior knowledge about the data other than field names and records is somewhat more complicated.

Essential components of a GP are the chromosome, selection scheme, crossover operation, mutation operation and fitness function. This chapter discusses these components specifically with regard to normalisation of relational databases. All these components are packaged as a Genetic Database Normalisation Algorithm (GDNA).

Experimental results are presented in the following chapter.

3.1 GDNA Source

The core source code of the Genetic Database Normalisation Algorithm (GDNA) is conceptually simple as given in figure 3.1. This fragment of GDNA source code, written in Java™1.4.1 [1], is the core algorithm. It is very similar to the general EC presented in section 2.2.

In the next sections, each of the components of the algorithm in figure 3.1 are discussed in detail. A prolog is given in section 3.2 describing the implementation and initialisation of the GDNA. Section 3.3 describes how a relation is represented as a chromosome. The GDNA crossover operator is described in section 3.4. Section 3.5 describes the fitness function developed for the GDNA. Following section 3.5 is section 3.6 describing how selection and mutation take place.

```
public class GDNA{
//------------------
    public static void main(String[]args){
    //---------------------------------------
    System.out.println("GDNA main program");
    int G = 10;
    int g = 0;
    GeneticProgram GP = new GeneticProgram("employeedata.txt","employee");
    GP.GenerateInitialPopulation(10);
    while (g < G){
        g++;
        GeneticProgram OP = (GeneticProgram) GP.clone();
        //crossover
        OP.tournamentCrossOver(7);
        //Get this fitness of the crossover Population
        OP.getFitness();
        //SelectSurvivors from both GP and OP
        GP.selectBestSurvivors(OP);
        //mutate
        GP.MutatePopulation();
        GP.reduceMutationRate();
    }
}
```

Figure 3.1: GDNA Source Code Listing

3.2 Prolog

GDNA is implemented using Java™, an object oriented Language [45]. The GDNA represents a class with one function, a "main" function that serves as the entry point for the algorithm. A class called "GDNA" is declared and the main function is installed.

In Object Oriented Programming, responsibilities can be delegated to different objects. The **GDNA** object created above has one responsibility i.e. to run a Genetic Program. The Genetic Program is re-

sponsible for population creation, maintenance and evaluation. The GDNA ties all the Genetic Program's functions together in an accepted EC structure.

The GDNA begins with the code *GeneticProgram GP = new GeneticProgram("employeedata.txt", "employee")* A **GeneticProgram**, which creates an object called "GP". The "GP" object receives data in a file called "employeedata.txt" as input. The corresponding relation is given as "employee". In Object Oriented Language jargon, an instance of **GeneticProgram**, called "GP", has been initialised using a two-parameter constructor.

All this constructor does is to assign a data filename and to assign a name for this data set. The GDNA is highly dependent on the data within a relation schema: fitness of a chromosome is measured in terms of the data within the original data file. The data file is a tab-delimited file with the first row acting as attribute names. The data file used in this example represents the relation depicted in table 2.1.

3.3 Chromosome Representation

Once the Genetic Program has been initialised with a data file, a population of chromosomes is created with *GP.GenerateInitialPopulation(10)*. The very first step in an EC is to create an initial population. In this GDNA, an initial population of 10 chromosomes is created. Suddenly three questions arise: what does a chromosome look like? how is a population maintained? and most importantly, how is chromosome created? These questions are answered below.

A relation schema is made up of a relation name, a list of attributes and a set of time varying tuples [17]. Written as $R = \{A_0, A_1, A_2, A_3, ..., A_n\}$ a relation schema is seen as a set of attributes [17]. Sets can contain other sets and therefore it is possible that relation schemas could contain other relation schemas, for example $R(A_0, A_1, A_2, S(X_1, X_2), A_3, ..., A_n)$. For the rest of this dissertation the term "nested relation" is used to describe just such a relation.

Programmatically, a relation set is seen as an array, named R, containing the elements A_0, A_1, A_2, A_3, ..., A_n. Since relations are allowed to contain other relations, so must this array be allowed to contain other arrays. Using Java™[45] (or any of the other Object Oriented Languages), such an array object can be created.

In terms of Java™, a relation schema could be implemented as a simple "Linked List" [59, 2] of objects that would grow and shrink dynamically. Each object within this linked list could be an instance of **Attribute** or an instance of **Relation**.

Outputting a **Relation** object is a matter of overriding the inherited `toString()` [45] method. A **Relation** is a descended of **Object** [3] with its own output requirements. The `toString()` method specifies how the object will be printed on the *standard output stream* [4]. An example of how a **Relation** object

will be output is:

$$R = (A_0, A_1, A_2, A_3, A_4, A_5, A_6, S(X_0, X_1, T(Y_0, Y_1))) \tag{3.1}$$

Genetic Programs require chromosomes to be represented as **Tree** [36, 59] objects. Tree objects are made up of terminals and functions. A terminal is a leaf node (a node with no children) and functions are the junctions in the trees [36]. In terms of database normalisation, all terminals are attributes and all non-terminals (functions) are the union $\bigcup$ operator. Therefore a relation is seen the union of all the sub-relations beneath it. Considering the example relation in equation 3.1, it is possible to derive a tree representation: R is the root of the tree with $A_0, .A_1, .., A_n, S$ as its child nodes. Since a set theory metaphor is being used, it is reasonable to say that the relation schema is the union of all its attributes [36]. Therefore $R = A_0 \bigcup A_1 \bigcup ... \bigcup A_n \bigcup S$, where S is the union of X_1 and X_2 and T is the union of Y_0 and Y_1.

A method is required to convert the linked representation of a **Relation** to a tree representation. Java™1.4.1 does not include a native tree data structure [59]. It does, unfortunately, include an ill suited, inflexible JTree [5] data structure. An alternative to creating a custom tree is to use a third party package. The package chosen is the JDSL 2 Library of Data Structures [6], chosen because of its strong research orientation and visualisation tools.

The JDSL 2 **NodeTree** [7] data structure proved ideal as a container for **Relation** objects. The research version of the JDSL 2 library ships with a set of tutorials; one of which draws a **NodeTree** object to screen. This author has taken the liberty of adapting this tutorial to draw **Relation** objects. Once drawn, the relation in equation 3.1 is illustrated in figure 3.2.

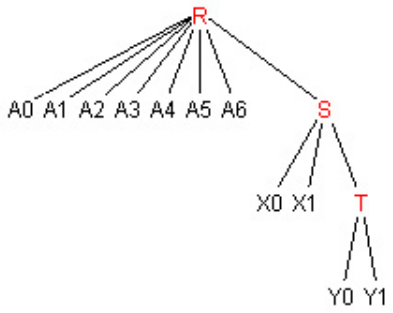

Figure 3.2: $R(A_0, A_1, A_2, S(X_0, X_1, T(Y_0, Y_1)), A_3, A_4, A_5, A_6)$

The root of each sub tree (relation) is the relation name, while terminal nodes are all the attributes in the underlying **Relation** object.

As for a population of chromosomes, a new class called **Population** was created. This class uses the JDSL 2 **NodeSequence** to store a list of chromosomes. Maintenance of a population is done with a series

of functions. There is a method to *add* new chromosomes to the population. This method does so by comparing the fitness of the new chromosome against the lowest fit chromosome in the population. If the new chromosome is more fit, the method replaces the chromosome it was compared with, ensuring the population size is always constant. A *sort* method sorts all chromosomes from highest fit to lowest fit making it easier to choose best fit chromosomes for inclusion into a subsequent generation. Amongst the less important methods is one to *print* each chromosome to either standard output or a file. Displaying chromosomes on the screen is handled by a different tailor made class. In simple terms, a **Population** is nothing more than a list of chromosomes.

How is a chromosome created? All generated chromosomes start life as a single relation with all attributes listed in the data file as in figure 3.3. These attributes are randomly placed to introduce some variety. Each chromosome is then subjected to a single mutation operation, which collects a randomly chosen number of attributes in a new relation and replaces all those chosen attributes with the single newly created sub-relation as shown in figure 3.4.

Figure 3.3: 1^{st} phase Employee Chromosome Creation

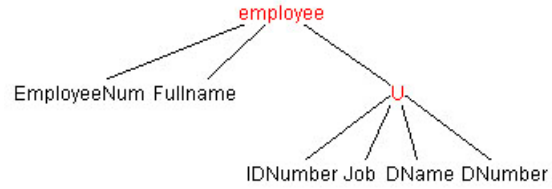

Figure 3.4: Mutation Population

3.4 The GDNA Crossover Operation

The new population is created from the current population using *GeneticProgram OP = (GeneticProgram) GP.clone()*. The copy of the population is manipulated by the crossover and mutation operators, then pitted against the original **GeneticProgram** in a competition so that the absolute best chromosomes are selected for the next population. In Java™ the only way to ensure a deep copy of any **Object** is to define a *clone()* method. Using this method ensures that the programmer is not tampering with objects that

shouldn't be tampered with. In this case, the current population inside **GeneticProgram** "GP" is to be left untouched until the selection phase. All manipulation is done on the population inside **GeneticProgram** "OP" (Offspring Program).

In the line of code, *OP.tournamentCrossOver(7)*, two very important operations of a GA are invoked. A tournament selection [30] to select parents for crossover is requested and then the actual crossover is performed.

A variety of selection mechanisms are detailed in section 2.2.2. By far the simplest mechanism to implement is a *Random selection*. Where two parents are chosen at random, allowed to breed and the resulting offspring are promoted to the next generation after being mutated. This function is called *randomCrossOver* in a GDNA.

D. E. Goldberg and K. Deb [30] suggest tournament selection as the best selection mechanism to use. Tournament selection has a better time complexity and the larger the tournament pool, the better the growth ratios: meaning, the bigger the tournament pool, the more likely the GDNA will choose the highest fit individuals resulting in a quicker algorithm covering a larger search space.

Crossover is the term used to describe the inter-breeding of parent chromosomes to create new child chromosomes referred to as 'offspring'. A portion of genetic material from each parent is exchanged with the other parent. The resulting chromosomes compete for inclusion into the next population. The objective of the crossover process is to accumulate the best characteristics of each chromosome in each generation.

In the GDNA a one-point crossover [37] is used. A randomly selected sub tree in the first parent is replaced with a randomly selected sub tree in the second parent as depicted in figure 2.4. Doing so introduces a great deal of variety into the population. It also introduces ill-formed chromosomes into the population; particularly chromosomes with repeating or missing attributes begin to appear. The fitness function defined next will swiftly deal with ill-formed chromosomes.

3.5 Fitness Function

In terms of this GDNA a normal form is seen as a fitness function. *OP.getFitness()* instructs the "GeneticProgram" to measure the fitness of each individual within the population. The source code in figure 3.5 summarizes the fitness function used to evaluate the fitness of each chromosome.

```
public float getFitness(){
//-------------------------
 //Calculate the fitness of 'this' relation schema
 getLocalRepeatingTuples(this);
 findAllGroups(this);
 generalFitness = collectFitnesses(this);

 //penalties
 //root relation's group penalty
 if(!isThisAGroup(this)) generalFitness += 10;
 //which relations, including root relation,
 //only have one attribute or sub-relation?
 float singles = countSingles(this);
 if(isLeafRelation()) singles += 1000;
 generalFitness += singles;

 //are all the original attributes present?
 float allAttributes = getMissingAttributePenalty()*1000;
 generalFitness += allAttributes;

 //return the fitness result
 return generalFitness;
}
```

Figure 3.5: GDNA Fitness Function

The fitness function is nothing more than the sum of the fitnesses associated with each form or normalisation and a penalty to hinder the success of ill formed chromosomes. Fitness is measured by *getLocalRepeatingTuples(this)* and *findAllGroups(this)*. The penalty is measured by *isThisAGroup(this)*, *countSingles(this)* and *getMissingAttributePenalty()*.

getLocalRepeatingTuples(this) recursively counts all the redundant tuples in each sub-relation, satisfying the requirement for $1NF$. A further requirement for $1NF$ is that relations do not contain repeating groups. *findAllGroups(this)* recursively asks the question: is this sub-relation a group?. If a sub-relation is a group, then its fitness becomes zero (because all repeating groups are in their own relation) else its fit-

ness remains untouched. A group is defined by a simple calculation: are the number of redundant tuples equal to number of tuples in the relation and are the number of redundant tuples equal to the number of repeating fields?

For a $2NF$ evaluation, each attribute in the sub-relation is checked for its dependency on the entire key. A single point is accumulated for each violation. The overall fitness of the relation schema is the sum of the fitness of each sub-relation calculated by *collectFitnesses(this)*.

Penalties are introduced to force the GDNA to discard ill formed relation schemas. An ill formed relation schema is one that, firstly, has a group in its root relation. A ten-point penalty is used to bias aesthetically pleasing chromosomes. A more serious penalty is attracted when a sub-relation within a relation schema has one attribute or sub-relation measured by *countSingles(this)*. Not doing so encourages chromosomes with one attribute per sub-relation and as many sub-relations as there are attributes. This type of configuration is wasteful and pointless. If the root of the relation schema only has attributes in it, or for whatever reason is empty, then the *singles* penalty is incremented by 1000. A penalty this large throws the chromosome completely out of contention.

The final, rather large, penalty is awarded to chromosomes that do not have the same attribute set as the original relation schema. *getMissingAttributePenalty()* tallies up the number of attributes the chromosome is missing. This number is then exploded by 1000 to force it out of competition. A well formed and correctly normalised, up to $2NF$, chromosome will carry a fitness value of 0.0.

3.6 Selection and Mutation

The selection operator is invoked with the statement *GP.selectBestSurvivors(OP)*. Selecting the best individuals from the parent population and the offspring population creates the following generation. The parent population is concatenated to the offspring population and then sorted. The concatenated population is trimmed to the size of the original parent population. This new population, comprising the best parent and offspring chromosomes, becomes the parent population for the next iteration.

This is an elitist selection mechanism. Every chromosome, from parent to offspring, has an equal chance of promoting to the next generation. The elitist selection mechanism allows individuals to survive for a number of generations. Consequently, the ratio of parent to offspring from generation to generation is always variable.

During the creation of the initial population, each chromosome is subject to various mutations when *GP.MutatePopulation()* is executed. The first mutation collects a random number of attributes and places them into a sub-relation as illustrated in figures 3.3 and 3.4. This method has been generalised to function on any sub-relation within the relation schema.

The second mutation moves an attribute from a randomly selected sub-relation to another randomly selected sub-relation. Doing so distributes the overall fitness more evenly amongst the sub-relations, as highly redundant attributes are moved to more fit sub-relations.

Mutation is performed uniformly on the offspring at a very low probability. Doing so introduces new unique chromosomes into the population. Ideally, each mutated chromosome will have a better fitness value than it did before. Practically though, mutating a chromosome could damage its fitness value especially for individuals close to the optimum.

In a GP, crossover is more important than mutation [36]; mutation should be a background function. In the GDNA the mutation rate is lowered exponentially by dividing the current mutation rate by 1.01 using *GP.reduceMutationRate()*. This ensures the mutation rate decreases quickly but never becomes zero.

This dynamic mutation rate allows the GP to cover a large part of the search space. As the number of generations increases, and the fitness of individuals are improved, the less diverse the search should be to prevent good individuals to loose their good genetic building blocks.

Decreasing the mutation rate linearly is also an option, although the effect is not quite as dramatic. It takes longer for the GDNA to find the best individual on a linearly decreasing mutation rate.

A linearly decreasing mutation rate does encourage a larger initial diversity of individuals than an exponentially decreasing mutation rate, but the a linear rate converges to zero much quicker than the exponential alternative. An exponentially decreasing mutation rate favours a much longer persistence of diversity even though that diversity may not be as large as that produced by a linearly decreasing rate. Not enough diversity is introduced into the population over the entire evolution to cover the large search space when using a linear mutation rate.

3.7 Epilogue

The GDNA is very simple, a while loop and a tournament pool size is all a programmer needs to control. Simple solutions, in this author's view, are more elegant and less error prone. There is a great deal of very complicated source code behind this GDNA, and that is where it should stay. The face of a GDNA has to be clear and understandable, which is why no complicated termination criteria are in play. Instead the GDNA stops when the maximum number of generations is reached.

Chapter 4

Experimental Results

In the previous chapter a GDNA was described. This chapter shows results of the application of the GDNA to normalise databases. The accuracy and efficiency of the GDNA are demonstrated.

Each experiment is given a single input data file, shown in the appendix, representing the relation that requires normalisation. Outputs from the GDNA are captured as graphs plotted and entity relationship diagrams depicting the optimal solutions found. The GDNA was altered to execute each experiment 30 times. At the end of 30 executions the absolute best experiment is stored. These absolute best experiments are what are presented in this chapter.

For each series of experiments, the initial population and a number of generations are specified. Graphical plots show the average fitness of the population and the fitness of the best individuals. Where applicable, the fitness of the best chromosome is plotted separately to give a better indication of its fitness in relation to the populations' average fitness.

ERDs representing optimal solutions are given. Relations in the ERD are highlighted to show what they are and what the resulting chromosome looked like.

Efficiency of the GDNA is tested by repeating all experiments a number of times with a fixed population size and evolutionary run aborted when the optimal individual is found. If no optimal candidate is found, the GDNA will terminate when the maximum number of generations is reached. The average number of generations it takes to find the optimal individual as well as the average time it take to run each experiment is given as an indication of the time complexity.

Finally, insights, into the GDNA and a $3NF$ fitness function are offered. The following chapter concludes this dissertation with a discussion of what was achieved and what future research possibilities exist.

4.1 Relation “gender”

The GDNA presented in the previous chapter was tested with a relatively complicated “employee” relation. This relation had a number of attribute redundancies and very distinct groups. How would the algorithm perform on a two-attribute relation? How long would it take to find the correctly normalised version of the “gender” relation in table 4.1?

(name	gender)
A	M
B	M
C	M
D	M
E	M
F	M
G	F
H	F
I	F
J	F
K	F

Table 4.1: gender

Running the GDNA over 10 generations with a large number individuals produced the plot in figure 4.1 and normalised relation schema in figure 4.2.

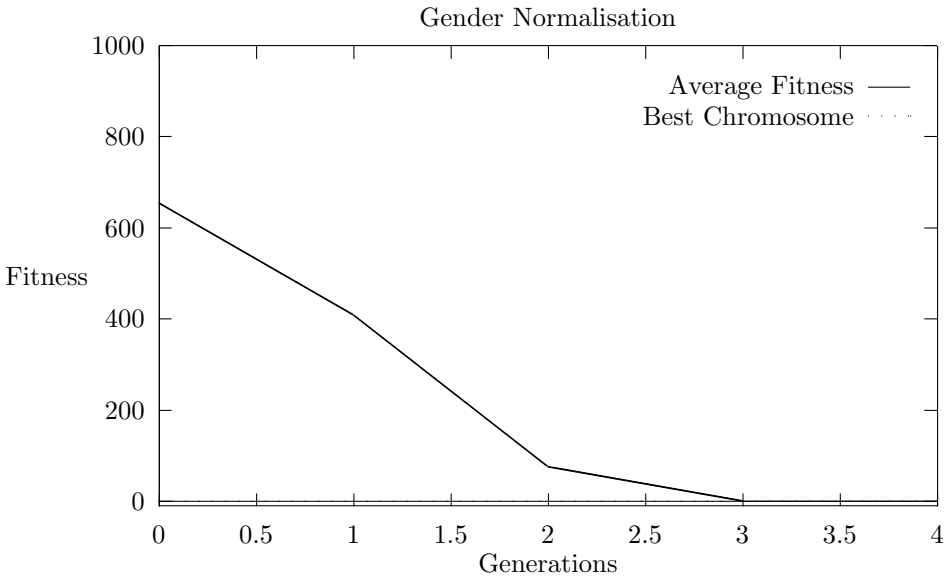

Figure 4.1: Average fitness of the “gender” relation normalisation

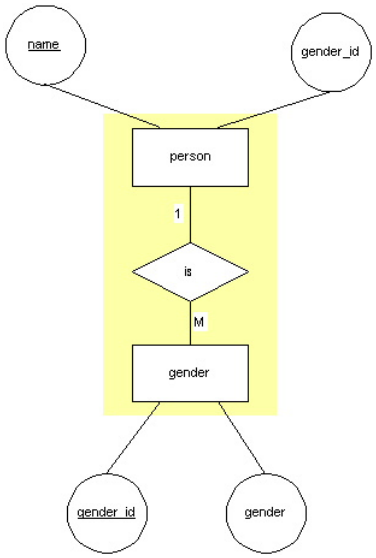

Figure 4.2: Gender ERD

The “Best Chromosome”, with a fitness value of 0, was found in the first generation and was retained throughout the 10 generations. The “Average Fitness” of all 250 individuals converged to zero after three generations, meaning that all individuals converged to the optimal solution.

The final relation schema as illustrated in figure 4.2 , is what was expected. The original relation was dissolved into two new relations, “person” and “gender”. The “person” relation has the *name* attribute

and a foreign key to the “gender” relation. The “gender” relation has a new primary key and the attribute called “gender”. A $1:M$ *is* relationship exists between person and gender. A ‘person’ can only have one ‘gender’, while the same gender can be allocated to many people.

The relation schema in figure 4.2 is correct for the relation in table 4.1. The *name* attribute is unique while the *gender* attribute has redundancy. By moving the redundant attribute to its own relation the first rule of normalisation is satisfied. By introducing a foreign/primary key pair and checking the primary key dependency in the new “gender”, relation the second rule of normalisation is satisfied. This new relation schema is perfect in terms of $1NF$ and $2NF$.

4.2 Relation “employee”

Going back to the “employee” relation in table 2.1, what relation schema will the GDNA offer with a randomly chosen population size of 150 chromosomes and an evolution of 100 generations?

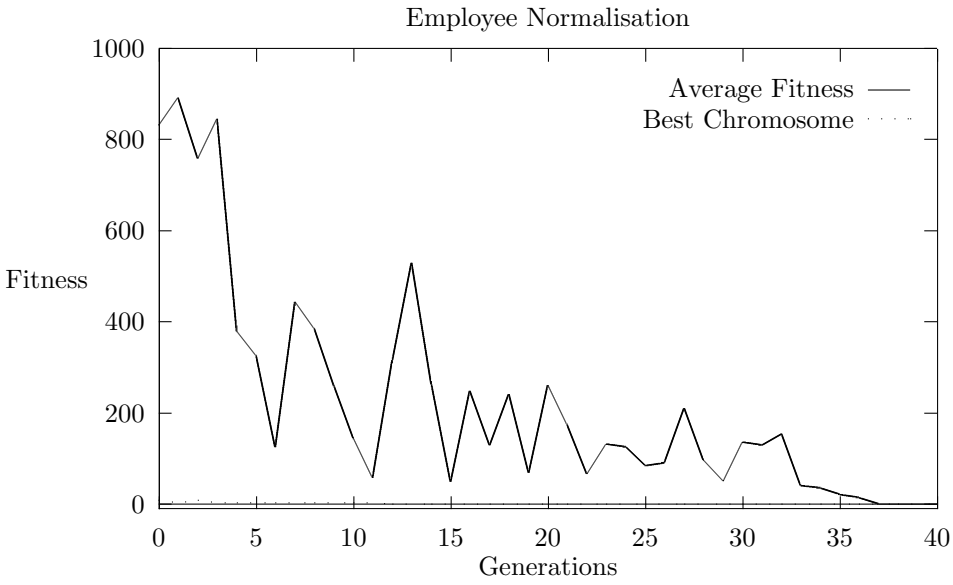

Figure 4.3: Average fitness of the “employee” relation normalisation

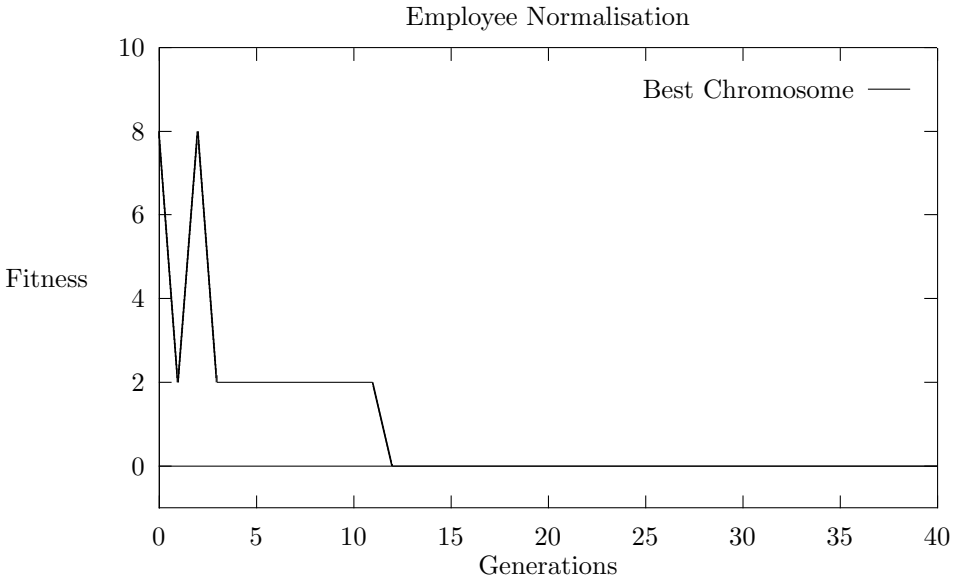

Figure 4.4: Fitness of the best "employee" chromosome

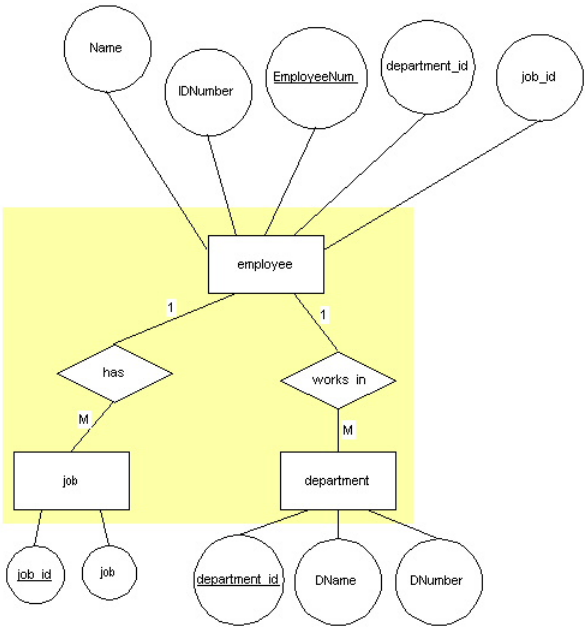

Figure 4.5: Employee ERD

The average fitness of the population is depicted in figure 4.3, while the fitness of the best chromosome is depicted in figure 4.4. The resulting relation schema is illustrated in 4.5. The GDNA did not create the best chromosome initially when the population was created. The GDNA did eventually find the best individual after 13 generations. Initially, the best chromosome's fitness was 8.0 and then went down to

2.0 and went up to 8.0 again. Eventually the fitness of the best chromosome came down to 2.0 and finally settled on 0.0 after 12 generations. The spikes in the best chromosome's fitness can be attributed to the mutation function. The mutation caused good genetic material to be replaced with bad genetic material. The mutation function follows a simple rule: if the fitness of an individual is anything other than zero it is subject to mutation at the current mutation probability rate. From this plot a clearer picture of the mutation rate emerges.

As for the final relation schema depicted in figure 4.5, a relation schema with three relations was found to be optimal. All redundant attributes are grouped together in their own relation. The "job" relation has less redundancy than the "department" relation, which is why these two relations exist separately. Measuring this relation schema against $1NF$ and $2NF$ shows that this relation schema is indeed in $2NF$. All repeating groups have been removed and all key dependencies are resolved.

4.3 Relation "customer"

This "customer" relation in table 4.2, taken from a data warehouse, exposes the core of the GDNA. From the initial relation in table 4.2 and the resulting relation schema illustrated in figure 4.8, it can be seen how the GDNA searched for groups (relations capturing redundancy). The GDNA was allowed to run over 220 generations with a randomly chosen population size of 50 chromosomes.

(customer_id	firstname	lastname	city	state	zipcode	gender	married)
65	Anderson	Anderson	Burlington	Vermont	15487	M	Married
104	Anderson	Anderson	Laramie	Wyoming	19367	M	Married
290	Anderson	Anderson	Salt Lake City	Utah	38559	M	Married
368	Anderson	Anderson	Madison	Wisconsin	45739	M	Married
430	Anderson	Anderson	Dallas	Texas	50800	M	Married
500	Anderson	Anderson	Seattle	Washington	57621	M	Married
503	Anderson	Anderson	Birmingham	Alabama	57750	M	Married
797	Anderson	Anderson	Charleston	West Virginia	81336	M	Married
835	Anderson	Anderson	Washington	DC	85208	M	Married
957	Anderson	Anderson	Richmond	Virginia	96972	M	Single

Table 4.2: customer(customer_id,firstname,lastname,city, state,zipcode,gender,married)

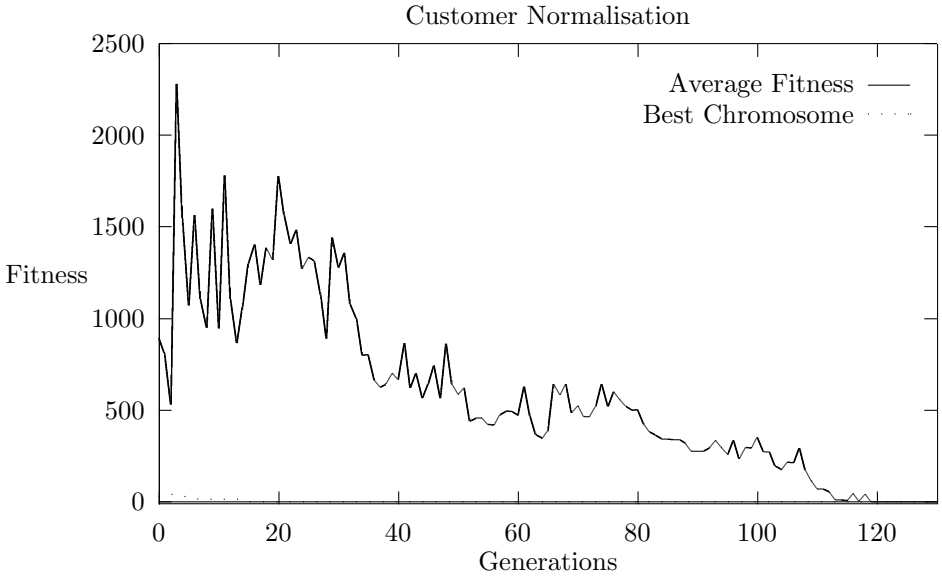

Figure 4.6: Average fitness of the “customer” relation normalisation

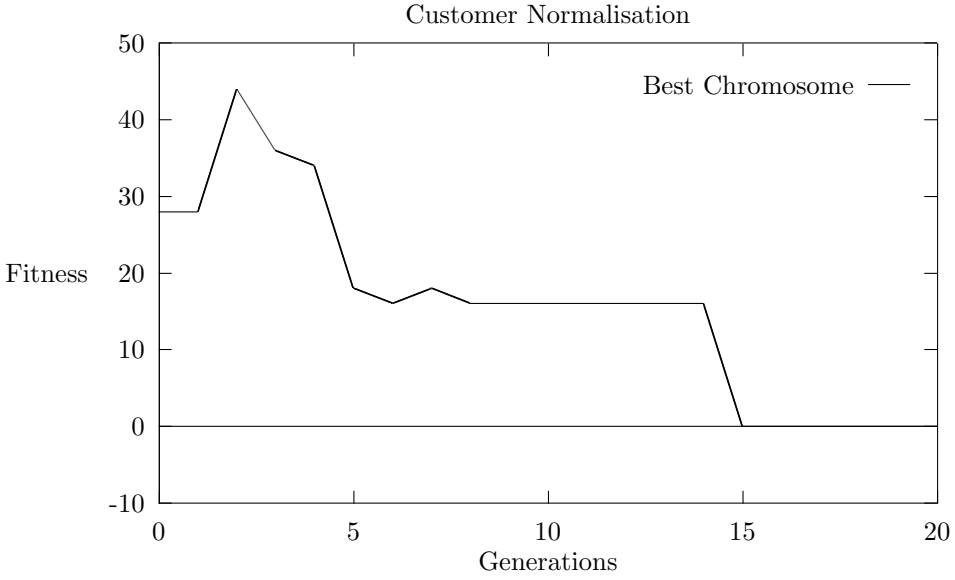

Figure 4.7: Fitness of the best “customer” chromosome

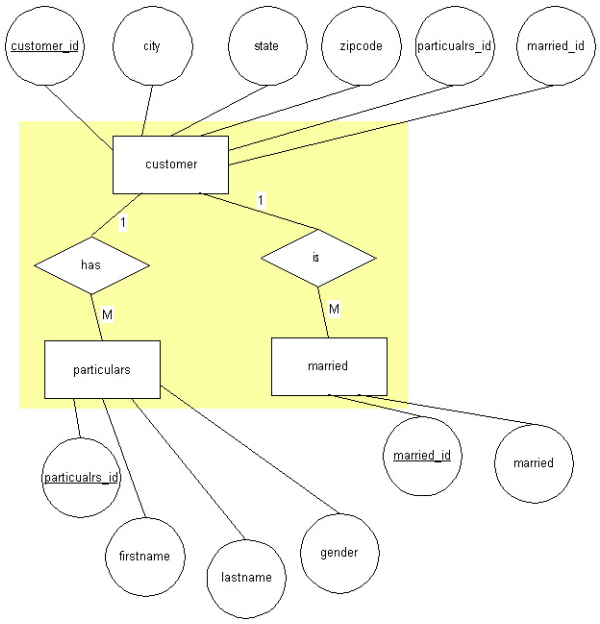

Figure 4.8: Customer ERD

Figures 4.6 and 4.7 confirm a trend: the best individuals are found before the average fitness converges to zero as expected from an EA. Convergence also occurs very quickly as all individuals in the population are replaced with the optimal individual. This trend is attributed to the decrease in the impact mutation has, since the mutation rate is exponentially reduced.

The resulting relation schema shown in figure 4.8 highlights one critical aspect of the GDNA: the GDNA does not know what attributes mean, nor what relationships exist between attributes. This conceptual information has been excluded from the GDNA on purpose. Thus the resulting relation schema, although normalised up to $2NF$, doesn't 'look' right. How can address information be stored in the root relation while customer particulars are stored in sub relations? Regardless of what the relation schema looks like it is still correct and acceptable.

The relation in table 4.2 has two groups within it. The attributes *firstname, lastname* and *gender* have the same values throughout the relation. Attribute *married* has only one value different from all the others. The GDNA is designed to search for groups and these are the only groups that exist and are found.

4.4 GDNA effeciency

From the previous successful experiments the GDNA results are summarised in table 4.4. These results are the best results produced by the GDNA after it had executed 30 times. Table 4.4 shows the results of

all the GDNA experiments performed.

Relation	Population Size	Total Generations	Generation in which best individual is found
Gender	50	10	3
Employee	150	100	12
Customer	50	220	15

Table 4.3: GDNA Summary

Relation	No. of Records	Attributes	Average no of Generations to Converge	Average Time
Gender	11	2	6.5	0:49
Employee	5	6	102.6	9:07
Customer	10	8	96	18:56

Table 4.4: Vital Statistics

The current GDNA is accurate for small data sets but not efficient at all. It is clear from table 4.4 that the GDNA takes longer to find optimal solutions the bigger the relation becomes. The number of generations the GDNA takes to find the best individual is unreasonably high for such small relations, considering that a trained database designer would normalise these relations in a few seconds. The GDNA's performance is acceptable for an untrained database designer, given that that designer need not do any work except execute the GDNA.

The bottleneck of this GDNA was found to be its fitness function (described in section 3.5). It takes too long to calculate redundant tuples and redundant fields. This GDNA was designed on the initial requirement that no information about the source relation schema be given. Doing so separates the GDNA from all previous work in the area of programmatically normalising database relation schemas (discussed in section 2.3).

4.5 $3NF$ Resolution

The $3NF$ is defined in chapter 2 as:

> Relations in third normal form ($3NF$) are in $2NF$ and all transitive dependencies are eliminated. In other words, no attribute of a relation is dependant on any other attribute of that relation other than the primary key [18, 19].

To satisfy this requirement the fitness function in section 3.5 would have to be adjusted. The adjustment would occur when sub-relations are measured for fitness. Included in this measurement will be a penalty for each transitive dependency found. Each attribute in each sub-relation would be compared to each other attribute in that sub-relation to find out if a transitive dependency exists. This calculation can be done in the GDNA by evaluating the relationship each field has with each other field in an exhaustive search. Alternatively, randomly chosen fields can be compared to uncover any transitive dependencies. Another, far simpler way, is to give all transitive dependencies up front thus destroying the foundation of the GDNA - no prior knowledge is offered. A $3NF$ measurement would slow the GDNA down more.

The GDNA in this dissertation has not been optimised at all. It does not make use of the multi-threading capabilities Java™has to offer. It does not maintain any information on the source relation schema as it is being processed. All this GDNA is designed to do is to prove that indeed a relation schema can be normalised with a Genetic Program.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Fundamental to this work is the question: "can a relation be normalised using genetic programming?" A Java™application and a standard GP algorithm answered this question. Together, a powerful and flexible application was created that normalises a given relation to $2NF$. It is not a fast application and it is not yet ready for commercial development, it is an experimental application that proves a "yes" answer to the question: "can the relations investigated in this dissertation be normalised using genetic programming?"

As sub-questions to the primary question this author wanted to know:

- can relational databases be normalised programmatically;
- is a genetic strategy simpler to implement?

The primary question was answered in Chapters 3 and 4 where a Genetic Database Normalising Algorithm (GDNA) was presented and tested. Chapter 3 undoubtly shows that a genetic strategy is straightforward, simple to implement and execute. The crux of the algorithm is the fitness function, the definition of which proved to be almost insignificant considering the amount of intellectual capital it incorporates.

Central to the GDNA is its non-reliance on meta-data about the source relation under evaluation. This unique feature negatively affects the speed of the GDNA. Another factor affecting the speed of the GDNA is a $3NF$ relation schema resolution. It was said that functional dependencies would have to be found for every individual in the population every time a new population was created (refer to section 4.5). This calculation was excluded from the scope of this dissertation.

5.2 Future Research

5.2.1 Improving Performance

The GDNA is not a quick algorithm as yet. Incorporating a learning element into the GDNA could serve to improve speed. Every chromosome the GDNA evaluates will generate some data about that chromosome. This data could be stored centrally and referenced for every sub-sequent chromosome. Information concerning functional dependencies for a $3NF$ relation schema resolution could be stored. Information on optimal groups (sub-relations) could also be stored. The fitness function would be redesigned to evaluate stored knowledge about the structure of relations under examination instead re-discovering this knowledge every time a new individual is evaluated.

5.2.2 Improving Value

The current GDNA is bias toward $1:1$ and $1:M$ relationships between relations. Future research can be undertaken to answer the question: "how to introduce and encode $M:M$ relationships between relations?" Along with further normalisation, this added dimension could make the GDNA viable for commercial application.

Further empirical investigations into better mutation schemas, optimal values for rate of crossover and mutation, different selection schemes need to be performed to find the best set of operators for the normalisation task.

5.2.3 Scalability

The GDNA presented in this dissertation was designed for small relations. An investigation into the GDNA's performance on larger, more complex, relations should be undertaken in future. It is possible that the penalty component of the fitness function will be inadequate for larger problems, since redundancy will exceed the penalty.

The fitness function, being the central component of the GDNA, is a multi-objective function. There is a fitness component to measure the accuracy of the resulting schema and penalty component to hinder the performance of ill formed chromosomes. It may be prudent to focus research into optimising the fitness function for all types of relations. The work of Coelle Coelle [22] and Coello Coello *et al.* [14] should be consulted in this regard.

Section 2.3.1 described several normalisation algorithms. Particularly, the two algorithms by Luchessi and Osborne to find keys of a relation. The effect of using these algorithms as part of the GDNA should also be tested in future to ascertain how well they would improve the performance of the GDNA.

5.2.4 Application

Currently the GDNA is nothing more than a toy. It has no real application except for, maybe, a demonstration tool to be used in a database theory course. It served its function as foundation for research but for it to become a viable commercial application it will have be to enhanced. The GDNA could become a serious research tool if further normalisation were included. Optimising the GDNA and adding packaging, GUI and XML or SQL input and output, could see the GDNA being incorporated into various existing CASE tools or becoming its own CASE tool. This is the direction this author would like the GDNA to take.

Bibliography

[1] http://java.sun.com, March 2003. 25

[2] http://java.sun.com/j2se/1.4/docs/api/java/util/LinkedList.html, March 2003. 27

[3] http://java.sun.com/j2se/1.4/docs/api/java/lang/Object.html, March 2003. 27

[4] http://java.sun.com/docs/books/tutorial/essential/system/iostreams.html, March 2003. 27

[5] http://java.sun.com/j2se/1.4/docs/api/javax/swing/JTree.html, March 2003. 28

[6] http://www.cs.brown.edu/cgc/jdsl/, January 2003. 28

[7] http://www.cs.brown.edu/cgc/jdsl/doc/jdsl/core/ref/NodeTree.html, January 2003. 28

[8] J. T. Alander. An indexed bibliography of genetic algorithms. In *Lance Chambers (Ed.), Practical Handbook of Genetic Algorithms: New Frontiers, Volume II, CRC Press*. 1995. 15

[9] J. T. Alander. Indexed bibliography of genetic algorithms in robotics, 1995. 15

[10] P. A. Bernstein. Synthesizing third normal form relations from functional dependencies. *ACM Transactions on Database Systems (TODS)*, 1(4):277–298, 1976. 22

[11] M. F. Bramlette and E. E. Bouchard. Genetic algorithms in parametric design of aircraft, in ref 16, 109, 1991. 15

[12] A. Brindle. *Genetic Algorithms for Function Optimization*. Unpublished doctoral dissertation, University of Alberta, Edmonton, Canada, 1981. 16

[13] R. A. Brooks and P. Maes (Editors). *Artificial Life IV*. MIT Press, Cambridge, MA, 1994. 15

[14] G. B. Lamont C. A. Coelle Coelle and David A. Van Veldhuizen. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Kluwer Academic Publishers, 2002. 45

[15] S. Ceri and G. Gottlob. Normalization of relations and prolog. *Communications of the ACM*, 29(6):524–544, 1986. 9, 22

[16] P. P. Chen. The entity-relationship model – toward a unified view of data. *ACM Transactions on Database Systems*, pages 9–36, Vol. 1, No. 1, March 1976. 11, 13

[17] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970. 6, 8, 9, 27

[18] E. F. Codd. Further normalization of the data base relational model. *IBM Research Report, San Jose, California*, RJ909, 1971. 7, 9, 42

[19] E. F. Codd. Normalized data base structure: A brief tutorial. In *ACM SIGFIDET Workshop on Data Description, Access, and Control.*, San Diego, California, 1971. 9, 42

[20] E. F. Codd. Further normalization of the data base relational model. In *Data Base Systems, Courant Computer Science Series, vol. 6, R. Rustin, Ed., Prentice-Hall, Englewood Cliffs, N.J.*, pages 33–64, 1972. 8

[21] E. F. Codd. Recent investigations into relational database systems. In *Proceedings of the IFIP Congress*, pages 1017–1021, Stockholm, Sweden, 1974. 8, 9

[22] C. A. Coelle Coelle. *An Empirical study of evolutionary techniques for multiobjective optimization in Engineering Design*. PhD thesis, Tulane University. 45

[23] M. W. Blasgen J. N. Gray W. F. King B. G. Lindsay R. Lorie J. W. Mehl T. G. Price F. Putzolu P. G. Selinger M. Schkolnick D. R. Slutz I. L. Traiger B. W. Wade D. D. Chamberlin, M. M. Astrahan and R. A. Yost. A history and evaluation of system r. pages 54–68, 1998. 6

[24] C. Darwin. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. Electronic Text Center, University of Virginia Library, 1859. 14

[25] R. Fagin. Normal forms and relational database operators. In *ACM SIGMOD International Conference on Management of Data.*, Boston, Mass, May 31–June 1, 1979. 9

[26] R. Fagin. Multivalued dependencies and a new normal form for relational databases. In *ACM Transactions on Database Systems 2 3.*, Sept. 1977. 9

[27] D. Fogel. Artificial intelligence through simulated evolution, 1960. 14

[28] A. S. Fraser. Simulation of genetic systems by automatic digital computers: S-linkage, dominance, and epistasis, 1960. 14

[29] A. S. Fraser. Simulation of genetic systems, 1962. 14

[30] D. E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In G. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 69–93, San Mateo, 1991. Morgan Kaufmann. 30

[31] J. J. Grefenstette and J. E. Baker. The genitor algorithm and selection pressure: why ranked-based allocation of reproductive trials is best. In *Schaffer, J. D., ed., Proceedings of the Third International Conference on Genetic Algorithms*, pages 239–255, 1989. 16

[32] J. J. Grefenstette and J. E. Baker. How genetic algorithms work: a critical look at implicit parallelism. In *Schaffer, J. D., ed., Proceedings of the Third International Conference on Genetic Algorithms*, pages 20–27, 1989. 16

[33] J. Holland. *Adaption In Natural and Artificial Systems*. University of Michigan Press, 1975. 14

[34] W. M. Spears K.A. De Jong and D. F. Gordon. Using genetic algorithms for concept learning. *Machine Learning*, 13:161–188, 1993. 14

[35] J. R. Koza. A genetic approach to econometric modeling. In Paul Bourgine and Bernard Walliser, editors, *Economics and Cognitive Science*, pages 57–75. Pergamon Press, Oxford, UK, 1991. 15

[36] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge,MA, USA: MIT Press, 1992. 14, 18, 28, 33

[37] J. R. Koza. Genetic programming: On the programming of computers by means of natural selection. *Statistics and Computing*, 4(2), 1994. 30

[38] J. L. Cox L. Davis, D. Orvosh and Y. Qiu. A genetic algorithm for survivable network design. In *Proceedings of the fifth Int. Conf. on Genetic Algorithms*. S. Forrest (Editor), Morgan Kaufmann, CA, 408, 1993. 15

[39] A. J. Owens L. J. Fogel and M. J. Walsh. *Artificial Intelligence through Simulated Evolution.* Wiley, New York, 1966. 17

[40] M. LaLena. Travelling salesman problem using genetic algorithms. http://www.lalena.com/ai/tsp, April 2003. 15

[41] W. B. Langdon. Genetic programming and databases. Technical Report IN/96/4, Gower Street, London WC1E 6BT, UK, 11 1996. 23

[42] C. L. Lucchesi and S. L. Osborn. Candidate keys for realtions. *J. Comput. Syst. Sci*, 17(2):270–280, Oct 1978. 22, 23

[43] Luger and Stubblefield. *Artificial Intelligence: Structures and strategies for complex problem solving*, chapter 1. Benjamin/Cummings Publishing Company, Inc. Redwood City, CA 94065, 1993. 13

[44] S. Zdonik D. DeWitt M. Atkinson, F. Banchilon and D. Dietrich. The object-oriented database system manifesto. In *Proceedings of the International Conference on Deductive Databases*, pages 40–56, Kyoto, Japan, 1989. 7

[45] K. Walrath M. Campione and A. Huml. *The Java^{TM}Tutorial: A Short Course on the Basics (3rd Edition)*. Addison-Wesley Pub Co, January 15, 2000. 26, 27

[46] M. Minsky and Papert. *Perceptron-An Essay in Computational Geometry*. MIT Press, 1969. 14

[47] B. Chopard M. Oussaidene R. Schirru P. Pictet, M. Dacorogna and M. Tomassini. Using genetic algorithms for robust optimization in financial applications. to appear, Neural network World, 1995. 15

[48] C. B. Lucasius P. van Bommel and T. P. van der Weide. Genetic Algorithms for Optimal Logical Database Design. *Information and Software Technology*, 36(12):725–732, 1994. 24

[49] C. B. Lucasius P. van Bommel and T.P. van der Weide. Pool Heuristics in Evolutionary Database Optimization. In N. Prakash, editor, *Proceedings of the International Conference on Information Systems and Management of Data (CISMOD 94)*, pages 76–90, Madras, India, 1994. 24

[50] B. Porter. Genetic design of control systems. *Transactions of the Society of Instrument and Control Engineers (Japan)*, 34(5):393–402, 1995. 15

[51] D. Powell and M. M. Skolnik. Using genetic algorithms in engineering design optimization with non-linear constriants. In *Proceedings of fifth Int. Conf. on Genetic Algorithms*. S. Forrest (Editor), Morgan Kaufmann, CA, 424, 1993. 15

[52] E. A. Feigenbaum R. K. Lindsay, B. G. Buchanan and J. Lederberg. *Application of Artificial Intelligence for Chemistry: The DENDRAL Project*. New York: McGraw-Hill, 1980. 14

[53] M. Ross R. Kimball, L. Reeves and W. Thornthwaite. *The Data Warehouse Lifecycle Toolkit*. Robert Ipsen, 1998. 7

[54] I. Rechenberg. *Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution*. Frommann-Holzboog, Stuttgart, 1973. 14, 17

[55] R. G. Reynolds. Cultural algorithms: Theory and applications. In David Corne, Marco Dorigo, and Fred Glover, editors, *New Ideas in Optimization*, pages 367–377. McGraw-Hill, London, 1999. 19

[56] S. G Roberts and M. Turega. Evolving neural networks: an evolution of encoding techniques. In *Proceedings Int. Conf. on Artificial Neural Nets and Genetic Algorithms*. D. W. Pearson, N. C. Steele and R. F. Albrecht(Editors), Springer-Verlang, 96, 1995. 15

[57] F. Rosenblatt. *Principles of Neurodynamics*. Spartan, 1961. 14

[58] P. Ross and D. Corne. Applications of genetic algorithms. *AISB Quarterly*, (89):23–30, 1994. 15

[59] T. A. Standish. *Data Structures in Java^{TM}*. Addison Wesley, 1997. 27, 28

[60] F. Hoffmeister T. Back and H. Schwefel. A survey of evolution strategies, 1991. 17

[61] D.M. Tsou and P.C. Fischer. Decomposition of a relation scheme into boyce-codd normal form. *ACM-SIGACT*, 14(3):23–29, Summer 1982. 22, 23

[62] A. M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950. 13

[63] P. van Bommel. A randomised schema mutator for evolutionary database optimisation. *Australian Computer Journal*, 25(2):61–69, 1993. 24

[64] P. van Bommel. Database Design Modifications based on Conceptual Modelling. In H. Jaakkola, H. Kangassalo, T. Kitahashi, and A. Márkus, editors, *Information Modelling and Knowledge Bases V: Principles and Formal Techniques*, pages 275–286, Amsterdam, The Netherlands, 1994. IOS Press. 24

[65] P. van Bommel. Experiences with EDO: An evolutionary database optimizer. *Data Knowledge Engineering*, 13(3):243–263, 1994. 24

[66] P. van Bommel. Implementation Selection for Object-Role Models. In T. A. Halpin and R. Meersman, editors, *Proceedings of the First International Conference on Object-Role Modelling (ORM-1)*, pages 103–112, Magnetic Island, Australia, 1994. 24

[67] P. van Bommel, G. Kovacs, and A. Micsik. Transformation of database populations and operations from the conceptual to the internal level. *Information Systems*, 19(2):175–191, 1994. 24

[68] P. van Bommel and T. P. van der Weide. Reducing the search space for conceptual schema transformation. *Data Knowledge Engineering*, 8:269–292, 1992. 24

[69] P. van Bommel and T. P. van der Weide. Towards Database Optimization by Evolution. In A. K. Majumdar and N. Prakash, editors, *Proceedings of the International Conference on Information Systems and Management of Data (CISMOD 92)*, pages 273–287, Bangalore, India, 1992. 24

[70] D. Wickert. Physical database design using a genetic algorithm approach. 23

[71] J. Stokes Z. Tari and S. Spaccapietra. Object normal forms and dependency constraints for object-oriented schemata. In *ACM Transactions on Database Systems*, volume 22, pages 513–569, December 1997. 7

[72] L. A. Zadeh. Knowledge representation in fuzzy logic. In R. R. Yager and L. A. Zadeh, editors, *An Introduction to Fuzzy Logic Applications in Intelligent Systems*, pages 1–25. Kluwer, Boston, 1992. 14

Appendix

The following appendix is a listing of all data-files used in the GDNA experiments.

Gender.txt

name	gender
A	M
B	M
C	M
D	M
E	M
F	M
G	F
H	F
I	F
J	F
K	F

Employee.txt

employeenum	name	idnumber	job	dnumber	dname
1	Jacob	12345789	Programmer	54	IT
2	Edwin	98765321	Programmer	54	IT
3	Gordon	45679123	Consultant	12	HR
4	Graham	89234567	Manager	12	HR
5	Henry	56789123	Analyst	54	IT

Customer.txt

customer_id	firstname	lastname	city	state	zipcode	gender	married
65	Anderson	Anderson	Burlington	Vermont	15487	M	Married
104	Anderson	Anderson	Laramie	Wyoming	19367	M	Married
290	Anderson	Anderson	Salt Lake City	Utah	38559	M	Married
368	Anderson	Anderson	Madison	Wisconsin	45739	M	Married
430	Anderson	Anderson	Dallas	Texas	50800	M	Married
500	Anderson	Anderson	Seattle	Washington	57621	M	Married
503	Anderson	Anderson	Birmingham	Alabama	57750	M	Married
797	Anderson	Anderson	Charleston	West Virginia	81336	M	Married
835	Anderson	Anderson	Washington	DC	85208	M	Married
957	Anderson	Anderson	Richmond	Virginia	96972	M	Single