The advantages and cost effectiveness of database improvement methods

This item was submitted to Loughborough University's Institutional Repository by the/an author.

Additional Information:

- A Doctoral Thesis. Submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy of Loughborough University.

Metadata Record: [https://dspace.lboro.ac.uk/2134/19225](https://dspace.lboro.ac.uk/2134/19225)

Publisher: © Abdulaziz Alkandari

Rights: This work is made available according to the conditions of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0) licence. Full details of this licence are available at: [https://creativecommons.org/licenses/by-nc-nd/4.0/](https://creativecommons.org/licenses/by-nc-nd/4.0/)

Please cite the published version.
Pilkington Library

Author/Filing Title ..............................................................

..............................................................

Vol. No. ............... Class Mark ........................................

Please note that fines are charged on ALL overdue items.

FOR REFERENCE ONLY

0402697766
The Advantages and Cost Effectiveness of Database Improvement Methods

By

Abdulaziz Alkandari

A Doctoral Thesis

Submitted in partial fulfilment of requirement for the award for doctor of Philosophy of Loughborough University

February 2002

© by Abdulaziz Alkandari 2002
Acknowledgments

Words are not enough to express my sincere gratitude to Mr. Ray Dawson, my supervisor, for his invaluable help, support, guidance and friendship during the course of my PhD work. I would like to thank Dr. Chris Hinde, my director of research, for his advise especially during the first year of this course.

My thanks goes to the Public Authority for Applied Education and Training (PAAET) for its financial support. Many thanks to Mr. Faisal Al-Qemlas, Ajeel Al-Toog, Khalid Sardar, Sarwat, and Kriss Sutherns for their help in gathering data needed for this research. Special thanks to Dr. Akmal Choudahry for his continuous help and advise.

Thanks are given to my parents, family, and friends for their encouragements. Special thanks goes to Dr. Thomas Jackson, my fellow colleague in the Computer Science Dept. for his continuous support and help.

Finally I would like to thank my fellow colleagues Dr. Waleed Ben Salama, Dr. Asad Al-Zaid, Abdulaziz Al-Roumi, Saleh Alenzi, Ahmad Al-Sharrah, Dr. Zahid Bin Ali Asghar, Mazhar Ali and Dr. Salman Ahmad for their moral support and kindness.
Relational databases have proved inadequate for supporting new classes of applications, and as a consequence, a number of new approaches have been taken (Blaha 1998), (Harrington 2000). The most salient alternatives are de-normalisation and conversion to an object-oriented database (Douglas 1997). De-normalisation can provide better performance but has deficiencies with respect to data modelling. Object-oriented databases can provide increased performance efficiency but without the deficiencies in data modelling (Blaha 2000).

Although there have been various benchmark tests reported, none of these tests have compared normalised, object oriented and de-normalised databases.

This research shows that a non-normalised database for data containing type code complexity would be normalised in the process of conversion to an object-oriented database. This helps to correct badly organised data and so gives the performance benefits of de-normalisation while improving data modelling.

The costs of conversion from relational databases to object oriented databases were also examined. Costs were based on published benchmark tests, a benchmark carried out during this study and case studies. The benchmark tests were based on an engineering database benchmark. Engineering problems such as computer-aided design and manufacturing have much to gain from conversion to object-oriented databases. Costs were calculated for coding and development, and also for operation. It was found that conversion to an object-oriented database was not usually cost effective as many of the performance benefits could be achieved by the far cheaper process of de-normalisation, or by using the performance improving facilities provided by many relational database systems such as indexing or partitioning or by simply upgrading the system hardware.
Abstract

It is concluded therefore that while object oriented databases are a better alternative for databases built from scratch, the conversion of a legacy relational database to an object oriented database is not necessarily cost effective.

Keywords:
Databases, Object-Oriented, Relational, Normalisation, De-normalisation, Performance, Benchmarking, Cost,
Contents

Chapter 1: Introduction
  1.1. The Research Problem
  1.2. Research Aims
  1.3. Research Objectives
  1.4. Structure of this Thesis

Chapter 2: Methodology
  2.1. Research Philosophy
    2.1.1. Positivism
    2.1.2. Interpretivism
    2.1.3. Discussion and Rationale for Choice of Approach
  2.2. Research Approach
  2.3. Research Strategy
    2.3.1. Case Study Research
  2.4. Summary

Chapter 3: Theory and Literature Review
  3.1. History of Database Systems
  3.2. Definition of database
  3.3. Filing system
  3.4. Network and Hierarchical Models
    3.4.1. Advantages of Network and Hierarchical Models
    3.4.2. Limitations
  3.5. The Relational model
    3.5.1. Advantages
    3.5.2. Limitations
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6</td>
<td>Object-Oriented model</td>
<td>25</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Advantages</td>
<td>28</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Main characteristics of Object-Oriented Systems</td>
<td>33</td>
</tr>
<tr>
<td>3.6.2.2</td>
<td>Encapsulation</td>
<td>33</td>
</tr>
<tr>
<td>3.6.2.3</td>
<td>Object identity</td>
<td>34</td>
</tr>
<tr>
<td>3.6.2.4</td>
<td>Classes</td>
<td>34</td>
</tr>
<tr>
<td>3.6.2.5</td>
<td>Inheritance</td>
<td>35</td>
</tr>
<tr>
<td>3.6.2.6</td>
<td>Overriding and late binding</td>
<td>35</td>
</tr>
<tr>
<td>3.6.2.7</td>
<td>Extensibility</td>
<td>36</td>
</tr>
<tr>
<td>3.6.3</td>
<td>Limitations</td>
<td>36</td>
</tr>
<tr>
<td>3.7</td>
<td>The Object-Oriented Model Versus the Relational Model</td>
<td>37</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Dealing with complex objects</td>
<td>38</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Object identity versus primary key</td>
<td>39</td>
</tr>
<tr>
<td>3.7.3</td>
<td>Storing programs and data</td>
<td>39</td>
</tr>
<tr>
<td>3.7.4</td>
<td>Typing and inheritance</td>
<td>40</td>
</tr>
<tr>
<td>3.8</td>
<td>Review of Existing Object-Oriented Database Systems</td>
<td>40</td>
</tr>
<tr>
<td>3.8.1</td>
<td>ODE</td>
<td>41</td>
</tr>
<tr>
<td>3.8.2</td>
<td>CLOSQQL</td>
<td>41</td>
</tr>
<tr>
<td>3.8.3</td>
<td>Oggetto</td>
<td>42</td>
</tr>
<tr>
<td>3.8.4</td>
<td>IDB Object Database</td>
<td>42</td>
</tr>
<tr>
<td>3.8.5</td>
<td>Avance (SYSLAB)</td>
<td>42</td>
</tr>
<tr>
<td>3.8.6</td>
<td>ConceptBase</td>
<td>43</td>
</tr>
<tr>
<td>3.8.7</td>
<td>COOL/COCOON</td>
<td>43</td>
</tr>
<tr>
<td>3.8.8</td>
<td>Encore</td>
<td>44</td>
</tr>
<tr>
<td>3.8.9</td>
<td>Exodus</td>
<td>44</td>
</tr>
<tr>
<td>3.8.10</td>
<td>MOOD4-PC</td>
<td>45</td>
</tr>
<tr>
<td>3.8.11</td>
<td>The Object System of STONE – OBST</td>
<td>45</td>
</tr>
<tr>
<td>3.8.12</td>
<td>OTGen</td>
<td>46</td>
</tr>
<tr>
<td>3.8.13</td>
<td>VODAC</td>
<td>46</td>
</tr>
<tr>
<td>3.8.14</td>
<td>ArtBASE (commercial system)</td>
<td>47</td>
</tr>
<tr>
<td>3.8.15</td>
<td>EasyDB</td>
<td>47</td>
</tr>
<tr>
<td>3.8.16</td>
<td>GemStone</td>
<td>47</td>
</tr>
</tbody>
</table>
### Chapter 4: Review of OO Improvement Methods for Legacy Database Systems

#### 4. Introduction

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.</td>
<td>Transforming a Relational Database to an OO System</td>
<td>52</td>
</tr>
<tr>
<td>4.2.</td>
<td>Using existing relational schema</td>
<td>53</td>
</tr>
<tr>
<td>4.3.</td>
<td>Transforming Relational Database Schemas into OO Schemas according to ODMG-93</td>
<td>54</td>
</tr>
<tr>
<td>4.4.</td>
<td>Connecting Objects To A Relational Database</td>
<td></td>
</tr>
<tr>
<td>4.4.1.</td>
<td>Ad hoc SQL bound to object</td>
<td>56</td>
</tr>
<tr>
<td>4.4.2.</td>
<td>Using a 4GL (Fourth Generation Language)</td>
<td>56</td>
</tr>
<tr>
<td>4.4.3.</td>
<td>Direct object mapping</td>
<td>58</td>
</tr>
<tr>
<td>4.4.4.</td>
<td>Bridging Object-Oriented programming to relational databases</td>
<td>59</td>
</tr>
<tr>
<td>4.5.</td>
<td>Interfacing Objects with the Relational DBMS</td>
<td>60</td>
</tr>
<tr>
<td>4.5.1.</td>
<td>COMMONBASE</td>
<td>60</td>
</tr>
<tr>
<td>4.5.2.</td>
<td>SMALLTALK/SQL</td>
<td>60</td>
</tr>
<tr>
<td>4.5.3.</td>
<td>POWERBUILDER</td>
<td>61</td>
</tr>
<tr>
<td>4.5.4.</td>
<td>NEXT’S DBKIT</td>
<td>62</td>
</tr>
<tr>
<td>4.5.5.</td>
<td>PERSISTENCE v.1.0</td>
<td>62</td>
</tr>
<tr>
<td>4.6.</td>
<td>Object-Relational DBMSs</td>
<td>63</td>
</tr>
<tr>
<td>4.7.</td>
<td>Summary</td>
<td>65</td>
</tr>
</tbody>
</table>

#### Chapter 5: The Claimed Advantages of Conversion to OO Databases

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.</td>
<td>Introduction</td>
<td>67</td>
</tr>
</tbody>
</table>
Chapter 6: An In-Depth Example to Illustrate the Advantages Of Conversion To O-O Databases

6. Introduction

6.1. Converting a Non-Normalised Relational Database to an Object-Oriented Database

6.2. The problem


Chapter 7: Overview of Existing Benchmarks

7. Introduction

7.1. Wisconsin Benchmark
Contents

11.4. Future research ....................................................... 164
References ................................................................. 174
Appendix A ................................................................. 174
Appendix B ................................................................. 183
Appendix C ................................................................. 192
Appendix D ................................................................. 204
Appendix E ................................................................. 211
Appendix F ................................................................. 217
1.1. The Research Problem

Until recently, the relational database model dominated the world of database systems. Data models must continuously evolve to be able to deal with the increasing complexity of information processing environments. Thus, it would be reasonable to expect that research on the next-generation of database systems would concentrate on selecting the most suitable database systems. However, database researchers and designers of relational Database Management Systems (DBMSs) face a complex choice problem today.

The success of the relational systems attracted the interest of users from new application domains. New users coming from different research areas (graphics, engineering) have realised the need to put some order to the unusual data they need to manipulate. It is the data of these application domains that require advanced database techniques for modelling and management (Blaha, 1998). These special applications, sometimes called complex or advanced, (Gardarin 1984) such as GIS (Geographical Information Systems), CAD (Computer-Aided Design), CAM (Computer-Aided Manufacturing), and CASE (Computer-Aided Software Engineering) have had a major impact on opening up the usage of computer hardware (multimedia) but also posing complex problems of data storage and representation. There is a need for the data of such systems to be persistent from one running of a program to the next, and the complexity of these data with many-to-many relationships gives storage problems for traditional database systems (Harrington, 2000). This is explained further in chapter 6.

A response to the requirements of these applications has been the development of a range of database systems that have been routed in varying ways in the emerging Object-Oriented technology. Object-Oriented databases have been described as being the next generation of database systems (Cattell,
In the commercial world, object-oriented databases are understood as systems that support a persistent object store for C++ Java or similar programs. The underlying philosophy of these systems is that a single object-oriented persistent programming language that supports procedural manipulation of data can resolve the differences between programming languages and databases.

Developers expect object-oriented tools and techniques (encapsulation, inheritance, object identity, etc.) to simplify designing and implementing newer real world problems compared with those based on the relational model (Blaha 1998). The relational model has made a great impact on the database market by offering a better way to manage applications than earlier approaches (Delobel 1995). The turn is for the object-oriented model to present its significant potential for databases.

This research focuses on two issues. Firstly, the relational model lacks the ability to model and manage complex data efficiently. It has difficulty in dealing with complex data such as recognising identification in data, accessing data using traversals, many-to-many relationships, and frequent use of type codes. This research looks at the possible advantages of moving to object-oriented databases. Secondly, the normalised relational database with complex data suffers from performance issues, which lead developers to look for alternative improvement methods such as de-normalisation (Douglas, 1997). This research seeks to answer the question whether a move to object-oriented databases is worthwhile for performance reasons alone. As the demand for Object-Oriented Databases (OODB) rises and the popularity of the relational database remains, this research intends to establish guidelines as to when each system should be used.

1.2. Research Aims

The aims of this research are:

- To determine what advantages may be gained from converting data from relational to an object-oriented schema for complex real world problems.
Chapter 1

Introduction

❖ To determine, by the application of suitable benchmark tests, the relative performance advantages for databases based on each database model/type.

❖ To discover which database systems give the best-cost savings in the short term and long term.

1.3. Research Objectives

❖ To evaluate the advantages and disadvantages of a non-normal complex relational database compared with an object-oriented database used for the same problem. By using suitable examples the comparison will show the weakness and strength of each model. The research will also show ways to convert a relational database to the object-oriented model where possible and necessary and the worked example will show the benefits in doing so.

❖ To evaluate, using benchmark tests, the difference in performance between a non-normalised relational database and a normalised one. Different published benchmark test will be examined to determine suitable tests for relational database systems and object-oriented database systems. The results can help in assessing each method to give a concise indication of the profitability of selecting either of the two methods. Finally, comparing the result with the object-oriented method would give the best of all solutions.

❖ To look at existing relational database systems in several government and business Information Technology departments, to show how using the above techniques can improve the performance of their systems. Case studies that involve organisations that have converted from using relational database systems to object-oriented systems will be examined. Other case studies where organisations have used de-normalisation to improve system performance will also be studied. Calculation of the cost of improving these systems will be used to compare the alternatives of de-normalising, converting to object-oriented databases or buying bigger and faster computers.
1.4. Structure of this Thesis

- **Chapter 1** - Starts with defining the problem of the new technology and the new human demand, then shows the aims and objectives of the research and the method used in this research to get as close as possible to the optimum solution.

- **Chapter 2** - Presents a literature review of major research methodologies. This chapter shows what was the author’s approach in his research and how the use of this methodology contributed towards the aims and objectives.

- **Chapter 3** - Presents a general literature survey of legacy databases. This chapter shows in brief the advantages and limitations of earlier and current database models to show the published claimed advantages of each system. The history of object-oriented modelling is presented to show how object-oriented modelling has been developed to give even greater advantages over other systems. The more significant research and advances in commercial object-oriented databases are identified. To satisfy the first aim of this research, the chapter ends by comparing the relational with the object-oriented models using selected criteria such as dealing with complex data, object identity versus primary key, storing program and data, and inheritance. Finally, the chapter gives a detailed review of existing object-oriented database systems with the purpose of showing whether commercial and research systems really do give the advantages claimed for object-oriented systems.

- **Chapter 4** - Reviews object-oriented improvement methods for legacy database systems to show how an organisation with legacy relational database systems can gain the advantage of object-oriented systems.

- **Chapter 5** - Shows the most notable claimed advantages of converting a relational database to object-oriented database.

- **Chapter 6** - Brings out the new advantage of converting a non-normalised relational database with complex data to an object-oriented database. This chapter shows that conversion informally normalises the non-normalised database up to
third normal form, which makes conversion to an object-oriented database easier to perform and a possible solution for a wider range of problems.

- **Chapter 7** - Shows different benchmark tests of early relational databases from the Wisconsin benchmark in 1981 up to the engineering object-oriented benchmark in 1991, which the purpose of establishing a suitable benchmark test for the comparison of performance of normalised and non-normalised databases.

- **Chapter 8** - Quantifies the difference in performance between the normalised and de-normalised database management systems as determined from the results of the benchmark tests.

- **Chapter 9** - Presents a methodology for comparing the costs of de-normalising a relational database with converting to an object-oriented database with the purpose of improving performance. The methodology calculates the break-even point then each method would give an overall cost advantage. The methodology then compares these costs with that of simply upgrading the hardware to a bigger and faster computer to obtain the necessary performance advantage.

- **Chapter 10** - Looks into existing improvement methods used in current information technology departments of mixed sectors (commercial and governmental). Real organisation metrics are gathered to enable a calculation to be made as to whether de-normalisation or changes to object-oriented systems are cost effective on the grounds of performance improvement.

- **Chapter 11** - Presents the conclusions of the research and examines possible directions for further work. The research is evaluated with respect to the initial aims and objectives and further work is proposed that would make additional contributions towards the research aims.
Chapter 2: Methodology

2. Introduction

The way in which research is conducted may be conceived in terms of the research philosophy subscribed to and the research strategy employed in the pursuit of the research objectives and the quest for the solution to the research question. The research question and research objectives have been outlined in Chapter 1. The purpose of this chapter is to:

- Discuss the research philosophy in relation to other philosophies.
- Explain the research strategy, including the research methodologies adopted.

2.1. Research Philosophy

A research philosophy is a belief about the way in which data about a phenomenon should be gathered, analysed and used. The term epistemology (what is known to be true) as opposed to doxology (what is believed to be true) encompasses the various philosophies of research approach. The purpose of science, then, is the process of transforming things believed into things known: doxa to episteme. Two major research philosophies have been identified in the tradition of science, namely positivist (sometimes called scientific) and interpretivist (also known as anti-positivist)(Galliers, 1992).

2.1.1. Positivism

Positivists believe that reality is stable and can be observed and described from an objective viewpoint (Levin 1988), i.e. without interfering with the phenomena being studied. They contend that phenomena should be isolated and that observations should be repeatable. This often involves manipulation of reality with variations in only a single independent variable so as to identify regularities.
Chapter 2

Methodology

in, and to form relationships between, some of the constituent elements of the social world. Predictions can be made on the basis of the previously observed and explained realities and their inter-relationships. Positivism has also had a particularly successful association with the physical and natural sciences.

There has, however, been much debate on the issue of whether or not this positivist paradigm is entirely suitable for the social sciences (Hirschheim 1985), many authors calling for a more pluralistic attitude towards Information Systems research methodologies (Kuhn 1970), (Bjørn-Andersen 1985), (Remenyi and Williams 1996). Indeed, some of the difficulties experienced in IS research, such as the apparent inconsistency of results, may be attributed to the inappropriateness of the positivist paradigm for the domain. Likewise, some variables or constituent parts of reality might have been previously thought unmeasurable under the positivist paradigm, and hence went unresearched (Galliers 1992).

2.1.2. Interpretivism

Interpretivists contend that only through the subjective interpretation of and intervention in reality can reality be fully understood. The study of phenomena in their natural environment is key to the interpretivist philosophy; together with the acknowledgement that scientists cannot avoid affecting those phenomena they study. They admit that there may be many interpretations of reality, but maintain that these interpretations are in themselves a part of the scientific knowledge they are pursuing.

2.1.3. Discussion and Rationale for Choice of Research Philosophy

Both research traditions start in classical Greek times with Plato and Aristotle (positivists) on the one hand, and the Sophists (anti-positivists) on the other. Well known positivists have included Bacon, Descartes, Mill, Durkheim, Russell and Popper. On the opposing side there are Kant, Hegel, Marx, Freud, Polanyi and Kuhn (Hirschheim 1985).
Vreede (1995) observes that, in both organisation science and information systems research, interpretive research used to be the norm, at least until the late 1970s. Since that time, however, the positivist tradition has taken a firm hold (Dickson and DeSanctis 1990), Orlikowski and Baroudi (1991) noting that 96.8% of researchs in the leading US IS journals conform to this theory. It has often been observed (e.g. Benbasat, Goldstein and Mead 1987) very accurately that no single research methodology is intrinsically better than any other methodology, many authors calling for a combination of research methods in order to improve the quality of research (e.g. Kaplan and Duchon 1988). Equally, some institutions have tended to adopt a certain "house style" methodology (Galliers 1992); this seems to be almost in defiance of the fact that, given the richness and complexity of the real world, a methodology best suited to the problem under consideration, as well as the objectives of the researcher, should be chosen (Benbasat 1984), (Pervan 1994).

The author's overriding concern is that the research undertaken should be relevant to the research aims, as set out in Chapter 1. The author has chosen the positivist philosophy, which seems appropriate for this purpose, i.e. the finding of a better ways to store and retrieve data in a particular information system. The author's research is concerned with capturing information from a laboratory in a scientific approach rather than from a social environment, which leads the author to select the positivist philosophy rather than the interpretivist philosophy. However, there is an element of interpretivism in the case studies used, as some subjective interpretation of the findings is necessary.

2.2. Research Approach

There are many different combinations of research approaches that could have been adopted and used as a framework to undertake the planned research experiments. The author has considered three broad styles of research approach. The three approaches are as follows:
Chapter 2 Methodology

- Constructive research methods
  - Conceptual development
  - Technical development

- Nomothetic research methods
  - Formal-mathematical analysis
  - Experiments, laboratory and field
  - Field studies and surveys

- Idiographic research methods
  - Case studies
  - Action research

The constructive approach is concerned with developing frameworks, refining concepts or pursuing technical developments. The approach allows models and frameworks to be created that do not describe any existing reality or do not necessarily have any "physical" realisation (Cornford and Smithson 1996). With the case of the author's planned research to be carried out in the comparative world of databases there are some requirements to create an artificial framework to capture the data.

Nomothetic research is concerned with exploring empirical data in order to test hypotheses of a general character about phenomena studied. Nomothetic research is concerned with a search for, and evidence to support, general laws or theories that will cover a whole class of cases. Such research emphasises systematic protocols and hypothesis testing within the scientific tradition. Thus, as discussed in this chapter the author's research is concerned with the positivist approach, which closely link to scientific experiments (Cornford and Smithson 1996).
In contrast to the constructive and nomothetic approaches, idiographic research is concerned with exploring particular cases or events and providing the richest picture of what transpires. The aim is to understand a phenomenon in its own, particular, context. Idiographic research emphasizes the analysis of subjective accounts based on participation or close association with everyday events. Within information systems there is a strong tradition of case studies, which might be seen as examples of idiographic research. In the case of the author's proposed research, case studies are likely to play a major role due to their non-restrictive variable approach, which means that the idiographic approach is suitable to be included in the framework for the author's research (Cornford and Smithson 1996). However, with the research approach decided, the different kinds of methodologies need to be considered to form a research strategy.

2.3. Research Strategy

A large number of research methodologies have been identified, Galliers (1992) for example listing fourteen, while Alavi and Carlson (1992), reported in Pervan (1994b), use a hierarchical taxonomy with three levels and eighteen categories. Table 3.1 presents a list of methodologies identified by Galliers (1992, p.149), indicating whether they typically conform to the positivist or interpretivist paradigms. Table 3.1 also shows, indicated by ticks, the research approach the author intends to use for his research. Before introducing the methodologies used in this research, the author will summarise the key features of the key methodologies in the table, identifying their respective strengths and weaknesses. In the following sections, the author will justify the choice of methodologies and explain how they operate.
Chapter 2 Methodology

<table>
<thead>
<tr>
<th>Scientific/Positivist</th>
<th>Interpretivist/Anti-positivist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laboratory Experiments</td>
<td>√ Subjective/Argumentative</td>
</tr>
<tr>
<td>Field Experiments</td>
<td>Reviews</td>
</tr>
<tr>
<td>Surveys</td>
<td>Action Research</td>
</tr>
<tr>
<td>Case Studies</td>
<td>√ Case Studies</td>
</tr>
<tr>
<td>Theorem Proof</td>
<td>Descriptive/Interpretive</td>
</tr>
<tr>
<td>Forecasting</td>
<td>Futures Research</td>
</tr>
<tr>
<td>Simulation</td>
<td>Role/Game Playing</td>
</tr>
</tbody>
</table>

Table 2-1 A Taxonomy of Research Methodologies

Laboratory experiments permit the researcher to identify precise relationships between a small number of variables that are studied intensively via a designed laboratory situation using quantitative analytical techniques with a view to making generalisable statements applicable to real-life situations. The key weakness of laboratory experiments is the "limited extent to which identified relationships exist in the real world due to oversimplification of the experimental situation and the isolation of such situations from most of the variables that are found in the real world" (Galliers, 1992, p.150).

Field experiments extend laboratory experiments into real organisations and their real life situations, thereby achieving greater realism and diminishing the extent to which situations can be criticised as contrived. In practice it is difficult to identify organisations that are prepared to be experimented on and still more difficult to achieve sufficient control to make replication viable.

Surveys enable the researcher to obtain data about practices, situations or views at one point in time through questionnaires or interviews. Quantitative analytical techniques are then used to draw inferences from this data regarding existing relationships. The use of surveys permits a researcher to study more variables at one time than is typically possible in laboratory or field experiments, whilst data can be collected about real world environments. A key weakness is that it is very difficult to realise insights relating to the causes of or processes involved in the phenomena measured. There are, in addition, several sources of
bias such as the possibly self-selecting nature of respondents, the point in time when the survey is conducted and in the researcher him/herself through the design of the survey itself.

Case studies involve an attempt to describe relationships that exist in reality, very often in a single organisation. Case studies may be positivist or interpretivist in nature, depending on the approach of the researcher, the data collected and the analytical techniques employed. Reality can be captured in greater detail by an observer-researcher, with the analysis of more variables than is typically possible in experimental and survey research. Case studies can be considered weak as they are typically restricted to a single organisation, and it is difficult to generalise findings since it is hard to find similar cases with similar data that can be analysed in a statistically meaningful way. Furthermore, different researchers may have different interpretations of the same data, thus adding research bias into the equation.

Simulation involves copying the behaviour of a system. Simulation is used in situations where it would be difficult normally to solve problems analytically, and it typically involves the introduction of random variables. As with experimental forms of research, it is difficult to make a simulation sufficiently realistic so that it resembles real world events.

Forecasting/futures research involves the use of techniques such as regression analysis and time series analysis to make predictions about likely future events. It is a useful form of research in that it attempts to cope with the rapid changes that are taking place in IT and to predict the impacts of these changes on individuals, organisations or society. However, it is a method that is fraught with difficulties relating to the complexity of real-world events, the arbitrary nature of future changes and the lack of knowledge about the future. Researchers cannot build true visions of the future, but only scenarios of possible futures.
Subjective/argumentative research, for example hermeneutics and phenomenology, requires the researcher to adopt a creative or speculative stance rather than act as an observer. It is a useful technique since new theories can be built, and new ideas can be generated and subsequently tested. However, as an unstructured and subjective form of research, there is a strong chance of researcher bias.

Action research is a form of applied research where the researcher attempts to develop results or a solution that is of practical value to the people with whom the researcher is working, at the same time developing theoretical knowledge. Through direct intervention in problems, the researcher aims to create practical outcomes while also aiming to re-inform existing theory in the domain studied. As with case studies, action research is usually restricted to a single organisation, making it difficult to generalise findings, while different researchers may interpret events differently. The personal ethics of the researcher are critical, since the opportunity for direct researcher intervention is always present. A summary of the different research approaches has been given in Table 2-2.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Key Features</th>
<th>Strengths</th>
<th>Weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laboratory Experiments</td>
<td>Identification of precise relationships between chosen variables via a designed laboratory situation, using quantitative analytical techniques, with a view to making general statements applicable to real-life situations</td>
<td>The solutions and control of a small number of variables which may then be studies intensively</td>
<td>The limited extent to which identified relationships exist in the real world due to oversimplification of the experimental situation and the isolation of such situations from most of the variables that are found in the real world.</td>
</tr>
<tr>
<td>Field Experiments</td>
<td>Extension of laboratory experiments into the real-life situations of organisations and/or society.</td>
<td>Greater realism; less artificial/sanitised than laboratory situation</td>
<td>Finding organisations prepared to be experimented on. Achieving sufficient control to enable replication, with only the study variables being altered</td>
</tr>
</tbody>
</table>
### Chapter 2

#### Methodology

| Surveys | Obtaining snapshots of practices, situations or views at a particular point in time (via questionnaires or interviews) from which inferences are made (using quantitative analytical techniques) regarding the relationships that exist in the past, present and future. | Greater number of variables may be studied than in the case of experimental approaches. Descriptions of real world situations. More easy / appropriate generalisations. | Likely that little insight is obtained relating to the causes/processes behind the phenomena being studied. Possible bias in respondents. |
| Case Studies | An attempt at describing the relationships, which exist in reality, usually within a single organisation or organisational grouping. | Capturing ‘reality’ in greater detail and analysing more variables than is possible using any of the above approaches. | Restriction to a single event / organisation. Difficulty in generalising, given problems of acquiring similar data from a statistically meaningful number of cases. Lack of control variables. Different interpretations of events by individual researchers. |
| Simulation | An attempt at copying the behaviour of a system that would otherwise be difficult/impossible to solve analytically, by the generation/introduction of random variables. | Provision of an opportunity to study situations that might otherwise be impossible to analyse. | Similar to experimental research in regard to the difficulties associated with devising a simulation that accurately reflects the real world situations. |
| Subjective argumentative research | Creative research based more on opinion / speculation than observation. | Useful in building theory that can be subsequently tested. | Unstructured, subjective nature of research process. A likelihood of biased interpretations. |
| Forecasting/futures research | Use of such techniques as regression analysis and time series analysis to deduce possible events | Provision of insights into likely future occurrences in situations where existing relationships may not hold true in the future. | Complexity and changing relationship of variables under study. Lack of real knowledge of future events. |

Table 2-2 A summary of the key features, strengths and weaknesses of alternative information systems research approaches (Galliers Robert, 1992)
Chapter 2 Methodology

The author as part of his research is going to use the case study approach, which has been defined below along with the justification for using the approach for his research.

2.3.1. Case Study Research

There are a number of important articles describing the case study approach to research. Key among these is Benbasat et al.'s (1987) paper with its comprehensive definitions and suggestions for the conduct of case research. The case study is considered by Benbasat et al. (1987, p.370) to be viable for three reasons.

- It is necessary to study the phenomenon in its natural setting.
- The researcher can ask "how" and "why" questions, so as to understand the nature and complexity of the processes taking place.
- Research is being conducted in an area where few, if any, previous studies have been undertaken.

Case studies are defined in various ways, and a standard does not exist. However, a definition compiled from a number of sources (Stone E. 1978), (Benbasat 1984), (Yin 1984), (Bonoma 1985) and (Kaplan 1985) in Benbasat et al. (1987, p.370), is as follows.

A case study examines a phenomenon in its natural setting, employing multiple methods of data collection to gather information from one or a few entities (people, groups or organisations). The boundaries of the phenomenon are not clearly evident at the outset of the research and no experimental control or manipulation is used.

When deciding whether to use the case study approach or not, there are a number of factors to consider. If there is a need to focus on contemporary events or phenomena in a natural setting, clearly the case study is advantageous. The same is also true if there is no strong theoretical base for the research, i.e. if it is a
theory building research project. "A rich and natural setting can be fertile ground for generating theories" (Benbasat, Goldstein and Mead 1987). However, if there were a need for control or manipulation of variables, then the case study would not be appropriate. It is important to clarify the need that should relate to the nature of the problem rather than the inability of the researcher to undertake research with a particular methodology. Within the case study approach there are a number of variations. A key feature of the design of case study research is the number of case studies that can be included in a project. Generally speaking it is better, i.e. more valid and generalisable, to include multiple cases, though there are instances where a single case is instructive (see e.g. Lee 1989). Exploratory studies are generally better served by single cases, i.e. where there is no previous theory. A single case can also be used to test an existing, well-formed theory. Multiple cases are preferable when the purpose of the research is to describe phenomena, and to develop and test theories. Multiple cases also permit cross-case analysis, a necessary feature for widespread generalisation of theories. The sites or locations where cases are to be conducted should be chosen with great care. As has already been indicated, case studies require multiple data collection methods, the results of which hopefully converge, in order to establish construct validity. Yin (1984, p.78) identifies these methods as including:

- Direct observation of activities and phenomena and their environment;
- Indirect observation or measurement of process-related phenomena;
- Interviews —(structured or unstructured);
- Documentation, such as written, printed or electronic information about the company and its operations (also newspaper cuttings);
- Records and charts about previous use of technology relevant to the case.

The case study approach is the most suitable for the author's research as it can address the implementation of a new database system with new features such as (CASE, CAD, CAM applications) or the improvement of an existing database
system with high cost and low performance efficiency. It is also useful for topics and areas of study, like benchmarking of de-normalised databases versus object-oriented databases, which are novel or which have little theory as yet. The case study approach will help build the author’s theories in a relatively new area of research.

2.4. Summary

This chapter has presented a detailed account of the research philosophy, approach and strategy according to which the research has been conducted. The author has placed the research in the positivist camp, utilising a mixture of experimental and case study research approaches though the case studies also give an element of interpretation. The author only intends to use multiple case studies incorporating surveys, which is a more a positivist approach, but the framework to be used for the research is partially an idiographic one, due to the real environment cases of different information technology departments interviewed and studied. Finally, the research strategy has been determined by using previous literature describing case studies, which has been valuable in identifying the most important points of the case study methodology, as well as illustrating the weaknesses associated with earlier research.
Chapter 3: Theory and Literature Review

3. Introduction

This chapter will present a general literature survey of legacy and object-oriented databases. It also shows, in brief, the advantages and limitations of earlier and current database models showing the published claimed advantages of each system. The history of object-oriented modelling is presented to show how object-oriented modelling has been developed to give even greater advantages over other systems. The more significant research and advances in commercial object-oriented databases are identified. Finally, to satisfy the first aim of this research, the chapter compares the relational with the object-oriented models using selected criteria such as dealing with complex data, object identity versus primary key, storing program and data, and inheritance.

This chapter gives a detailed review of existing object-oriented database systems with the purpose of showing whether commercial and research systems really do give the advantages claimed for object-oriented systems.

3.1. History of Database Systems

The idea of database was conceived in the 1960s. It was during the period when the storage device technology evolved, that applications for managing large quantities of data were developed. Prior to the development of the first database management system (DBMS), application programs that accessed flat files provided access to data. These first systems had lack of integrity and inability to represent data relationships that in turn lead to the invention of data modelling. The first data model was the hierarchical data model (Harrington 2000).

3.2. Definition of database

A Database is a structured set of data items that can be accessed by the computer in order to satisfy several users simultaneously, within an appropriate time (Claude, 1995). Also it is a collection of data that represents information
concerning a certain real-world application; because of its size, it is usually held in secondary storage (Lausen 1997). The database field is concerned with the management of large amount of persistent, reliable and shared data. “Large” means too big to fit in conventional main memory, “Persistent” means that data persist from one session to another. “Reliable” means recoverable in case of hardware or software failures. “Sharable” means that several users should be able to access the data in an orderly manner (Bancilhon 1988).

### 3.3. Filing system

The first generation was file systems, such as ISAM and VSAM. These first systems held their data in files and used the standard access methods via programming languages such as Fortran, PL/I, Pascal, and Cobol (Delobel 1995). During this period considerable effort was put into improving the physical file supports and developing techniques for accessing files and records.

### 3.4. Network and Hierarchical Models

The first hierarchical DBMS -IMS and its DL/1 language- was developed by IBM and North American Aviation (Rockwell International) in the late 1960s (Elmasri 1994). In the hierarchical model data is organised in a tree structure. Each node in the tree corresponds to a class of entities in the real world and the arcs between the nodes represent the links between the objects.

Charles Bachman did early work on the network model during the development of the first commercial DBMS, IDS (Bachman and Williams, 1964) at General Electric and later at Honeywell. Bachman also introduced the earliest diagrammatic technique for representing relationships in database schemas, called data structure diagrams (Bachman, 1969) or Bachman diagrams (Elmasri, 1994).

The network model is an extension of the hierarchical model in which the graph of objects is not limited. It allows objects to be shared and represents cyclic links between objects. This model is used by CODASYL (derived from the Conference on Data System Language) systems. In the network model, a conceptual schema is composed of record definitions, which define the entities
and the links between those entities, and of set definitions, which express the multi-valued links between the records (Delobel 1995).

3.4.1. Advantages of Network and Hierarchical Models

These systems realised the sharing of an integrated database among many users within an application environment (Kim 1990). For example, the data that needed to be held in order to manage a company was becoming increasingly complex and more and more interdependent. The user of various data files wanted to integrate files and applications, which had previously been used independently. They also wanted to be able to represent more complex links between the various records.

The idea of the Database Management System (DBMS) started with this generation. A DBMS is the software package used to define and manipulate the database kept in the storage media. More specifically, the aims of the DBMS are:

- Links between data.
- Data consistency.
- Ease of access to data.
- Data security.
- Data independence.
- Performance.
- Administration and control.

3.4.2. Limitations

The tree structure of the graph of objects in the hierarchical model becomes limiting when the sharing of data needs to be modelled. The network model in a way resolves this limitation, but it makes it too complicated to understand and manipulate. The lack of data independence and the tedious navigational access to the database was another drawback of this model. These problems with the
hierarchical and network models lead to the development of relational database technology.

3.5. The Relational model

Dr. Codd first introduced the Relational Database Model (RDBM) when he presented a paper in a conference in 1970 (Codd 1970). The Relational Database System is based on a newer and a simpler approach to data organisation that was a major change from the earlier data models. In particular, the relationships do not become a part of a relational database. Only primary key-foreign key matches that are connected as needed by the join operator represent logical relationships. The relational model is therefore not navigational and provides superior ad hoc query support to any of the earlier data models (Lausen 1997). Mainly it satisfies the management of large, persistent, reliable, and, sharable data, which are the most essential aspects of database systems.

3.5.1. Advantages

There are many advantages of using the relational model, this thesis will only comment on the better known ones. One of the features of a RDBMS is the simplicity of the Structured Query Language (SQL) which is a Fourth Generation Language (4GL) used by most RDBMSs. Its independence of defining and manipulating data -using its Data Definition Language (DDL), and its Data Manipulation Language (DML)- gave the relational model the lead in introducing a systematic study of database principles.

The reason behind this leadership is the standardisation in designing and implementation of database systems. Mathematical methods such as relational algebra and relational calculus provide the theory that underpins the standardisation of relational database systems. Relational database technology is distinguished by its declarative query language that fulfils, to a great extent, the needs of business-type applications.

Many books (e.g. (Delobel 1995), (Elmasri 1994), (Vermulen 1996)) show different advantages but they all agree that a RDBMS gives the following:
A simple data-storage concept, tables, and a standard query language.

A logical separation of the database from application programs.

A powerful data consistency and security mechanism. There are several features relational databases provide to ensure that data remain consistent, secure, and synchronised. 1) Constraints and consistency checks: that is a mechanism, which causes an error when an SQL statement is issued if any of the consistency checks is violated. Also, 2) Triggers are used to define an action which happens to a table, such as an update, delete, or insert. A trigger is fired behind the scenes and the trigger code is stored in the database to be referred to by the application program or a user when using SQL. 3) Security there are special statements used to ensure authority and access rights to data such as the GRANT statement in SQL, that gives each user a specified type of operation that the user is allowed to perform on tables. 4) Stored procedures in the same way that triggers can be stored in the database, on some RDBMSs more complex procedural statements can be stored there as well. This is usually done to centralise special query execution in the database callable by name so they do not need to be run on a slower client machine. A procedure can be run once and live on a central server, where it is executed.

Other factors of a RDBMS are as follows:

- The ability to have multiple concurrent users, with transactions.
- Systems are data-centric which means that the data is represented in a form that is independent of any particular application. This allows data structures to be reused by multiple applications and it can support concurrent access by multiple users. This provides significant gains in productivity and accessibility of data.
- The simplicity of the schema, which is made of a list of tables and the physical data, is defined separately.

---

1 (e.g. data type checking, ensuring certain values are not NULL, ensuring data values meet certain constraints, enforcing primary key integrity, enforcing "referential integrity" of foreign keys)
A good theoretical basis. There was no formal theory for databases before the relational databases.

A high degree of data independence. An application that handles data using a file system is strongly dependent on its data. The application must know how the files are structured and the methods for accessing them. If, for some reason, the way the files are structured or the access methods have to be changed, this cannot be done without requiring modification to the application. In contrast, the RDBMS allows applications to be written without the programmer having to worry about the physical structure of the data and the associated access methods. Thus the system can evolve to take account of new needs without disturbing applications that have already been written.

An improvement in integrity and security by using high-level languages with the facility to specify integrity constraints.

The possibility of optimising accesses to the database. A large part of relational technology has been devoted to improving optimisation techniques; this is why relational systems have become faster.

3.5.2. Limitations

A growing class of application domain is either difficult or impossible for the relational model to deal with efficiently. The main limitations of relational systems stem from the fact that they provide an oversimplified data model and manipulation languages that are too limited for certain data complexities (Delobel, 1995). Usually it deals with data that are stored in simple tables with basic relationships among data items expressed as references to values in other tables.

Relational Database Systems Models are set-based and structured as fixed-format tables (Elmasri, 1994). Although the structure provides flexibility it pays the price through slow performance, because of the different joins that reassemble complex data every time it is accessed. Each table is named and can have any number of columns and rows. Each column holds a particular type of data and
each row of values is conceptually like a record. Only one type of data can be stored in any given column and the developers cannot extend the set of available data types.

In addition, users must conform to their data model, which is inadequate for storing anything other than atomic values. As a result, composite entities must be disassembled into tables and many of the advantages of the whole real-world case approach are lost. The developer and the database administrator must understand two ways of approaching the same problem, the real world model and the relational model. This in turn decreases productivity.

Tables and SQL are flags of fame for relational databases for business applications in the market. The difficulties in building some applications using relational databases forced database specialists to look for a substitute. There are some real-world applications that by their nature are difficult or clumsy to be represented using tables (Douglas, 1996) such as:

1. **Composites or Hierarchies:** the most difficult entity to represent in a relational database is a composite or hierarchical entity, especially when the entity's subparts are recursively defined. There are many examples of these types of entities:

   ✷ Assemblies, which are composed of subassemblies or individual parts.
   
   ✷ Departments and sub-departments.
   
   ✷ Companies and their divisions.

   ✷ For example, an automobile consists of parts, such as an engine, silencer, doors, windows, wheels, and so on. Some of these are composed of subparts. An engine may contain a carburettor, many cylinder heads, many gaskets, spark plugs; etc. the carburettor has subparts, and so on. This is interesting because one may want to reason about the composite objects, and at other times about its subcomponents for example if one wants to determine the cost to build an automobile, broken down by subassembly, such as engine, brakes, etc., each with its own costs.
Because there is an arbitrary number of layers in a composite object, there is no clear-cut way to create an SQL statement that can return the entire composite. Some database vendors have implemented SQL extensions to allow some ability to deal with composites, but they can be difficult to use. There are other reasons why representing composites or hierarchies in a relational database can often be unsatisfactory such as:

- A single logical entity is often spread out over many tables.
- Query performance is very slow, due to multiple joins.
- Recursive queries often require the use of a programming language to be used with the SQL.

2. Many-to-Many relationships: when two base tables have a many-to-many relationship (e.g. students register in many courses and courses serve many students) we need a third table called an intersection entity table to resolve the many-to-many relationship. This is because relational database normalisation rules do not allow repeating values to be stored in fields. This means every many-to-many relationship not only requires an extra table to maintain, but it also requires extra joins, which can be slow when performing queries.

3. Performing non-standard derivation and analysis: the SQL language is a powerful set-manipulation language. But there are some things SQL cannot easily do. One example is the creation of cross-tabulation reports. Say we wanted to create a result table that showed the number of students of a computer science department attending any sessions in the whole schedule and who are also supervised by the head of the department. It is nearly impossible unless we use some non-intuitive special tricks to overcome this problem.

3.6. Object-Oriented model

Over the last decade, people have been talking more and more about the terms "object database" or "object database management system" (ODBMS). Before
diving into the technical details it is informative to cover the basics and consider just what object databases are. Randal V. Zoeller states that databases started with structured databases, which were characterised by the explicit definition of relationships between database records. It attained high performances in traditional transaction processing. These databases supported a standard - CODASYL (mentioned in 1.3) - that promised increased data independence while improving data structuring capabilities (Zoeller 1995).

Objects have entered that database world in two ways (Harrington, 2000):

Pure object-oriented DBMSs: a pure object-oriented DBMS is based solely on the object-oriented data model.

Hybrid, or post-relational, DBMSs: a hybrid DBMS is primarily relational but stores objects in relations.

The relational database came from an upstart database technology seeking to revolutionise the industry. Instead of a standard document, a mathematical definition formed the basis of this new technology. Developers were not locked to a particular design. The data model could change and evolve without major difficulty because the relationships between the data could be defined later on.

The new databases are known as object databases. However, the important difference is that relational databases are now the entrenched systems, and a new upstart technology (object database) is the one making big waves. Although Zoeller (Zoeller 1995) says that relational and object databases have the same strength, Michael Stonebraker's book shows why object-relational DBMSs will replace relational systems (Stonebraker 1996). Others, like Barry Douglas, put a research ODBMS prototype, a pioneering ODBMS product, and an RDBMS product through their paces on the same computer system. He found out that the ODBMS product required less code and ran between 7 to 100 times faster than the RDBMS (Douglas 1996).

To help explain how OOT works it is useful to examine the aspects of the real world that had to be modelled. For example, in order to issue a student
schedule, a number of objects need to be related such as Courses, Students, Departments, Instructors, along with different constraints. Once these are correctly represented the model can be used to solve a wide variety of jobs.

Code developed using Object-Oriented Design Technology (OODT) tend to be well modularised through the use of source code building blocks called Objects. An Object is made up of a collection of related procedures and data. Objects can be defined and maintained independently of one another with each object forming a neat self-contained universe. Everything an object knows is expressed in its data. Everything an object does is expressed in its procedures.

Objects interact with one another by sending messages to carry out their procedures. A message is simply the name of an object followed by the name of the procedure that the object knows how to execute.

Objects can be grouped into classes. A class is a template that defines the procedures and data to be included in a particular type of object. Classes allow developers to organise complex systems in a rational orderly way. Classes can be nested resulting in a tree like structure called a class hierarchy. A class hierarchy represents the relationships among sets of classes. This is found to replace the Type Complex data in the relational model (Douglas 1996).

The great power of class hierarchy is that it applies general rules to broad groups of objects while also accommodating exceptions to these rules. The relational approach uses type code in its table to create relations between its objects, which can easily be converted to a class hierarchy in the Object-Oriented approach (Alkandari 1999).

There are three main factors contributing to the interest in object-oriented database systems as Bancilhon (Bancilhon 1988) mentions in his paper, which will be explained in detail in the section of advantages and benefits of Object-Oriented model.

1. The need for database functionality to build object-oriented systems.
2. The modelling power of the object-oriented approach and the semantic data model aspect. The emphasis is on solving the database design problem by providing more powerful tools to model the real world.

3. A solution for the problem of "impedance mismatch" requires integrating database and programming language technology. Application development requires the communication between a relational query language and a programming language. These two types of languages do not mix well, because they have different types, they have different computational models. Relational systems are set-at-a-time while programming languages are record-at-a-time. The overall objective is to integrate database technology and the object-oriented approach in a single system.

3.6.1. Advantages

In most of the books and articles there is an agreement that the drawbacks of relational databases could be solved by Object-Oriented databases. Some authors such as Bancilhon (Bancilhon 1988), Stonebraker (Stonebraker 1996), and Douglas (Douglas 1996) came to a conclusion that object-oriented databases are going to replace relational databases. They agree that the OODB represents real-world problems clearly. Relations between substances of the object system are well defined, as it is part of the definition of the Object identity. Class hierarchy in the object-oriented model makes it possible to use composite objects, inheritance and reusability of code, which is very useful for database technology.

'Objectstore' is one of the leading developers in Object Database Management System Design; providing storage management for data in the object-oriented environment. Their system provides three major benefits using ODBM, which are as follows:

\* It allows for highly efficient, but easy access from object programming languages.
It hides the complexities of distribution of objects across network sites.

It allows multiple users and applications to share protected access to the objects.

An application is built with objects where the object database maintains the natural relationship among data and also between data and behaviour of each object. Objects used by the applications are stored in exactly the same format in the database. By maintaining the relationship among data providing single level storage and supporting behaviour written in object-oriented languages, object databases are dramatically easier to use than traditional database management systems.

Object databases are also extremely fast for interactive applications, which use complex structure, and they are highly compatible with many object-oriented development tools used in object-oriented analysis, object-oriented design, and object-oriented programming.

An object database benefits from the experience that developers have had with older database structures. Developers have been able to take the best features from hierarchical, network, and relational databases while discarding features that have become obsolete. While maintaining the best features from traditional databases, they also extend the state of the art functionality by adding new features.

The temptation of using Object-Oriented database comes from the power of Object-Oriented programming. Objects provide an encapsulation of attribute values and behaviours. Unlike SQL, Objects package attributes and behaviours (methods) together and only expose what is important to the outside. Object-Orientation looked promising to solve the impedance mismatch perceived by the database community because it provided a framework to represent and manage both programs and data.

Object-oriented databases allow objects made using object-oriented programming languages to be stored permanently. They are said to make such
objects persistent, which means that the object can be accessed even after the
program terminated (Cattell, 1994). Objects are stored exactly the same way as
they are used in the program. This means that there is no need for mapping code
as there is when using a relational database. The way in which the object is stored
also permits the object to be accessed by other applications. Entities are stored
exactly as they appear in real life, and they are not split across many tables as in
relational systems.

Objects form the equivalent to tuples in relational databases. Objects are
constructed from a class type defined by the programmer. Unlike tuples objects
can also contain methods, these methods hold calculations or procedures that can
be used to update or manipulate the object.

Attributes of a class are not restricted as they are in relational databases
and user defined types can be stored and database operations such as queries and
selections can be performed on them. In addition, these user-defined types can
have behaviours assigned to them by user-defined methods.

Each object created from a class type is given a unique object identifier
(OID), which acts as an address or reference to the object that can be called or
held in other objects.

In object-oriented databases, attributes in the user-defined classes can
include such data types as arrays and lists that allow more than one item per
attribute to be stored. These data types are not supported by relational databases,
as each tuple must be of the same size due to the rules of normalisation. For
instance, in relational databases another relation must be set up if employees are
allowed more than one telephone number. However, in object-oriented databases
telephone numbers can be stored in an array and hence fewer joins are used.

Object-oriented databases also allow the OID (object identifier) reference
of other objects to be stored in a collection. A collection can have multiple values.
This provides the way in which objects relate to each other in object-oriented
databases. It also eliminates the need for a relationship table used by relational
databases for many-to-many relationships.
There are many reasons behind the interest in moving to objects because they solve many problems that programmers have every day. The following are some of the features that enable OO languages to help in solving complex business problems:

**Enhanced modelling power**

Object orientation is a strategy for organising systems as a collection of objects that combine data and behaviour. An object can represent entities in the real world. It associates not only attributes, but also behaviour, directly with those entities. Object can be classified and sub-classified, gaining more specific attributes and methods at each level. This is how people think about objects in the real world. In other words, OO languages try to mimic human cognitive functions to provide a powerful programming metaphor, allowing us to model the real world and its interactions in a computer. It is cumbersome, in a relational database, to represent a single entity across many tables, as it requires many joins between tables to re-synthesise the original entity. This problem occurs most frequently when dealing with complex entities, such as composites and hierarchies. OO languages allow the developer to treat a complex object as a single entity, although it is a clear violation to the first normal form rule (Anstey 1998).

**Extensibility using class libraries**

SQL has a fixed set of data types, such as INTEGERS and DATES. It cannot be extended. In theory, any object can be stored in the new binary large object (BLOB) data types now available, which many databases are supporting for multimedia objects such as digitised video and images, simply converting that object to a stream of bytes.

**Representation of Multi-valued attributes**

A data value in any particular field may be a singularity (cardinality = 1), or it may be a collection (i.e., cardinality> 1). For example, an automobile may have a single colour or many colours. Having repeating values in a single table
violates rules for normalisation (first normal form). This is one reason why an explosion in the number of database tables occurs. A single cognitive entity cannot be described in a single table, due to the need for resolution tables, known as intersection tables as mentioned earlier. In OO language, however, this problem does not occur. All data about an object, even Multi-valued data, is stored in the object itself. The ability to support collections is one of the most powerful features of an OO system. A collection may logically resemble a list of items, for example if we have students with more than one major (Aziz: {computer science, math}) or an instructor who teaches in two different departments (Ray {computer science, physics}). This object is described in a logical notation. It does not hint at how the collection is actually implemented.  

OO languages support the following kinds of collections:

- Arrays or lists, such as {red, green, blue} in which order matters
- Sets or Bags, in which order doesn’t matter
- Queues, First-In, Last-Out (FILO) or Last-In, First-Out (LIFO).

Representing composites and hierarchies

It was clarified earlier how difficult it is to represent these kinds of objects in relational database. In an OO language, however, a composite class can be represented recursively (using pseudo code):
CLASS COMPOSITE:

...  
SUBCOMPONENTS: SET OF COMPOSITE
...  
So an actual automobile composite object under this class may resemble:

AUTOMOBILE_487:

SUBCOMPONENT: {ENGINE_876, WINDSHIELD_211, WHEEL_1, ...}  
...  
And an actual engine object may be defined as:
ENGINE-876:

SUBCOMPONENTS: {CYLINDER_HEAD-876, INTAKE_MANIFOLD_733, SPARK_PLUGS_1, ...} (Vermulen, 1996)

Representing relationships.

It is possible to establish bi-directional relationship between objects in which referential integrity is maintained transparently. In other words, if an employee is added to a department, the department reference is also stored in the employee object automatically.

3.6.2. Main characteristics of Object-Oriented Systems

There are essential features that one should consider in implementing a DBMS to make it Object-Oriented. Those features represent the more original ideas and have the most impact on programmer productivity. Object-Oriented Database Systems (OODBS) would be enriched if they include the following programming features:

3.6.2.2. Encapsulation

Encapsulation is a principle that packages data and operations in a single object. An object has an interface part and an implementation part. The interface part is the specification of the set of operations, which can be performed on the object. It is the only visible part of the object. The implementation part has a data
part and an operation part. The data part is the memory of the object and the operation part describes the implementation of each operation.

### 3.6.2.3. Object identity

An object in a model has an existing object identity (OI) that is independent of its value. Thus two objects can either be identical (they are the same object) or they can be equal (they have the same value). In the object-oriented model the object identity is system generated rather than artificially made like in the relational model. In relational databases, tables must have a unique field or set of fields called the primary key that a programmer can use for reference from another table with a field called a foreign key.

### 3.6.2.4. Classes

In object-oriented terminology, the template on which similar objects are based is known as a class. When a program creates an object from a class, it provides data for the object's variable. The object then can use the methods that have been written for its class. All the objects from the same class share the same procedures for their methods. They also have the same type of data but the values differ (Harrington, 2000).

A class is also a data type. Actually, a class is an implementation of an Abstract Data type, which is another term for a user-defined data type. For example a class can handle data about employees in some organisation. The attributes of the class might include employeeID, the first name, the last name and the address. The address itself is made up of a street, city, province, and postal code. Therefore, one would probably create an address class with those attributes and then, rather than duplicating those attributes in the employee class, simply indicate that an object of the employee class will include an object created from the address class to contain the employee's address.
3.6.2.5. Inheritance

This is probably the most powerful concept in object-oriented programming: it allows objects of different structures to share operations related to their common part (Lausen, 1997). In a relational system, the database designer defines a relation for each entity (Student, Employee, etc...) and writes the code for each operation (calc-grades, calc-salary, etc...). Thus the application programmer writes extra programs. In an object-oriented system, using the inheritance property, we recognise that Employee and Students entities are Persons; thus, they have something in common (the fact of being a person), and they also have something specific. We introduce a type, Person, who has attributes name and age and we write the operations die and marry for this type. Then, we declare that employees are special types of persons, who inherit attributes name and age and have an extra attribute salary, and that students are special kind of person, with a specific set-of-grades attribute and a special operation, GPA computation. In this case, we have written less code. This has two advantages: it is a powerful modelling tool, because it gives a concise and precise description of the world. It helps code reusability because every program element is at the level in the hierarchy at which the largest number of objects can share.

3.6.2.6. Overriding and late binding

There are cases where, one wants to have the same name used for different operations. Consider for instance the display operation: it takes an object as input and displays it on the screen. Depending on the object, we want to use different kinds of display. If the object is a picture, then we want it to appear on the screen. If the object is a person, then we want some form of a tuple being printed, and if the object is a graph, then we will want its graphical representation. Consider now the problem of displaying a set of objects, whose type is unknown at compile time.

In a standard system, we have three operations: display-person, display-bitmap and display-graph. The programmer will test for the type of each object
and use the corresponding display operation. This forces the programmer, when
he displays an object, to be aware of the type of the object (extra knowledge at
compile time) and to be aware of the associated display operation and to use it
accordingly (more information to remember).

In an object-oriented system, the programmer defines the display
operation at the object type level the most general type in the system). Thus,
display has a single name and can be used on graphs, persons and pictures.
However, he/or she refines the body of the operation for each of the types
according to the type specificity (this is called overriding) (Bancilhon, 1988). This
result in a single name “display” denoting three different functions known as
methods which is called overloading (Bancilhon, 1988). To display the set
elements, we simply apply the display method to each element.

The programmer has an advantage when using object-oriented systems.
He or she still writes the same number of programs, but does not have to worry
about which of the three methods is used. The code written is simpler. There is no
case statement switching on types. Finally, the code is also re-usable. If, in the
future, a new type is to be introduced in the system and in the set of objects to be
displayed, the same display program works (provided the programmer overrides
the display method for the new type). To offer this new functionality, the system
cannot bind operation names to programs at compile time. Therefore operation
names are resolved (translated into program addresses) at run time (this is called
late binding) (Bancilhon, 1988).

3.6.2.7. Extensibility

This is a major advantage of object-orientation; by adding a new type (or
classes) to the system, one can extend its capabilities. This is especially important
when adapting the system to new types of applications.

3.6.3. Limitations

It is important to have a clear notion of what is the advantage brought by the
introduction of database functionality in the object-oriented system. In other
Words, what is needed in an object-oriented system to make it into a database system.

- **Set programming**, there is no set construct in object-oriented programming. Traditional programming languages do not, in general, have the concept of sets; they use other constructs to implement sets such as arrays, lists or files. This is why they have no specific operation to manipulate sets. The relational model introduced algebra based on sets with the associated operations. By defining selection and join as the major operations, the relational model gave the description of the operations to optimise.

- **Persistence and reliability**, most object-oriented systems do not offer persistence. Thus, in order to keep data from one session to another a file system is needed to save the necessary data. The exception is POET, which allows C++ objects or structures to be made persistent by prefixing the declaration with the keyword Persistent.

- **Sharing**, object-oriented systems are single user and do not provide a control over the concurrent access to the same data.

- **Managing large volumes of data**, application programs are limited in the size of the data they manipulate to the virtual address space, because the system runs in main or virtual memory (Bancilhon, 1988).

### 3.7. The Object-Oriented Model Versus the Relational Model

Current relational systems together with their connection to general purpose programming languages can do almost everything. They ensure persistency, reliability, data sharing; they can model any data and perform any possible computation. However, even though every application can be written on top of such a relational system, it might be extremely hard to do or it might be incredibly slow. Thus, to compare a new system to a relational system, the criterion for improvement is not computing power, but it is computing speed, ease of modelling and implementation, and finally maintainability and extendibility.

Randal V. Zoeller states:
“I dare say that relational databases won out over structured databases because they offered a better way to do things. Although I do feel object databases have significant potential, I would not say that object databases are inherently better than relational databases—or vice versa. Instead, object databases are best viewed as an enabling technology that allows developers to do certain things more easily than with the relational model. Object databases will not replace relational databases. Rather, technology from both types of systems will live on and become entrenched in the particular area of need. I like to call it the “right tool” approach—pick the right tool to help you get the job done, but don’t be afraid to change if necessary” (Zoeller, 1995).

Barry Douglas has put a research ODBMS prototype, a pioneering ODBMS product, and an RDBMS product through their paces on the same computer system. He found that the ODBMS ran between 7 and 100 times faster than it ran on the RDBMS, and not only did it run faster, it required much less code (Douglas, 1996).

In many regards object oriented systems are superior to relational systems, as they handle complex objects in a more natural way. That is because the object-oriented model is a direct outgrowth of the object-oriented paradigm. The entity objects used by object-oriented programs are directly analogous to database entities used by pure object-oriented databases, with one major difference: program objects disappear once the program stops running; database objects must persist.

3.7.1. Dealing with complex objects

The ability to store and manipulate complex objects is a feature of many object-oriented systems, while relational systems are restricted to store and manipulate only flat tuples. This gives more modelling power; because a complex structure can be represented directly, and does not have to be mapped onto a lower level relational structure.
3.7.2. Object identity versus primary key

The notion of object identity, as introduced in an object-oriented system, is clearly a bonus in modelling power. A relational database represents data relationships by having matching primary key-foreign key data. There are no data structures within the database that form links between the tables; the relationships are used as needed by joining tables. In direct contrast, a pure object-oriented database "hard codes" its relationships by including object identifiers within an object to indicate other objects to which it is related.

An object identifier is an internal database identifier for each individual object. Users, whether, they be programmers or end users working with an interactive query tool, never see or manipulate these identifiers directly. Object identifiers are assigned and used only by the DBMS.

3.7.3. Storing programs and data

Database languages are in general not complete, because they don't allow computable queries to a database to be expressed. For example to achieve a loop construct in SQL we have to embed a general programming language that is equipped with control structures into the query language of the DBMS and then ensure full computational power. The side affect of this procedure is called impedance mismatch since databases and programming languages have different data types.

The aim of object-orientation is to reduce or eliminate the impedance mismatch between database languages and programming languages. The tool used to overcome this pitfall is encapsulation (storing programs and data together). Encapsulation means more; it is actually an abstraction of some tangible real world case. The main idea is to hide the internal implementation of the object but make visible what the object can do.

On one hand, the operations that each data needs are attached to it and not separated in another place or library. On the other hand, object-oriented databases are written in object-oriented languages, which leaves no room for any language
mismatches. It might be difficult to code in the beginning but the shorter, more natural code pays off in the long run with its reduced maintenance requirement.

3.7.4. Typing and inheritance

There are several reasons why object-oriented paradigm has become so pervasive in programming. Among the perceived benefits are the following:

- An object-oriented program consists of modular units that are independent of one another. These units can therefore be reused in multiple programs, saving development time. For example, if there is a well-debugged object class, this class is reusable for any program that requires data about that class.

- This class independence mentioned above can help reduce the maintenance requirement. As long as the class’s interface is kept unchanged we can update the unseen operations without having to alter the interface code since they are independent of each other. This process helps to eliminate any unexpected side effects when modifying the program.

- Inheritance adds logical structure to a program by relating classes in a general to specific manner, making the program easier to understand and therefore easier to maintain.

3.8. Review of Existing Object-Oriented Database Systems

Much research has been conducted and is still continuing because of the rich characteristics of the Object-Oriented model (Cattle, 1994). This research represents the main trend in object-oriented databases. Some of the research deals with extending the relational database where the other research specialises in the pure object-oriented model. The following (from ODE to VODAC) is the list of research systems using the object-oriented model. The rest (from ArtBASE to POET) is the list of commercial systems.
3.8.1. ODE

ODE is a database system and environment based on the object paradigm. It offers one integrated data model for both database and general-purpose manipulation. The database is defined, queried and manipulated in the database programming language O++ that is based on C++. O++ borrows and extends the object definition facility ‘class’ of C++. Classes support data encapsulation and multiple inheritances. It provides facilities for creating persistent and versioned objects, defining sets, and iterating over sets and clusters of persistent objects. It also provides facilities to associate constraints and triggers with objects.

3.8.2. CLOSQL

CLOSQL is a research prototype OODB designed primarily for prototyping various schema evolution and view mechanisms based on class versioning. The system is built using Common LISP. It would really only be of interest to other parties as a research tool. It requires Common LISP with the CLOS standard. The graphical user interface requires the Harliquin LispWork Tool-Kit. The system was built on Sun4 and has not been tested on any other platform (Monk, 1992).

As a prototype CLOSQL is not robust enough to sell. The system is single user and does not properly support persistence, as the data has to be loaded and saved explicitly. The query language is quite good making use of the functional nature of the environment. Methods (LISP and query language only), class versioning and multiple inheritances are all supported in the data model. Type checking information is held in the database, but is not enforced at present. The GUI is notable for its support for schema evolution, but otherwise rather ordinary.

On schema evolution CLOSQL implements a class versioning scheme (like ENCORE see section 3.8.8), but employs a conversion adaptation strategy. Instances are converted when there is a version conflict, but unlike ORION and GemStone, CLOSQL can convert instances to older versions of the class if necessary (Monk, 1992).
3.8.3. Oggetto

Developed locally at the University of Lancaster, UK Oggetto is an object-oriented database layered over a triple store. (Mariani, 1992). It supports its own language. It supports relational algebraic operators such as "select", "join" and "project". However, it is not very efficient. It has been used as the target for a number of research and under-graduate projects. It has been used as the basis of a C++ language binding, distributed databases and transaction management. Oggetto has client-server architecture, which is supported by the use of SUN RPC (remote procedure calls). This means there is a clean and clear interface between client and server. Oggetto related projects have taken full advantage of this interface by interposing a transparent filter between clients and server. This filter has been developed in two ways; one to support transactions and one to support distributed data.

3.8.4. IDB Object Database

IDB Object Database is a distributed object-oriented database programmable in ANSI C. It supports multiple inheritance, polymorphism, binding, transactions for concurrency control, versioning, dynamic linking, heterogeneous networks and exceptions. It includes an interactive schema designer and database browser. Data and applications are portable across all supported platforms. It is a product of Persistent Data Systems Incorporation™.

3.8.5. Avance (SYSLAB)

This is an Object-Oriented, distributed database programming language. Its most interesting feature is the presence of system-level version control, which is used to support schema evolution, system-level versioning (as a way of improving concurrency), and objects with there own notion of history. The system consists of a programming language (PAL) and distributed persistent object manager (Björnerstedt, 1988).

Persistent Data Systems, Inc. P.O. Box 38415 Pittsburgh, PA 15238 U.S.A.
3.8.6. ConceptBase

ConceptBase is a deductive object manager mainly intended for conceptual modeling and the coordination of design environments. It follows client-server architecture. Client programs can connect to the ConceptBase server and exchange data via inter-process communication. The X11-based ConceptBase user interface offers a palette of graphical, tabular and textual tools for editing and browsing the object base. The ConceptBase programming interface allows the users to create their own client programs in C or Prolog.

3.8.7. COOL/COCOON

The COCOON project was intended to extend the concepts and the architecture of relational database management systems beyond nested relational to object-oriented ones. The prototype implementation of the COCOON model was built based upon the nested relational DBMS kernel (DASDBS). The goals of the project are:

- To develop a general formal framework for investigations of all kinds of schema changes in object-oriented database systems (including schema design, schema modification, schema tailoring, and schema integration).
- To find implementation techniques for evolving database schemas, such that changes on the logical level propagate automatically to adaptations of the physical level (without the need to modify all instances, if possible).

In their paper (Tresch, 1992), schema evolution is used as example of a general framework for change in Object-Oriented Databases (OODBs), supporting change on three levels of database objects: data objects, schema objects, and meta-schema objects.

---

3 ConceptBase can be obtained from ftp.informatik.rwth-aachen.de in /pub/CB/CB_3.2.4 (released 26-Apr-1994 for Sun/SPARC, SunOS 4.1.3)
3.8.8. Encore

Encore is an object-oriented database system targeted at large scale software engineering applications, which are involved in data modelling. It was developed at Brown University in the late 1980s. It is notable for its special support for cooperative translations, popular in design applications, and its support for class versioning. Objects are never converted, rather, classes are versioned, and the user can specify filters to make old-style instances appear as new instances to new applications (and vice versa) (Hornick, 1987).

3.8.9. Exodus

Exodus is an extensible database system project that is addressing data management problems posed by a variety of challenging new applications. The goal of the project is to facilitate the fast development of high-performance, application-specific database systems. Exodus provides certain kernel facilities, including a flexible storage manager. In addition, it provides an architectural framework for building application-specific database systems; powerful tools to help automate the generator and a persistent programming language; and libraries of generic software components (e.g., access methods) that are likely to be useful for many application domains (Carey, 1990). The programming language is called E, an extension of C++ (Richardson, 1989).

However, the project was not able to provide schema evolution. Furthermore, emulation is rejected by the authors, who claim that the addition of the layer between the Exodus storage manager and E program would seriously reduce efficiency. Automatic conversion, whether lazy (uses the minimum processing to accomplish the task) or eager (uses the maximum processing to provide best possible solution), is also rejected, as it does not mesh well with C++ data layout. To implement immediate references to other classes and structures, C++ embeds class and structure instances within its referent. The resulting change in the size of the object might invalidate remote pointer references.
3.8.10. MOOD4-PC

MOOD4-PC is Material's/Miniature Object Oriented Database Prototype for NEC/IBM-PC. MOOD4-PC is an object-oriented database system (OODBS) program developed in the course of research project MOOD. The aim of the project is to develop a material database system to handle raw material data which are produced and accumulated in materials research and referred to by material experts when they face scientific or engineering problems where the expected behavior of particular material in particular environments are crucial importance. Since the conventional database systems do not fulfill these requirements, MOOD serves well for bibliographic databases or fact databases which deal with the standard properties of standard materials.

MOOD as described by Noboru Ono is a general purpose OODBS. This is not in the sense that it is capable to develop application programs on it, but in the sense that it generally supports the essential capabilities of OODBS:

1. The abstract data type.
2. The nesting of structured data objects.
3. The class hierarchy.
4. The inheritance of attributes along the hierarchy.
5. Matching between objects along their structures with the knowledge of the class hierarchy.

3.8.11. The Object System of STONE – OBST

The persistent object of management system OBST was developed by Forschungszentrum Informatik (FZI) as a contribution to the STONE project supported by the German Ministry of Research. OBST was originally designed to

---

1 Noboru Ono, Dept. of Machine Intelligence and Systems Engineering, Faculty of Engineering, Tohoku University. ono@mood.mech.tohoku.ac.jp
2 Forschungszentrum Informatik (FZI), OBST Projekt, Haid-und-New-Strasse 10-14, D-76131 Karlsruhe, Germany. obst@fzi.de
serve as the common persistent object store for the tools in software engineering environments.

3.8.12. **OTGen**

OTGen is a design for a system to support schema evolution in object-oriented databases. The chief contribution of OTGen is support for programmer extensibility of transformation functions to allow a system to support a wide range of schema changes, not just those that can be easily automated. OTGen was never implemented, it is based on the implementation of TransformGen, a system to support the evolution of the specialized database used by Gandalf programming environments (Lerner, 1990).

3.8.13. **VODAC**

The VODAC Model Language VML (Klas, 1993) homogeneously integrates the concept of meta-classes and the separation of types and classes with other object-oriented concepts such as properties, methods, inheritance, and object identity. Complex nested data structures can be defined using the set, array, tuple, and dictionary type constructors. VML supports its own programming language for implementing methods, specifying transactions and an ad hoc query language.

In VML classes are used to organize a set of objects corresponding to real world entities and relationships between them. Object types define the structure of an object and the operations defined on these structures. They are associated with classes in order to determine the structure and behaviour of the class instances. Meta-classes are first class objects whose instances are classes. Meta-classes are associated with three object types: an (optional) own-type extending their instances (which are classes), and an instance-instance-type specifying the behaviour of the instances of their instances. Meta-classes can be organised in an instantiation hierarchy of arbitrary depth.

This approach leads to an open, adaptable data model, which provides for the specification of additional modelling primitives at a Meta layer of the database schema. The concept of Meta-class and the separation of classes and types allow
determination of the structure and behaviour of objects and the individual inheritance behaviour via semantic relationships between arbitrary objects already at the Meta layer for the application specific classes.

The system architecture consists of a central database environment and several external database environments to which the user wants to have integrated access. Each of these environments consists of an object manager, a message handler, a transaction manager, and a communication environment includes a database interface module, which realise the access to an external database system.

The first version of a C++ based prototype of VODAK is available for Sun Sparc Station under certain conditions. It implements all the features specified e.g. Meta-classis, transactions, and remote message execution.

3.8.14. ArtBASE (commercial system)

ArtBASE is an Object-Oriented Data Model, uses Objectworks\Smalltalk by ParcPlace Systems, Inc. and it runs on different platforms like Unix, PC Windows, and Macintosh. A single user version is available. The Distributed Multi User Server was presented at the OOPLSA’93 at Washington D.C. in September 1993 for Unix environment and PCs.

3.8.15. EasyDB

EasyDB is a single or multi user Distributed Object Database Management System. Well integrated with C, C++ and Ada.

3.8.16. GemStone

The GemStone Object-Oriented Database, from GemStone Systems, Inc. is a product of Servio Corporation of San Jose, California and Beaverton, Oregon was first introduced in 1987 (Bretl, 1988). It is the oldest commercial ODBMS and still available today. GemStone is particularly well suited for use in complex
multi-user, multi-platform client/server applications. It supports concurrent access from multiple external languages, including Smalltalk (VisualWorks, VisualAge, and Visual Smalltalk), C++ and C. GemStone also provides Smalltalk as an internal DML, which can execute methods or entire applications in the database.

GemStone incorporates gateways or data bridges that allow object applications to integrate legacy data, whether in SQL or other formats. The level of integration between GemStone and legacy data and applications can range from simple query access to extensive read-write operations (Cattle, 1994).

3.8.17. ITASCA

ITASCA, a product of Itasca Systems, Inc. of Minneapolis, Minnesota, is a commercial ODMS based upon the ORION prototype be described in the next section 3.8.18. Development of ORION began at Microelectronic and Computer Technology Corporation (MCC) in 1985 and prototypes were delivered in three phases through 1989. The first release of ORION was a stand-alone system. The second release featured a single-server, multiple-client (remote access) architecture. The third release featured a multiple-server, multiple-client (distributed database) architecture. ITASCA is based upon this final release of ORION and has been shipping since August 1990.

Applications written in C++, C, Lisp, and Ada may communicate directly with ITASCA through their respective application programming interfaces. ITASCA stores and manages objects in a neutral format, yet presents the objects to an application in the format of its native language. This allows applications written in various languages to share objects. For example, a C++ application may create some objects and store them in ITASCA. An Ada application may then access those objects and modify them.

3.8.18. ORION

ORION is an Object-Oriented ODMS built at MCC in Austin, Texas. ORION incorporates object identifiers, multiple inheritance, composite objects, versions, indexing, queries, distributed databases, dynamic schema evolution, and access
authorization. It is implemented in Lisp. Several applications have been built on ORION at MCC, so ORION has seen some use even though it was constructed as a research prototype.

An area where ORION has put more emphasis than other systems is distributed databases. Another area that ORION has emphasized more than have other systems is schema evolution. (Kim, 1988) discussed schema versioning, a technique to allow multiple versions of the schema to be maintained simultaneously. Objects are identified with the version of the schema with which they are currently constructed. A mapping can be maintained from the old schema to the new, and an object can be converted automatically to the new schema format at the time it is accessed, rather than immediately on schema update.

3.8.19. Objectivity/DB

Objectivity/DB\(^7\) provides an integrated C++ programming interface with an emphasis on the DBMS engine for robustness and scalability. It supports a distributed client-server, rather than central-server architecture, with all operations working transparently over a mixture of multiple databases, schemas, users, and computers, and over heterogeneous hardware, operating systems, and networks. The language interface includes a C++ class library interface, which is based on ODMG-93. Objectivity/DB platform support currently includes all Sun, All DEC, HP/9000 series, IBM RS/6000, NCR 3300, SGI, Windows 3.1, and Windows NT.

3.9. VERSANT

The VERSANT Object-Oriented DBMS is a product of Versant Object Technology of Menlo Park, California. Like GemStone, VERSANT provides database programming language interfaces to both C++ and Smalltalk. VERSANT also offers a C library interface. VERSENT is designed for multi-user applications in distributed database environments.

Because VERSANT is designed for multiple programming languages, it supports its own object model which includes features of C++ such as multiple inheritance, and features of Smalltalk such as dynamic class creation. VERSANT requires neither a proprietary C++ compiler nor pre-processor because the interface of VERSANT is implemented entirely in an ANSI-standard C++ class library. The VERSANT Smalltalk interface uses standard Smalltalk and thus does require use of a server-side interpreter to access data.

3.9.1. POET

POET 8 is the final Object-Oriented Database considered, and bears resemblance to those discussed in previous sections. POET is based on C++. It runs on PC, Macintosh, and Unix platforms. A client can access a database stored on any platform. A single-user version is available in addition to the client/server version.

POET allows C++ objects or structures to be made persistent by prefixing the declaration with the keyword persistent. POET's pre-processor produces C++ compiler input from these declarations. POET supports C++ encapsulation, object identity, inheritance and polymorphism. It includes a library of extensions to C++, including predefined types such as dates and Binary Large Objects (BLOB), and parameterised types for sets.

3.10. Summary

The findings in this chapter are based on the general literature survey of database models. The object-oriented data model is the latest in a sequence of data models that has been evolving since the early 1960s. In this chapter we have given an overview of the history of data modelling to get a better understanding why the object-oriented data model is a step forward and it entered the database world in two ways.

---

Chapter 3 Theory and Literature Review

It has also shown, in brief, the advantages and limitations of earlier and current database models showing the published claimed advantages of each system. The history of object-oriented modelling is presented to show how object-oriented modelling has been developed to give even greater advantages over other systems. The more significant research and advances in commercial object-oriented databases are identified.

This chapter shows an overview of existing object-oriented DBMSs that provide database capabilities in an existing programming language. The ones that are covered such as (GemStone, ORION, ITASCA, Objectivity/DB, and VERSANT) provide a query language, object types with inheritance, caching of objects in main memory, transaction management, and remote database access. VERSANT and GemStone provide access to the database from more than one programming language.

Object-Oriented DBMSs, particularly those based on C++, have been a popular approach to object data management—-the number of products using this approach shows that many people believe this is a promising direction.

To satisfy the first aim of this research, the chapter compares the relational with the object-oriented models using selected criteria such as dealing with complex data, object identity versus primary key, storing program and data, and inheritance.
Chapter 4: Review of OO Improvement Methods for Legacy Database Systems

4. Introduction

This chapter investigates the different object-oriented improvement methods for legacy database systems. Published literature shows that object-oriented methods are used to tackle database problems with complex data to improve performance. These methods vary in their techniques, some completely convert the legacy database system to an object-oriented one, while other techniques try to use legacy database systems and object-oriented database systems in conjunction to bring out the strength of each. This chapter shows how converting is better than other techniques.

4.1. Transforming a Relational Database to an OO System

Since relational database technology started it kept increasing its share in the market. SQL is a de facto standard, and it is a standard for exchanging data between heterogeneous systems. Even if objects take over the world, exchanging data between systems is likely to remain in relational form. Thus, there is a need for a relational interface to object-oriented databases. The belief of many that object-oriented database systems have a reasonable chance to overtake relational systems is based on the following:

1. Object-orientation will win as a programming paradigm; because it has some obvious qualities – mentioned earlier in chapter 2 -. It is very appealing, it is fashionable, it mixes with variety of different styles of programming, and finally it has shown to be very successful in areas such as Artificial Intelligence (AI) and User Interface Development (UID).

2. Object-orientated databases blend the most successful programming paradigm with database technology, thus solving the impedance mismatch.
3. Their ability to extend easily will allow them to evolve and accommodate new
types of data types and new functionality.

4.2. Using existing relational schema

Modelling complex data using the relational model is a tedious task. For
example each many-to-many relationship with relational schema requires an
intersection entity. Many studies have been carried out to examine the use of
object-oriented instead of relational systems to solve or improve such problems.
The question that comes to mind is what is going to happen to the existing
relational schemas if there is a move to object technology? How are we going to
handle the data that has been stored in a RDBMS? There are two choices shown
by (Douglas, 1996):

1. To convert data to an Object-Oriented Database Management System
(ODBMS).

2. To keep it in the RDBMS but access the data as needed.

For each type of data complexity there is one technique or more for
converting relational schema to object-oriented schema. For example, the
conversion of a type code complexity is handled by reworking the relational
schema into an object schema. The technique is to use the type codes and
relational views to get ideas about the object class hierarchy. After checking the
type code complexity the process proceeds to generate tentative classes.

By separating objects, the common attributes of different objects can be
seen and then gathered in a specific class. Furthermore, some attributes are
assigned to some distinct objects. By looking at the result, the common attributes
are gathered in a super-class and can be used inherently. The other specified
attributes are kept to where they belong in the sub-class.
4.3. Transforming Relational Database Schemas into OO Schemas according to ODMG-93

The article written by Fahrner et al. (Fahrner, 1995) describes a transformation of relational database schemas into object-oriented schemas according to the Object Database Management Group (ODMG-93) where developers and manufacturers of object-oriented database systems have joined together to form the ODMG and have proposed a standard for object-oriented database systems that is ODMG-93 (Lausen, 1997). It discusses the problem of migrating database applications from relational systems to object-oriented systems. The article indicates that such migration involves schema conversion, and could be automated.

The article states that, if an Object Oriented Database (OODB) is created from scratch, its design can make use of a variety of existing techniques that include methodologies based on the Entity Relationship model (ER) or on Object-Oriented Analysis (OOA) and Object-Oriented Design (OOD). These techniques typically make full use of the features of the target OO model at hand. However, if a database already exists, the task is to transform its schema into a schema of an OODB. Clearly, another task is to convert the contents of the given database.

There are two design decisions of central importance for the methodology described in the above article. Firstly, the authors transformed relational schemas directly into an object-oriented database i.e., without using an intermediate model. Secondly, they imposed as few requirements as possible on a given relational schema, whilst taking all given information into account. Hence the starting point could be an arbitrary relational database schema with relational schemas and possibly integrity constraints; additionally, a database instance might be available.

The goal is to obtain a schema in the target model consisting of classes, whose attributes can be complex and can reference other classes; additionally, classes can be arranged in an inheritance hierarchy. This is the basic view supported by ODMG-93, where class-valued attributes are termed relationships.
When the designer of a given relational schema has made its semantics as explicit as possible, i.e. if integrity constraints relevant to a transformation are declared in the schema and the schema is properly annotated, a transformation to ODMG-93 can be done in an almost canonical way. Since the Object-Oriented Database (OODB) model has more expressive power, then both structural and semantic information has to be enhanced when moving from relational to an OO description, which implies that the input to a transformation has first to be completed, both structurally and semantically.

The transformation proposed in the above article has a three-step process: first, the given relational schema is completed. This includes an identification of synonyms and homonyms, a classification of attributes, the determination of keys and Functional Dependencies (FD), a particular Third Normal Form (3NF) normalisation, an analysis of Inclusive Dependencies (ID), a removal of redundant attributes and of redundant relational schema, and an identification of inheritance structures.

In the second step, a canonical transformation is made, which basically transforms each relation schema into an ODMG-93 class. Finally, the OO schema that is obtained is restructured with respect to various OO aspects, including an elimination of artificial keys, an elimination of relation schemas representing binary relationships, an identification of complex attribute structures, and the redefinition of objects as litorals (types). In essence, the initial object-oriented schema is improved to better exploit the options available in the object-oriented paradigm.

The importance of the approach above is the observation that the transformation of relational schema into ODMG-93 model is straightforward, provided the source schema satisfies certain requirements. Therefore, a major portion of the exposition has to answer the question of how to achieve these prerequisites for arbitrary given schema.
4.4. Connecting Objects To A Relational Database

There are other approaches besides transforming schemas from a relational database to an object-oriented database. The idea is to use both models in conjunction to bring out the strength of each. By bringing the relational data into objects, where the data can be navigated and the applications built. Consequently, the altered data can be brought back to a relational database and it can be securely kept (Vermulen, 1996). There are two different ways to link objects with relational data. One way is using ad hoc SQL bound to an object and the other is direct object mapping.

4.4.1. Ad hoc SQL bound to object

One way to bring the relational data into the object world is to bind the results from SQL statements to objects. The value for each field returned is bound to one of the attributes of the object. For example, the data for the "Hire date" field, for a particular row in the SQL query results, can be bound to a particular object's "Hire date" attribute. This technique is similar to having the program storing data in the ram and before the termination of the run the data is kept in the secondary memory. In other words it is making data persist or remain in memory for future use like the idea of "persistence objects".

4.4.2. Using a 4GL (Fourth Generation Language).

Relational database vendors have developed their own procedural languages that extend SQL, and are well integrated with it. These are known as fourth generation languages (or 4GL). For example, using Informix's NewEra™ language, rows in the database can be bound to program variables.
Chapter 4 Review of OO Improvement Methods for Legacy Database Systems

FUNCTION process_order (company_id, prod_id, quantity)
DEFINE company_id INTEGER,
    prod_id INTEGER,
    Quantity INTEGER,
    Amount_each MONEY(6)
SELECT amount
INTO amount_each
FROM products
WHERE product_id = prod_id
IF (status = NOTFOUND)
    THEN
        ERROR "Product", prod_id, "does not exist"
        RETURN (FALSE)
    END IF
CALL
    process_company_order(company_id, prod_id, quantity,
    amount_each)
END FUNCTION

Program Segment 4.4.2-1 Rows in the database are bound to program variables.

In the Program Segment 4.4.2-1 the “process_company_order” call is a stored procedure. This example shows that fourth-generation language (4GL) syntax is certainly more readable than C or C++ (Vermulen, 1996). However, using 4GLs also has some disadvantages. For example, they cannot import class libraries from C++, the lowest-common denominator language of choice. Also they require more work to integrate or embed with other systems.

SQL can be embedded in another language, such as C or C++ (Anstey, 1998). An example of an Oracle program using embedded SQL is shown in Program Segment 4.4.2-2. It shows that the results of query, in this case the person_name, are bound to a variable in C:

main() {
    exec sql include sqlca;
    exec SQL begin declare section;
        int person_id
        char person_name(20);
    exec sql end declare section;
    printf("Enter person's id:");
    scanf("%d", &person_id);
    exec sql select name
        from people_table
Improvement Methods for Legacy Database Systems

Chapter 4

where id = :person_id
into :person_name;
printf("That person's name is: %s",person_name);
exit();
}

Program Segment 4.4.2-2 Oracle code to bind the results of a query to a variable.

Embedded SQL can be used to store a class hierarchy in a relational database. But this can be a difficult exercise. "Collapsing the hierarchy into a single table can be unsatisfactory" (Loomis, 1990) because some columns are irrelevant when there is no equivalent attribute in a subclass. In other words, it is possible to place two subclasses of EMPLOYEE (HOURLY_EMPLOYEE and SALARIED_EMPLOYEE) into the same table with a little work. The Program Segment 4.4.2-3 assumes that a C++ employee (emp) object has been defined, and that emp_type can be used to determine which subclass the employee falls under, in order to store the object in a relational database:

```
EXEC SQL INSERT INTO Employee_Table (soc_no, emp_name, birth_date, dept_name);
VALUES (:emp->soc_no, :emp->name, :emp->birth_date, :emp->dept_name);
If (emp->type ==1)
  EXEC SQL INSERT INTO Hourly_employee_Table (soc_no, hourly_rate, overtime_rate, max_overtime_hours);
VALUES (:emp->soc_no, :emp->hourly_rate, :emp->overtime_rate, :emp->max_overtime_hours);
else if (emp->type ==2)
  EXEC SQL INSERT INTO Salaried_employee_Table (soc_no, monthly_rate, bonus_pct);
  VALUES (:emp->bonus_pct);
EXEC SQL COMMIT WORK RELEASE;
```

Program Segment 4.4.2-3 Storing an object in a relational database.

4.4.3. Direct object mapping

An alternative approach to linking a relational database with objects is to establish a direct object mapping, or correspondence between classes and tables, which allows objects to be persistently stored in a relational database. This is the
approach taken by Sybase™ with its “Object Connect” product, which it created by licensing software from Persistence Software, Inc.™ Object mapping software usually provides a way to specify how classes and their attributes map to tables (Vermulen, 1996).

4.4.4. Bridging Object-Oriented programming to relational databases

One of the techniques used to connect OO concepts to existing relational databases is to build an OO application that accesses relational data. Persistence Software is an application development tool that uses an automatic code generator to merge C++ applications with relational data. As described in (Keller, 1993) this automation could improve the productivity and quality of C++ applications. The approach provides a way for companies to move to C++ applications while leveraging investments in relational data.

As it is indicated by Keller that there is an alternative approach for interfacing C++ classes to a relational database through a C++ class library containing classes that model relational entities such as tables, tuples and fields. To build an application, the developer customises these building blocks by specifying a mapping between a generic tuple instance and a specific class instance. Inheritance, associations and runtime behaviours must be hand-coded.

In contrast, a code generator produces C++ classes directly from the application object model information. The developer can concentrate on the correctness of the model while the code generator automatically creates the database interface portion of the application. The code generator can also create a table schema based on the object model, or map the object model into an existing table schema.

The application object model supplies the information about inheritance, attributes and associations; the code generator translates these inputs into appropriate table structures and C++ class definitions. Classes may be related via
4.5. Interfacing Objects with the Relational DBMS

With the context of object features in place, it is possible to see the problems as well as the opportunities that appear when trying to use a relational database as the data store for an object application. Five different products are described in this section to show the distinction in what they objectify (SQL statements, screen entities, or the ER model). The most powerful mediators seem to be those that objectify elements of the ER model (Thomson, 1993). The five products are COMMONBASE™, SMALLTALK/SQL™, POWERBUILDER™, NEXT'S DBKIT™, and PERSISTENCE™. They are described not because they are the best, but to give different flavours of the available options.

4.5.1. COMMONBASE

Marketed by Image soft™, COMMONBASE is a C++ class library that objectifies the SQL statements. It encapsulates the low-level components of SQL statements so that the programmer can "code in objects" without having verbose, proprietary and obscure call-level SQL code sprinkled throughout the classes and methods. One of the product's limitations is that it knows about tables, but it knows nothing about their keys and relationships. Its level of abstraction is low. As a result, it offers no automated code generation of join conditions, concatenated keys, or subtype / super-type relationships that can help to create a class hierarchy.

4.5.2. SMALLTALK/SQL

A product of Synergistic Solutions Inc.™, SMALLTALK/SQL is a set of classes that insulates the developer from proprietary call-level SQL details and offers a higher level of abstraction of which to operate (Thomson, 1993). Smalltalk/SQL objectifies tables but it does not gain detailed knowledge about keys and relationships among tables. As a result, the tool does not simplify the
management of joins or subtype/super-type relationships. An interesting feature is the ability to auto-generate Smalltalk classes from existing database tables as well as tables from Smalltalk classes. Source code is provided so that custom classes can be easily created.

4.5.3. POWERBUILDER

A product of Powersoft Corp., POWERBUILDER is based on a proprietary scripting language, PowerScript. PowerScript's syntax is based on the (procedural) BASIC model, so its object-orientation is not strictly enforced. The product's object orientation lies mainly in the inheritance capabilities of four types of objects: windows, user objects, DataWindows (a GUI widget such as a list-box), and menus. The emphasis is on GUI programming rather than on internal objects such as tables (as in Smalltalk/SQL). The widget can be directed to inherit changes made to the parent widget, or ignore them. The inheritance can include the widget's visual characteristics and the SQL statement that populates it.

An object library can also be set up so that the preceded objects can be reused across a project team. This object library is designed in an open fashion so it will work with an emerging object communications protocols, such as Object Management Group's Object Request Broker.

A possible disadvantage of the proprietary strategy is that it may not be easy to integrate PowerBuilder objects with new commercial class libraries as they become available with the increasing popularity of object orientation. Also, since the objects created have a close relation to screen widgets, ER diagram details such abstraction of concatenated keys or subtypes within super-types may not be well suited for integration into PowerBuilder's object model. For the same reason, it could be difficult for batch programs to share code with the online side. Similar problems occur if one needs to replace the GUI with a radically different input device, such as a bar-code reader.
4.5.4. NEXT’S DBKIT

NEXT’S DBKIT is based on Objective C, a C variant that supports dynamic binding in the Smalltalk style. In contrast to PowerBuilder’s emphasis on GUI objects, DBKIT objectifies the ER model. The first task in building a DBKIT application is to define such a model through the Model Builder, a visual ER tool that allows the specification of standardised join conditions and other data relationships. This tool makes a connection between the native DBMS Adapter and the Data Access layer. It is easy to use, but offers a limited set of graphic capabilities, far short of a specialised ER diagram tool.

Once the model is built, the developer turns to the visual objects available in the Data Access Layer (for example, scrollable browsers, display objects for Rich Text Format, and display windows for images and other binary large objects), as well as the standard widgets of the Application Kit (AppKit) and Interface Builder. This action is done in co-ordination with a target / action selection (for example, Save Changes) in the Interface Builder’s Button Inspector Window. A developer can click on an interface object (for example a button) and drag the mouse over the DBModule icon, representing the ER model created with the Model Builder.

4.5.5. PERSISTENCE v.1.0

PERSISTENCE v.1.0 is produced by Persistence Software™ of San Mateo, California (Cattle, 1994). This product returns us to the C++ world and a class-based, compiled approach, in place of DBKIT’s visual mode of operation. Similar to DBKIT, Persistence objectifies elements of the ER model. The development process is similar to DBKIT’s, with the ER model’s description for the underlying relational tables. The developer uses five specialised windows for the purpose: the main window, Class Editor, Attribute Editor, Relationship Editor, and Foreign Key Editor.

As an example of the rich set of capabilities in Persistence, the modeller supports both Oracle and Rumbaugh ER notations. Since the underlying code is
C++, the modeller naturally supports higher levels of abstraction that are typical in a RDBMS application, such as classes within classes and subtype/super-type relationships: for instance, Employee and Customer classes could be defined as subtypes of an abstract Person class. In this case, Persistence would create tables only for Employee and Customer, although the developer could still perform queries on the Person class and define methods and attributes for Person, which Employee and Customer would inherit. Alternatively, if the ER model specifies that each Employee must be related to a Person, Persistence would map Person and Employee to separate tables.

All of these mediators above provide some type of “objectified” view of relational data, whether the object is an SQL statement (CommonBase), screen widget (PowerBuilder), or an element of the ER diagram (Smalltalk/SQL, DBKit, and Persistence). The most powerful (probably DBKIT and Persistence) leverage the ER diagram to create higher levels of abstraction that promise to simplify SQL data handling considerably.

4.6. Object-Relational DBMSs

The Object-Relational DBMS is called the new promising class of database management systems that will replace relational systems to become the next great wave of database technology (Stonebraker, 1996). The spectrum of application areas covered by the object-relational DBMS ranges from video and graphic asset management in the entertainment industry to time series analysis problems in the financial services market, scientific databases, and Geographic Information Systems (GISs). In addition, the exploring market for multimedia data, often accessed through the World Wide Web, is best served by object-relational technology.

The four quadrants matrix of Stonebreaker divides database applications into four categories, as Figure 4.6-1 shows Object-Relational Databases are most suitable for applications with complex data that require queries.
# Chapter 4  
Review of OO Improvement Methods for Legacy Database Systems

<table>
<thead>
<tr>
<th>Simple Data</th>
<th>Filing Database systems</th>
<th>Relational Database systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex Data</td>
<td>Object-Oriented Database systems</td>
<td>Object-Relational Database Systems</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No Query</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 4.6-1 Database applications are designed according their needs</td>
<td></td>
</tr>
</tbody>
</table>

Object-Relational DBMSs have characteristics of both an RDBMS and an ODBMS in that Object-Relational DBMSs store both objects and tuples in the DBMS. Figure 4.6-2 shows five different products that allow using objects in DBMSs. Those products are compared by three criteria (Object Programming Language Binding, Database Engine, and Database Storage Format). The chart shows each type of product relative to increasing number of users and increasing data complexity.

![Diagram showing areas where each product type generally applies](image)

**Figure 4.6-2 Areas where each of product types generally apply**

It is obvious how the Object-Relational DBMSs cover a much larger spectrum, mainly because the category has three different types of client architectures, depending on the product. Those products that have transparent
program integration with an object programming language can handle data as complex as that handled by an object DBMSs. On the other hand, the other possible clients for Object-Relational DBMSs are much like Relational DBMSs, Object-Relational Mapping, or Object Data Manager.

4.7. Summary

There are many ways to integrate a relational model with an object-oriented model. The main objective of this integration is to use object-oriented features on existing relational database systems. Converting a relational database system to an object-oriented system seems to be the most suitable type of such integration. Other alternatives, such as (1) connecting objects to a relational database by either binding SQL statements to objects or using a 4GL that extends SQL or (2) Using direct object mapping technique that allows objects to be persistently stored in a relational database or (3) Bridging, which uses object-oriented programming language in an application that accesses an existing relational database and finally (4) Interfacing object-oriented capabilities to relational databases which objectifies certain aspects of relational database like SQL statements or Entity Relationship (ER) modelling to give a better picture when using the database system.

The reason behind the suitability of the conversion method is that other mentioned object-oriented improvement methods do not exploit the full benefit of object-oriented tools. For example, when extending the converted database system, it is the new system that is being worked on and all the object-oriented concepts can be applied directly. However, when using other object-oriented improvement methods the extension will be applied to the original legacy database system and, as a result, there would be a need to rewrite the object code that corresponds to the schema in the legacy database system to suit the new extension. The next chapter will show some existing advantages gained from converting to object-oriented database model.
In general there are four types of DBMSs as Michael Stonebraker (Stonebraker, 1996) shows in his well known (DBMS matrix). There is no one answer to the question of whether an RDBMS or an ODBMS is better, as Barry Douglas claims (Douglas, 1996). The answer depends on several factors, such as the ability of the model to meet the complexity of the data structure for the new system. Finally, we should consider the most appropriate cost of the selected model with the relation to other selected competitive models.
Chapter 5: The Claimed Advantages of Conversion to OO Databases

5. Introduction

Before diving into details about what are the advantages of converting a Relational Database System to an Object-Oriented Database System, it is important to look at the reasons why the relational model may be unsatisfactory and why in particular, an object-oriented model may be better. There are many who discourage using the relational model for certain applications and advise using an object-oriented model instead. One argument is that the relational model lacks the ability to model and manage complex data efficiently (Bancilhon, 1988). It's claimed that the object-oriented modeling has more expressive power to design and model database applications (Blaha, 1998).

The relational model has been an impressive success for the developing business applications and is still being used effectively in that area. The relational database systems together with high-level languages can develop almost any application if we disregard how slow they can be. This success was of interest in many other fields such as Computer Aided Design (CAD), Computer Aided Software Engineering (CASE), and Computer Aided Manufacturing (CAM), but unfortunately relational databases failed to comply with the performance needs of those new technologies.

The relational model has made a great impact on the database market. It offered a better way to manage applications than earlier approaches. The turn is for object-oriented model to present its significant potential for databases. Object-orientation is expected to be an enabling technology that allows developers to do certain things more easily that with the relational model.

The idea of converting an existing relational model to an object-oriented model does not mean that the latter would replace the earlier and make it history. Converting to an object-oriented model is really an approach that might be called
"the right tool for the job" or taking advantage of the right tool to help accomplish the job.

5.1. Advantages of Conversion to Object-Oriented Databases

5.1.1. Better Understandability

A major problem with currently existing database systems is that the people actually do not have a conceptual understanding of the data of the system. Such understanding could be obtained by describing the data using a conceptual data model (Johnannesson, 1989). Oracle 8 is Oracle's first version of the database to incorporate object-oriented technology. It defines an object-oriented database as a one that can store data, the relationship of the data, and the way it interacts with other data.

Unlike the relational database approach that deals with data at the lowest possible level, a series of columns and rows the object-oriented approach deals with data at a much higher level; it deals with the objects surrounding the data. With object-oriented databases, when dealing with the customer, you deal with an object called "customer". When dealing with an order, you reference an object called "order". Since the object database understands the object customer and all its relationships, it can easily deal with the object customer and all that is needed to work with it.

Oracle 8 uses the SQL declarative language to define and manipulate data in the database. The SQL Create command is used to build a table in a relational database model. The syntax is quite simple e.g.

```
Create student_table Table ( StId NUMBER NOT NULL, StName VARCHAR2(200) NOT NULL, Street VARCHAR2(200) NOT NULL, City VARCHAR2(200) NOT NULL, PostCode VARCHAR2(200) NOT NULL, StTel VARCHAR2(20), PRIMARY KEY (StId));
```

Currently Oracle 8 uses objects as user defined data types, which is a step forward in data modelling to enrich the expressiveness of the relational model.
Chapter 5 The Claimed Advantages Of Conversion To OO Databases

The student in the above example could be considered as an object, as well as the address and the telephone number if desired. Address and telephone number objects could be nested or sub-objects of student object or may be referenced by a relation between different classes. The create type is the new command in Oracle 8 for defining an object which, as a result, gives birth to the new command create table of objects. The following listing demonstrates creating a new data type as an object called Student Type.

**Create type** Student_Type as object (StId NUMBER NOT NULL,
StName VARCHAR2(200) NOT NULL,
Street VARCHAR(200) NOT NULL,
City VARCHAR(200) NOT NULL,
PostCode VARCHAR(200) NOT NULL,
StTel VARCHAR(20));

To use the created type in a table simply we use the command Create Table Student_Table of (object) Student_Type or we can include some other attribute along with what is already defined: Create Table Module (Room_name varchar (20), undergradst Student_Type primary StId);

This facility of object user defined data type works as a template for other object tables. Because there is a type Student_Type, it is possible to create numerous object tables of the same type. For example, it is possible to create an object table Module_Post_Grads also of type Student_Type. Without this ability, one would need to define each table individually.

Another advantage is the ability to introduce variations when creating multiple tables such as Under_grad table, Post_Grad table etc... . The Create table statement that created Under_Grad object table type may have defined a primary key constraint on the StId column. This constraint applies only to this object table (Under_Grad). Another object table of the same type (Post_Grad) might not have this constraint and could use another column as a primary key e.g.
Chapter 5 The Claimed Advantages Of Conversion To OO Databases

Create Table Module2 (Room_Name VARCHAR(20),
  Research_ST Student_Type primary StTel);

5.1.2. Usage in Heterogeneous Databases

A system for managing a heterogeneous mix of databases is important as a migration path from relational databases to object-oriented databases, and is essential for object-oriented database systems to take root. It could be highly desirable to allow the user to access a heterogeneous mix of databases under the illusion of a single common data model (Kim, 1990). For example, suppose that an Employee database is managed by a relational database system, a Product database is managed by a hierarchical database system and a Company database is managed by an object-oriented database system. An object-oriented data model may be used as the common data model for presenting the schemas of these different databases to the user.

The richness of an object-oriented data model makes it appropriate for use as the common data model for representing a broad range of data models. Further, since object-oriented design and programming promotes extensibility, it may be used for designing and implementing a system to manage a heterogeneous mix of databases, which can accommodate the addition of new types of databases.

As large centralised databases are being replaced by distributed databases (Client/Server), these databases may be implemented with a conventional network and hierarchical DBMS, the current relational DBMSs, or the object-oriented DBMS. The systems which are gaining popularity use more than one database model, which then require a means of communication between systems or a transformation of the schema to more expressive schemas. Conversion is usually preferable to a better understandable model in such heterogeneous systems (Stanisic, 1999)

5.1.3. Future Extensibility

The question that arises is: how can a currently existing database that has been built over time be converted to more desirable model? One of the solutions is to
Chapter 5 The Claimed Advantages Of Conversion To OO Databases

extract a conceptual Entity Relationship Model (ERM) from the existing relational database. This conceptual model then can be used in any of the translation process to convert the relational database to a different DBMS (Davis, 1987). Having the relational model presented in an ERM makes it easy to convert to an object-oriented model (Biskup, 1995).

The traditional use of a RDBMS has been limited by the rather minimal set of data types (string, integer, floating-point number, date, currency) supported by the mainstream data management systems. For Object Relational Database Management Systems (ORDBMSs), extensibility means that the ORDBMS has provisions for the user to define and support new data types. Initially, it may mean the vendor, or its value added resellers, supply the extended support (e.g., for multimedia data types such as images, audio, and video).

5.1.4. New features

One of the claimed advantages of moving from the relational database using object-oriented tools is the continuous growth of consumers' demands to get better facilities. The Department of Agricultural Economics and Animal Science, Michigan State University carried out an experiment of this advantage for the process of storing information on cows. To further extend the relational database used, the application employed had to save photographic pictures of calves for breeding purposes (Rahn 1992).

The advantage of object-oriented databases in this situation cannot be gained using other alternatives such as de-normalisation of a normal relational database or replacing the current hardware with a more powerful one. It is the need to store various data types such as photographic pictures that gives a good reason to put aside the limited data type of the relational data model. The relational systems do not have the tools to express different forms of data especially where the rules for normalised data forces the data to have an atomic value in each cell of the table (i.e. an inability to express audio, visual, pictorial or video information). A solution to this dilemma is using Binary Large Objects
(BLOB) which are clearly not atomic but they meet the violations of normalisation rules.

5.1.5. Computational advantages

The RDBMSs typically confine all processing to the SQL language and its operations (SELECT/PROJECT/JOIN and INSERT/UPDATE/DELETE), ODBMSs allow the use of host object languages like C++, Java, and Smalltalk directly on the objects "in the database"; that is, instead of translating back and forth between application language structures (COBOL, C, etc.) and database structures (SQL), application programmers can simply use the object language to create and access objects through the methods. The database system maintains the persistence, integrity, recoverability, and concurrency of those same objects (McClure, S 1997).

5.1.6. Development advantages

One of the advantages of using ODBMS is that Objects in ODBMS are used in the same way they are stored; they don't have to be translated into a database sub language such as SQL with Open Database Connectivity (ODBC) or Java Database connectivity (JDBC) instead, developers write in an object-oriented programming language such as Java or C++. For this reason, developers write less code than if they were using a relational database management system. In many cases, this code could be as much as 40 percent less (www.odbmsfacts.com).

5.1.7. Production advantages

Another benefit, which is related to the way objects are stored and used, occurs in production. With complex data ODBMS can give a performance ten to a thousand times faster than a RDBMS (Barry & Associates, Inc.). That's because when the data is read off the disk, it is already in the format that Java or C++ uses. No translation is needed from SQL to another language, as is the case with ODBC or JDBC. The range in performance gain depends on the complexity of the data (www.odbmsfacts.com).
5.1.8. Active Objects

A new generation of spatial databases, Graphic Information Systems (GIS) and mapping systems, is adopting Object-Oriented concepts - such as Encapsulation, Referencing, Inheritance, and Polymorphism - and applying them to storage, retrieval, analysis and display of location-related information. Mapping and geodata systems use OO to model data, with the features of a map (the buildings, roads and forests found in the real world) becoming objects. Object classes are defined to determine the common behaviors of similar features, and these are arranged in an inheritance hierarchy (e.g. a school is a kind of building). In an OO spatial database such as Laser Scan’s Gothic, the data model or schema can be defined by the customer, and then populated with objects corresponding to the real world.

To show the merit of using object-oriented database systems let us take, for example, house cartography. A house object has not just coordinates and other numeric or character attributes. Because it belongs to the object class House, and inherits from Building, it knows about the properties of houses (that they have owners, addresses and neighbours, and they have areas, heights and dates of building). It can also know about relationships with other objects (neighbouring houses, access from a particular road, ownership of a garage). Finally, it has behaviours (methods), which make it behave differently from other classes of different features (Hardy, P.G, 2000).

5.1.9. Using XML

The need to efficiently store and manage large amounts of XML data is rapidly increasing due to the growing use of XML as an improved web format, as the native data format for a variety of applications and as a standard interchange format especially in the e-business domain (Bohme,T, 2001). Mapping between XML and objects is the simplest form of mapping because the object model and XML model are very similar. In fact, if we parse XML, we get a tree, which, is a common data structure for an object model (Barry & Associates, 1999). Mapping between XML and relational tables is more complicated than mapping.
between XML and objects because the two models are different as figure 6.1-1 shows.
Chapter 5  The Claimed Advantages Of Conversion To OO Databases

Figure 5.1-1 Advantage of mapping between XML and objects over mapping between XML and tables
5.2. Summary

One might ask why objects? In Management Science they say that knowing the reason behind a failure is the first step to success. Information Systems Managers have suffered for a long time striving to improve software quality, with shorter time development to market, increased programming productivity and greater reusability of code. They also expect easier application maintenance and improved ability to deal more effectively with the increased complex applications.

Object technology with its new rich features arrived with a promising intention to change the way of designing and building software. Actually, object technology is a family of techniques and tools that make it easier to do good software engineering. Such tools express the data model in a more understandable way, which gives object-oriented database systems supremacy over other database management systems.

The consequences of proper data modelling are extensibility, easier development and production of code and reusability. In addition, the object-oriented model manages some of the new demands such as the storage of audio, visual, pictorial, and video information in a more natural way. The advantages of managing such objects have meant that spatial database systems have adopted object-oriented concepts to manage their data.

All these different advantages of improving the performance of the database management system and thinking ahead for better maintainability and extensibility is an advantage in reducing the cost of the enterprise in the long-run. There are other alternative improvement methods but converting to object-orientation in most cases fits more naturally.
Chapter 6: An In-Depth Example to Illustrate the Advantages Of Conversion To O-O Databases

6. Introduction

This chapter is based on the paper "Systems Evolution: Converting a Non-Normalised Relational Database to an Object-Oriented Database" given at the 12th International Conference Software & Engineering and their Applications ICSSAE '99 in Paris in December 1999. See appendix A.

The transformation of any system to another requires an investigation of not only their components but also their sub-components. The investigation then will reveal the important similarities and differences that will help the conversion process from the source system to the target one; database systems are no exception (Stonebraker, 1996).

The relational model lacks the ability to model and manage complex data efficiently. It has difficulty in dealing with complex data such as recognizing identification in data, accessing data using traversals, many-to-many relationships, and frequent use of type codes. Object-Oriented Databases (OODB) seem to handle this complex data in a more natural way, and as a result, it gives better performance and retrieval of data (Timothy, 1995).

As the demand for OODB rises and the popularity of relational databases remains, developers are urged to seek a quick fix via an object-to-relational transformation layer. This method offers the benefits of object-orientation to new systems, while it retains the information in the legacy systems.

The pure relational model is inadequate because it does not have the facility to describe complex data such as Computer Aided Design (CAD), Computer Aided Manufacturing (CAM), and Computer Aided Software Engineering (CASE). While the purely object-oriented model is not entirely appropriate for database use because it is not mature enough to cope with the needs of database requirements such as permanent memory. As a result, existing
Chapter 6  An In-Depth Example to Illustrate the Advantages Of Conversion to O-O Databases

relational databases have found some way out of their dilemma by using some object-oriented concepts to solve the problems that they are facing. Oracle8 uses nested tables and aggregate objects despite the fact that this is a clear violation of first normal form (Anstey, 1998).

There are other ways to deal with complex data by using techniques such as interfacing, mapping, bridging, and conversion from legacy systems to the object-oriented model these methods are described in chapter 4. All of the above approaches show an advantage over the existing relational database systems. This chapter shows another advantage of converting a relational database system to an object-oriented system.

This chapter gives an example of a conversion of a relational database to an object-oriented database. The relational system had type code complexity and many-to-many relationships. The example illustrates the benefits to be gained by converting to an object-oriented database as a result of the inheritance and collection types. The chapter also shows a new advantage of conversion when relational system is not fully normalised- the conversion process acting like an informal normalisation of the data.

6.1. Converting a Non-Normalised Relational Database to an Object-Oriented Database

To design with an Object-Oriented Database Model (OODM) it is important to bear in mind its various distinctive features such as object identity, encapsulation, and inheritance. Many researchers state that a suitable OODM is expected to use such tools in its model (Kim, 1990), (Lausen, 1997). The following concepts distinguish the OODM from conventional conceptual models:

- **Nesting**: It allows one object to be a component of another.

- **Object Identifier**: Any real world entity is an object, with which is associated a system-wide unique identifier. The system should provide an Object Identity independent of any value of its attributes.
Chapter 6  
An In-Depth Example to Illustrate the Advantages Of Conversion to O-O Databases

- **Attributes and Methods**: Objects can be equipped with behaviour, which links specific operations exclusively to these entities. An object has one or more attributes, and one or more methods that operate on the values of the attributes. The value of an attribute of an object is also an object. An attribute of an object may take on a single value or a set of values.

- **Class Hierarchy and Inheritance**: The structure and behaviour of objects can be inherited hierarchically. The class in a system forms a hierarchy or a rooted directed acyclic graph. Suppose, for a class C and a set of lower-level classes \( \{S_i\} \) connected to C, a class in the set \( \{S_i\} \) is a specialisation of the class C, and conversely the class C is the generalisation of the classes in the set \( \{S_i\} \). The classes in \( \{S_i\} \) are subclasses of the class C; and any class in \( \{S_i\} \) inherits all the attributes and methods of the class C and may have additional attributes and methods.

- **Extendibility**: They have the ability to extend the structure and behaviour function in a logical sense (Lausen, 1997).

- **Object-oriented database systems**: allow the modelling of objects, relationships, and complex structures in a way that in many applications is more appropriate than traditional systems can offer. For this reason alone they are of increasing interest in a host of applications. If an OODB is created from scratch, its design can make use of several existing techniques. However, if a database already exists, then it is a different story, for this situation many researchers (Fahrner, 1995), (Stanisic, 1999) have proposed a step-by-step transformation from the relational to the object-oriented model.

### 6.2. The problem

Previous research shows that the normalisation process is an important factor for conversion, for example, (Johnannesson, 1989) indicates that to simplify the translation process of a relational schema to a conceptual schema the former needs to be in a third normal form (3NF). Others, like (Davis, 1987), also place a
restriction on the relations to be in 3NF. This implies that it necessary to
normalise a relational database system to be converted into an object-oriented
system, but is this really necessary? If it is, what is the degree of normalisation
required (1NF, 2NF, or 3NF)?

(Fahrner, 1995) shows that the relational database must be normalised (at
least up to 3NF) in order to convert its schema to an object-oriented one or to
another conceptual data model. Other research shows the possibility of converting
an entity relationship model to an object oriented model (Biskup, 1995) but a
properly formulated entity relationship model will give a data structure
normalised at least up to 3NF.

The richness of the object-oriented data model makes it possible to
represent a broad range of databases including relational databases. However, to
directly convert a non-normalised relational database to an object-oriented
database would eliminate the need for the intermediate process of normalisation.
Furthermore, we will be able to see if the conversion process will handle the
partial and transitive dependencies (i.e. normalisation). Alternatively, it would be
beneficial to find out if the conversion process would still preserve the different
anomalies in the source database and see the impact on the resulting object­
oriented database.

6.2.1. Method of Converting

In most database applications, there are numerous objects (data) of the same
type. This kind of data is called the Type Code (Douglas, 1996) and is usually
represented by a column in a relational database table. (Harrington, 2000)
explains it as the “Whole-Part” relationship. For example a Person type could be a
Student, Lecturer, Administrator, etc. Object-oriented models type code
complexity in a more natural way than conventional systems since a class in an
object-oriented model represents a set of objects of a particular type (such as
Person).
Let us use the example of a non-normalised relational database with a type code complexity feature as shown in Schema 6.2-1. Attributes are defined as follows:

Id# represents person's identity number (made by database designer). Modu# stands for module number where Modnam stands for module name. Room# stands for the room number where the lecture is taking place and Time stands for the time of that lecture. Ptype represents the type of person and Pname stands for the person's name where D.O.B stands for his date of birth. Hdate represents employees hire date, where Tdate stands for professor tenure date and Entdate stands for students' date of enrolment.

Schema 6.2-1: The relational schema.

In order to normalise the Schema 6.2-1 to 2NF we need to get rid of all partial dependencies as shown in the Schema 6.2-2.

To eliminate redundancy, the Schema 6.2-1 needs to be broken up into Person Schema 6.2-3 and Module Schema 6.2-4.


Schema 6.2-3 needs to be split into two schemas, one that is dependant on Modu# as in Schema 6.2-5 and the other one is not as in Schema 6.2-6. The
Chapter 6: An In-Depth Example to Illustrate the Advantages of Conversion to O-O Databases

Modu# dependency distinguishes between persons that study or teach and persons that do not (such as technicians, caretakers, etc.).

<table>
<thead>
<tr>
<th>Id#</th>
<th>Modu#</th>
<th>Ptype</th>
<th>Pname</th>
<th>D.O.B</th>
<th>Hdate</th>
<th>Tdate</th>
<th>Entdate</th>
</tr>
</thead>
</table>

Schema 6.2-5: Non-support schema dependent on Modu#.

<table>
<thead>
<tr>
<th>Id#</th>
<th>Ptype</th>
<th>Pname</th>
<th>D.O.B</th>
<th>Hdate</th>
</tr>
</thead>
</table>

Schema 6.2-6: Support schema dependent on Modu#.

In Schema 6.2-5 the Hire date, Entry date and Module number attributes distinguish the type of person according to the attribute dependency. For example, Modu# and Entry date describe a student schema (Schema 6.2-7), while Modu# and Hire date describe a teacher schema (Schema 6.2-8). Thus, the 2NF decomposition of Non-Support schema will result into two different schemas (Schema 6.2-7 and Schema 6.2-8)

<table>
<thead>
<tr>
<th>Id#</th>
<th>Modu#</th>
<th>Ptype</th>
<th>Pname</th>
<th>D.O.B</th>
<th>Entdate</th>
</tr>
</thead>
</table>

Schema 6.2-7: Student schema dependent on Entry date.

<table>
<thead>
<tr>
<th>Id#</th>
<th>Modu#</th>
<th>Ptype</th>
<th>Pname</th>
<th>D.O.B</th>
<th>Hdate</th>
<th>Tdate</th>
</tr>
</thead>
</table>

Schema 6.2-8: Teacher schema dependent on Hire date.

Schema 6.2-8 is in 2NF but not in 3NF because of the transitive dependency of Tenure date of Professor on the Id#. A 3NF decomposition of Schema 6.2-8 is shown in Schema 6.2-9 and Schema 6.2-10.

<table>
<thead>
<tr>
<th>Id#</th>
<th>Modu#</th>
<th>Ptype</th>
<th>Pname</th>
<th>D.O.B</th>
<th>Hdate</th>
</tr>
</thead>
</table>

Schema 6.2-9: Non-Professor Schema in 3NF.

<table>
<thead>
<tr>
<th>Id#</th>
<th>Ptype</th>
<th>Tdate</th>
</tr>
</thead>
</table>

Schema 6.2-10: Professor Schema in 3NF.

There are two many-to-many relationships in this model, the first is between Staff and Module; the second is between Student and Module. The way to solve this complexity is to add intersection entities to each many-to-many relationship. This will force differentiation between Student Identity number (StId#) as in (Schema 6.2-11) and Teacher Identity number (TeId#) as in Schema 6.2-12 to get rid of any redundant storage of data. As a result, the identity number...
Chapter 6  An In-Depth Example to Illustrate the Advantages Of 
Conversion to O-O Databases

of work-Study will appear in both schemas Teacher schema and Student schema 
and will use up storage in two different places.

<table>
<thead>
<tr>
<th>StId#</th>
<th>Modu#</th>
</tr>
</thead>
</table>

Schema 6.2-11: Student/Module intersection entity to remove redundant 
storage of data.

<table>
<thead>
<tr>
<th>TeId#</th>
<th>Modu#</th>
</tr>
</thead>
</table>

Schema 6.2-12: Teacher/Module intersection entity

The intersection entities were added to ensure integrity and eliminate 
redundancy as mentioned earlier. Modu# will be removed from both schemas 
(Student and Teacher). At the same time, StId# and TeId# will be removed from 
Module schema. Because these attributes are primary keys in their original 
schemas they will be called foreign keys in the intersection entities and will have 
referential integrity constraint.

Now all types have been classified in their 3NF schemas and are ready to be 
converted to object-oriented schema.


The process of converting a relational schema with a type code complexity to 
an object-oriented schema is as follow:

- Generating a tentative class for each type of the type code complexity.
- Factoring attributes up higher in the class hierarchy.
- Some attributes are specialised to certain entities.

6.3.1. Generate Tentative Classes

From Schema 6.2-7, student tentative class can be generated as shown in 
Figure 6.3.1-1 because every student has an entry date.
Chapter 6

An In-Depth Example to Illustrate the Advantages Of Conversion to O-O Databases

Student Work-study

| Id# | Id# |
| Modu# | Modu# |
| Name | Name |
| Entry date | Hire date |

Figure 6.3.1-1: Student tentative class.

Student and Work-study classes have the same attributes for being students except that Work-study class has a special attribute (Hire date). The common attributes will be factored up in a higher class and will be named Student class where Work-study is a sub-class (see Figure 6.3.1-2) of Student class.

![Figure 6.3.1-2: Work-study tentative class.](image)

From Schema 6.2-8 we can generate tentative classes for Teachers since they all have Hire date and Modu#.

Professor

| Id# | Id# |
| Modu# | Modu# |
| Name | Name |
| Hire date | Hire date |
| Tenure date | |

Instructor

| Id# |
| Modu# |
| Name |
| Hire date |

Work-study

| Id# |
| Modu# |
| Name |
| Hire date |
| Entry date |

Figure 6.3.1-3: Tentative classes for teachers

All common attributes could be factored out except for Entry date of Work-study as explained earlier and Tenure date of Professor. The new supper class of the three sub classes could be named Teacher as in Figure 6.3.1-4.
An In-Depth Example to Illustrate the Advantages Of Conversion to O-O Databases

Figure 6.3.1-4: Teacher super class with its subclasses

From Schema 6.2-6 we can generate a tentative class of Support who has the Hire date attribute but does not have the Modu# attribute. This class is considered part of Employee class that shares the rest of attributes except the teaching attribute. The rest of common attributes in Teacher class can be factored up in another super class called employee as in Figure 6.3.1-5.
Figure 6.3.1-5: Factoring the common attributes to form Employee class.

Looking at Student class in Figure 6.3.1-1 it shows that the class shares the same common attributes as in Teacher class except for the Hire date, besides it has an Entry date in its class. We can factor up all common attributes that the two classes (Employee and Student) share and form another super class called Person as in Figure 6.3.1-6.

Work-study has an Entry date attribute in Student Class and Hire date in the Teacher class thus it can inherit from both classes (multiple inheritance). Multiple inheritance in this model solves the problem of Student identity and Work-Study identity redundancy appeared earlier in the relational model. The super class Person will keep one object identity for each Student object that is Work-Study.

At this stage the Module class will be added to the object-oriented schema. The inverse relation in object-oriented will replace the intersection entity job in
the relational model. The object identity on both sides of the relationship must match. Each (Student or Teacher) must relate to his/her Module.

Figure 6.3.1-6: Factoring up common attributes for Employee and Student.

To ensure that when object identifier of (Student or Teacher) object is inserted into a Module object, the object identifier of the same Module object is inserted into the (Student or Class) object. This type of relationship integrity, which is somewhat analogous to referential integrity in the relational data model, is handled by specifying inverse relationships. The Module class is added to the object-oriented schema.
Chapter 6  An In-Depth Example to Illustrate the Advantages Of Conversion to O-O Databases

![Diagram showing a hierarchical structure of objects and attributes]

**Figure 6.3.1-7: Object-Oriented schema with Module class.**

The Student class has an attribute called modules (Student.modules, defined as a set). At the same time, the Module class has an attribute called students (Module.students, also defined as a set).

The object-oriented DBMS will ensure the integrity of the relationship by allowing the database designer to include syntax specifying where the inverse object identifier should appear, such as:

```plaintext
modules: (set) Module
  inverse is Module.students
```

for the Module class and

```plaintext
students: (set) Student
  inverse is Student.modules
```

for the Student class.
Chapter 6  An In-Depth Example to Illustrate the Advantages Of Conversion to O-O Databases

Whenever a user or application program inserts or removes an object identifier from the students attribute in Module object, the DBMS will automatically update the modules attribute in related Student object. When a modification is made to the Student object, the DBMS will automatically propagate it to the Module object.

Just as it is up to the database designer to specify referential integrity rules for a relational database, it is the responsibility of an object-oriented database designer to identify the inverse relationship to the DBMS when creating a database schema (Harrington, 2000).

This example shows the advantage of conversion from Relational to Object-Oriented databases. The multiple inheritance facility of the OO database greatly simplifies the structure into a more intuitive schema that eliminates the need for one or more type codes. The set types in object oriented databases and the concept of object identifiers enables objects to efficiently and effectively link to other objects without the need for the time consuming searching that occur when tables are joined in a Relational Databases. These advantages of an OO database allow the extraction of data to be far more efficiently and quickly performed.

6.3.2. Converting a non-normalised schema to an object-oriented schema

The technique use in 6.3 for looking up type code was by using the functional and transitive dependencies on the prime key. This classification categorised each person type to the task appointed to him/her. In the non-normalised version we will use the type code and relational view to get ideas about the object class hierarchy. There is another way of classifying the type code in relational model that is by using the program code as shown in Figure 6.3.2-1
Chapter 6  An In-Depth Example to Illustrate the Advantages Of Conversion to O-O Databases

if Ptype is Student then
    ... specific processing for Student ....
else if Ptype is Professor then
    ... specific processing for Professor ....
else if Ptype is Instructor then
    ... specific processing for Instructor ....
else if Ptype is Work-Study then
    ... specific processing for Work-Study ....

Figure 6.3.2-1: Type checking in application code.

6.3.3. Base table and relational views

The conversion process will use the procedure used in section 6.2.1 using the same example. Table 3 is the base table of Schema 6.2-1 to produce relational views that will give ideas about the object class hierarchy.

<table>
<thead>
<tr>
<th>ID#</th>
<th>Module</th>
<th>Modnam</th>
<th>Room#</th>
<th>Time</th>
<th>Ptype</th>
<th>Pnam</th>
<th>D.O.B</th>
<th>Tdate</th>
<th>Tdate</th>
<th>Enddate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cs101</td>
<td>Intro</td>
<td>HS201</td>
<td>9:00</td>
<td>Student</td>
<td>Jack</td>
<td>1-2-1980</td>
<td>10-1999</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Cs101</td>
<td>Intro</td>
<td>HS201</td>
<td>9:00</td>
<td>Instructor</td>
<td>Pat</td>
<td>1-9-1965</td>
<td>Jun 1990</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>MS401</td>
<td>Project</td>
<td>Sc36</td>
<td>11:00</td>
<td>Professor</td>
<td>Donna</td>
<td>11-11-1948</td>
<td>Aug 1978</td>
<td>Jan 1985</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>MS401</td>
<td>Project</td>
<td>Sc36</td>
<td>11:00</td>
<td>Work-study</td>
<td>Kimi</td>
<td>22-6-1972</td>
<td>Sep 1994</td>
<td></td>
<td>10-1997</td>
</tr>
<tr>
<td>22</td>
<td>Ec301</td>
<td>Micro</td>
<td>Eh109</td>
<td>15:00</td>
<td>Professor</td>
<td>Fred</td>
<td>15-3-1955</td>
<td>Jul 1985</td>
<td>Jan 1992</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Base table using schema 7.2.1

A base table is the way the data is stored physically using an RDBMS. We can see the use of Person Type (Ptype attribute) in this base table to distinguish the various types of people at the university. This is a common way to efficiently store data in a RDBMS. By inspecting the data definition we can determine the type data. Type codes provide hints about the nature of the class hierarchy to be constructed. Type codes provide a good place to start creating a class hierarchy as considering the conversion of the data definition from a relational schema to an object schema. A class hierarchy in an object schema takes the place of type codes in a relational schema. In an ODBMS, type codes are known and
"understood" as the class hierarchy. In a relational schema, type codes are embedded in the data and are not actually "understood" by the DBMS.

An example of this ambiguity, Kim is a postgraduate research student (Work-study) and by looking at the table it is difficult to tell whether Kim is teaching any of the modules or actually studying them. In a relational database the type code is actually numeric or alphanumeric and not spelled out as in the example shown above. Having the relational database designer and developer code each type separately would lead to a more difficulty to code more difficulty to use, more prone to error and therefore more costly.

### 6.3.4. Non-decomposable views

First, we will look at the situation where one type code cannot be decomposed. In the university example, all Persons who are not teachers of any kind are classified as "support". By looking at the view we will be able to tell that it cannot be decomposed any further. A view is a virtual table defined on the base table. Figure 6.3.4-1 shows the definition of the Support View. It selects the one type code of "Support".

<table>
<thead>
<tr>
<th>Name</th>
<th>Date of Birth</th>
<th>Hire Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Don</td>
<td>12-2 1960</td>
<td>May 1993</td>
</tr>
</tbody>
</table>

Select Person Type of Support

<table>
<thead>
<tr>
<th>ID#</th>
<th>Mod#</th>
<th>Modnam</th>
<th>Room#</th>
<th>Time</th>
<th>Ptype</th>
<th>Pnam</th>
<th>D.O.B</th>
<th>Date</th>
<th>Time</th>
<th>Enddate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cs101</td>
<td>Intro</td>
<td>HS201</td>
<td>9:00</td>
<td>Student</td>
<td>Jack</td>
<td>1-2-1980</td>
<td></td>
<td></td>
<td>10-1999</td>
</tr>
<tr>
<td>11</td>
<td>Cs101</td>
<td>Intro</td>
<td>HS201</td>
<td>9:00</td>
<td>Instructor</td>
<td>Pat</td>
<td>1-9-1965</td>
<td>Jun 1990</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>MS401</td>
<td>Project</td>
<td>Sc36</td>
<td>11:00</td>
<td>Professor</td>
<td>Donna</td>
<td>11-11-948</td>
<td>Aug 1978</td>
<td>Jan 1985</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>MS401</td>
<td>Project</td>
<td>Sc36</td>
<td>11:00</td>
<td>Work-study</td>
<td>Kim</td>
<td>22-6-1972</td>
<td>Sep 1994</td>
<td>10-1997</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Cs101</td>
<td>Intro</td>
<td>HS201</td>
<td>10:00</td>
<td>Work-study</td>
<td>Kim</td>
<td>22-6-1972</td>
<td>Sep 1994</td>
<td>10-1997</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Ec301</td>
<td>Micro</td>
<td>Eh109</td>
<td>15:00</td>
<td>Professor</td>
<td>Fred</td>
<td>15-3-1955</td>
<td>Jul 1985</td>
<td>Jan 1992</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.3.4-1: Support view.
From the Support view in Figure 6.3.4-1 we can construct the tentative class definition. This is simply a direct translation of the columns (in relational terms) or attributes (in object terms) to the class definition. This is the initial step in translating a relational schema to an object schema (see Figure 6.3.4-2).

**Figure 6.3.4-2: Tentative Support Class.**

We can follow the same process to create additional tentative classes for the object schema definition. Inspecting the view definition, it can be seen that the views for professor and Instructor also select on single type codes. So the columns in the views can be used to create the attributes for the tentative class definitions of Professor and Instructor as well. Figure 6.3.4-3 shows the Professor View is based on selecting a single type code.
### Chapter 6
An In-Depth Example to Illustrate the Advantages Of Conversion to O-O Databases

<table>
<thead>
<tr>
<th>Id#</th>
<th>Modu#</th>
<th>Modnam</th>
<th>Room#</th>
<th>Time</th>
<th>Ptype</th>
<th>Pnam</th>
<th>D.O.B</th>
<th>Hdate</th>
<th>Tdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>MS401</td>
<td>Project</td>
<td>Sc36</td>
<td>11:00</td>
<td>Donna</td>
<td>11-11-1948 Aug 1978 Jan 1985</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Ec301</td>
<td>Micro</td>
<td>Eh109</td>
<td>1500</td>
<td>Fred</td>
<td>15-3-1955 Jul 1985 Jan 1992</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Professor View**

**Select Person Type of Professors**

**Display columns for Professors**

**Figure 6.3.4-3: Professor View.**

Figure 6.3.4-4 provides the tentative Professor class.

<table>
<thead>
<tr>
<th>Id#</th>
<th>Modu#</th>
<th>Modnam</th>
<th>Room#</th>
<th>Time</th>
<th>Ptype</th>
<th>Pnam</th>
<th>D.O.B</th>
<th>Hdate</th>
<th>Tdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cs101</td>
<td>Intro</td>
<td>HS201</td>
<td>9:00</td>
<td>Student</td>
<td>Jack</td>
<td>1-2-1980</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Cs101</td>
<td>Intro</td>
<td>HS201</td>
<td>9:00</td>
<td>Support</td>
<td>Don</td>
<td>12-2-1960 May 1993</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>MS401</td>
<td>Project</td>
<td>Sc36</td>
<td>11:30</td>
<td>Professor</td>
<td>Donna</td>
<td>11-11-1948 Aug 1978 Jan 1985</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>MS401</td>
<td>Project</td>
<td>Sc36</td>
<td>11:00</td>
<td>Work-study</td>
<td>Kim</td>
<td>22-6-1972 Sep 1994 10-1997</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Ec301</td>
<td>Micro</td>
<td>Eh109</td>
<td>15:00</td>
<td>Professor</td>
<td>Fred</td>
<td>15-3-1955 Jul 1985 Jan 1992</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Professor View**

**Figure 6.3.4-4: Tentative Professor class.**
Chapter 6 An In-Depth Example to Illustrate the Advantages Of Conversion to O-O Databases

The same process applies for Instructors and Work-Study by inspecting the code and creates tentative classes for them. At the end of this process we gather all tentative classes as in Figure 6.3.4-5.

<table>
<thead>
<tr>
<th>Support</th>
<th>Professor</th>
<th>Instructor</th>
<th>Work-study</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id#</td>
<td>Id#</td>
<td>Id#</td>
<td>Id#</td>
</tr>
<tr>
<td>PName</td>
<td>Modu#</td>
<td>Modu#</td>
<td>Modu#</td>
</tr>
<tr>
<td>D.O.B</td>
<td>Modnam</td>
<td>Modnam</td>
<td>Modnam</td>
</tr>
<tr>
<td>Hdate</td>
<td>Room#</td>
<td>Room#</td>
<td>Room#</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>Time</td>
<td>Time</td>
</tr>
<tr>
<td></td>
<td>PName</td>
<td>PName</td>
<td>PName</td>
</tr>
<tr>
<td></td>
<td>D.O.B</td>
<td>D.O.B</td>
<td>D.O.B</td>
</tr>
<tr>
<td></td>
<td>Hdate</td>
<td>Hdate</td>
<td>Hdate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Entdate</td>
</tr>
</tbody>
</table>

**Figure 6.3.4-5: All tentative classes to check the repeated attributes**

By looking at the tentative classes we will check for repeated attributes like Id#, Pname, D.O.B and Hdate. All common attributes will be factored up leaving these tentative classes to become a leaf classes in the class hierarchy. Additionally we check for additional attributes like Tenure date for Professor and Entry date for Work-study.

6.3.5. Factoring up common attributes

Now we should expand the class hierarchy to intermediate or non-leaf classes. This is achieved by looking at the repeated attributes in the tentative classes or by looking at the multi-type views as defined in the relational schema. We can have a multi-type view such as the Teacher View in Figure 6.3.5-1. The Teacher View is multi-type because there are several types of teachers. The Employee View is multi-type because there are several types of employees in this example all of them are employees. These can be used as hints for creating intermediate classes above the leaf classes shown in Figure 6.3.5-1.

<table>
<thead>
<tr>
<th>ID#</th>
<th>Mod#</th>
<th>Modnam</th>
<th>Room#</th>
<th>Time</th>
<th>Pname</th>
<th>D.O.B</th>
<th>Hdate</th>
<th>Tdate</th>
<th>Entdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cs101</td>
<td>Intro</td>
<td>HS201</td>
<td>9:00</td>
<td>Jack</td>
<td>1-2-1980</td>
<td></td>
<td></td>
<td>10-1999</td>
</tr>
<tr>
<td>11</td>
<td>Cs101</td>
<td>Intro</td>
<td>HS201</td>
<td>9:00</td>
<td>Pat</td>
<td>1-9-1965</td>
<td>Jan 1990</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>MS401</td>
<td>Project</td>
<td>Sc36</td>
<td>11:00</td>
<td>Donna</td>
<td>11-11-1948</td>
<td>Aug 1978</td>
<td>Jan 1985</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>MS401</td>
<td>Project</td>
<td>Sc36</td>
<td>11:00</td>
<td>Kim</td>
<td>22-6-1972</td>
<td>Sep 1994</td>
<td></td>
<td>10-1997</td>
</tr>
<tr>
<td>2</td>
<td>Cs101</td>
<td>Intro</td>
<td>HS201</td>
<td>10:00</td>
<td>Kim</td>
<td>22-6-1972</td>
<td>Sep 1994</td>
<td></td>
<td>10-1997</td>
</tr>
<tr>
<td>22</td>
<td>Ec301</td>
<td>Micro</td>
<td>Eh109</td>
<td>15:00</td>
<td>Fred</td>
<td>15-3-1955</td>
<td>Jul 1985</td>
<td>Jan 1992</td>
<td></td>
</tr>
</tbody>
</table>
An In-Depth Example to Illustrate the Advantages Of Conversion to O-O Databases

<table>
<thead>
<tr>
<th>ID#</th>
<th>Modu#</th>
<th>Modnam</th>
<th>Room#</th>
<th>Time</th>
<th>Type</th>
<th>Pnam</th>
<th>D.O.B</th>
<th>Idate</th>
<th>Tdate</th>
<th>Endate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CS101</td>
<td>Intro</td>
<td>HS201</td>
<td>9:00</td>
<td>Student</td>
<td>Jack</td>
<td>1-2-1980</td>
<td></td>
<td></td>
<td>10-1999</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Support</td>
<td>Don</td>
<td>12-2-1960</td>
<td>May 1993</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>CS101</td>
<td>Intro</td>
<td>HS201</td>
<td>9:00</td>
<td>Instructor</td>
<td>Pat</td>
<td>1-9-1969</td>
<td>Jun 1999</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>MS401</td>
<td>Proj</td>
<td>Sc36</td>
<td>11:00</td>
<td>Professor</td>
<td>Donna</td>
<td>11-11-948</td>
<td>Aug 1978</td>
<td>Jan 1988</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>MS401</td>
<td>Proj</td>
<td>Sc36</td>
<td>11:00</td>
<td>Work-study</td>
<td>Kim</td>
<td>22-6-1972</td>
<td>Sep 1994</td>
<td>10-1997</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>CS101</td>
<td>Intro</td>
<td>HS201</td>
<td>10:00</td>
<td>Work-study</td>
<td>Kim</td>
<td>22-6-1972</td>
<td>Sep 1994</td>
<td>10-1997</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>EC301</td>
<td>Micro</td>
<td>EN109</td>
<td>15:00</td>
<td>Professor</td>
<td>Fred</td>
<td>15-3-1955</td>
<td>Jul 1985</td>
<td>Jan 1992</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.3.5-1: Multi-type view (Teacher view)

To create the intermediate classes above the leaf classes, the multi-type view will be raised above the tentative classes (Teacher above (Professor, Instructor, and Work-study)) as shown in Figure 6.3.5-2. Tenure date for professors and Entry date of students are special attributes as seen in the Figure 6.3.5-1 otherwise the relational view shows all the common attributes that are ready to factored up.

![Diagram of class hierarchy]

Figure 6.3.5-2: Creating class hierarchy by factoring attributes
Then we create other intermediate classes above the intermediate classes until we reach the super class or the root class.

![Diagram of Employee super class]

**Figure 6.3.5-3 Employee super class**

This process is achieved by factoring up the repeated attributes in the lower classes. In this example, Id#, Pname, D.O.B, and Hdate could be factored up again to create the Employee class, which contains Teacher and Support as sub-classes as shown in Figure 6.3.5-3.

Consequently, Student class will be added to the hierarchy by factoring up the common attributes it shares with both classes Employee class and Teacher class. On one hand, it shares Id#, Pname, and D.O.B attributes with Employee class forming another super class we will call it Person. On the other hand, it shares Modu#, Modnam, Room#, and Time with Teacher class. In addition, it shares the attribute Entdate with the Work-Study class. Work-Study class inherits information about Hdate attribute from Employee class and also inherits information pertaining to students from Student class. This multiple inheritance resolves the problem of ambiguity found in highlighted rows of Table 3. A simple traversal using the object identity would distinguish whether the student object
Chapter 6  An In-Depth Example to Illustrate the Advantages Of Conversion to O-O Databases

belongs to the Teacher class or to the Student class see Figure 6.3.5-4. This is a further illustration of the advantage of converting to an object-oriented model form a relational database.

![Diagram of Person, Employee, Student, Support, Teacher, Professor, Instructor, Work-Study classes]

Figure 6.3.5-4 Adding Student class

Finally, the common attributes of Student class and Teacher class belong to another class (Module class). Module class holds a relationship between Student class and Teacher class that can be called teaching module/studying module relationship. These are actually two relationships. One is a teaching/module many-to-many relationship between Teacher class and Module class; the other is studying/module many-to-many relationship between Student class and Module class as shown in Figure 6.3.5-5.
6.3.6. Summary

It is possible to construct an object-schema from either normalised or non-normalised relational schema. The information about the relational type code forms the basis of the object class hierarchy. The process of constructing the object-oriented class hierarchy using the information about the relational type code is like informally using the normalisation principles. Attributes that are partially dependent along with type code are factored out to the super-class in the class hierarchy using the type code class as a route. By using the conversion process from a relational to an object-oriented database it can be seen how
normalisation is grouping out different attributes to where they semantically belong.

Previous research (Fahrner, 1995) has indicated that in order to covert a relational database to an object-oriented database it is first necessary to ensure the relational database is normalised at least up to third normal form. In this chapter it has been shown that for relational databases containing type code complexity this normalisation requirement is unnecessary, as the normalisation will take place informally during the process of conversion.

With the growing development of object-oriented database management systems more and more data owners are looking to convert their legacy data. However, much of this legacy data is not fully normalised, for reasons of performance, convenience of access or even from poor modelling in the first instance. In this chapter we have shown that this lack of normalisation need not prevent the use of the step-by-step conversion methodologies proposed by earlier researchers.

This chapter has provided an in-depth example to illustrate the points made. A relational database with type code complexity and many-to-many relations is converted to an object-oriented database taking advantage of the multiple inheritance and collection attributes available. The example clearly shows the advantage of using the object-oriented database for problems with these complexities, and in addition, it shows how these advantages of conversion can be obtained even when the original database is not fully normalised.
Chapter 7: Choosing and using a benchmark test to compare normalised, de-normalised and object-oriented databases

7. Introduction

In the context of this research a benchmark can be defined as a set of guidelines and limitations that a customer or a vendor may specify for a particular type of database system to be implemented to their particular field of business/industry. According to the article written be Jim Gray of Digital Equipment Corp (Gray 1993), he states that each benchmark tries to answer the question: “What computer should I buy?” Clearly, the answer to the question is “The system that does the job with the lowest cost-of-ownership”. The cost-of-ownership includes, programming costs, operations costs, hardware costs, and software costs.

To determine these costs can be difficult. It is necessary to examine as many working examples as possible to try and identify the costs that can be incurred. Examples of this are described in chapter 10. However, what can be done if no examples are available? One means of calculating operational costs is to examine the performance of the systems to be evaluated. The length of time taken for an SQL query, for example, will directly affect the efficiency of the person performing the query. The cumulative time lost by the person waiting for a reply to each query could quickly mount up with a very slow database.

Suitable benchmarking of different databases will help determine the operator time used for each query and this will enable the operational costs to be calculated. This chapter looks at the standard benchmark tests available for testing database performance in order to select suitable tests for this research and to enable the second objective of this research to be carried out.

7.1. Wisconsin Benchmark

In 1981 the Computer Science Department at University of Wisconsin developed a benchmark –The Wisconsin Benchmark- that could evaluate their
implemented system, DIRECT, both relative to itself and to the university version of Ingres. At that time no standard database benchmark existed. There were a few application-specific benchmarks, which were very hard to understand simply because they were application-specific (DeWitt, 2001).

The intension of the benchmark, on one hand, was to test the performance of the major components of a relational database system using a set of queries. On the other hand the intention was to understand the semantics and statistics of the underlying relations so that it is easy to add new queries.

The Wisconsin benchmark became very successful due to its technical quality that gave the first evaluation containing impartial measures of real products. The benchmark identified products by name and that triggered a series of ‘benchmark wars’ between commercial database products.

The developing of the Wisconsin benchmark had a major effect of encouraging the development of the Datamation benchmark by a large group lead by Jim Gray (Gary, 1993). By 1990 the DebitCredit transaction of the Datamation benchmark replaced the Wisconsin benchmark as the standard for relation products. The Wisconsin benchmark had one critical failing, being a single user benchmark. A single user does not need to test concurrency control and recovery subsystems of a database system. Another version of the Wisconsin benchmark was developed for multi-user systems but could not compete against the DebitCredit benchmark.

7.1.1. Evaluation

The Wisconsin benchmark did a good job of discovering performance anomalies in the early relational DBMS products. While it no longer receives much public attention, a number of vendors and users still run it as part of their standard quality assurance and performance test suites. With scaled-up relations, it remains a fairly thorough single-user evaluation of the basic operations that a relational system must provide (DeWitt DJ, 2000).
Chapter 7  

Overview of Existing Benchmarks

The Wisconsin benchmark was criticised for a number of deficiencies (Bitton, 88), (O'Neil, 91). These criticisms are certainly valid particularly, its single-user nature, the absence of bulk updates, database load and unload tests, lack of outer join tests, and the relative simplicity of the various complex join queries.

Other than its single user nature, the other most significant problem with the benchmark today is that the join queries it contains are too simple. The benchmark should be augmented to include much more complex join queries with a wider range of join selectivity factors. In addition, a cost component should have been included so that one could meaningfully compare the response times of systems with different costs. For these reasons it was decided not to use this benchmark in this research.

7.2. Transaction Processing Performance Council (TPC)

In early 1980s there was an interest in characterising the performance of On Line Transaction Processing (OLTP) systems. This interest was driven by the increased demand for very high performance transaction systems. This demand was rising from the increasing automation of common, daily business transactions such as the proliferation of automatic teller machines (ATMs) in the banking industry (Serlin, 1990). IBM was able to publish the results of an OLTP test, known as the "1K" test – systems that are capable of sustaining 1000 transactions per second (tps), which, as its name implied, was meant to prove specifically that an IBM mainframe could indeed perform 1,000 tps.

By the end of 1984, three standard performance tests were adopted for OLTP systems: one on-line transaction processing test, dubbed "DebitCredit"; and two batch tests, a sort and a scan. The DebitCredit test was developed to become a more standard benchmark by the Transaction Processing Performance Council (TPC). TPC-A was published in late 1989.
Chapter 7 Overview of Existing Benchmarks

TPC-B uses the same transaction profile, ACID\(^9\) requirements, and costing formula as TPC-A; but it permits the use of batch transaction generator processes, instead of terminal emulation, for creating incoming transactions. The term "residence time" takes the place of the term "response time" since the concept of "user" is omitted using the batch configuration.

The TPC-A Benchmark continued to be the leading tool for comparing OLTP systems until its retirement in 1995. Since then, TPC-C has replaced it in that role and gained even greater recognition. TPC-C added several major characteristics beyond TPC-A to be able to flow with the enormous improvements in hardware and software after 1993. One of the significant extensions that TPC-C added to the basic TPC-A OLTP benchmark model is having multiple transactions of different types.

The most recognised evolution of TPC-C appears in developing solutions to replace user terminals with web browsers as the Internet was growing in popularity. It is clear that the TPC-C requirements allow such radical shift in the end-user connectivity paradigm. The great majority of published TPC-C results have used a web based user interface.

This benchmark is a possible candidate for use in this research. However, it is designed for banking and commercial sector whereas the areas quoted a being most likely to benefit from conversion to object-oriented databases or where complex data exists such as in computer aided design and manufacture. For this reason the TPC benchmark were not used in this research.

7.3. \(\text{AS}^3\text{AP}\)

An ANSI SQL Standard Scalable and Portable (\(\text{AS}^3\text{AP}\)) Benchmark for Relational Database Systems is a paper written at the University of Illinois at Chicago (Turbyfill, 1987). The only measurement required by the \(\text{AS}^3\text{AP}\) benchmark is query elapsed time, with a 12-hour limit. The tradeoff in the design

---

\(^9\)ACID database properties: A stands for Atomicity, C for Consistency, I for Integrity and D for Durability.
of AS\textsuperscript{3}AP is tractability versus completeness. The benchmark is not a test of SQL completeness and is not intended to model a specific workload or query mix. The tractability is achieved by setting the time limit for the benchmark at 12 hours to accomplish the most fundamental and important queries rather than specialised queries.

Systems tested with the AS\textsuperscript{3}AP benchmark are assumed to support common data types, and to provide a complete relational interface with basic integrity, consistency, and recovery mechanisms. Other than that, systems may range from single-user microprocessors to a high-performance parallel or distributed DBMS. It is a strict requirement that all the basic tests (single or multi-user) run in a total time of less than 12 hours. The benchmark can be run as one large program, or as a set of independent modules if the host system is not available in standalone mode for a 12 hours interval of testing.

The benchmark database contains five relations. One, the tiny relation, is a one tuple, one column relation, used only to measure overhead. The database generator, with the appropriate scaling, generates the rest of relations. These relations are named as follows:

- **Unique**: a relation where all attributes have unique values.
- **Hundred**: a relation where most of the attributes have exactly 100 unique values, and are correlated. This relation provides absolute selectivity of 100, and projections producing exactly 100 multi-attribute tuples.
- **Tenpct**: a relation where most of the attributes have 10\% unique values. This relation provides relative selectivities of 10\%.
- **Updates**: a relation customized for update. Different distributions are used and three types of indices are built on this relation.

The values of every attribute in the test database are randomly generated with a uniform and non-uniform distribution. In addition there is an assumption of three types of indices supported by the DBMS under the test which are B-tree
clustered index, a B-tree secondary non-clustered index and a hashed secondary non-clustered index.

Single-user tests are logically divided into operational issues and user queries. Operational issues include, backup, building indices, and a check of referential integrity. Queries include retrievals, single-tuple updates, and bulk updates. All are run in standalone mode, and the elapsed time for each test or query is measured.

There are four multi-user tests, each modelling a different workload profile. These tests are for measuring the throughput as a function of the number of concurrent database users.

While this benchmark offers a good test for database it does not offer test for inserts and updates. For this reason this benchmark was not used in this research.

7.4. The Neal Nelson Database Benchmark:

Neal Nelson & Associates is an independent benchmarking firm based in Chicago. They create benchmarks as well as offering consulting services on computer performance topics. The firm has developed a database benchmark with a unique operation and methodology.

Since the benchmark is coded in industry standard SQL, it can run with almost any database product that support SQL on many platforms ranging from PC LANs to classical mainframes, including the wide variety of Unix-based machines.

The benchmark is run by using Remote Terminal Emulation benchmarking techniques. With this methodology two computers are connected “back to back” so that characters transmitted from ports on one machine (RTE) are sent into corresponding ports of a second machine (the System Under Test or SUT).
Command files called "scripts" are executed on the RTE that transmits keystroke sequences that emulate users performing various database activities. The RTE test methodology allows the test suite to be run with one user, two users, and so forth until a database/computer combination has been adequately stressed.

This benchmark was a possible candidate for this research. However, the need to use two or more computers made it less convenient to use. For this reason, therefore, this benchmark was not used in this research.

7.5. The Set Query Benchmark

While DebitCredit and TPC (see 7.2) measure a single-record update of OLTP, they are not appropriate benchmarks for some of the new systems such as Marketing Information Systems, Decision Support Systems, Management Reporting or Direct Marketing. These application use "set queries", queries that need to refer to data from a potentially large set of table rows for an answer. The set query benchmark actually reflects the activity in these commercial systems.

7.5.1. Document Search

In this application, the user begins by specifying one or more qualities desired in a set of retrieved rows. In return the application returns the count of rows selected to the user. The user can add new qualities to the previous specified ones and again gets a count of the rows retrieved from the application. The aim is to get down to a small number of documents, which deserve closer scrutiny, at which point more detail from the record is printed out.

As this benchmark was also only designed for queries and not for inserts and updates, this benchmark was not used in this research.

7.6. The Engineering Database Benchmark

7.6.1. Introduction

In order to accept an object-oriented database system in the database marketplace it should prove its competitiveness compared to existing relational
database systems. The challenge would be clear by measuring the performance of engineering applications such as Computer-Aided Software Engineering (CASE), Computer-Aided Design (CAD) and Computer-Aided Manufacturing. Because traditional database system benchmarks (Bitton, DeWitt, & Turbyfill (Bitt, 84), Anon et. al. (ANON, 85), TPC (TPC89)) do not measure the performance of features essential to engineering applications. The Engineering Database Benchmark (EDB) was designed for this purpose and has run on a dozen database products and prototypes (Cattel, 91).

The EDB differs in a number of ways from the TPC-A and Wisconsin benchmarks. TPC-A is designed to measure transaction throughput with large numbers of users, while EDB focuses on an engineer with negligible contention with other users (Cattel, 1991). Some of the Wisconsin measures are relevant to engineering performance but not accurate, as they tend to focus on the intelligence of the query optimiser on complex queries. Such set-oriented operations are rare in engineering applications.

The generic benchmark measures in EDB are operations expected to be most frequent in engineering applications, based on interviews with CASE and CAD engineers and feedback on the earlier Sun benchmark (Rubenstein et.al.). EDB substantially improves upon the earlier work, simplifying and focusing the measurement on these engineering database operations.

7.6.2. Engineering Database Performance

The EDB benchmark measures performance of database systems such as object-oriented, relational, network, or hierarchical database systems, and even B-tree packages or custom application-specific database systems. It is designed to be scaleable, and representative of small database working sets that can be cached in main memory and large ones that require efficient access methods. It is portable since it is defined in term of ANSI C and ANSI SQL.

There is a big gap between the performances provided by in-memory programming language data structures and that provided by disk-based structures.
in a conventional database management system. Disk-based structure systems respond to random read queries in tenth of a second. In contrast, simple lookups using in-memory data structures can be performed in microseconds, which is the factor of 100,000:1 in response time.

Sometimes modelling factors can make a big difference in performance. By exploiting some of the characteristics of engineering applications the performance improvement can be quite noticeable in comparison to business applications. For example, and engineer may “check out” part of a design and work on it for hours, with no requirements for concurrent updates by other engineers, and the data can largely be cached in main memory on the engineer’s workstation.

As Cattell states in his paper, “differences in data models (Cattell, 1991) (e.g., relational and object-oriented) can be dwarfed by architectural differences”. Large improvements in engineering database performance are probably not going to be accomplished through minor improvements in the data model, physical representation, or query languages. The substantial improvements come from major changes in DBMS architecture: caching a large working set of data in main memory, and minimising overhead in transferring data between programming and query languages.

7.6.3. The Benchmark Database

The EDB benchmark database is independent of the data model provided by the DBMS. Its data structure depends fully on the type of application to be studied and measured. The database stores the information of the abstract definition of the engineering application. For example, if the engineering application is pertinent to CAD then the data definition would possibly be as two or more tables of some record types in a relational system as in the following example:

```sql
create table part (id integer not null primary key,
```
A database $N$ of parts will have dense unique part numbers (part.id) in the range $[1..N]$, such a database will have $3*N$ connections, with exactly three connections from each part to other parts. The X, Y and length field values are randomly distributed in the range $[0..99999]$, the type fields have values randomly selected from the strings {"part-type0"...part-type9}, and the build date is randomly distributed in a 10-year range.

### 7.6.4. Benchmark Measures

The benchmark measures response time and is run by a single user. This is consistent with the model of an engineer "checking out" a segment of data for exclusive use. On the other hand, it is difficult to achieve better performance with highly concurrent access by multiple users. However, it is important for the DBMS to allow multi-user access to the database. EDB requires that the data used be locked or versioned in order to support multiple users.

The following three operations are the EDB benchmark measures. Each measure is run ten times, measuring response time for each run to check consistency and caching behaviour.

1. **Lookup**: Generate 1000 random part ids and fetch the corresponding parts from the database. For each part, call a null procedure written in any host language, passing the x, y position and type of the part.
Chapter 7 Overview of Existing Benchmarks

2. Traversal: Find all parts connected to a randomly selected part, or to a part connected to it, and so on. For each part, call a null programming language procedure with the value of the x and y fields, and the part type. Perform the traversal depth-first. Also measure time for reverse traversal, swapping "from" and "to" directions, to compare the results obtained.

3. Insert: Enter 100 parts and 3 connections from each to other randomly selected parts. Time must be included to update indices or other access structures used in the execution of Lookup and Traversal. Call a null programming language procedure to obtain the x, y position for each insert. Commit the changes to the disk.

7.6.5. Running EDB

In order to reproduce comparable results, it is necessary to run the benchmarks and DBMS on a similar configuration. The hardware and software specifications must be set along with DBMS architecture and benchmark implementation. Finally, there should be a general order for executing a specific benchmark such as the following:

Clear the operating system cache
Open the database
Start timer
Execute benchmark measures (Look up, Traverse, Insert, ...etc)
Stop timer
Restore the database to its original state
Close the database

The EDB has been implemented on three products that vary quite dramatically in their overall design, remote data access, concurrency control implementation, and traditional DBMS properties. The three products are as follow:

1. OODBMS: the pre-release ("beta") of a commercial object-oriented DBMS. It supports objects, classes, inheritance, and persistence storage, multi-user sharing of objects, transactions and remote access.
2. **RDBMS:** the UNIX-based production release of a commercial relational DBMS. It provided features such as atomic transaction, full concurrency support, network architecture, and support for journaling and recovery.

3. **INDEX:** is a B-tree package on Sun named "INDEX". The system consisted of parts file, a connection file interconnected to the parts file with record ids, and a B-tree file used for the lookup on parts.

Figure 7.6-1 shows the EDB measurements for the most important scenario, the small remote database (i.e., 20,000 parts accessed over the network) (Cattell, 1991).

<table>
<thead>
<tr>
<th>Measure</th>
<th>Cache</th>
<th>INDEX</th>
<th>OODBMS</th>
<th>RDBMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB size</td>
<td></td>
<td>3.3 MB</td>
<td>3.7 MB</td>
<td>4.5 MB</td>
</tr>
<tr>
<td>Load Time</td>
<td>1100*4</td>
<td>267</td>
<td>370</td>
<td></td>
</tr>
<tr>
<td>Reverse traverse</td>
<td>cold</td>
<td>23</td>
<td>6.3</td>
<td>95</td>
</tr>
<tr>
<td>Lookup</td>
<td>cold</td>
<td>7.6</td>
<td>20</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>warm</td>
<td>2.4</td>
<td>1.0</td>
<td>19</td>
</tr>
<tr>
<td>Traversal</td>
<td>cold</td>
<td>17</td>
<td>17</td>
<td>94</td>
</tr>
<tr>
<td></td>
<td>warm</td>
<td>8.4</td>
<td>1.2</td>
<td>84</td>
</tr>
<tr>
<td>Insert</td>
<td>cold</td>
<td>8.2</td>
<td>3.6</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>warm</td>
<td>7.5</td>
<td>2.9</td>
<td>20</td>
</tr>
<tr>
<td>Total</td>
<td>cold</td>
<td>33</td>
<td>41</td>
<td>143</td>
</tr>
<tr>
<td></td>
<td>warm</td>
<td>19</td>
<td>5</td>
<td>123</td>
</tr>
</tbody>
</table>

**Figure 7.6-1 Small Remote Database (benchmark elapsed times in seconds)**

This is the general benchmark with all measurements. Other tests could take place by omitting one or more of the specified testing measurements. The
benchmark measures were performed at least ten times, the first time with no data cached. The repetitions provided a consistency check on the variation on the results and also show the performance (on the second, third and remaining iterations) as successively more data is in memory. There was some cache hits even on the first iteration, depending on database system parameters such as page size, clustering, and caching strategy. As a result, the average cache hit rate for the first iteration can be as low as 10% or as large as 90%. The results of the first iteration are called the cold start results, and the asymptotic best times (the tenth iteration, when the cache is fully initialised) are called the warm start results.

7.7. Normalised, De-normalised

7.7.1. De-Normalisation Measurements

Ideally the benchmark of the two DBMS –normalized, de-normalised, could follow the same procedure of the EDB benchmark:

Clear the operating system cache
Open the database
Start timer
Execute benchmark measures (Look up, Traverse, Insert, ...etc)
Stop timer
Restore the database to its original state
Close the database

The EDB benchmark fits more than other benchmarks mentioned earlier. Because, on one hand, the EDB compares two different models (RDBMS and OO ODBMS) and on the other hand compares the two with another highly optimised access method that is a B-tree package on Sun.

The EDB benchmark was also designed for engineering problems such as computer-aided design. As it is claimed that these problems are particularly suited for conversion to object-oriented databases, this benchmark offers a rigorous test for comparing de-normalisation with conversion to object-oriented databases.

The above procedure measures the performance by calculating the elapsed time for each query. But we need to include the time spent on extra code for de-
normalisation to ensure integrity (Hours per programmer). Also we should consider how frequently the query is used everyday. The frequency of the query used per day shows the need for the improvement in performance. Some queries are rarely used and may have acceptable performance.

The objective of calculating the time spent on extra coding is to look ahead and measure when the performance gain of de-normalised code will give a return on the investment in its DBMS. Thus, it is very important to assess how much the de-normalisation will reduce the elapsed time for the query or, in other words, improve the performance.

By multiplying that number of times the query is used everyday by the elapsed time this will give us an indication how much time it takes to run a particular query a day. Moreover, the programmers will shoot for a better target on how much they need to improve the performance and is it feasible to de-normalise it or just leave it the way it is. The calculations are as follow:

For any x query

1. Total time used by x in normal form = elapsed time by x * frequency of x in a day
2. Total time used by x in de-normal form = elapsed time by x * frequency of x “
3. Number of days to break even = (time spent for extra coding to ensure integrity)/ (step 2 - step 1)

The break-even point could be figured out by dividing the time spent for extra coding to ensure integrity by the difference between the total times spent by the query in normalised and de-normalised forms. This formula will enable the decision makers to select between de-normalising and sustaining the old system.

Thus: the difference in total time elapsed = time of extra coding / number of days to break even By using this formula we can determine when we will break even and the performance required to give an indication to weather we
should go ahead with the de-normalisation or just live with the current slow performance of the normalised database or get a bigger and faster machine.

7.7.2. Object-Orientation Measurements

The EDB benchmark was made to measure performance of engineering applications and show that object-oriented database suites such applications better. However, many existing relational databases that have complex data are not designed and implemented the same as such engineering applications. The EDB benchmark used a relational database to run an engineering problem to show the shortcoming of the relational model. In this research we are testing an object-oriented database plus an existing relational database with complex data modeling to see whether the object-oriented model is superior to the relational model or not.

To be fair in our comparison tests, we need to add the time to be spent on converting the relational database model to object-oriented model or the time spent to design and implement the object-oriented model from scratch. This time resembles the time spent on extra code to ensure integrity for de-normalisation. Knowing the amount of time spent will help the benchmark and will give a better assessment to establish the conversion or keep the legacy system as it is.

To compare the object-oriented database management system with the de-normalised and normalised database management systems we must look at joined queries. Some queries need to join many tables to produce certain results. These queries need to be tested because they take a longer time. We also need to look at queries that are used more frequently. This comparison will show the strength and weaknesses of each database model.

The cost of training the staff on the new concepts of object-orientation is a compound expense of the time spent to learn and the actual training costs. Object-oriented databases such as POET, Objectstore and Gemstone usually use one of the object-oriented programming languages such as Java, C++, or SmallTalk along with a tool to save classes and objects (persistence objects). This involves the cost of an extra training and a new license. All these costs add to the expenses
incurred by the company when converting from a relational to an object-oriented database so care must be taken to ensure the overall investment is beneficial.

The benefits obtained from improving performance using an object-oriented database must overcome the summation of all other costs involved in building the object-oriented system. If it is the case that the object-oriented database will not improve the overall performance when compared with the legacy system in use, then the assessment should result in recommendation to keep the current RDBMS and look for other alternative improvements. These are important factors that previous benchmarks such as EDB did not include in their measurements when they compared the relational model with object-oriented model.

There are other calculations that affect the cost in the long run such as maintenance and extendibility costs. Another important cost is reusability, which is a promising feature in object-oriented model. These useful object-oriented features can also affect the comparison tests.

This research uses the same concepts used in previous benchmarks but it adds a few important measurements that will give more accurate results. Some of these factors are difficult to measure. Other factors were not included because of the big differences found between the two paradigms. At the end we must include every possible variable that can affect the decision of whether to migrate to the object-oriented model.

A benchmark test to measure the performance of the two DBMSs would clarify the functional preferences and most suitable cost that would be appropriate to the budget of a company. The following factors must be taken in account to show the costs and possible investments or losses when using the object-oriented model: Time spent for training, Cost of training, Cost of converting or building from scratch, Maintenance, Reusability of code, Cost of the License. These additional costs are examined in chapters 9 and 10.
7.8. Summary

This chapter has shown a brief evolution of database benchmarks. The survey shows that none of the previous benchmarks has measured the performance of normalised database versus de-normalised or non-normalised database. However, the EDB benchmark uses an engineering application to compare the performance of relational database versus an object-oriented database rather than using a business application that has a complex data.

As many of the quoted examples of complexities that are difficult to handle with relational databases are from engineering type problems (e.g. CAD, CAM) this makes the EDB benchmark particularly suitable for use in this research.

The author will use the same technique used in EDB benchmark to compare normalised database against de-normalised database. The benchmark will be used later in chapter 10 and chapter 11 to calculate the cost of existing maintenance cost and the cost of improvement method used to upgrade the database system.

By comparing the time difference between normalised and de-normalised databases we can determine the number of days to break even. This is done by dividing the difference in total time elapsed over the time spent for extra coding. By determining the time period up to breaking even, decision makers will have at their disposal a better analysis of what improvement should be taken place.
Chapter 8: Benchmarking of De-Normalised Databases

8. Introduction

Non-normalised databases are the ones that do not comply with the rules of normalization. However, there is a difference between a de-normalised and non-normalised database. The database could be non-normalised for many different reasons such as a lack of awareness of normalisation rules or bad design of the database system to start with. However, database development experts may deliberately de-normalise databases or make them non-normalised to improve performance.

The well-studied entity-relationship data models produce tables that contain no redundant or derived data and tables that are well structured by the way of relational theory. To meet extraordinary demands for high performance, developers might sometimes have to modify the data model in ways that are undesirable from a theoretical standpoint. This chapter describes some of these modifications and their associated costs.

Normalised tables don't contain redundant or derived data. Every attribute appears in only one table and data that can be computed from existing attributes is selected as an expression based on those attributes. Thus, normalised tables minimise the use of disk space and makes updating the tables as easy as possible. However, normalised tables can force the use of joins and aggregate functions, which can be time consuming when retrieving data.

A correct relational data model avoids redundancy by keeping any attribute only in the table for the entity that it describes. To present the attribute data in a different context we need to make the connection by joining tables. To avoid consuming extra time by joining tables we can duplicate the joined data for future use in another table. There is an integrity risk in duplicating data when we want to update the data and also there is extra space used on the disk but it pays off with better performance.
This chapter quantifies the performance gain that can be achieved with a non-normalised database by using benchmark tests, thus achieving the second of the objectives set out for this research.

8.1. Methods of De-Normalisation

There are two possible ways to de-normalise a database, firstly at the design stage, and secondly dynamically when the system is in use.

8.1.1. Design De-Normalisation

Design de-normalisation is where the database for a given system is not fully normalised at the design stage. If this is performed properly, the design must first be normalised to the maximum extent possible. From this point, with knowledge about how the system is going to work and how it will be used, particular parts of the data structure can be de-normalised.

The system designers have to successfully decide how the system will be used to know where to de-normalise. However, most failures in software projects are due to incorrect system specification, so therefore it is quite possible for the structure to be incorrect. As such, if the system works, the structure could cause a performance or data integrity shortfall where the opposite was in fact intended.

Furthermore, the database structure should ideally be as correct as possible so that any improper design decisions would not affect any future systems that may replace the currently designed one. Such structures often impose some form of legacy onto the software designers of future systems. Indeed, old systems often impose a legacy on new systems that could actually be designed better if they were constructed from scratch.

In some cases it is hard to distinguish between design de-normalisation and poor design. Poor design is usually when the data model is never fully normalised, because the data relations / dependencies are not identified. As stated earlier, however, design de-normalisation requires the data model to be fully normalised before de-normalisation takes place. The same result is required from
either process (de-normalisation or poor design), but poor design means that the dangers of loss of integrity may not be realised and the extra code required to protect the system from these dangers would not then be written.

8.1.2. Adaptive Performance De-Normalisation

The drive to include an abstraction layer to a DBMS is such that the database engine can be optimised to execute fewer JOIN operations. An abstraction layer is a layer within the code of the DBMS that provides the programmer with exactly the same interface and data structure as before. In fact, as far as a programmer would know, nothing would be changed at all. However, the way in which the DBMS accesses the data could be modified without the knowledge of the programmer.

JOIN operations are expensive in terms of processor time, but read or update operations can be executed directly on a TABLE with very little processor time.

Therefore, when database tables are created, the abstraction layer could be used to optimise the table design for performance. In this process, it is necessary to assess how often each table will require updates. This would allow those lookup tables that might not need to be changed often, to be de-normalised. A virtual copy of the original table would then exist.

The system designer would see no change in the execution of their design; the abstraction layer hides this. Therefore, they may well use an SQL statement to JOIN the two tables, but the abstraction layer would interpret this as a straight “SELECT x,y,z FROM A;” vastly reducing the result time.

When insertion or update occurs, the process of input validation is required. The system designer would still attempt to JOIN the two (or more) tables, but here comes the abstraction layer to perform a “SELECT x,y,z FROM B;” to validate the input before performing “UPDATE A SET x,y,z WHERE ? ;” which would reduce the time taken slightly when using large tables (for very
small tables such as less than 100 records this approach would be undoubtedly slower).

This idea is intended for use in large databases, where downtime is not allowed and integrity is required along with performance. It relies on the fact that many queries are executed over one 'master' table and one or two lookup tables. It is intended for use where there are possibly hundreds of tables within the system, effectively reducing them so that there is far less need to perform table JOIN operations. The downside of this approach is that of increased storage space. Data normalisation as devised by Boyce and Codd is such that it would be extremely hard to find an approach that is more frugal with storage space. This approach allowed them to maintain data integrity, because at the time, storage space was very expensive by today's standards, so data were optimised for the amount of storage space required. In current times, the problem is that of performance as in most cases we no longer have to worry so much about the cost of storage space (see Figure 8.1-1). Therefore de-normalising a database can be seen as a way forward in performance terms as long as there is a protection scheme that can maintain the data integrity.

Figure 8.1-1 Predicted Price Of Hard Disk Drive (HDD)
8.1.3. The Correctness of using the Relational Model

Problems do exist with the relational database model. Performance aspects have been under consideration for a long time. The creation of join operations is an area where performance gains could be made.

It is well known, however, that the relational data model is correct with regard to the storage of data in an environment least likely to violate data integrity, since each item of data exists only once, with no repetition if the data modelling is performed correctly.

The role of the DBMS is much the same as that of memory or of a file scheme within an operating system. That is to say that it must provide sharing and protection to the data and user sessions, at the same time as providing a response to the user within a reasonable amount of time.

Protection of the database takes the form of preventing any activity that may violate either the referential integrity rules, or the data integrity / validation rules. These must be strictly adhered to or else the loss of integrity will make the data useless within a short period of time.

Sharing comes in the form of allowing multiple, concurrent accesses to the same database from a multitude of clients. This may be of the form of a server machine on the Internet for example.

Any query requested by the user should expect a response within a reasonable amount of time. As such, time constraints are not critical, but are still important, as the user does not want to be left without a response of any kind. Any response is either to give the user the data requested, or to process data that has been sent. As such a response is required so that the user is provided with 'closure' when operating the system, therefore it is not a real time system.

8.2. Experimental Criteria Required for the Benchmark Tests

• The server and clients of any database should remain the same throughout testing the exact models. The specifications of these machines will have to
be included in the final project report. The point is to prevent any
differences in hardware that could alter the timings.

- The testing should use the same data regardless of the level of
  normalization in the database. This restriction will test the data structure
  only.

- The queries used will have to be as comparable as possible i.e. the output
data sets will have to be identical and the queries will have to be
  semantically identical. This will ensure the identical functional mapping of
  the query regardless of the data structure.

- The operating environments of the machines must be the same when either
  set of tests is performed for the normalised and de-normalised databases.
  The build of the operating system on the server and client machine must
  be such that the tests can be reproduced by another person and comparable
  results obtained.

- The build of the operating system and software on the machine must be
  fully documented so that the same environment can be reproduced exactly
  anywhere.

- Between tests it may be a wise idea to restart the DBMS service on the
  server machine so that no caching of data can interfere with the speed of
  data access. Some DBMS engines can cache results to eliminate it the
  service should be stopped and then restarted.

- The measurement will be the process time of the server in running the
  SQL query. This is to prevent distortion from any other performance
  features such as client speed or network bandwidth.

8.3. Tests

Tests have been numbered in accordance to the type of operation and join
used; any selection is regarded as a minor alteration. Each test is to have the SQL
modified to be suitable for the different DBMSs, but shall remain semantically
(and hopefully functionally) identical; therefore testing shall take place as the same test number.

Some data has been obtained from Alstom Combustion Services Ltd., The data came from their database for the wiring of computer and telephone sockets in their office, names have been blanked out and the data multiplied to allow for a suitably sized data set. The database is of a very simple structure. The system was based on a flat-file legacy system and was redesigned during the summer of 2000 by Kris Suthems (Suthems, 2000). 60 different tests cover the operation of this system (although the queries in the test plan are different to those in use in the system).

A test plan for the Alston Combustion Services database was available, however not all queries were used in the benchmark test. This is because the test plan has to include all types of joins that could be produced by the data, not necessarily bothering with the semantics of what the data actually means. The reason for this is that when designing a database system it is often a good idea to also consider that most queries only run on one table. However, in the case of an application type interface, one user operation may in fact run a script running multiple queries in quick succession. The structure of the database is as follows:

![Database Structure Diagram]

The test plan itself contains Generic SQL statements; these may require modification for particular database instances or DBMSs.
8.4. Normalised Test Plan

Figure 8.4-1 gives part of the test plan for the first test run in normalised form. A full test plan is given in Appendix B.

<table>
<thead>
<tr>
<th>Test Number</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SELECT * FROM Floor_Sockets FULL OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch</td>
</tr>
<tr>
<td>13</td>
<td>UPDATE Floor_Sockets SET Owner = 'yyyyyyyyy' WHERE Floor_Sockets.Owner is not null AND Floor_Sockets.Owner IN (SELECT Floor_Sockets.Owner FROM Floor_Sockets FULL OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch)</td>
</tr>
<tr>
<td>25</td>
<td>INSERT INTO Floor_Sockets (Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch) SELECT Floor_Sockets.Tech_Primary, Floor_Sockets.Floor_Socket, Floor_Sockets.Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets FULL OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch</td>
</tr>
<tr>
<td>37</td>
<td>DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN (SELECT Floor_Sockets.Tech_Primary FROM Floor_Sockets, BT_Patch WHERE Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch (+))</td>
</tr>
</tbody>
</table>

Figure 8.4-1 Part of test plan for the first test run in normalised form

8.5. De-normalised test plan

Figure 8.5-1 shows part of the test plan for the experiment running in de-normalised form. The full de-normalised test plan is given in Appendix C.

<table>
<thead>
<tr>
<th>Test Number</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SELECT * FROM AllLand2</td>
</tr>
<tr>
<td>13</td>
<td>UPDATE AllLand2 SET Owner = 'yyyyyyyyy' WHERE AllLand2.Owner is not null AND AllLand2.Owner IN (SELECT AllLand2.Owner FROM AllLand2)</td>
</tr>
<tr>
<td>25</td>
<td>INSERT INTO AllLand2 (Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch, BT_Patch_2, TN_Prefix, TN_Suffix, Telephone, Fax_modem) SELECT AllLand2.Tech_Primary, AllLand2.Floor_Socket, AllLand2.Computer, AllLand2.Owner, AllLand2.BT_Patch, AllLand2.BT_Patch_2, AllLand2.TN_Prefix, AllLand2.TN_Suffix, AllLand2.Telephone, AllLand2.Fax_modem FROM AllLand2</td>
</tr>
<tr>
<td>37</td>
<td>UPDATE AllLand2 SET Tech_Primary = Null, Floor_Socket = Null, Owner = Null, Comments = Null, Computer = Null, BT_Patch = Null, Tech_Primary = Null, Floor_Socket = Null, Owner = Null, Comments = Null, Computer = Null, BT_Patch = Null</td>
</tr>
</tbody>
</table>

Then run:
DELETE FROM AllLand2 WHERE AllLand2.Tech_Primary is null AND AllLand2.BT_Patch_2 is null

Figure 8.5-1 Part of the test plan running in de-normalised form.

8.6. Test Log

Figure 8.6-1 is part of the test log to show the time taken by each test performed as part of this experiment. The full test is given in Appendix C.
Chart 8.6-1 shows the difference of time in seconds between normalised and de-normalised executions for selection queries. The de-normalised execution time is far less (such that it can hardly be seen) than the execution time for the normalised tests. This is where a notable time saving can be seen. Some of the queries show a big difference because of the number of joins made like in test numbers 25 and 28.

Statically, by applying the t test formula for two population means (normalised and de-normalised) using paired samples (Weiss 1993). The results of calculations provide sufficient evidence to conclude that the mean performance of all queries when de-normalisation is applied is greater than the mean performance of all queries when de-normalisation is not applied. In other words, it appears that the de-normalisation is effective in increasing performance in the select queries (see statistical analysis at the end of appendix D).

Thus at 90% confidence interval for (mean1 – mean2) is from 1.1 to 5.8 seconds. We can be 90% confident that the difference between the mean performance of all queries when de-normalisation is used and the mean performance of all queries when de-normalisation is not used is somewhat between 1.1 and 5.8 seconds. In particular, we can be 90% confident that, on the average de-normalisation reduces the retrieval time by at least 1.1 seconds.
Chart 8.6-1: Normalised de-normalised select query comparison.

Chart 8.6-2 shows the difference of time in seconds between normalised and de-normalised executions for update queries. The results for update queries look disappointing. The normalised database has somewhat better performance that the normalised one, except for few points where large amounts of data are concerned. The reason behind this performance similarity between normalised and de-normalised database for using update queries is the time taken of maintaining data integrity that provides a massive performance problem. For small to medium data sets (such as the group selection operations) there is no real benefit to de-normalisation.
Chapter 8 Benchmarking of De-Normalised Databases

Chart 8.6-2: Normalised and de-normalised update query comparison.

Chart 8.6-2 shows the difference of time in seconds between normalised and de-normalised executions for update queries. By inspecting the test numbers (25-37) we can notice that the improvement stem from reading of data (Select operations), for the queries of Insert (Chart 8.6-3) and Delete (Chart 8.6-4) there is not an advantage to de-normalisation. This can be shown statistically by the t test at the end of Appendix D. The overheads of de-normalised tables being updated can be seen especially by tests (25-48.2).

Chart 8.6-3: Normalised and de-normalised insert query comparison.
Chapter 8 Benchmarking of De-Normalised Databases

Chart 8.6-4: Normalised and de-normalised delete query comparison

It was also noticed that during the select operations there was a problem that although the query execution time was vastly reduced, the production of the HTML table took far longer, somewhat neutralising the results as can be seen by Chart 8.6-5.

This comparison actually shows that where a large amount of data is involved, there is still a time saving. However, when displaying these large amounts of data, it always takes a long time to display for the end user because of the amount of memory that is required by the browser. So even if the server takes less time to generate the output, it would still take a very large amount of time to actually display the page.
Chapter 8  Benchmarking of De-Normalised Databases

Chart 8.6-5: Comparison of normalised and de-normalised HTML tables.

The results for queries other than Select look disappointing. The de-normalised database has similar performance to the normalised one, except where large amounts of data are concerned, when the cost of maintaining data integrity provides a massive performance problem.

8.7. Summary

The basic message that this provides is that the de-normalisation of data does improve performance times, but at the cost primarily of storage space. De-normalisation also produces a 'new' overhead, that is the cost of processor time to implement a protection system, and the fact that one normalised query may in fact be multiple queries on a de-normalised system.

The time saving of the execution of de-normalised joins is of a large order of magnitude; therefore the time complexity savings increase very quickly as the size of data increases.

While additional overheads are introduced, the time complexity of these does not seriously affect the time taken when using a de-normalised database.
Chapter 8 Benchmarking of De-Normalised Databases

With large amounts of data, it is true that numerous queries on a single table are far quicker than performing a join on two tables.

When small amounts of data are being used, it is true that de-normalisation can cause more problems than the benefits are worth. This is because the machines that run these programs are so fast that when processing small amounts of data the integrity and protection is easier to handle in normalised form. Even if the structure is not at its most efficient it would not necessarily be noticed, if this were such a serious problem for small amounts of data many people that develop systems within applications such as Microsoft Access would suffer huge performance penalties if the systems worked at all.

Overall de-normalisation is not as good as it is sometimes made out to be. Most queries do not get a benefit out of de-normalisation. The benefits only really become effective where a large number of select queries are performed on large data sets.

The noticeable improvements all stem from the reading of data (SELECT operations), for the rest of the queries there is not an advantage to de-normalisation. The overheads of de-normalised tables being updated can be seen especially by tests of INSERT and DELETE operations (Tests 25-48.2).

This does lead us to believe that further research is required into this area. An actual implementation of a database abstraction layer capable of adaptive de-normalisation would be quite a feat. A system with enough rules built-in to perform the task would take a large amount of design and programming. Not all of the answers of how to do this are contained here, merely the ideas and concepts important to the design of such a system.

This research does show that any improvement to be made would most likely come from the reduction in the time taken to perform table joins. The time saving for this type of data retrieval are used in the calculations of operational cost saving in chapters 9 and 10. The result reported in this chapter helps satisfy both the second and third aims of the research given in chapter 1.
Chapter 9: Cost Comparison Of Conversion To OODB With Other Methods

9. Introduction


The innovation of normalisation concepts in database systems helped to achieve such desirable features as safety, consistency and maintainability (Delobel, 1995). The DBMS guarantees reliable data definitions and manipulations when data are properly modelled. In the case of a relational database, the data must be well normalised to attain reliability. However, the normalised model for complex data usually suffers from many side effects concerning performance and response time. Therefore, working databases are often not fully normalised (Elmasri, 1994).

A problem area for normalised databases concerns many-to-many relationships, where an additional table (intersection entity) is required for each relationship to ensure consistency, and reduce maintenance cost but the number of table joins for data retrieval increases performance cost (Douglas 1996).

This chapter gives a methodology for evaluating whether de-normalisation or conversion to object-oriented modelling is worthwhile to improve system performance or whether simply using a bigger, faster computer would be more cost effective.

9.1. De-normalisation

RDBMS performance improvements may be achieved through “de-normalisation”. In the case of many-to-many relationships this achievement is accomplished by moving away from intersection entities (Douglas 1996). By
Chapter 9 Cost comparison of conversion to OODB with other methods

decreasing the number of tables the number of extra joins will also decrease, as a result the retrieval time will improve. This technique is actually used in INFORMIX™ to improve performance. Unfortunately such improvement is usually accompanied with undesirable costs see Figure 9.1-1.

9.1.1. Side Effect

By de-normalising we are forcing an improper data model in the database system that can cause many side effects:

- The DBMS requires extra code to prevent any anomalies that might occur.
- As a result, the database system may face higher maintenance cost
- The system would be vulnerable to errors.

However, some companies are prepared to accept the extra development and error risk to achieve a performance gain.

Figure 9.1-1 Steps that a RDBMS application developer may take to improve performance by de-normalisation.
For clarity, let us consider the example of a many-to-many relationship in a relational database model. Figure 9.1-2 shows a university example of a many-to-many relationship between different databases, Student, Teaching Module, Departments, and Teachers.

Figure 9.1-2 Many-to-many relationship

Figure 10.1-3 shows one of the many-to-many relationships, the Students table and the Modules table. One of the drawbacks in the relational model is that it does not handle many-to-many relationships easily and it can create the delete and update anomalies. In order to delete a record in a table the redundant information must also be deleted from the second table as well.

For example if we delete the first row (Sam, 10/11/68, CSc 201) from the Students' table then we must also delete the third row (CSc 201, Computing, Sam) from Modules' table. To get around this problem in the relational model we must add an extra table (intersection table).
Chapter 9  Cost comparison of conversion to OODB with other methods

One solution to gain consistency and integration in the database is to normalise the data. The many-to-many relationship must be decomposed into three tables, which are the two original entities and another table that combines the primary key from each table.

<table>
<thead>
<tr>
<th>Student Table</th>
<th>Intersection Table</th>
<th>Module Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>D.O.B</td>
<td>Name</td>
</tr>
<tr>
<td>Sam</td>
<td>10/11/1968</td>
<td>Sam</td>
</tr>
<tr>
<td>Susan</td>
<td>15/10/1970</td>
<td>Sam</td>
</tr>
<tr>
<td>Thomas</td>
<td>07/07/1971</td>
<td>Susan</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Susan</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Thomas</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Thomas</td>
</tr>
</tbody>
</table>

Figure 9.1-4 Intersection Table for a Many-to-Many Relationship

For our example in Figure 9.1-2 we need to add another three intersection tables to the existing four original tables to normalise the database and get rid of any anomalies that might occur to the data in the database (see Figure 9.1-5). To produce a report of (Student name, Date of birth, Module code and number, Department, and Teacher) we must join all seven tables. The join operation can be a performance issue for RDBMSs, and many joins can drastically reduce performance. As a result, developers are willing to risk the consistency of duplicating by de-normalising to improve performance. This will require extra code to try and remove the possibility of inconsistencies in the data but this is time consuming to produce and more difficult to maintain and may in itself be prone to error.

Figure 9.1-5 An example of Many-to-Many Relationship with intersection tables
Chapter 9 Cost comparison of conversion to OODB with other methods

9.2. Conversion to Object-Oriented Model

An alternative method for improving performance of a complex normalised database system is to convert it to an object-oriented system. The advantage of this method is that it will preserve the proper data modelling as in the normalised database, as well as giving the overall performance of a de-normalised database. The Object-Oriented model has the merit of long-term improvement by increasing maintainability. This leads to further advantages of fewer errors, easier extension and reduced maintenance costs. This is the opposite of the effect of de-normalisation.

![Object-Oriented schema for the university example](image)

The object-oriented model handles many-to-many relationships easily as Figure 9.2-1 shows using the university example. Each class in the complex relationship has an attribute of the related class as a reference. For example, the Student class would have Module number as a collection attribute in its class.

There would be no joins and the data is accessed by traversing the relationship (takes/has) using object identities. Because, no joins are needed, the performance is obtained without de-normalisation and no extra code is needed to maintain reliability and consistency of data. Delete/update anomalies are taken care of since there are no redundant data in the database.

Published papers on methodologies for converting relational to object-oriented databases assume the relational database is based on a proper, normalised
Chapter 9 Cost comparison of conversion to OODB with other methods

data model. However, many working databases in the real world are not
normalised for performance reasons or for reasons of convenience or simply
because the developers did not fully understand how to normalise the database in
the first place! However, as shown in Chapter 6, this need not prevent direct
conversion to an object-oriented database (Al-Kandari, 1999), the conversion
process will itself informally normalise the data.

9.3. Better machine

A third, and perhaps even more significant alternative method of improving
performance needs to be considered. Could the same level of improvement be
achieved simply by getting a bigger and faster computer? As one industrial
employee reported, “Once I had to make changes to 3 forms to add various
triggers to get the update/insert things to work properly after the affected tables
were de-normalised. The performance gain was a response below 3 seconds
compared to 15 seconds. It took about 3 days of coding on the forms (working 18-
hours each day!). If the database was on a slightly better machine (more RAM,
disk, etc.) I wonder if there would have been any need to do the de-
normalisation.” Clearly this alternate must be costed and compared with the de-
normalised and object-oriented database methods of improving performance.
Especially, when we look at Figure 8.1-1 given by IBM showing the decrease in
the price of data storage.

Figure 9.3-2 shows the increase in efficiency and retrieval time. IBM drives
have consistently demonstrated progressively lower seek times through the use of
advanced actuator designs, improved, lighter materials and shorter stroke lengths.
Seek times below 4 ms to 3.4 ms are available in IBM Ultrastar 15K RPM drives
with a disk diameter of 70 mm. The sum of seek and lower latency time (due to
faster rotation) has resulted in accessing times less than 6 milliseconds, with
indications that times much less than this are attainable in the near future. Disk
drive performance is increasing concurrently with capacity while price per
megabyte is decreasing correspondingly.
Chapter 9 Cost comparison of conversion to OODB with other methods

Figure 9.3-1 Performance increase until year 2010.

9.4. **A Methodology for evaluating the cost of a system.**

Normally, the better quality the merchandise usually means the greater the cost or value. For database systems the cost savings from improving the performance has to be measured against the cost incurred as a result of the improvement to determine the best quality solution.

We can constitute a new methodology by looking at Figure 9.4-1 for simplicity this methodology has been split into four steps:

**Step 1 - Determine the cost of de-normalisation**

*Consider the costs of the de-normalisation process, which will include:*

- The cost of project management for the de-normalisation process such as risk analysis, configuration management and quality control.
- The cost of writing the extra code to prevent anomalies.
- The cost of maintaining the extra code to prevent anomalies.
Chapter 9  Cost comparison of conversion to OODB with other methods

- The cost of slower performance of data insertion, deletion and update.
- The cost of additional errors occurring in the data because of the anomalies missed.

_Consider the benefits of de-normalisation, which will include:_

- Reduced employee time to use the system due to the increased performance of data retrieval.
- The end result will produce the overall cost of implementing and operating a relational database system with a de-normalised data structure.

**Step 2 - Determine the cost of conversion to an object-oriented database**

_Consider the costs to implement an object-oriented conversion process. Which will include:_

- The cost of project management for the conversion process such as risk analysis, configuration management & quality control.
- The cost of the object-oriented database management system.
- The cost of employee time to convert the data to the object oriented model.
- The possible costs to train employees in object-oriented methods.

_Consider the benefits of conversion to the object-oriented, which will include:_

- A saving of employee time in using the database due to the increased performance.
- A possible reduction in maintenance costs if the system is modified or extended
- The end result will produce the overall cost of conversion to and operating an object-oriented database.

**Step 3 - Compare the costs from step 1 and step 2.**
Costs need to be determined over time as the de-normalisation method may give short term gains but the object-oriented system’s greater integrity and resulting reliability is likely to give longer term gains. (See Figure 9.4-1.)

Figure 9.4-1 Cost comparison for performance improvement alternatives

Step 4 – Determine if a simple hardware upgrade would be as cost effective

See if processor and disk storage hardware upgrades are available that will give the increase in performance required and if so, determine the cost of the upgrade and compare the costs of de-normalisation and conversion to an object-oriented system.

9.5. Carrying out the methodology.

A quick examination of the steps of the methodology indicates that the outcome is bound to depend on the circumstances of each individual system. For
example, a system, which is rarely used to extract data from joined tables, may not achieve any benefit from de-normalisation or conversion to an object-oriented system. A system where many future enhancements are envisaged will more able to benefit from the proper data model of the object-oriented system whereas the improper model of the de-normalised system could make the enhancements considerably more difficult. It is clear that database owners will need to research the individual circumstances of their own systems.

The suggested research methodology is to carry out a mixture of performance benchmark tests and case studies. Ideally, a database system will need to be compared in a fully normalised relational format, a de-normalised relational format and an object-oriented format. Benchmark tests will need to be made using the available alternative database management systems. For example, non-normalised data systems could be compared with normalised systems using databases such as Microsoft Access or MySQL. This benchmark data can also be compared with published data concerning comparisons of RDBMS and OODB such as provided in the report by STR (STR, 1997).

Both paths (de-normalisation and conversion to object-orientation) in Figure 9.4-1 are likely to improve the performance but the question is which will give the greater cost savings? This is where case study data can be examined to provide the costs of operating and maintaining de-normalised and object-oriented databases, assuming such data can be found for similar systems to the system under consideration.

Suggested methods for obtaining the necessary data from case studies are as follows:

Data should be obtained from the database system log or by examining the process carried out by employees to determine the frequency of data retrieval involving the join operation. Data indicating how often de-normalised data tables are updated would also be useful.
Chapter 9 Cost comparison of conversion to OODB with other methods

Data should be obtained from employee timesheet records for employees involved in creating and maintaining the systems. Case study employee time data is needed from similar organisations that have:

- De-normalised a database, including adding extra code to prevent data anomalies.
- Corrected errors caused by de-normalisation.
- Carried out enhancements to a de-normalised database.
- Converted a relational database to an object-oriented database.
- Carried out enhancements to an object-oriented database.

Other company data that may be available are the fault reports for the database systems, possibly available from the company helpdesk. The faults attributable to integrity errors resulting from table de-normalisation need to be identified and the frequency of such faults needs to be calculated. The helpdesk may also give an indication of the time taken and the associated costs to correct the errors.

A skills audit should be carried out to determine employee abilities in object-oriented methods and the training required to meet any shortfall. The training costs then need to be determined. Training can be via an external training agency, in which case the cost of that training plus the employee time to attend the training must be calculated. Training may also be internal to an organisation in which case the time of the employees involved needs to be determined. Where possible case studies of similar organisations that have undertaken the training will give a more accurate estimate of training costs.

The above case study data will not all be required in every case. For example, for a system expected to remain stable with little or no enhancements would not need the employee times to carry out this form of maintenance. Furthermore if a careful design and implementation of extra coding to prevent anomalies from de-normalised databases will be undertaken it could be argued that data anomaly errors resulting from de-normalisation will never occur!
The predictions of the future costs depend on the data structure of the system, for a complex data structure we should be concerned with the following:

- Short-term costs of conversion to object-oriented versus the on going cost of maintenance as Figure 9.5-1 shows.

<table>
<thead>
<tr>
<th>Cost of external training agency</th>
<th>Conversion cost</th>
<th>De-normalisation cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cost of employee time to train</th>
<th>Conversion cost</th>
<th>De-normalisation cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cost of time for employees to overcome difficult cases</th>
<th>Conversion cost</th>
<th>De-normalisation cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes (Need of expertise sometimes)</td>
<td></td>
<td>No</td>
</tr>
</tbody>
</table>

Figure 9.5-1 Short term costs of conversion versus on going cost of maintenance

- Long-term savings of conversion versus the on going cost of maintenance as Figure 9.5-2 shows.

<table>
<thead>
<tr>
<th>Extendibility</th>
<th>Conversion</th>
<th>De-normalisation cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reusability</th>
<th>Conversion</th>
<th>De-normalisation cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Less code</th>
<th>Conversion</th>
<th>De-normalisation cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stability (proper model)</th>
<th>Conversion</th>
<th>De-normalisation cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td></td>
<td>No</td>
</tr>
</tbody>
</table>

Figure 9.5-2 Long-term conversion savings versus on going maintenance costs.

Based on the above figures the short-term costs of conversion will add to the expenses of the company for some time but after that there would a long-term
Chapter 9 Cost comparison of conversion to OODB with other methods

savings that could pay back the short-term costs in addition to degrading the ongoing of maintenance cost. Somewhere along that line is the break-even point. The break-even point is the key factor for decision makers in the company to convert or de-normalise.

9.6. Summary

This chapter has identified two alternatives to moving to object-oriented databases to improve performance. De-normalisation will also improve performance but at a cost of increasing the number of errors and/or increasing the code to be maintained. The second, simpler alternative is to obtain a bigger and faster computer but it is uncertain whether this can give the same improvement or cost benefits. This identifies a need to properly cost the reported advantages of speed and low maintenance of object-oriented databases and to compare these costs with the alternative methods of improving performance.

This chapter outlines the methodology to be undertaken to establish the costs of creating, operating, and maintaining databases in normalised and non-normalised relational systems and also in object-oriented systems. The next chapter concentrates on case studies comparing the performance of a normalised database system with complex data before and after any changes applied to it and evaluating the associated costs. The aim is to verify that a conversion to an object-oriented database gives greater cost benefit in the long term and, if possible, to determine the break-even point where it becomes the least expensive option.
Chapter 10: Case Studies of Current Relational Improvement Methods

10. Introduction

This chapter is based on the paper “A methodology to assess the cost of database performance improvement.” Given at the Issues Of Quality Management And Process Improvement, SQM 2002 in Limerick, Ireland in March 2002. See appendix F.

This chapter considers database systems used in different Information Technology departments of governmental sectors and scientific research institutes. The intention was to see the difference in modeling capabilities and performances in the state of Kuwait between legacy systems and object-oriented systems used in these IT departments. In addition the objective was to compare and calculate the cost of de-normalising the relational database and the migration from the relational model to object-oriented model.

10.1. The Zakat House

Zakat House is one of the biggest governmental charity sectors. The “House” as it is dubbed, provides its charity services to more than 41643 families inside the country and approximately half of that number outside the country according to statistics of the year 1999. The department of Information Technology was established to meet an urgent need for a database to keep up with the large amount of data and to better serve its customers.

The department is using Oracle8 at the moment and trying to upgrade their system with Oracle9. The system analysts and programmers are of mixed experiences some of more than 20 years. They have a very good relational database background but unfortunately no object-oriented knowledge except for the chairman Mr. Ajeel Al-Toog. At the time of my visit all programmers and analysts (about 10 of them) were attending an introduction course for Java. The intention is to use the tools provided by Oracle.
Chapter 10 Current relational improvement methods

The cost of the course was about 15000 pounds (1500 pounds per trainee). After that course the best three out of the all trainees will progress onto another advanced course on Java at a cost of 6000 pounds (2000 pounds per trainee). After finishing the advanced course the three trainees must convey what they have learnt to the rest of the staff in the IT department.

It was difficult to get any other results from the House since there was no object-oriented system designed or a converted version of any existing system. There would have been a chance to see some differences in performance between relational and object-relational model if the IT department were to use at least an object-relational data model such as defining an object in a relational database by using “Create object” and “Create table” of that object in any running relational database system.

Lack of expertise was the main reason for not doing such an experiment. The idea was too good to just forget about it, that is why they were motivated to train all staff by spending so much money and learn object-oriented concepts and programming in Java. I think in the very near future there will be a good chance to cooperate with the House to get some good benchmark comparison tests of relational versus object-oriented and object-relational models.

The only result I can rely on is the cost of training their staff, which it is about 21,000 pounds worth of training itself in addition to the cost of time spent out of the House. The cost of the licence of Oracle 8 is already included since they use it for their relational systems. This leaves the cost of the Java Database Connectivity JDBC that they will use in future and how it will save by improving the performance relative to their RDBMS.

10.2. Ministry of Planning

The Ministry of Planning controls the future plans of the country and works as a coordinator between different governmental sectors. The Information Technology department uses Oracle8 at the moment, which is running on a mainframe that is managed by some external software company. The senior
analyst, Mr. Khalid Parvees, and his team are running most of the database jobs in the IT department. They are using a relational database model on their systems.

I had a long discussion with a number of programmers about the benefits of object-relational tools in Oracle8 and why they were not used in their projects. I found out that the programmers and analysts in the department are not aware of any object-relational or object-oriented tools or modeling. The only person in the department who had some knowledge of object-oriented concepts was the senior analyst, Mr. Khalid Parvees.

I met the manager of the IT department who did not have much knowledge on computing. He was annoyed with the long delays in producing many reports needed from him by other departments. Some reports took the IT department about three to four days to be ready. Those delays were because of the large volume of data stored (data for the whole country sometimes) and the complex relationships between the databases, which is modeled using the relational model. It would have been a very good place for testing the difference in performance among the alternative improvement methods of normalised, denormalised, and object-relational databases.

10.3. Ministry of Defence

In the department of Ammunition Support in Ministry of Defence in the state of Kuwait they are using Oracle8 to store information pertained to everything stored in their warehouse. An external software company using the relational model is building the new database system. The developers in the software company are not using any object-oriented or object-relational improvement techniques. Due to the lack of object-oriented knowledge developers are using some of the other improvement techniques that Oracle8 provides to speed up some of their queries.

Mr. Javeed, a senior programmer, explained to me one of the improvement methods he used to speed up the retrieval time of some data. The technique he used was a new feature that Oracle8 provides which is to partition a table
physically in the disk into several partitions. The table would look logically as one
table to the user who can access any part of it. Many users can access the table
without locking. This technique brought down the waiting time for processing a
query from 107 seconds to 4 seconds.

Users of the Oracle database at the technical stores of the Kuwaiti Army
were finding queries relatively slow on their front-end tools. The queries were
used to check previous transactions of spare parts. These transactions included

1. Orders
2. Receipts
3. Returns
4. Issues
5. Non-Availability Declaration

Upon receipt of a demand for spares, the users need to process it in the
following way:

A. The part numbers on the demand have to be checked for previous issues/Non-
Availability Declarations.
   - Against a particular vehicle number (if it is a vehicle demand)
   - Against the demanding unit (brigade, battalion etc..) if it is a stock
demand.

B. The application has screens for users to query the database, allowing them to
enter specific criteria like part number or vehicle number. These screens are
heavily used by the users every day before each demand is approved for issue.
The approximate usage rate was more than 300 times in 5 hours.

Before improvement, each transaction was taking approximately 14
seconds real time, which means approximately 70 minutes of retrieval time
excluding the time taken to get hard copy approval (signatures, etc..) and also the
time taken to actually post the transaction into the database. Indexing was used as
an improvement method.
Chapter 10 Current relational improvement methods

After the creation of indexes on the concerned tables, using columns used by the queries as well as columns used to define relationships within the database, a vast improvement was observed in the performance of the queries. The retrieval time went down drastically from 14 seconds to 2 seconds real-time.

The size of the extra code for improvement was only about ten lines. These were ordinary SQL commands written in less than one hour. No extra code was necessary to ensure code integrity because no updating was needed. This shows that some improvement methods can be quicker and safer than others.

10.4. Kuwait Institute for Scientific Research (KISR)

KISR is one of the largest scientific research institutes in the gulf region. It supports all sorts of scientific research. The Information Technology department uses Oracle8 for their database systems. The institute uses a Local Area Network (LAN) with Novell operating system and Sun Spark 2000. The senior analyst, Mr. Faisal Al-Quimlas has been in the department since 1984 and he is very interested in object-orientation but has not used it yet.

There is no direct need to convert from relational database to object-oriented database at the moment. Such conversion might take place only for the sake of research. After discussing which system to use for experimentation I found some previous results of an old system used.

The store system in the KISR calculates the cost of various stationary consumptions for each department in the whole institute. There are several queries to get the results needed from the database. Some queries need a long period for the calculations. For example they need to give a report calculated over five years to the general accounting department for the Kuwaiti government. Such a query took more than four hours on a stand-alone machine in 1991. The same query took 5 minutes in 1995 using a LAN server. It takes less than a second now using a Sun Spark 2000.

This query is de-normalised to give better performance. The extra code written to ensure integrity, including the associated project management, took
approximately 320 working hours. The de-normalisation was implemented for research purposes and not for commercial use. In industry they would be more concerned about the frequency of using the query in the DBMS and is it feasible to spend all that time to increase performance.

10.5. What can be learned from the case studies?

The case studies have shown that a full set of data is difficult to obtain from any one source. This is largely because uses of object-oriented databases do not appear to be widely used in the Government circles and public bodies in Kuwait. There is the capability of producing data with an object-relational model as several organisations use Oracle8, which has object-relational facilities. However, it would seem the object facilities are not being used, mainly because of a lack of experience and training in object-oriented techniques. Some steps towards object-orientation are being made and in the future the organisations studied could yield a useful source of data. This means that, at the moment, no conclusive cost results can be obtained, though when the case studies are taken together there are some interesting findings that enables tentative cost results to be obtained:

- The fact that a number of organizations already used Oracle8 which has object-oriented facilities built in shows that the cost of obtaining the software to implement an object-oriented system (or at least an object-relational system) is likely to be zero.

- In the case of the Zakat House the extensive training was a major concern to try using object-oriented tools provided by Oracle8. The training in turn, had a high cost in both the cost of training itself and the cost of time taken in training. Assuming that their training requirement is typical for an organization that does not have object-oriented database experience it is useful to calculate the total training cost incurred.

- Cost of the object-oriented course (1500 Pounds per trainee). 10 * 1500 = 15000 pounds
Current relational improvement methods

- Cost of advanced object-oriented course (2000 Pounds per trainee). Best three were chosen $3 \times 2000 = 6000$ pounds.

- Cost of time spent out of Zakat house for programmers to study the introduction course (2200 pounds per trainee) $10 \times 2200 = 22000$ pounds (assuming £50,000 salaries).

- Cost of time spent out of Zakat house for the best three programmers to study the advanced course. $3 \times 2200 = 6600$ pounds.

- Cost of employees’ time training each other. $10 \times 2200 = 22000$ pounds

- Cost of object-oriented licence.

The total cost in Zakat house came to the sum of 71600 pounds to use object-oriented tools of Oracle8 in addition to the cost of object-oriented software licence.

- The Ministry of Defence case study showed that queries that are heavily used can have a frequency of over 300 uses in 5 hours, or more than 60 uses each hour. Assuming this level of use is not untypical we can deduce the time taken for such queries if we know how long each one takes. Before indexing was used each took an average of 14 seconds, afterwards each took an average of 2 seconds.

  The cost/hour of waiting for the transactions at 14 seconds would be $14 \times 60 = 840$ seconds which is 23% of the operator’s time each hour. The cost/hour of waiting for the transactions at 2 seconds would be $2 \times 60 = 120$ seconds which is 3% of the operator’s time each hour. Assuming operators are slightly less paid, perhaps £40,000 salaries, this means a saving of 20% of operator’s time, which is £8,000/year.

- The cost of de-normalisation was reported by the KISR at 320 hours of employee time. This effort is not out of line with the experience reported in Section 9.3. Here it was reported the extra coding for just 3 forms totalled 54 man-hours of effort. The 320 hours experienced at KISR is probably typical of a system where de-normalisation is extensively used for performance considerations.
Chapter 10  

Current relational improvement methods

The 320 hours cost the KISR 7.6 weeks of work for one employee, where each week costs KISR 961 pounds. The total cost = 7.6 * 961 = 7304 pounds assuming £50,000 salary.

- It was not possible to obtain the cost of data anomaly errors resulting from de-normalisation. However, the KISR case study shows that where de-normalisation is done intentionally the organization is likely to make a significant effort to produce the necessary code to safeguard against the risk of data anomalies. The experience quoted in section 9.3 also verifies this. For the purposes of this research, therefore, it will be assumed that the effort put in to the extra code will prevent any data anomaly errors occurring.

- The one significant piece of data missing from the case studies is the cost of conversion of a relational database to an object-oriented database. However, it was discovered that many organizations already use software with object-relational facilities. As this can support ordinary relational data modelling as well as the object modelling it suggests that where legacy databases exist, it would probably only be worth conversion of the data to the object-oriented model where complexity exists such as many to many relationships. There would be no need to completely remodel the whole system. It is likely, therefore, that the only remodelling that would take place would be in the same places that de-normalisation could be used as an alternative.

For the purposes of this research, therefore, it will be assumed that, if the employees are fully trained, the time it would take them to convert the necessary parts of the system to the object model will be similar to that taken for the de-normalisation process. This is a very big assumption. In practice, this can be considered to be a lower bound for the time taken - changing from one data model to another will almost certainly take more time than the relatively simple de-normalisation process.

The case studies also gave us other unexpected information that did not help in the calculation of the costs of de-normalisation or object-orientation but
nevertheless were relevant to the exercise of working out what was the most cost effective method of performance improvement:

- In the case of the Ministry of Defence, the table partitioning used gave an improvement of a substantial speed increase (approximately 25 times the original speed). This shows that the claimed advantage of object-oriented migration or de-normalisation should be reconsidered in the light of results of the partitioning improvement technique. Thus, we should not jump into higher cost solutions before we exhaust available tools that may better suit the budget of the company and accomplishes the job just as effectively.

- By looking at the indexing improvement method (7 times increased) we can see that it is equal to the lower bound of improvement using an object-oriented database (Douglas, 1997). Again this is a simple, virtually cost free method of achieving a performance improvement.

- Another area that we should look at is the hardware improvements as seen in Kuwait Institute for Scientific Research and how the upgrading of the computer have lead to around a 50 fold speed increase in the four years 1991 to 1995 and a further 300 fold increase in the speed in the five years to the year 2000. This shows a the speed was nearly trebled each year for the first four years and more than trebled each year in the next five years. As a result, if any performance improvement is going to take any significant time there is a danger that the equivalent benefits would have come about naturally just by upgrading to the latest hardware at the end of the conversion period!

10.6. A calculation of the comparative costs of de-normalisation and conversion to object-orientation

The cost of using object-orientation at Zakat house is more than £71600. On the other hand, the cost of using a legacy system in the Ministry of Defence is about 23% of employee’s time, which is about £9200 before improvement using indexing technique. After improvement the cost was reduced to £1200, which is about 3% of the employee’s time. The saving is about £8000 per employee.
The cost of de-normalisation at KISR was £7304 to increase performance but by upgrading the hardware the performance was increased dramatically from more than four hours to 5 minutes and finally to less than a second when using Sun Spark 2000.

Results form chapter8 shows that de-normalisation is better for data when there is no need to maintain data integrity, which could cause massive amount of extra coding. Another pitfall was when producing HTML tables. Other than that the cost saving was drastic and obvious by simply looking at the retrieving time comparison between the normalised and de-normalised queries in chart 8.6-1and 8.6-2.

In the article “How to Store Java Objects” by Tomaz Domajnko, the performance degrades in both relational and object-oriented systems for a system that uses JDBC on the relational system see Table 10.6-1 and Table 10.6-2. Moreover, the degradation in performance is less for the object-oriented system when moving from simple data type to complex objects, see Figure 10.6-1.

<table>
<thead>
<tr>
<th></th>
<th>Poet</th>
<th>Oracle 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>4503.6</td>
<td>9545.1</td>
</tr>
</tbody>
</table>

Table 10.6-1 the Instantiation time in milliseconds for a set of 1000 simple objects.

<table>
<thead>
<tr>
<th></th>
<th>Poet</th>
<th>Oracle 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>6055.2</td>
<td>18293.8</td>
</tr>
</tbody>
</table>

Table 10.6-2 Instantiation times in milliseconds for a set of 1000 complex objects.
The set of tests that was designed to test queries in databases were classified by their complexity and the number of objects involved. The distance of a query was defined as a number of object types involved in a query or a number of table joins for the relational database. The results are given in Table 10.6-3 in milliseconds for performing 100 queries; each query selected a single object from a database with 45,000 stored objects.

<table>
<thead>
<tr>
<th>Distance</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poet</td>
<td>4797.2</td>
<td>5072.4</td>
<td>6599.5</td>
<td>7044.7</td>
</tr>
<tr>
<td>Oracle7</td>
<td>3327.8</td>
<td>5154.4</td>
<td>10267.5</td>
<td>18780.5</td>
</tr>
</tbody>
</table>

Table 10.6-3 Performance degradation from query distance.

Figure 10.6-2 shows that with growing complexity of queries the time needed for query execution grows, but it grows slowly in the case of Poet while in the case of relational database the query execution times rise rapidly because of the time consumed in table joins.
The effort needed to get the last results was measured using LOC (Lines Of Code) one of the popular software metrics. The measurements show that by using object database management systems and by storing native objects there will be only a slight increase development effort (less cost). In the case of using relational database management system there was a substantial amount of additional effort needed when compared to native Java implementation (more cost).

10.7. An Illustration of the Application of the Cost Methodology

To illustrate the application of the cost methodology it is useful to apply the costings to an imaginary organisation that has characteristics similar to those found at the Kuwait organisations studied. Let us assume the organization is using Oracle8 or Oracle9 already. This means no extra licensing costs will be incurred when introducing object orientation. It also means they have the capability of upgrading to an object oriented model but can be selective as to where this upgrade is applied.

As discussed in the last section the greatest benefits of object-orientation is gained when the data is complex in nature, so let us assume the organisation is
only interested in a performance gain where queries on complex data are regularly performed.

Let us assume the number of development staff in the organisation is the same as that for Zaket House and that a similar training strategy is applied. Let us also assume that the complex data queries are performed on average 60 times an hour as at the Ministry of Defense, and also as at the Ministry of Defense each query takes 14 seconds before any improvements are applied. Let us assume that the de-normalisation of these complex queries would take 320 working hours as at the Kuwait Institute for Scientific Research including the project management of the process. This effort includes that to produce additional code to preserve the integrity of the database and, because this code has been applied, the resulting number of database errors is not significantly greater than before the de-normalisation. We will assume that development staff cost the company £50,000/year and the database operator staff cost the company £40,000/year.

As shown in the previous sections this means that:

- Training in object oriented methods will cost the organisation a total of £71,600
- The cost of de-normalising a single query is approximately £7,300
- The cost of converting this part of the database to the object oriented model will be at least £7,300 and is likely to be a lot more.
- The cost per year of running each complex query is 23% of £40,000 = £9,200/year.

According to Barry Douglas (Douglas, 1996) the improvement in performance using object orientation will normally be in the range of 7 to 100 times increase in speed. This means the cost of waiting for the query would be reduced to between £1,314 and £92 each year. Because we can be selective as to where the object-orientated model is applied, let us assume the performance benefits will be towards the end of the range giving the best performance gain.
Eg. Assume the cost of the query would be £200/year giving a saving of £9,000/year on the query using the relational model.

The results of the benchmark testing in chapter 9 show that de-normalisation can give similar improvement in performance for complex queries as that which can be achieved by conversion to the object oriented model. So let us assume the saving would also reduce the waiting time for a query to between £1,314 and £92 each year. Taking £700 as being in the middle of the range, this gives a saving of £8,500/year on the query using the relational model.

These calculations allow us to calculate the break even points to determine how long it takes before de-normalisation or conversion to object orientation will start to give an overall saving.

For de-normalisation the initial cost of £7,300 is recovered at £8,500/year. This gives a break-even point of between 10 and 11 months.

For conversion to the object-oriented model the break even point depends on how many queries are subject to conversion. For a single query the costs would be £71,600 for training and £7,300 + for conversion, recovered at £9,000/year. This gives a break-even point would not be until at least 8 years and 9 months. However, a single query conversion is unlikely. If 10 such queries were subject to conversion to the object oriented model the break-even point would be reduced to something over 1 year and 4 months.

However, both the de-normalisation and the object oriented model conversion calculations assume no other improvements had been made. The Ministry of Defense reduced the query time to 2 seconds simply by introducing indexing. This gave a saving of £8,000/year with negligible implementation costs. It would take of the order of 15 years for de-normalisation to become less expensive than indexing and object oriented model conversion nearly 9 years to become less expensive than indexing. Clearly this makes indexing by far the best alternative.
Chapter 10
Current relational improvement methods

Could the option of de-normalisation or object-orientation be used in addition to indexing? Certainly this may be possible but it is unlikely that de-normalisation or object orientation could achieve a further 7 to 100 times performance improvement if indexing had previously been introduced. However, for the sake of comparison let us assume a 100-fold increase in addition to that achieved through indexing.

Each query takes only 2 seconds after indexing is introduced giving a total waiting time cost of just over £1,300/year. The 100-fold improvement would reduce the waiting to negligible time and so give a saving of £1,300/year. This would give a break even point for de-normalisation of between 5 and 6 years and for object oriented model conversion for 10 queries a break even point of about 11 years!

This shows that if indexing is available as an option then neither de-normalisation, nor conversion to the object-oriented model is a worthwhile method of improving performance for an organisation such as the one considered here. Other improvement methods may be possible. The Ministry of Defense obtained a 27 fold increase in performance using table partitioning on the disk storage, this again would make the application of de-normalisation or object orientation not cost effective.

Furthermore, the improvement in performance has to be compared with the improvement that would naturally occur through upgrading the hardware. The experience of KISR showed that every two to three years the improved speed of hardware upgrades would give performance gains comparable with that of de-normalisation or object orientation. This will significantly reduce any long-term performance benefits of de-normalisation or conversion to object orientation.

Other organisations may have different characteristics that would make de-normalisation or conversion to object orientation more cost-effective options. This chapter has shown the importance of examining the costs of each option carefully if the performance cost is an issue causing concern.
Chapter 10  

Current relational improvement methods

There are, of course, other reasons to convert a database to the object-oriented model. Object orientation, it is claimed, gives a system that is easier to change and extend, and as is shown in chapter 7, conversion to the object-oriented model can correct the errors of a badly modelled relational database. There may also be political reasons for converting to the object oriented model if, for example, it made the data compatible with that in other parts of the organisation. Each of these reasons must be considered on their own merit but this chapter has shown that if the motivation is solely to improve performance then the organisation should proceed with caution and should carry out the cost comparison methodology described in chapter 10.

10.8. Summary

This chapter has shown many basic improvement methods in different environments. In some cases there is no need to convert to an object-oriented system or de-normalise data, while in other cases one of the two choices could cost less with an increased performance.

Unfortunately no single organisation was able to supply all data for the methodology to compare costs to be carried out. However, by assuming the data for each organisation studied is typical, and by using published benchmark data and the benchmark data derived in chapter 9 it is possible to show the working of the methodology for a “typical” organisation.

We can conclude that a methodology could greatly help in giving hints in choosing certain model for each and every specific case depending on the performance desired and the approximate budget of the company.

It was found that for the “typical” organisation used for the methodology neither the de-normalisation of the data nor the conversion of the data to object-oriented model is likely to be worthwhile for performance reasons alone. Other, simpler improvements such as table partitioning or indexing may be available, and if so, these would give similar improvements at negligible cost. Furthermore,
the continual improvement of computer hardware is likely to give an equivalent improvement in performance within a few years.

Many questions remain unanswered in the process of choosing a database management system. One of them is the presence of object-relational database management systems. Vendors assure that this technology brings the best from both worlds, but this research suggests that conversion to object-oriented databases is unlikely to be justified on performance improvement reasons alone.
Chapter 11: Conclusions and future research recommendations

11. Conclusions

This research has looked in depth at the shortcomings of the relational model, especially the part concerned with modelling and manipulating complex data. Real world problems that contain complex data exposed the relational model deficiencies in either expressing the problems easily or/and in performance of DBMS. These reasons combine together to reveal the unnecessary extra cost imposed on the database systems. Thus researchers are very interested in many different improvement methods to achieve a better performance and savings.

Many of the existing relational databases are deliberately de-normalised for a better performance. For example a many-to-many (M-N) relationship needs an extra intersection table for each (M-N) relationship to prevent any possible anomaly. The alternative is for more tables to be added resulting in more joins being needed and these extra joins can significantly reduce performance.

One of the well-known improvement techniques used is converting a relational database to an object-oriented database because the later describes complex data more naturally. Previous research has indicated that in order to convert a relational database to an object-oriented database the relational database must be in third normal form (3NF). This thesis shows that a relational database containing type code complexity does not need the normalisation requirement. In addition, the conversion process informally normalises the non-normal model.

11.1. De-normalisation conclusions

By benchmarking the normalised relational database against de-normalised database it was shown that the de-normalisation of data does improve performance times, but at the cost primarily of storage space. De-normalisation also produces a 'new' overhead, that is the cost of program development time to implement a protection system, to preserve integrity and the fact that one normalised query may in fact be multiple queries on a de-normalised system.
The time saving of the execution of de-normalised joins is of a large order of magnitude; therefore the time complexity savings increase very quickly as the size of data increases.

While additional overheads are introduced, the time complexity of these does not seriously affect the time taken when using a de-normalised database. With large amounts of data, it is true that numerous queries on a single table are far quicker than performing a join on two tables.

When small amounts of data are being used, de-normalisation can cause more problems than the benefits are worth because of the integrity checks for any anomalies that might occur. This is because the machines that run these programs are fast enough when processing small amounts of data and joining tables is easier to handle in normalised form. Even if the structure of the normalised format is not at its most efficient it would not be noticed.

Overall de-normalisation is not as good as it is sometimes made out to be. Most queries do not get a benefit out of de-normalisation. The benefits only really become effective where a large number of select queries are performed on large data sets.

The noticeable improvements all stem from the reading of data (SELECT operations), for the rest of the queries there is not an advantage to de-normalisation. The overheads of de-normalised tables being updated can be seen especially by tests of INSERT and DELETE operations.

This research does show that any improvement to be made would most likely come from the reduction in the time taken to perform table joins.

11.2. Conclusions from the case studies

This thesis has identified two alternatives to moving to object-oriented databases to improve performance. De-normalisation will improve performance but at a cost of increasing the number of errors and/or increasing the code to be maintained. The second, simpler alternative is to obtain a bigger and faster computer but it is uncertain whether this can give the same improvement or cost.
benefits. This identifies a need to properly cost the reported advantages of speed and low maintenance of object-oriented databases and to compare these costs with the alternative methods of improving performance.

This thesis outlines the methodology to be undertaken to establish the costs of creating, operating, and maintaining databases in normalised and non-normalised relational systems and also in object-oriented systems. This methodology clarifies the way to the optimum preferred solution by scaling the cost savings from improving the performance against the cost incurred because of the improvement. The methodology has been divided into four steps as follow:

- Determine the cost of de-normalisation including the subsequent costs of writing extra code to prevent, and maintain anomalies against the benefits from reducing employee time to use the system due to the increased performance of data retrieval.

- Determine the cost of conversion to an object-oriented database including the cost of implementation process, employee time to convert the data to object-oriented model and the possible costs to train employees in object-oriented methods against the benefits of conversion by having the long-term savings as mentioned in chapter 9.

- Compare the above.

- Determine if a simple hardware upgrade would be as cost effective as in the case of KIRS in chapter 10.

11.3. Guidelines for performance improvement

When trying to improve the performance of a system, the guidelines that we can deduce from this research are as follows:

If performance is a problem then first thing to be looked at is the available methods of improvement that are easy to implement, such as partitioning or indexing before de-normalisation or moving to an object-oriented system.
Chapter 11 Conclusions and future research recommendations

If the system uses a lot of select joins then de-normalisation might be the solution, but there is a need to estimate the break-even length. Case study data for any similar systems and organizations may give some indication of the likely costs.

If it is going to take a long time to introduce object-oriented databases then the improvement could simply be achieved from improved hardware in the machine. If the current system is running on hardware more than a year old then upgrading the hardware to the latest version may give all the performance improvement required.

The cost of training redevelopment with the move to object-oriented is likely to be very high and could easily eliminate any benefit of conversion.

All conclusions depend on circumstances- this thesis has given guidelines and methodology for each individual company to make their own assessment.

11.4. Future research

- Further research is required into de-normalisation. An actual implementation of a database abstraction layer capable of adaptive de-normalisation would be quite a feat. A system with enough rules built-in to perform the task would take a large amount of design and programming. Not all of the answers of how to do this are contained here, merely the ideas and concepts important to the design of such a system.

- Choosing a database management system is a task that needs careful study based on performance and cost. The presence of object-relational database management system should be taken in the consideration. Vendors assure that this technology brings the best from both worlds. These products should be evaluated in the future.

- Further case studies are required to show whether the results obtained in this research are typical. Case studies involving commercial companies with large amount of complex data types that are using relational database system where cost is crucial and saving matter greatly.
A case study is required for a large amount of data that is stored in legacy system. Through the thesis the factor presented was choosing a database system. Another factor should be taken into account that is when it comes to the data that already exists in the business environments. Many possibilities exist that should be thoroughly examined such as using data in a new system of transferring that data into a new system. There is a need to find single organisations from which all data required can be obtained.

There is an emergent need to know how long it would take to convert to an object-oriented system. In addition the need to know how much does it cost to convert to an object-oriented system is very important for decision-making.

Oracle8 uses object-oriented tools on relational database or using objects in relational tables. Benchmarking of OO should be carried out on a object-relational system such as the one used in Oracle8 so that OO and de-normalisation can be directly compared on the same system.

Through out this research many improvement methods were found some of them were very efficient and saved a lot. We should take a close look and need to further investigate other improvement methods before trying the hard way.
References


Herbert, J. (2001). *Comparison Of Relational Database Management Systems And Object-Oriented Database Management Systems*, final year project, Department of Computer Science, Loughborough University, U.K.


Appendix A

Systems Evolution: Converting a Non-Normalised Relational Database to an Object-Oriented Database

Abdulaziz Al-Kandari and Ray Dawson

Dept. of Computer Science
Loughborough University,
Loughborough
Leics. LE11 3TU
UK

Tel. +44 1509 222679
Fax. + 44 1509 211586

A.AlKandari@Lboro.ac.UK
R.J.Dawson@Lboro.ac.UK

Abstract

With the growing development of object-oriented database management systems more and more data owners are looking to convert their legacy data stored on relational database systems. Previous researchers have proposed step by step methodologies to convert a relational database to an object-oriented database, where the relational database has previously been normalised at least to third normal form. This paper shows that this normalisation requirement is unnecessary for problems with type code complexity as the conversion process will itself informally normalises the data.

Keywords

Database, Object-Oriented, Normalisation, Evolution

1. Introduction
The transformation of any system to another requires an investigation of not only their components but also their sub-components. The investigation then will reveal the important similarities and differences that will help the conversion process from the source system to the target one; database systems are no exception. Before we start thinking of converting any database system to another we should study the need and necessity for such conversion.

The relational model lacks the ability to model and manage complex data such as images, documents, video, audio, animation, and composite objects (such as nested bill of materials or time series) efficiently. The relational databases (RDB) have had difficulties to accommodate itself with the new changing face of computing. One of the main reasons, as Barry Douglas shows [5], is the difficulties of dealing with complex data such as recognising identification in data, accessing data using traversals, many-to-many relationships, and frequent use of type codes. Object-Oriented Databases (OODB) seem to handle this complex data in a more natural way, and as a result, it gives better performance and retrieval of data [11]. For this reason, RDB is loosing its position as "industry standard" to OODB. As the demand for OODB rises and the popularity of relational databases remains, developers are urged to seek a quick fix via an object-to-relational translation layer. This method offers the benefits of object-orientation to new systems, while it retains the information in the legacy systems.

The pure relational model is inadequate while the purely object-oriented model is not entirely appropriate for database use. As a result, existing relational databases have found some ways out of their dilemma by using some object-oriented concepts to solve the problems that they are facing. Oracle8 uses nested tables and aggregate objects despite this being a clear violation of first normal form [1]. There are other ways to deal with complex data by using methods such as interfacing, mapping, converting schemas, or conversion of type code into class hierarchy where most of these methods have been shown to result in a better performance [3].

To design with an Object-Oriented Database Model (OODM) it is important to bear in mind its various distinctive features such as object identity, encapsulation, and inheritance. Many researchers such as Kim [8] and Lausen [9] state that a suitable OODM is expected to use such tools in its model. The following concepts distinguishes OODM from conventional conceptual models:

- **Nesting.** Nesting allows one object to be a component of another.

- **Object and Object Identifier.** Any real world entity is an object, with which is associated a system-wide unique identifier. The system should provide an Object Identity independent of the value of its attribute.

- **Attributes and Methods.** Objects can be equipped with behaviour, which links specific operations exclusively to these entities. An object has one or more attributes, and one or more methods that operate on the values of the attributes. The value of an
attribute of an object is also an object. An attribute of an object may take on a single value or a set of values.

- **Class Hierarchy and Inheritance.** The structure and behaviour of objects can be inherited hierarchically. The classes in a system form a hierarchy or a rooted directed acyclic graph, called class hierarchy. Suppose, for a class C and a set of lower-level classes \( \{S_i\} \) connected to C, a class in the set \( \{S_i\} \) is a specialisation of the class C, and conversely the class C is the generalisation of the classes in the set \( \{S_i\} \). The classes in \( \{S_i\} \) are subclasses of the class C; and any class in \( \{S_i\} \) inherits all the attributes and methods of the class C and may have additional attributes and methods.

- **Extendibility.** The ability to extend the structure and behaviour function in a logical sense. [9]

Object-oriented database systems allow the modelling of objects, relationships, and complex structures in a way that in many applications is more appropriate than traditional systems can offer. For this reason alone they are of increasing interest in a host of applications. If an OODB is created from scratch, its design can make use of several existing techniques. However, if a database already exists, then it is a different story, for this situation many researchers [6,10] have proposed a step by step transformation from the relational to the object-oriented model.

2. Problem

Previous research shows that the normalisation process is an important factor for conversion, for example, Johannesson [7] indicates that to simplify the translation process of a relational schema to a conceptual schema the former needs to be in a third normal form (3NF). Others, like Davis [10], also place a restriction on the relations to be in 3NF. This implies that it necessary to normalise a relational database system to be converted into an object-oriented system but is this really necessary? If it is, what is the degree of normalisation required (1NF, 2NF, or 3NF)? Previous researchers, such as [6], show that the relational database has to be normalised (at least up to 3NF) in order to convert its schema to object-oriented one or to another conceptual data model. Other research shows the possibility of converting an entity relationship model to an object oriented model [2], but a properly formulated entity relationship model will give a data structure normalised at least up to 3NF.

The richness of an object-oriented data model makes it possible to represent a broad range of databases including relational databases. However, to directly convert a non-normalised relational database to an object-oriented database would eliminate the need for the intermediate process of normalisation. Furthermore, we will be able to see if the conversion process will handle the partial and transitive dependencies (i.e. normalisation). Alternatively, it would be beneficial to find out if the conversion process would still preserve the different anomalies in the source database and see the impact on the resulting object-oriented database.
3. Method of Converting

In most database applications, there are numerous objects (data) of the same type. This kind of data is called the Type Code [5] and is usually represented by a column in a relational database table. For example a person type could be a student, lecturer, administrator, etc. Object-oriented models type code complexity in a more natural way than conventional systems since a class in an object-oriented model represents a set of objects of a particular type (such as Person). Type or Class Hierarchy in the object-oriented model takes the place of a type code in the relational schema. The conversion process of a non-normalised relational database to an object-oriented is meant to exploit the inheritance feature provided in the Object-Oriented modelling paradigm. An example of a non-normalised relational database with a type complexity feature is given in Table 1. The relational schema of the base table is as shown in Table 1.

<table>
<thead>
<tr>
<th>ID#</th>
<th>Sub#</th>
<th>Rm#</th>
<th>Time</th>
<th>Ename</th>
<th>Etype</th>
<th>D.O.B</th>
<th>Hdate</th>
<th>Tdate</th>
<th>Sname</th>
</tr>
</thead>
</table>

Table 4: The relational schema of the base table

If the above schema were implemented using one of the relational databases, such as Access™, then such implementation would reveal the anomalies found in the non-normalised database. When this is converted for implementation on object-oriented database software, such as Visual Works™, it will show whether the object-oriented converted schema would preserve the anomalies found in the non-normalised relational database.

<table>
<thead>
<tr>
<th>ID N</th>
<th>ENAME</th>
<th>ETYPE</th>
<th>D.O.B</th>
<th>HIRE.D</th>
<th>TEN.D</th>
<th>S.NO</th>
<th>SNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Don</td>
<td>Support</td>
<td>22-5-1993</td>
<td>22-5-1993</td>
<td></td>
<td></td>
<td>Compisci1,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Compisci2,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Math1</td>
</tr>
<tr>
<td>2</td>
<td>Pat</td>
<td>Instructor</td>
<td>10-6-1990</td>
<td>10-6-1990</td>
<td></td>
<td>C101,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C102,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M101</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Donna</td>
<td>Professor</td>
<td>15-8-1978</td>
<td>15-8-1978</td>
<td>15-1-1985</td>
<td>S1,</td>
<td>System,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D1</td>
<td>Database</td>
</tr>
<tr>
<td>4</td>
<td>Kim</td>
<td>Work-study</td>
<td>20-9-1994</td>
<td>20-9-1994</td>
<td></td>
<td>I1,</td>
<td>Intro,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C101,</td>
<td>Compisci1,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M101</td>
<td>Math</td>
</tr>
<tr>
<td>5</td>
<td>Fred</td>
<td>Professor</td>
<td>17-7-1985</td>
<td>17-7-1985</td>
<td>22-1-1992</td>
<td>S1,</td>
<td>System,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P1</td>
<td>Physics1</td>
</tr>
</tbody>
</table>

Table 5: The original non-normalised table
Table 2 shows an example of a non-normalised database where the type codes are embedded in the data. When inspecting the table schema it is not particularly obvious that the ETYPE column represents a type code. Even when the table content is examined this can still be easily overlooked. However, we need to know if the table has data complexity involving type codes in order to make the conversion to object-oriented data. Each employee will generate a tentative class since it is a member of the same Type Code as in figure 1. By looking at the tentative classes generated from the original table it can be seen that all Teachers (Professor, Instructor, and Work-study) have some attributes in common, however the Support type or class does not share these particular attributes. The attributes Subject Number, Subject Name, RoomNo, Time, and Payment fall in the Teacher's category forming a super class.

Figure 3: Tentative classes
Figure 4: Class hierarchy created according to type code (ETYPE)

The attributes in common will be factored out and placed in the Teacher super-class. The tenure date will be restricted to professor type only so in that case it will be added to the professor class. The different Teacher types will form sub-classes for the super-class Teacher and they will inherit the common attributes, generating the class hierarchy shown in figure 2.

In order to normalise the original table to the first normal form we need to add an extra row for each redundant atomic value in the subject columns (according to Codd's first normal form rule). The normalised table will then look like that given in Table 3.

<table>
<thead>
<tr>
<th>ID</th>
<th>ENAME</th>
<th>ETYPE</th>
<th>D.O.B</th>
<th>HIRE.D</th>
<th>TEN.D</th>
<th>S.NO</th>
<th>SNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Don</td>
<td>Support</td>
<td>22-5-1993</td>
<td>22-5-1993</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Pat</td>
<td>Instructor</td>
<td>10-6-1990</td>
<td>10-6-1990</td>
<td></td>
<td>C101,</td>
<td>Comp sci1,</td>
</tr>
<tr>
<td>2</td>
<td>Pat</td>
<td>Instructor</td>
<td>10-6-1990</td>
<td>10-6-1990</td>
<td></td>
<td>C102,</td>
<td>Comp sci2,</td>
</tr>
<tr>
<td>2</td>
<td>Pat</td>
<td>Instructor</td>
<td>10-6-1990</td>
<td>10-6-1990</td>
<td></td>
<td>M101</td>
<td>Math1</td>
</tr>
<tr>
<td>3</td>
<td>Donna</td>
<td>Professor</td>
<td>15-8-1978</td>
<td>15-8-1978</td>
<td>15-1-1985</td>
<td>S1</td>
<td>System</td>
</tr>
<tr>
<td>3</td>
<td>Donna</td>
<td>Professor</td>
<td>15-8-1978</td>
<td>15-8-1978</td>
<td>15-1-1985</td>
<td>D1</td>
<td>Database</td>
</tr>
<tr>
<td>5</td>
<td>Fred</td>
<td>Professor</td>
<td>17-7-1985</td>
<td>17-7-1985</td>
<td>22-1-1992</td>
<td>S1</td>
<td>System</td>
</tr>
<tr>
<td>5</td>
<td>Fred</td>
<td>Professor</td>
<td>17-7-1985</td>
<td>17-7-1985</td>
<td>22-1-1992</td>
<td>P1</td>
<td>Physics1</td>
</tr>
</tbody>
</table>

Table 6: The original table in first normal form

In order to normalise the above schema in 2NF we need to get rid of all partial dependencies. Suppose we a relation schedule as in Table 4, which is not in 2NF.
Table 7: The relation schedule with all partial dependencies.

The attributes Employee name, Employee type, D.O.B, Hire date, and Tenure date are partially dependent on (part of the prime key) ID#. The attribute Subject name is partially dependent on the Sub#. Thus, the 2NF decomposition will result into three different tables as in Tables 5, 6 and 7.

<table>
<thead>
<tr>
<th>ID#</th>
<th>Sub#</th>
<th>Rm#</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8: Prime key of the relation schedule.

<table>
<thead>
<tr>
<th>ID#</th>
<th>Ename</th>
<th>Etype</th>
<th>D.O.B</th>
<th>Hdate</th>
<th>Tdate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9: Partial dependencies on the Id number.

<table>
<thead>
<tr>
<th>Sub#</th>
<th>Sname</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 10: Partial dependency on the subject name.

Table 5 shows that all attributes that are partially dependent on part of the prime key ID# along with the (Code Type) Employee Type attribute are part of the attribute set in the super class Employee in Figure 2. Tables 5 and 7 are out of the Teachers class hierarchy route as Figure 2 shows. The process of factoring out all common attributes in the tentative classes is actually like informally applying the normalisation principles by eliminating all partial and transitive dependencies.

Table 5 is in 2NF but not in 3NF since Tenure date is transitively dependent on the ID#. To get rid of the transitive dependency we need to decompose it into two 3NF tables as in Tables 8 and 9.

<table>
<thead>
<tr>
<th>ID#</th>
<th>Ename</th>
<th>Etype</th>
<th>D.O.B</th>
<th>Hdate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 11: All attributes in the Employee super class.

<table>
<thead>
<tr>
<th>Etype</th>
<th>Tdate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 12: Tenure date for Professor.

The tenure date is restricted (specialised) only to Professor type, as shown in the professor tentative class; thus it would be an attribute of the professor object alone. If there was another attribute, for example a special type of payments for Teachers, this attribute would be transitively dependent on the ID# and also common, but specialised to all teachers, thus it would be part of each teachers' tentative class. As a result, such attributes reside in Teacher Class where all sub-classes can inherit them.
4. Conclusion

Object and relational approaches have a fundamental difference regarding normalisation, in particular 1NF. A table is considered in 1NF if all columns contain only atomic values. Clearly, objects stored in an object-relational database need not be atomic, that is why it is called object-relational. It is possible to construct an object-schema from a non-normalised relational schema. The information about the relational type code forms the basis of the object class hierarchy. The process of constructing the object-oriented class hierarchy using the information about the relational type code is like informally using the normalisation principles. Attributes that are partially dependent along with type code are factored out to the super-class in the class hierarchy using the type code class as a route. By using the conversion process from relational to object-oriented one can see how normalisation is grouping out different attributes to where they semantically belong.

Previous research has indicated that in order to covert a relational database to an object-oriented database it is first necessary to ensure the relational database is normalised at least up to third normal form. This paper has shown that for relational databases containing type code complexity this normalisation requirement is unnecessary, as the normalisation will take place informally during the process of conversion.

Further work is required to examine other forms of data complexity in the relational model to discover the necessity for normalisation such as in the cases many to many relationships, non-unique identity and reflective relationships (i.e. Where an entity has a relationship with its own type). This complexity exists not only in the data structure but also in the access to the data such as data traversal.

With the growing development of object-oriented database management systems more and more data owners are looking to convert their legacy data. However, much of this legacy data is not fully normalised, for reasons of performance, convenience of access or even from poor modelling in the first instance. This paper has shown that this lack of normalisation need not prevent the use of the step by step conversion methodologies proposed by earlier researchers.

5. Acknowledgements

Access™ is a trademark of the Microsoft Corporation
VisualWorks™ is a trademark of ObjectShare, Inc
6. References


6. Christian Fahner and Gottfried Vossen: Transforming Relational Database Schemas into Object-Oriented Schemas according to ODMG-93. Lecture Notes of Computer Science, 10(3), pp429-446, 1995


11. Timothy J. Waltz, David (Chi-Chung) Yen and Sooun Lee: "Object-Oriented database systems: an implementation plan." Industrial management & Data systems 95,6 95(6), pp8-17, 1995
Appendix B

Test plan for a normalised database.

<table>
<thead>
<tr>
<th>Test Number</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SELECT * FROM Floor_Sockets FULL OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch</td>
</tr>
<tr>
<td>1.1</td>
<td>SELECT * FROM Floor_Sockets FULL OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Tech_Primary &gt; 200 AND Floor_Sockets.Tech_Primary &lt; 400</td>
</tr>
<tr>
<td>1.2</td>
<td>SELECT * FROM Floor_Sockets FULL OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Floor_Socket = '2487'</td>
</tr>
<tr>
<td>2</td>
<td>SELECT * FROM Floor_Sockets LEFT OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch</td>
</tr>
<tr>
<td>2.1</td>
<td>SELECT * FROM Floor_Sockets LEFT OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Tech_Primary &gt; 200 AND Floor_Sockets.Tech_Primary &lt; 400</td>
</tr>
<tr>
<td>2.2</td>
<td>SELECT * FROM Floor_Sockets LEFT OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Floor_Socket = '2487'</td>
</tr>
<tr>
<td>3</td>
<td>SELECT * FROM Floor_Sockets RIGHT OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch</td>
</tr>
<tr>
<td>3.1</td>
<td>SELECT * FROM Floor_Sockets RIGHT OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Tech_Primary &gt; 200 AND Floor_Sockets.Tech_Primary &lt; 400</td>
</tr>
<tr>
<td>3.2</td>
<td>SELECT * FROM Floor_Sockets RIGHT OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Floor_Socket = '2487'</td>
</tr>
<tr>
<td>4</td>
<td>SELECT * FROM Floor_Sockets INNER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch</td>
</tr>
<tr>
<td>4.1</td>
<td>SELECT * FROM Floor_Sockets INNER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Tech_Primary &gt; 200 AND Floor_Sockets.Tech_Primary &lt; 400</td>
</tr>
<tr>
<td>4.2</td>
<td>SELECT * FROM Floor_Sockets INNER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Floor_Socket = '2487'</td>
</tr>
<tr>
<td>5</td>
<td>SELECT FROM BT_Patch FULL OUTER JOIN Line_Types ON BT_Patch.TN_Prefix = Line_Types.TN_Prefix WHERE Line_Types.TN_Prefix &gt; 100 AND Line_Types.TN_Prefix &lt; 200</td>
</tr>
<tr>
<td>5.1</td>
<td>SELECT FROM BT_Patch FULL OUTER JOIN Line_Types ON BT_Patch.TN_Prefix = Line_Types.TN_Prefix WHERE Line_Types.TN_Prefix &gt; 100 AND Line_Types.TN_Prefix &lt; 200</td>
</tr>
<tr>
<td>5.2</td>
<td>SELECT FROM BT_Patch FULL OUTER JOIN Line_Types ON BT_Patch.TN_Prefix = Line_Types.TN_Prefix WHERE BT_Patch.BT_Patch = 1412</td>
</tr>
<tr>
<td>6</td>
<td>SELECT FROM BT_Patch FULL OUTER JOIN Line_Types ON BT_Patch.TN_Prefix = Line_Types.TN_Prefix WHERE BT_Patch.BT_Patch = 1412</td>
</tr>
<tr>
<td>6.1</td>
<td>SELECT FROM BT_Patch LEFT OUTER JOIN Line_Types ON BT_Patch.TN_Prefix = Line_Types.TN_Prefix WHERE BT_Patch.BT_Patch = 1412</td>
</tr>
<tr>
<td>6.2</td>
<td>SELECT FROM BT_Patch LEFT OUTER JOIN Line_Types ON BT_Patch.TN_Prefix = Line_Types.TN_Prefix WHERE BT_Patch.BT_Patch = 1412</td>
</tr>
<tr>
<td>7</td>
<td>SELECT FROM BT_Patch RIGHT OUTER JOIN Line_Types ON BT_Patch.TN_Prefix = Line_Types.TN_Prefix WHERE BT_Patch.BT_Patch = 1412</td>
</tr>
<tr>
<td>7.1</td>
<td>SELECT FROM BT_Patch RIGHT OUTER JOIN Line_Types ON BT_Patch.TN_Prefix = Line_Types.TN_Prefix WHERE BT_Patch.BT_Patch = 1412</td>
</tr>
<tr>
<td>7.2</td>
<td>SELECT FROM BT_Patch RIGHT OUTER JOIN Line_Types ON BT_Patch.TN_Prefix = Line_Types.TN_Prefix WHERE BT_Patch.BT_Patch = 1412</td>
</tr>
<tr>
<td>8</td>
<td>SELECT FROM BT_Patch INNER JOIN Line_Types ON BT_Patch.TN_Prefix = Line_Types.TN_Prefix WHERE Line_Types.TN_Prefix &gt; 100 AND Line_Types.TN_Prefix &lt; 200</td>
</tr>
<tr>
<td>8.1</td>
<td>SELECT FROM BT_Patch INNER JOIN Line_Types ON BT_Patch.TN_Prefix = Line_Types.TN_Prefix WHERE Line_Types.TN_Prefix &gt; 100 AND Line_Types.TN_Prefix &lt; 200</td>
</tr>
</tbody>
</table>
Appendix B

8.2 SELECT * FROM BT_Patch INNER JOIN Line_Types ON BT_Patch.TN_Prefix = Line_Types.TN_Prefix WHERE BT_Patch.BT_Patch = 1412

9.1 SELECT * FROM Floor_Sockets , BT_Patch , Line_Types WHERE (BT_Patch.TN_Prefix = Line_Types.TN_Prefix (+)) AND (Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch (+))

9.2 SELECT * FROM Floor_Sockets , BT_Patch , Line_Types WHERE (BT_Patch.TN_Prefix = Line_Types.TN_Prefix (+)) AND (Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch (+)) AND Floor_Sockets.Floor_Patch = '2487'

10 SELECT * FROM Floor_Sockets , BT_Patch , Line_Types WHERE (BT_Patch.TN_Prefix = Line_Types.TN_Prefix (+)) AND (Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch (+)) AND Floor_Sockets.Floor_Patch = '2487'

11 SELECT * FROM Floor_Sockets , BT_Patch , Line_Types WHERE (BT_Patch.TN_Prefix = Line_Types.TN_Prefix (+)) AND (Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch (+)) AND Floor_Sockets.Floor_Patch = '2487'

12 SELECT * FROM Floor_Sockets , BT_Patch , Line_Types WHERE (BT_Patch.TN_Prefix = Line_Types.TN_Prefix (+)) AND (Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch (+)) AND Floor_Sockets.Floor_Patch = '2487'

13 UPDATE Floor_Sockets SET Owner = 'yyyyyyyyy' WHERE Floor_Sockets.Owner is not null AND Floor_Sockets.Owner IN (SELECT Floor_Sockets.Owner FROM Floor_Sockets FULL OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch)

13.1 UPDATE Floor_Sockets SET Owner = 'yyyyyyyyy' WHERE Floor_Sockets.Owner is not null AND Floor_Sockets.Owner IN (SELECT Floor_Sockets.Owner FROM Floor_Sockets FULL OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Floor_Patch = '2487')

13.2 UPDATE Floor_Sockets SET Owner = 'yyyyyyyyy' WHERE Floor_Sockets.Owner is not null AND Floor_Sockets.Owner IN (SELECT Floor_Sockets.Owner FROM Floor_Sockets FULL OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Floor_Patch = '2487')

14 UPDATE Floor_Sockets SET Owner = 'yyyyyyyyy' WHERE Floor_Sockets.Owner is not null AND Floor_Sockets.Owner IN (SELECT Floor_Sockets.Owner FROM Floor_Sockets FULL OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Floor_Patch = '2487')

15 UPDATE Floor_Sockets SET Owner = 'yyyyyyyyy' WHERE Floor_Sockets.Owner is not null AND Floor_Sockets.Owner IN (SELECT Floor_Sockets.Owner FROM Floor_Sockets FULL OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Floor_Patch = '2487')
### Appendix B

<table>
<thead>
<tr>
<th>16.1</th>
<th>UPDATE Floor_Sockets SET Owner = 'yyyyyyyyy' WHERE Floor_Sockets.Owner is not null AND Floor_Sockets.Owner IN (SELECT Floor_Sockets.Owner FROM Floor_Sockets RIGHT OUTER JOIN BT_Patch ON Floor_Sockets.BT_patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Tech_Primary &gt; 200 AND Floor_Sockets.Tech_Primary &lt; 400)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.2</td>
<td>UPDATE Floor_Sockets SET Owner = 'yyyyyyyyy' WHERE Floor_Sockets.Owner is not null AND Floor_Sockets.Owner IN (SELECT Floor_Sockets.Owner FROM Floor_Sockets INNER JOIN BT_Patch ON Floor_Sockets.BT_patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Tech_Primary &gt; 200 AND Floor_Sockets.Tech_Primary &lt; 400)</td>
</tr>
<tr>
<td>16.3</td>
<td>UPDATE Floor_Sockets SET Owner = 'yyyyyyyyy' WHERE Floor_Sockets.Owner is not null AND Floor_Sockets.Owner IN (SELECT Floor_Sockets.Owner FROM Floor_Sockets INNER JOIN BT_Patch ON Floor_Sockets.BT_patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Tech_Primary = 'yyyyyyyyyyyy')</td>
</tr>
</tbody>
</table>

**Line_Types:**

- **Line_Types.TN_Prefix** FROM BT_Patch FULL OUTER JOIN Line_Types ON BT_Patch.TN_Prefix = Line_Types.TN_Prefix WHERE Line_Types.TN_Prefix > 100 AND Line_Types.TN_Prefix < 200

**UPDATE Line_Types SET Digital = True WHERE Line_Types.TN_Prefix IN (SELECT Line_Types.TN_Prefix FROM BT_Patch FULL OUTER JOIN Line_Types ON BT_Patch.TN_Prefix = Line_Types.TN_Prefix WHERE Line_Types.TN_Prefix > 100 AND Line_Types.TN_Prefix < 200)
21.1  UPDATE Floor_Sockets SET Owner = 'yyyy yyyy yyyy' WHERE Floor_Sockets.Owner is not null AND Floor_Sockets.Owner IN (SELECT Floor_Sockets.Owner FROM Floor_Sockets , BT_Patch , Line_Types WHERE (BT_Patch.TN_Prefix = Line_Types.TN_Prefix (+) ) AND (Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch (+) ) AND Floor_Sockets.Tech_Primary > 200 AND Floor_Sockets.Tech_Primary < 400)

21.2  UPDATE Floor_Sockets SET Owner = 'yyyy yyyy yyyy' WHERE Floor_Sockets.Owner is not null AND Floor_Sockets.Owner IN (SELECT Floor_Sockets.Owner FROM Floor_Sockets , BT_Patch , Line_Types WHERE (BT_Patch.TN_Prefix = Line_Types.TN_Prefix (+) ) AND (Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch (+) ) AND Floor_Sockets.Floor_Socket.Floor_Socket = '2487' )

22  UPDATE Floor_Sockets SET Owner = 'yyyy yyyy yyyy' WHERE Floor_Sockets.Owner is not null AND Floor_Sockets.Owner IN (SELECT Floor_Sockets.Owner FROM Floor_Sockets , BT_Patch , Line_Types WHERE (BT_Patch.TN_Prefix = Line_Types.TN_Prefix (+) ) AND (Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch (+) )

23  UPDATE Floor_Sockets SET Owner = 'yyyy yyyy yyyy' WHERE Floor_Sockets.Owner is not null AND Floor_Sockets.Owner IN (SELECT Floor_Sockets.Owner FROM Floor_Sockets , BT_Patch , Line_Types WHERE (BT_Patch.TN_Prefix = Line_Types.TN_Prefix (+) ) AND (Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch (+) )

24  UPDATE Floor_Sockets SET Owner = 'yyyy yyyy yyyy' WHERE Floor_Sockets.Owner is not null AND Floor_Sockets.Owner IN (SELECT Floor_Sockets.Owner FROM Floor_Sockets , BT_Patch , Line_Types WHERE (BT_Patch.TN_Prefix = Line_Types.TN_Prefix (+) ) AND (Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch (+) )

25  INSERT INTO Floor_Sockets ( Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch ) SELECT Floor_Sockets.Tech_Primary, Floor_Sockets.Floor_Socket, Floor_Sockets.Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets WHERE (BT_Patch.BT_Patch = Floor_Sockets.Floor_Socket.Floor_Socket = '2487' )

25.1  INSERT INTO Floor_Sockets ( Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch ) SELECT Floor_Sockets.Tech_Primary, Floor_Sockets.Floor_Socket, Floor_Sockets.Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets WHERE (BT_Patch.BT_Patch = Floor_Sockets.Floor_Socket.Floor_Socket = '2487' )

25.2  INSERT INTO Floor_Sockets ( Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch ) SELECT Floor_Sockets.Tech_Primary, Floor_Sockets.Floor_Socket, Floor_Sockets.Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets WHERE (BT_Patch.BT_Patch = Floor_Sockets.Floor_Socket.Floor_Socket = '2487' )
### Appendix B

| 26 | INSERT INTO Floor_Sockets (Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch) SELECT Floor_Sockets.Tech_Primary, Floor_Sockets.Floor_Socket, Floor_Sockets.Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets FULL OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Floor_Socket = '2487'. |
| 26.1 | INSERT INTO Floor_Sockets (Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch) SELECT Floor_Sockets.Tech_Primary, Floor_Sockets.Floor_Socket, Floor_Sockets.Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets LEFT OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch |
| 26.2 | INSERT INTO Floor_Sockets (Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch) SELECT Floor_Sockets.Tech_Primary, Floor_Sockets.Floor_Socket, Floor_Sockets.Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets RIGHT OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Floor_Socket = '2487'. |
| 27 | INSERT INTO Floor_Sockets (Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch) SELECT Floor_Sockets.Tech_Primary, Floor_Sockets.Floor_Socket, Floor_Sockets.Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets LEFT OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch |
| 27.1 | INSERT INTO Floor_Sockets (Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch) SELECT Floor_Sockets.Tech_Primary, Floor_Sockets.Floor_Socket, Floor_Sockets.Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets RIGHT OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Floor_Socket = '2487'. |
| 27.2 | INSERT INTO Floor_Sockets (Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch) SELECT Floor_Sockets.Tech_Primary, Floor_Sockets.Floor_Socket, Floor_Sockets.Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets RIGHT OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Floor_Socket = '2487'. |
| 28 | INSERT INTO Floor_Sockets (Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch) SELECT Floor_Sockets.Tech_Primary, Floor_Sockets.Floor_Socket, Floor_Sockets.Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets INNER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch |
| 28.1 | INSERT INTO Floor_Sockets (Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch) SELECT Floor_Sockets.Tech_Primary, Floor_Sockets.Floor_Socket, Floor_Sockets.Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets INNER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Floor_Socket = '2487'. |
| 28.2 | INSERT INTO Floor_Sockets (Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch) SELECT Floor_Sockets.Tech_Primary, Floor_Sockets.Floor_Socket, Floor_Sockets.Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets INNER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Floor_Socket = '2487'. |
Appendix B

```
30
```

```
30.1
```

```
30.2
```

```
31
```

```
31.1
```

```
31.2
```

```
32
```

```
32.1
```

```
32.2
```

```
33
INSERT INTO Floor_Sockets (Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch) SELECT Floor_Sockets.Tech_Primary, Floor_Socket, Floor_Socket,Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets, BT_Patch, Line_Types WHERE (BT_Patch.TN_Prefix = Line_Types.TN_Prefix (+)) AND (Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch (+))
```

```
33.1
INSERT INTO Floor_Sockets (Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch) SELECT Floor_Sockets.Tech_Primary, Floor_Socket, Floor_Socket,Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets, BT_Patch, Line_Types WHERE (BT_Patch.TN_Prefix = Line_Types.TN_Prefix (+)) AND (Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch (+)) AND Floor_Sockets.Floor_Socket '2497'
```

```
33.2
INSERT INTO Floor_Sockets (Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch) SELECT Floor_Sockets.Tech_Primary, Floor_Socket, Floor_Socket,Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets, BT_Patch, Line_Types WHERE (BT_Patch.TN_Prefix = Line_Types.TN_Prefix (+)) AND (Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch (+)) AND Floor_Sockets.Floor_Socket '2497'
```

```
34
INSERT INTO Floor_Sockets (Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch) SELECT Floor_Sockets.Tech_Primary, Floor_Socket, Floor_Socket,Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets, BT_Patch, Line_Types WHERE (BT_Patch.TN_Prefix = Line_Types.TN_Prefix (+)) AND (Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch (+)) AND Floor_Sockets.Floor_Socket '2497'
```

```
34.1
INSERT INTO Floor_Sockets (Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch) SELECT Floor_Sockets.Tech_Primary, Floor_Socket, Floor_Sockets.Computer, Floor_Sockets.Owner,
```

188
Appendix B

34.2

INSERT INTO Floor_Sockets (Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch) SELECT Floor_Sockets.Tech_Primary * 2, Floor_Sockets.Floor_Socket, Floor_Sockets.Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets, BT_Patch, Line_Types WHERE (BT_Patch.TN_prefix = Line_Types.TN_prefix (+)) AND (Floor_Sockets.BT_Patch = BT_Patch.BT_Patch (+)) AND Floor_Sockets.Tech_Primary > 400 AND Floor_Sockets.BTech_Primary < 400

35

INSERT INTO Floor_Sockets (Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch) SELECT Floor_Sockets.Tech_Primary * 2, Floor_Sockets.Floor_Socket, Floor_Sockets.Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets, BT_Patch, Line_Types WHERE (BT_Patch.TN_prefix = Line_Types.TN_prefix (+)) AND (Floor_Sockets.BT_Patch = BT_Patch.BT_Patch (+)) AND Floor_Sockets.Tech_Primary > 400 AND Floor_Sockets.BTech_Primary < 400

35.1

INSERT INTO Floor_Sockets (Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch) SELECT Floor_Sockets.Tech_Primary * 2, Floor_Sockets.Floor_Socket, Floor_Sockets.Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets, BT_Patch, Line_Types WHERE (BT_Patch.TN_prefix = Line_Types.TN_prefix (+)) AND (Floor_Sockets.BT_Patch = BT_Patch.BT_Patch (+)) AND Floor_Sockets.Tech_Primary > 400 AND Floor_Sockets.BTech_Primary < 400

35.2

INSERT INTO Floor_Sockets (Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch) SELECT Floor_Sockets.Tech_Primary * 2, Floor_Sockets.Floor_Socket, Floor_Sockets.Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets, BT_Patch, Line_Types WHERE (BT_Patch.TN_prefix = Line_Types.TN_prefix (+)) AND (Floor_Sockets.BT_Patch = BT_Patch.BT_Patch (+)) AND Floor_Sockets.Tech_Primary > 400 AND Floor_Sockets.BTech_Primary < 400

36

INSERT INTO Floor_Sockets (Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch) SELECT Floor_Sockets.Tech_Primary * 2, Floor_Sockets.Floor_Socket, Floor_Sockets.Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets, BT_Patch, Line_Types WHERE (BT_Patch.TN_prefix = Line_Types.TN_prefix (+)) AND (Floor_Sockets.BT_Patch = BT_Patch.BT_Patch (+)) AND Floor_Sockets.Tech_Primary > 400 AND Floor_Sockets.BTech_Primary < 400

36.1

INSERT INTO Floor_Sockets (Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch) SELECT Floor_Sockets.Tech_Primary * 2, Floor_Sockets.Floor_Socket, Floor_Sockets.Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets, BT_Patch, Line_Types WHERE (BT_Patch.TN_prefix = Line_Types.TN_prefix (+)) AND (Floor_Sockets.BT_Patch = BT_Patch.BT_Patch (+)) AND Floor_Sockets.Tech_Primary > 400 AND Floor_Sockets.BTech_Primary < 400

36.2

INSERT INTO Floor_Sockets (Tech_Primary, Floor_Socket, Computer, Owner, BT_Patch) SELECT Floor_Sockets.Tech_Primary * 2, Floor_Sockets.Floor_Socket, Floor_Sockets.Computer, Floor_Sockets.Owner, Floor_Sockets.BT_Patch FROM Floor_Sockets, BT_Patch, Line_Types WHERE (BT_Patch.TN_prefix = Line_Types.TN_prefix (+)) AND (Floor_Sockets.BT_Patch = BT_Patch.BT_Patch (+)) AND Floor_Sockets.Tech_Primary > 400 AND Floor_Sockets.BTech_Primary < 400

37

DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN (DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary FROM Floor_Sockets WHERE Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch (+))

37.1

DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN (DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary FROM Floor_Sockets WHERE Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch (+) AND BT_Patch.TN_prefix = '247')

37.2

DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN (DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary FROM Floor_Sockets WHERE Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch (+) AND BT_Patch.TN_prefix = '2847')

38

DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN (DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary FROM Floor_Sockets WHERE Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch (+))

38.1

DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN (DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary FROM Floor_Sockets WHERE Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch AND BT_Patch.TN_prefix = '247')

38.2

DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN (DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary FROM Floor_Sockets WHERE Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch AND BT_Patch.TN_prefix = '2847')
<table>
<thead>
<tr>
<th>Line</th>
<th>SQL Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td><code>DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN ((SELECT Floor_Sockets.Tech_Primary FROM Floor_Sockets, BT_Patch WHERE Floor_Sockets.BT_Patch = BT_Patch.BT_Patch (+))</code></td>
</tr>
<tr>
<td>39.1</td>
<td><code>DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN ((SELECT Floor_Sockets.Tech_Primary FROM Floor_Sockets, BT_Patch WHERE Floor_Sockets.BT_Patch = BT_Patch.BT_Patch (+) AND BT_Patch.TN_Prefix = '247'))</code></td>
</tr>
<tr>
<td>39.2</td>
<td><code>DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN ((SELECT Floor_Sockets.Tech_Primary FROM Floor_Sockets, BT_Patch WHERE Floor_Sockets.BT_Patch = BT_Patch.BT_Patch (+) AND BT_Patch.TN_Prefix = '248'))</code></td>
</tr>
<tr>
<td>40</td>
<td><code>DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN ((SELECT Floor_Sockets.Tech_Primary FROM Floor_Sockets, BT_Patch WHERE Floor_Sockets.BT_Patch = BT_Patch.BT_Patch (+))</code></td>
</tr>
<tr>
<td>40.1</td>
<td><code>DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN ((SELECT Floor_Sockets.Tech_Primary FROM Floor_Sockets, BT_Patch WHERE Floor_Sockets.BT_Patch = BT_Patch.BT_Patch (+) AND BT_Patch.TN_Prefix = '247')</code></td>
</tr>
<tr>
<td>41</td>
<td><code>DELETE FROM BT_Patch WHERE BT_Patch.BT_Patch IN ((SELECT BT_Patch.BT_Patch FROM BT_Patch, Line_Types WHERE BT_Patch.TN_Prefix (+) = Line_Types.TN_Prefix (+) AND Line_Types.Digital = True))</code></td>
</tr>
<tr>
<td>41.1</td>
<td><code>DELETE FROM BT_Patch WHERE BT_Patch.BT_Patch IN ((SELECT BT_Patch.BT_Patch FROM BT_Patch, Line_Types WHERE BT_Patch.TN_Prefix (+) = Line_Types.TN_Prefix (+) AND Line_Types.Digital = True))</code></td>
</tr>
<tr>
<td>41.2</td>
<td><code>DELETE FROM BT_Patch WHERE BT_Patch.BT_Patch IN ((SELECT BT_Patch.BT_Patch FROM BT_Patch, Line_Types WHERE BT_Patch.TN_Prefix (+) = Line_Types.TN_Prefix (+) AND Line_Types.Digital = True))</code></td>
</tr>
<tr>
<td>42</td>
<td><code>DELETE FROM BT_Patch WHERE BT_Patch.BT_Patch IN ((SELECT BT_Patch.BT_Patch FROM BT_Patch, Line_Types WHERE BT_Patch.TN_Prefix (+) = Line_Types.TN_Prefix (+) AND Line_Types.Digital = True))</code></td>
</tr>
<tr>
<td>43</td>
<td><code>DELETE FROM BT_Patch WHERE BT_Patch.BT_Patch IN ((SELECT BT_Patch.BT_Patch FROM BT_Patch, Line_Types WHERE BT_Patch.TN_Prefix (+) = Line_Types.TN_Prefix (+) AND Line_Types.Digital = True))</code></td>
</tr>
<tr>
<td>44</td>
<td><code>DELETE FROM BT_Patch WHERE BT_Patch.BT_Patch IN ((SELECT BT_Patch.BT_Patch FROM BT_Patch, Line_Types WHERE BT_Patch.TN_Prefix (+) = Line_Types.TN_Prefix (+) AND Line_Types.Digital = True))</code></td>
</tr>
<tr>
<td>45</td>
<td><code>DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN ((SELECT Floor_Sockets.Tech_Primary FROM Floor_Sockets, BT_Patch, Line_Types WHERE Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch (+) AND BT_Patch.TN_Prefix = Line_Types.TN_Prefix (+))</code></td>
</tr>
<tr>
<td>45.1</td>
<td><code>DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN ((SELECT Floor_Sockets.Tech_Primary FROM Floor_Sockets, BT_Patch, Line_Types WHERE Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch (+) AND BT_Patch.TN_Prefix = Line_Types.TN_Prefix (+) AND Line_Types.Digital = True))</code></td>
</tr>
<tr>
<td>45.2</td>
<td><code>DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN ((SELECT Floor_Sockets.Tech_Primary FROM Floor_Sockets, BT_Patch, Line_Types WHERE Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch (+) AND BT_Patch.TN_Prefix = Line_Types.TN_Prefix (+) AND Floor_Sockets.Floor_Socket = '248')</code></td>
</tr>
<tr>
<td>46</td>
<td><code>DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN ((SELECT Floor_Sockets.Tech_Primary FROM Floor_Sockets, BT_Patch, Line_Types WHERE Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch (+) AND BT_Patch.TN_Prefix = Line_Types.TN_Prefix (+) AND Floor_Sockets.Floor_Socket = '249'))</code></td>
</tr>
</tbody>
</table>
Appendix B

| 46.1 | DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN (SELECT Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch AND BT_Patch.TN_Prefix (+) = Line_Types.TN_Prefix) |
| 46.2 | DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN (SELECT Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch AND BT_Patch.TN_Prefix (+) = Line_Types.TN_Prefix AND Floor_Sockets.Floor_Socket = '2487') |
| 47   | DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN (SELECT Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch AND BT_Patch.TN_Prefix (+) = Line_Types.TN_Prefix AND Floor_Sockets.Floor_Socket = '2487') |
| 47.1 | DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN (SELECT Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch AND BT_Patch.TN_Prefix (+) = Line_Types.TN_Prefix AND Line_Types.Digital = True) |
| 47.2 | DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN (SELECT Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch AND BT_Patch.TN_Prefix (+) = Line_Types.TN_Prefix AND Floor_Sockets.Floor_Socket = '2487') |
| 48   | DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN (SELECT Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch AND BT_Patch.TN_Prefix (+) = Line_Types.TN_Prefix) |
| 48.1 | DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN (SELECT Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch AND BT_Patch.TN_Prefix (+) = Line_Types.TN_Prefix AND Line_Types.Digital = True) |
| 48.2 | DELETE FROM Floor_Sockets WHERE Floor_Sockets.Tech_Primary IN (SELECT Floor_Sockets.BT_Patch (+) = BT_Patch.BT_Patch AND BT_Patch.TN_Prefix (+) = Line_Types.TN_Prefix AND Floor_Sockets.Floor_Socket = '2487') |
Appendix C

Test Plan for a De-normalised Database

<table>
<thead>
<tr>
<th>Test Number</th>
<th>SQL Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SELECT * FROM Allland2</td>
</tr>
<tr>
<td>1.1</td>
<td>SELECT * FROM Allland2 WHERE Allland2.Tech_Primary &gt; 200 AND Allland2.Tech_Primary &lt; 400</td>
</tr>
<tr>
<td>1.2</td>
<td>SELECT * FROM Allland2 WHERE Allland2.Floor_Socket = '2487'</td>
</tr>
<tr>
<td>2</td>
<td>SELECT * FROM Allland2 WHERE (Allland2.BT_Patch = Allland2.BT_Patch2 OR Allland2.Tech_Primary is not null)</td>
</tr>
<tr>
<td>2.1</td>
<td>SELECT * FROM Allland2 WHERE (Allland2.BT_Patch = Allland2.BT_Patch2 OR Allland2.Tech_Primary is not null) AND Allland2.Tech_Primary &gt; 200 AND Allland2.Tech_Primary &lt; 400</td>
</tr>
<tr>
<td>2.2</td>
<td>SELECT * FROM Allland2 WHERE (Allland2.BT_Patch = Allland2.BT_Patch2 OR Allland2.Tech_Primary is not null) AND Allland2.Floor_Socket = '2487'</td>
</tr>
<tr>
<td>3</td>
<td>SELECT * FROM Allland2 WHERE (Allland2.BT_Patch = Allland2.BT_Patch2 OR Allland2.BT_Patch2 is not null) AND Allland2.Tech_Primary &gt; 200 AND Allland2.Tech_Primary &lt; 400</td>
</tr>
<tr>
<td>3.1</td>
<td>SELECT * FROM Allland2 WHERE (Allland2.BT_Patch = Allland2.BT_Patch2 OR Allland2.BT_Patch2 is not null) AND Allland2.Floor_Sockets.Floor_Socket = '2487'</td>
</tr>
<tr>
<td>3.2</td>
<td>SELECT * FROM Floor_Sockets RIGHT OUTER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Floor_Socket = '2487'</td>
</tr>
<tr>
<td>4</td>
<td>SELECT * FROM Floor_Sockets INNER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch</td>
</tr>
<tr>
<td>4.1</td>
<td>SELECT * FROM Floor_Sockets INNER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Floor_Socket = '2487'</td>
</tr>
<tr>
<td>4.2</td>
<td>SELECT * FROM Floor_Sockets INNER JOIN BT_Patch ON Floor_Sockets.BT_Patch = BT_Patch.BT_Patch WHERE Floor_Sockets.Floor_Socket = '2487'</td>
</tr>
<tr>
<td>5</td>
<td>SELECT * FROM All2and3</td>
</tr>
<tr>
<td>5.1</td>
<td>SELECT * FROM All2and3 WHERE All2and3.TN_Prefix3 &gt; 100 AND All2and3.TN_Prefix3 &lt; 200</td>
</tr>
<tr>
<td>5.2</td>
<td>SELECT * FROM All2and3 WHERE All2and3.BT_Patch = 1412</td>
</tr>
<tr>
<td>6</td>
<td>SELECT * FROM All2and3 WHERE (All2and3.TN_Prefix = All2and3.TN_Prefix3 OR All2and3.BT_Patch is not null)</td>
</tr>
<tr>
<td>6.1</td>
<td>SELECT * FROM All2and3 WHERE (All2and3.TN_Prefix = All2and3.TN_Prefix3 OR All2and3.BT_Patch is not null) AND All2and3.TN_Prefix &gt; 100 AND All2and3.TN_Prefix &lt; 200</td>
</tr>
<tr>
<td>6.2</td>
<td>SELECT * FROM All2and3 WHERE (All2and3.TN_Prefix = All2and3.TN_Prefix3 OR All2and3.BT_Patch is not null) AND All2and3.BT_Patch = 1412</td>
</tr>
<tr>
<td>7</td>
<td>SELECT * FROM All2and3 WHERE (All2and3.TN_Prefix = All2and3.TN_Prefix3 OR All2and3.BT_Patch is not null)</td>
</tr>
<tr>
<td>7.1</td>
<td>SELECT * FROM All2and3 WHERE (All2and3.TN_Prefix = All2and3.TN_Prefix3 OR All2and3.BT_Patch is not null) AND All2and3.TN_Prefix &gt; 100 AND All2and3.TN_Prefix &lt; 200</td>
</tr>
<tr>
<td>7.2</td>
<td>SELECT * FROM All2and3 WHERE (All2and3.TN_Prefix = All2and3.TN_Prefix3 OR All2and3.BT_Patch is not null) AND All2and3.BT_Patch = 1412</td>
</tr>
<tr>
<td>8</td>
<td>SELECT * FROM All2and3 WHERE All2and3.TN_Prefix = All2and3.TN_Prefix3</td>
</tr>
<tr>
<td>8.1</td>
<td>SELECT * FROM All2and3 WHERE All2and3.TN_Prefix = All2and3.TN_Prefix3</td>
</tr>
<tr>
<td>8.2</td>
<td>SELECT * FROM All2and3 WHERE All2and3.TN_Prefix = All2and3.TN_Prefix3</td>
</tr>
<tr>
<td>9</td>
<td>SELECT * FROM Alldata WHERE NOT (Alldata.Tech_Primary IS NULL AND Alldata.BT_Patch2 IS NULL AND Alldata.TN_Prefix3 IS NOT NULL) AND Alldata.BT_Patch2 IS NULL AND Alldata.TN_Prefix3 IS NOT NULL</td>
</tr>
<tr>
<td>132</td>
<td>131</td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
<tr>
<td>UPDATE Alland2.OWNER = 'yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alland2.OWNER IN (SELECT Alland2.OWNER FROM Alland2 WHERE (Alland2.BT_Patch = Alland2.BT_Patch_2 OR Alland2.BT_Patch_2 IS NOT NULL) AND Alland2.TECH_Primary &gt; 200 AND Alland2.TECH_Primary &lt; 400)</td>
<td></td>
</tr>
</tbody>
</table>
Appendix C

16.1 \textbf{UPDATE} All1and2 \textbf{SET} Owner = 'yyyy yyyyyy' \textbf{WHERE} All1and2.Owner is not null AND All1and2.Owner IN (SELECT All1and2.Owner FROM All1and2 \textbf{WHERE} (All1and2.BT_Patch = All1and2.BT_Patch_2))

16.2 \textbf{UPDATE} Floor_Sockets \textbf{SET} Owner = 'yyyy yyyyyy' \textbf{WHERE} Floor_Sockets.Owner is not null AND Floor_Sockets.Owner IN (SELECT Floor_Sockets.Owner \textbf{FROM} Floor_Sockets \textbf{WHERE} Floor_Sockets.Floor_Socket = '2487')

17. \textbf{UPDATE} All12and3 \textbf{SET} Digital = True \textbf{WHERE} All12and3.TN_Prefix IN (SELECT All12and3.TN_Prefix \textbf{FROM} All12and3 \textbf{WHERE} All12and3.BT_Patch = 1412)

18. \textbf{UPDATE} All12and3 \textbf{SET} Digital = True \textbf{WHERE} All12and3.TN_Prefix = (SELECT All12and3.TN_Prefix \textbf{FROM} All12and3 \textbf{WHERE} All12and3.Tech_Primary > 200 AND All12and3.Tech_Primary < 400)

19. \textbf{UPDATE} All12and3 \textbf{SET} Digital = True \textbf{WHERE} All12and3.TN_Prefix > 100 AND All12and3.TN_Prefix = (SELECT All12and3.TN_Prefix \textbf{FROM} All12and3 \textbf{WHERE} All12and3.TN_Prefix > 100 AND All12and3.TN_Prefix = 1412)

20. \textbf{UPDATE} All12and3 \textbf{SET} Digital = True \textbf{WHERE} All12and3.TN_Prefix = (SELECT All12and3.TN_Prefix \textbf{FROM} All12and3 \textbf{WHERE} All12and3.Tech_Primary > 200 AND All12and3.Tech_Primary < 400)

21. \textbf{UPDATE} All1data \textbf{SET} Owner = 'yyyy yyyyyy' \textbf{WHERE} All1data.Owner is not null AND All1data.Owner IN (SELECT All1data.Owner \textbf{FROM} All1data \textbf{WHERE} All1data.BT_Patch = All1data.BT_Patch_2)

22. \textbf{UPDATE} All1data \textbf{SET} Owner = 'yyyy yyyyyy' \textbf{WHERE} All1data.Owner is not null AND All1data.Owner IN (SELECT All1data.Owner \textbf{FROM} All1data \textbf{WHERE} All1data.BT_Patch = All1data.BT_Patch_2 OR All1data.Tech_Primary IS NOT NULL AND All1data.Tech_Primary > 200)

194
### Appendix C

| 23   | UPDATE Floor_Sockets SET Owner = 'yyyy yyyy yyyy' WHERE Floor_Sockets.Owner is not null AND Floor_Sockets.Owner IN (SELECT Floor_Sockets.Owner FROM Floor_Sockets, BT_Patch, Line_Types WHERE (BT_Patch.TN_Prefix = Line_Types.TN_Prefix + Line_Types.TN_suffix) AND (Floor_Sockets.BT_patch = BT_Patch.BT_patch)) |
| 23.1 | UPDATE Floor_Sockets SET Owner = 'yyyy yyyy yyyy' WHERE Floor_Sockets.Owner is not null AND Floor_Sockets.Owner IN (SELECT Floor_Sockets.Owner FROM Floor_Sockets, BT_Patch, Line_Types WHERE (BT_Patch.TN_Prefix = Line_Types.TN_Prefix + Line_Types.TN_suffix) AND (Floor_Sockets.BT_patch = BT_Patch.BT_patch)) AND Floor_Sockets.Floor_Socket = '2487' |
| 23.2 | UPDATE Floor_Sockets SET Owner = 'yyyy yyyy yyyy' WHERE Floor_Sockets.Owner is not null AND Floor_Sockets.Owner IN (SELECT Floor_Sockets.Owner FROM Floor_Sockets, BT_Patch, Line_Types WHERE (BT_Patch.TN_Prefix = Line_Types.TN_Prefix + Line_Types.TN_suffix) AND (Floor_Sockets.BT_patch = BT_Patch.BT_patch)) AND Floor_Sockets.Floor_Socket = '2487' |
| 24   | UPDATE Alldata SET Owner = 'yyyy yyyy yyyy yyyy yyyy yyyy yyyy' WHERE Alldata.Owner is not null AND Alldata.Owner IN (SELECT Alldata.Owner FROM Alldata WHERE (Alldata.Tech_Primary = Alldata.BT_patch_2 OR Alldata.Tech_Primary IS NOT NULL) AND NOT (Alldata.Tech_Primary IS NULL AND Alldata.BT_patch_2 IS NULL AND Alldata.TN_Prefix_3 IS NOT NULL) AND Alldata.Floor_Socket = '2487') |
BT_Patch, BT_Patch_2, TN_Prefix, TN_Suffix, Telephone, Fax_modem


27

27.1


27.2


28


28.1


28.2


29


29.1


29.2


30


30.1
### Appendix C

<table>
<thead>
<tr>
<th>Table Number</th>
<th>SQL Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.2</td>
<td><code>INSERT INTO All2and3 (BT_Patch, TN_Prefix, TN_Suffix, Telephone, Fax_modem, TN_Prefix_3, Digital) SELECT All2and3.BT_Patch, All2and3.TN_Prefix, All2and3.TN_Suffix, All2and3.Telephone, All2and3.Fax_modem, All2and3.TN_Prefix_3, All2and3.Digital FROM All2and3 WHERE All2and3.TN_Prefix = All2and3.TN_Prefix_3 OR All2and3.BT_Patch &gt; 100 AND All2and3.TN_Prefix &lt; 200</code></td>
</tr>
<tr>
<td>31</td>
<td><code>INSERT INTO All2and3 (BT_Patch, TN_Prefix, TN_Suffix, Telephone, Fax_modem, TN_Prefix_3, Digital) SELECT All2and3.BT_Patch, All2and3.TN_Prefix, All2and3.TN_Suffix, All2and3.Telephone, All2and3.Fax_modem, All2and3.TN_Prefix_3, All2and3.Digital FROM All2and3 WHERE All2and3.TN_Prefix = All2and3.TN_Prefix_3 OR All2and3.BT_Patch = 1412</code></td>
</tr>
<tr>
<td>31.1</td>
<td><code>INSERT INTO All2and3 (BT_Patch, TN_Prefix, TN_Suffix, Telephone, Fax_modem, TN_Prefix_3, Digital) SELECT All2and3.BT_Patch, All2and3.TN_Prefix, All2and3.TN_Suffix, All2and3.Telephone, All2and3.Fax_modem, All2and3.TN_Prefix_3, All2and3.Digital FROM All2and3 WHERE All2and3.TN_Prefix = All2and3.TN_Prefix_3 OR All2and3.BT_Patch = 1412</code></td>
</tr>
<tr>
<td>31.2</td>
<td><code>INSERT INTO All2and3 (BT_Patch, TN_Prefix, TN_Suffix, Telephone, Fax_modem, TN_Prefix_3, Digital) SELECT All2and3.BT_Patch, All2and3.TN_Prefix, All2and3.TN_Suffix, All2and3.Telephone, All2and3.Fax_modem, All2and3.TN_Prefix_3, All2and3.Digital FROM All2and3 WHERE All2and3.TN_Prefix = All2and3.TN_Prefix_3 OR All2and3.BT_Patch = 1412</code></td>
</tr>
<tr>
<td>32</td>
<td><code>INSERT INTO All2and3 (BT_Patch, TN_Prefix, TN_Suffix, Telephone, Fax_modem, TN_Prefix_3, Digital) SELECT All2and3.BT_Patch, All2and3.TN_Prefix, All2and3.TN_Suffix, All2and3.Telephone, All2and3.Fax_modem, All2and3.TN_Prefix_3, All2and3.Digital FROM All2and3 WHERE All2and3.TN_Prefix = All2and3.TN_Prefix_3</code></td>
</tr>
<tr>
<td>32.1</td>
<td><code>INSERT INTO All2and3 (BT_Patch, TN_Prefix, TN_Suffix, Telephone, Fax_modem, TN_Prefix_3, Digital) SELECT All2and3.BT_Patch, All2and3.TN_Prefix, All2and3.TN_Suffix, All2and3.Telephone, All2and3.Fax_modem, All2and3.TN_Prefix_3, All2and3.Digital FROM All2and3 WHERE All2and3.TN_Prefix = All2and3.TN_Prefix_3</code></td>
</tr>
</tbody>
</table>
### Appendix C

Appendix C


36.2


37

UPDATE Alldata SET Tech_Primary is null, Floor_Socket is null, Owner is null, Comments is null, Computer is null, BT_Patch is null

Then run:
DELETE FROM Alldata WHERE Alldata.Tech_Primary is null AND Alldata.BT_Patch_2 is null

37.1

UPDATE Alldata SET Tech_Primary is null, Floor_Socket is null, Owner is null, Comments is null, Computer is null, BT_Patch is null WHERE Alldata.TN_Prefix = '247'.

Then run:
DELETE FROM Alldata WHERE Alldata.Tech_Primary is null AND Alldata.BT_Patch_2 is null

37.2

UPDATE Alldata SET Tech_Primary is null, Floor_Socket is null, Owner is null, Comments is null, Computer is null, BT_Patch is null WHERE Alldata.BT_Patch_2 = '2847'.

Then run:
DELETE FROM Alldata WHERE Alldata.Tech_Primary is null AND Alldata.BT_Patch_2 is null

38

UPDATE Alldata SET Tech_Primary is null, Floor_Socket is null, Owner is null, Comments is null, Computer is null, BT_Patch is null WHERE Alldata.BT_Patch_2 OR Alldata.Tech_Primary is not null

Then run:
DELETE FROM Alldata WHERE Alldata.Tech_Primary is null AND Alldata.BT_Patch_2 is null

38.1

UPDATE Alldata SET Tech_Primary is null, Floor_Socket is null, Owner is null, Comments is null, Computer is null, BT_Patch is null WHERE Alldata.BT_Patch_2 OR Alldata.Tech_Primary is not null AND Alldata.TN_Prefix = '247'.

Then run:
DELETE FROM Alldata WHERE Alldata.Tech_Primary is null AND Alldata.BT_Patch_2 is null

38.2

UPDATE Alldata SET Tech_Primary is null, Floor_Socket is null, Owner is null, Comments is null, Computer is null, BT_Patch is null WHERE Alldata.BT_Patch_2 OR Alldata.Tech_Primary is not null AND Alldata.BT_Patch_2 = '2847'.

Then run:
DELETE FROM Alldata WHERE Alldata.Tech_Primary is null AND Alldata.BT_Patch_2 is null

39

UPDATE Alldata SET Tech_Primary is null, Floor_Socket is null, Owner is null, Comments is null, Computer is null, BT_Patch is null WHERE Alldata.BT_Patch_2 OR Alldata.BT_Patch_2 is not null

Then run:
DELETE FROM Alldata WHERE Alldata.Tech_Primary is null AND Alldata.BT_Patch_2 is null

39.1

UPDATE Alldata SET Tech_Primary is null, Floor_Socket is null, Owner is null, Comments is null, Computer is null, BT_Patch is null WHERE Alldata.BT_Patch_2 OR Alldata.BT_Patch_2 is not null AND Alldata.TN_Prefix = '247'.

199
Then run:
DELETE FROM Alland2 WHERE Alland2.Tech_Primary is null AND Alland2.BT_Patch_2 is null

39.2

UPDATE Alland2 SET Tech_Primary = Null, Floor_Socket = Null, Owner = Null, Comments = Null, Computer = Null, BT_Patch = Null WHERE Alland2.BT_Patch = '2847'

Then run:
DELETE FROM Alland2 WHERE Alland2.Tech_Primary is null AND Alland2.BT_Patch_2 is null

40

UPDATE Alland2 SET Tech_Primary = Null, Floor_Socket = Null, Owner = Null, Comments = Null, Computer = Null, BT_Patch = Null WHERE Alland2.Tech_Primary is not null AND Alland2.BT_Patch_2 is not null

Then run:
DELETE FROM Alland2 WHERE Alland2.Tech_Primary is null AND Alland2.BT_Patch_2 is null

40.1

UPDATE Alland2 SET Tech_Primary = Null, Floor_Socket = Null, Owner = Null, Comments = Null, Computer = Null, BT_Patch = Null WHERE Alland2.Tech_Primary is not null AND Alland2.BT_Patch_2 is not null AND Alland2.TN_Prefix = '247'

Then run:
DELETE FROM Alland2 WHERE Alland2.Tech_Primary is null AND Alland2.BT_Patch_2 is null

40.2

UPDATE Alland2 SET Tech_Primary = Null, Floor_Socket = Null, Owner = Null, Comments = Null, Computer = Null, BT_Patch = Null WHERE Alland2.Tech_Primary is not null AND Alland2.BT_Patch_2 is not null AND Alland2.BT_Patch_2 = '2847'

Then run:
DELETE FROM Alland2 WHERE Alland2.Tech_Primary is null AND Alland2.BT_Patch_2 is null

41

UPDATE All2and3 SET BT_Patch = Null, Telephone = Null, Fax_modem = Null, TN_Prefix =Null, TN_Suffix = Null

Then run:
DELETE FROM All2and3 WHERE All2and3.BT_Patch is null AND All2and3.TN_Prefix_3 is null

41.1

UPDATE All2and3 SET BT_Patch = Null, Telephone = Null, Fax_modem = Null, TN_Prefix =Null, TN_Suffix = Null WHERE All2and3.Digital = True

Then run:
DELETE FROM All2and3 WHERE All2and3.BT_Patch is null AND All2and3.TN_Prefix_3 is null

41.2

UPDATE All2and3 SET BT_Patch = Null, Telephone = Null, Fax_modem = Null, TN_Prefix =Null, TN_Suffix = Null WHERE All2and3.TN_Prefix = All2and3.TN_Prefix_3 OR All2and3.BT_Patch is not null AND All2and3.Digital = True

Then run:
DELETE FROM All2and3 WHERE All2and3.BT_Patch is null AND All2and3.TN_Prefix_3 is null

42

UPDATE All2and3 SET BT_Patch = Null, Telephone = Null, Fax_modem = Null, TN_Prefix =Null, TN_Suffix = Null WHERE All2and3.TN_Prefix = All2and3.TN_Prefix_3 OR All2and3.BT_Patch is not null AND All2and3.Digital = True

Then run:
DELETE FROM All2and3 WHERE All2and3.BT_Patch is null AND All2and3.TN_Prefix_3 is null
Appendix C

42.2 UPDATE All2and3 SET BT_Patch = Null, Telephone = Null, Fax_modem = Null, TN_Prefix = Null, TN_Suffix = Null WHERE All2and3.TN_Prefix = All2and3.TN_Prefix_3 OR All2and3.BT_Patch is not null AND All2and3.TN_Prefix_3 = 247

Then run:
DELETE FROM All2and3 WHERE All2and3.BT_Patch is null AND All2and3.TN_Prefix_3 is null

43 UPDATE All2and3 SET BT_Patch = Null, Telephone = Null, Fax_modem = Null, TN_Prefix = null, TN_Suffix = null WHERE All2and3.TN_Prefix = All2and3.TN_Prefix_3 OR All2and3.TN_Prefix_3 is not null

Then run:
DELETE FROM All2and3 WHERE All2and3.BT_Patch is null AND All2and3.TN_Prefix_3 is null

43.1 UPDATE All2and3 SET BT_Patch = Null, Telephone = Null, Fax_modem = Null, TN_Prefix = null, TN_Suffix = null WHERE All2and3.TN_Prefix = All2and3.TN_Prefix_3 OR All2and3.TN_prefix_3 is not null AND All2and3.Digital = True

Then run:
DELETE FROM All2and3 WHERE All2and3.BT_Patch is null AND All2and3.TN_prefix_3 is null

43.2 UPDATE All2and3 SET BT_Patch = Null, Telephone = Null, Fax_modem = Null, TN_Prefix = null, TN_Suffix = null WHERE All2and3.BT_Patch is not null AND All2and3.TN_prefix_3 is not null AND All2and3.TN_Prefix_3 = 247

Then run:
DELETE FROM All2and3 WHERE All2and3.BT_Patch is null AND All2and3.TN_Prefix_3 is null

44 UPDATE All2and3 SET BT_Patch = Null, Telephone = Null, Fax_modem = Null, TN_Prefix = null, TN_Suffix = null WHERE All2and3.BT_Patch is not null AND All2and3.TN_prefix_3 is not null

Then run:
DELETE FROM All2and3 WHERE All2and3.BT_Patch is null AND All2and3.TN_prefix_3 is null

44.1 UPDATE All2and3 SET BT_Patch = Null, Telephone = Null, Fax_modem = Null, TN_Prefix = null, TN_Suffix = null WHERE All2and3.BT_Patch is not null AND All2and3.TN_prefix_3 is not null AND All2and3.Digital = True

Then run:
DELETE FROM All2and3 WHERE All2and3.BT_Patch is null AND All2and3.TN_prefix_3 is null

44.2 UPDATE All2and3 SET BT_Patch = Null, Telephone = Null, Fax_modem = Null, TN_Prefix = null, TN_Suffix = null WHERE All2and3.BT_Patch is not null AND All2and3.TN_prefix_3 is not null AND All2and3.TN_prefix_3 = 247

Then run:
DELETE FROM All2and3 WHERE All2and3.BT_Patch is null AND All2and3.TN_prefix_3 is null

45 UPDATE Alldata SET Tech_Primary = Null, Floor_Socket = Null, Owner = Null, Comments = Null, Computer = Null, BT_Patch = Null WHERE NOT (Alldata.Tech_Primary is null AND Alldata.BT_Patch_2 is null AND Alldata.TN_prefix_3 is not null)

Then run:
DELETE FROM Alldata WHERE Alldata.Tech_Primary is null AND Alldata.BT_Patch_2 is null AND Alldata.TN_prefix_3 is null

45.1 UPDATE Alldata SET Tech_Primary = Null, Floor_Socket = Null, Owner = Null, Comments = Null, Computer = Null, BT_Patch = Null WHERE NOT (Alldata.Tech_Primary is null AND Alldata.BT_Patch_2 is null AND Alldata.TN_prefix_3 is not null) AND Alldata.Digital = True

Then run:
DELETE FROM Alldata WHERE Alldata.Tech_Primary is null AND Alldata.BT_Patch_2 is null AND Alldata.TN_prefix_3 is null

45.2 UPDATE Alldata SET Tech_Primary = Null, Floor_Socket = Null, Owner = Null, Comments = Null, Computer = Null, BT_Patch = Null WHERE NOT (Alldata.Tech_Primary is null AND Alldata.BT_Patch_2 is null AND Alldata.TN_prefix_3 is null) AND Alldata.Digital = True

Then run:
DELETE FROM Alldata WHERE Alldata.Tech_Primary is null AND Alldata.BT_Patch_2 is null AND Alldata.TN_prefix_3 is null

201
Comments = Null, Computer = Null, BT_Patch = Null WHERE NOT (Alldata.Tech_Primary is null AND Alldata.BT_Patch_2 is null AND Alldata.TN_Prefix_3 is not null) AND Alldata.TN_Prefix_3 = 247

Then run:
DELETE FROM Alldata WHERE Alldata.Tech_Primary is null AND Alldata.BT_Patch_2 is null AND Alldata.TN_Prefix_3 is null


Then run:
DELETE FROM Alldata WHERE Alldata.Tech_Primary is null AND Alldata.BT_Patch_2 is null AND Alldata.TN_Prefix_3 is null


Then run:
DELETE FROM Alldata WHERE Alldata.Tech_Primary is null AND Alldata.BT_Patch_2 is null AND Alldata.TN_Prefix_3 is null


Then run:
DELETE FROM Alldata WHERE Alldata.Tech_Primary is null AND Alldata.BT_Patch_2 is null AND Alldata.TN_Prefix_3 is null

Then run:
DELETE FROM Alldata WHERE Alldata.Tech_Primary is null AND Alldata.BT_Patch_2 is null AND Alldata.TN_Prefix_3 is null

48.1

Then run:
DELETE FROM Alldata WHERE Alldata.Tech_Primary is null AND Alldata.BT_Patch_2 is null AND Alldata.TN_Prefix_3 is null

48.2
UPDATE Alldata SET Tech_Primary = Null, Floor_Socket = Null, Owner = Null, Comments = Null, Computer = Null, BT_Patch = Null WHERE (Alldata.BT_Patch = Alldata.BT_Patch_2) AND NOT (Alldata.Tech_Primary IS NULL AND Alldata.BT_Patch_2 IS NULL AND Alldata.TN_Prefix_3 IS NOT NULL) AND Alldata.Floor_Socket = '2487'

Then run:
DELETE FROM Alldata WHERE Alldata.Tech_Primary is null AND Alldata.BT_Patch_2 is null AND Alldata.TN_Prefix_3 is null

203
Appendix D

Test Log results for the tests given in Appendix B and Appendix C.

<table>
<thead>
<tr>
<th>Test Number</th>
<th>Normalised Execution only</th>
<th>De-normalised HTML Table generation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time is in seconds</td>
<td>Time is in seconds</td>
</tr>
<tr>
<td>1</td>
<td>15.031</td>
<td>0.040</td>
</tr>
<tr>
<td>1.1</td>
<td>1.558</td>
<td>0.028</td>
</tr>
<tr>
<td>1.2</td>
<td>1.399</td>
<td>0.114</td>
</tr>
<tr>
<td>2</td>
<td>6.440</td>
<td>0.040</td>
</tr>
<tr>
<td>2.1</td>
<td>1.596</td>
<td>0.030</td>
</tr>
<tr>
<td>2.2</td>
<td>1.397</td>
<td>0.117</td>
</tr>
<tr>
<td>3</td>
<td>2.495</td>
<td>0.030</td>
</tr>
<tr>
<td>3.1</td>
<td>1.506</td>
<td>0.032</td>
</tr>
<tr>
<td>3.2</td>
<td>1.398</td>
<td>0.116</td>
</tr>
<tr>
<td>4</td>
<td>2.828</td>
<td>0.029</td>
</tr>
<tr>
<td>4.1</td>
<td>1.439</td>
<td>0.025</td>
</tr>
<tr>
<td>4.2</td>
<td>1.387</td>
<td>0.115</td>
</tr>
<tr>
<td>5</td>
<td>0.847</td>
<td>0.033</td>
</tr>
<tr>
<td>5.1</td>
<td>0.726</td>
<td>0.077</td>
</tr>
<tr>
<td>5.2</td>
<td>0.315</td>
<td>0.093</td>
</tr>
<tr>
<td>6</td>
<td>0.817</td>
<td>0.032</td>
</tr>
<tr>
<td>6.1</td>
<td>0.696</td>
<td>0.074</td>
</tr>
<tr>
<td>6.2</td>
<td>0.347</td>
<td>0.108</td>
</tr>
<tr>
<td>7</td>
<td>0.812</td>
<td>0.036</td>
</tr>
<tr>
<td>7.1</td>
<td>0.719</td>
<td>0.077</td>
</tr>
<tr>
<td>7.2</td>
<td>0.341</td>
<td>0.096</td>
</tr>
<tr>
<td>8</td>
<td>1.185</td>
<td>0.034</td>
</tr>
<tr>
<td>8.1</td>
<td>0.428</td>
<td>0.035</td>
</tr>
<tr>
<td>8.2</td>
<td>0.304</td>
<td>0.110</td>
</tr>
<tr>
<td>9</td>
<td>35.839</td>
<td>0.078</td>
</tr>
<tr>
<td>9.1</td>
<td>1.696</td>
<td>0.035</td>
</tr>
<tr>
<td>9.2</td>
<td>1.674</td>
<td>0.142</td>
</tr>
<tr>
<td>10</td>
<td>13.038</td>
<td>0.065</td>
</tr>
<tr>
<td>10.1</td>
<td>1.688</td>
<td>0.034</td>
</tr>
<tr>
<td>10.2</td>
<td>1.518</td>
<td>0.147</td>
</tr>
<tr>
<td>11</td>
<td>3.393</td>
<td>0.032</td>
</tr>
<tr>
<td>11.1</td>
<td>1.558</td>
<td>0.031</td>
</tr>
<tr>
<td>11.2</td>
<td>1.552</td>
<td>0.149</td>
</tr>
<tr>
<td>12</td>
<td>3.107</td>
<td>0.032</td>
</tr>
<tr>
<td>12.1</td>
<td>1.571</td>
<td>0.036</td>
</tr>
</tbody>
</table>
## Appendix D

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12.2</td>
<td>1.523</td>
<td>0.146</td>
<td>1.527</td>
<td>1.175</td>
</tr>
<tr>
<td>13</td>
<td>5.113</td>
<td>5.962</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.1</td>
<td>2.967</td>
<td>2.904</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.2</td>
<td>3.206</td>
<td>3.062</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>4.935</td>
<td>5.431</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.1</td>
<td>3.001</td>
<td>3.095</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.2</td>
<td>3.204</td>
<td>2.963</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>4.987</td>
<td>4.204</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.1</td>
<td>2.993</td>
<td>3.130</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.2</td>
<td>3.354</td>
<td>2.895</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>5.065</td>
<td>3.773</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16.1</td>
<td>3.036</td>
<td>3.029</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16.2</td>
<td>3.228</td>
<td>3.016</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>0.802</td>
<td>1.746</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17.1</td>
<td>0.376</td>
<td>0.748</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17.2</td>
<td>0.307</td>
<td>0.415</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>0.814</td>
<td>1.598</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18.1</td>
<td>0.384</td>
<td>0.710</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18.2</td>
<td>0.302</td>
<td>0.496</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>0.784</td>
<td>1.685</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19.1</td>
<td>0.390</td>
<td>0.689</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19.2</td>
<td>0.322</td>
<td>0.416</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0.796</td>
<td>1.595</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20.1</td>
<td>0.367</td>
<td>1.581</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20.2</td>
<td>0.299</td>
<td>0.412</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>5.490</td>
<td>6.015</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21.1</td>
<td>3.623</td>
<td>3.252</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21.2</td>
<td>3.256</td>
<td>3.010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>5.632</td>
<td>5.433</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22.1</td>
<td>3.639</td>
<td>3.216</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22.2</td>
<td>3.450</td>
<td>3.127</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>5.518</td>
<td>4.200</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23.1</td>
<td>3.435</td>
<td>3.137</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23.2</td>
<td>3.555</td>
<td>3.098</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>5.683</td>
<td>3.768</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24.1</td>
<td>3.433</td>
<td>3.028</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24.2</td>
<td>3.387</td>
<td>3.063</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>7.647</td>
<td>30.906</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25.1</td>
<td>1.624</td>
<td>1.260</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25.2</td>
<td>1.413</td>
<td>1.180</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>6.344</td>
<td>5.833</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26.1</td>
<td>1.588</td>
<td>1.264</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-----</td>
<td>-----</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26.2</td>
<td>1.346</td>
<td>1.217</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>3.749</td>
<td>3.441</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27.1</td>
<td>1.480</td>
<td>1.232</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27.2</td>
<td>1.364</td>
<td>1.214</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>3.658</td>
<td>2.614</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28.1</td>
<td>1.047</td>
<td>1.262</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28.2</td>
<td>1.361</td>
<td>1.205</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>2.234</td>
<td>2.113</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29.1</td>
<td>0.922</td>
<td>0.641</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29.2</td>
<td>0.318</td>
<td>0.300</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>2.230</td>
<td>2.106</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30.1</td>
<td>0.924</td>
<td>0.702</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30.2</td>
<td>0.319</td>
<td>0.327</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>2.187</td>
<td>2.076</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31.1</td>
<td>0.970</td>
<td>0.723</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31.2</td>
<td>0.305</td>
<td>0.345</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>2.550</td>
<td>2.125</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32.1</td>
<td>0.689</td>
<td>2.137</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32.2</td>
<td>0.304</td>
<td>0.337</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>8.873</td>
<td>30.918</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33.1</td>
<td>1.772</td>
<td>1.252</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33.2</td>
<td>1.435</td>
<td>1.067</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>7.819</td>
<td>6.053</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34.1</td>
<td>1.745</td>
<td>1.204</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34.2</td>
<td>1.401</td>
<td>1.045</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>4.743</td>
<td>3.462</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35.1</td>
<td>1.564</td>
<td>1.160</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35.2</td>
<td>1.449</td>
<td>1.047</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>3.694</td>
<td>2.648</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36.1</td>
<td>1.524</td>
<td>1.056</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36.2</td>
<td>1.407</td>
<td>1.056</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>3.786</td>
<td>9.594</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37.1</td>
<td>2.273</td>
<td>1.120</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37.2</td>
<td>0.763</td>
<td>1.166</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>3.663</td>
<td>9.566</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38.1</td>
<td>2.295</td>
<td>2.330</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38.2</td>
<td>0.825</td>
<td>4.939</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>3.794</td>
<td>2.387</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39.1</td>
<td>1.549</td>
<td>2.325</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39.2</td>
<td>0.804</td>
<td>2.241</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>3.718</td>
<td>4.719</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40.1</td>
<td>2.319</td>
<td>1.095</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40.2</td>
<td>0.918</td>
<td>1.100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>2.646</td>
<td>1.694</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41.1</td>
<td>1.513</td>
<td>0.820</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41.2</td>
<td>0.485</td>
<td>0.288</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>2.586</td>
<td>1.883</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42.1</td>
<td>1.516</td>
<td>1.703</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42.2</td>
<td>0.395</td>
<td>7.403</td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>2.758</td>
<td>1.379</td>
<td></td>
<td></td>
</tr>
<tr>
<td>43.1</td>
<td>1.494</td>
<td>1.774</td>
<td></td>
<td></td>
</tr>
<tr>
<td>43.2</td>
<td>0.519</td>
<td>1.794</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>2.549</td>
<td>2.078</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44.1</td>
<td>1.536</td>
<td>0.910</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44.2</td>
<td>0.472</td>
<td>0.302</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>4.476</td>
<td>7.284</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45.1</td>
<td>3.818</td>
<td>1.789</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45.2</td>
<td>1.571</td>
<td>1.245</td>
<td></td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>4.570</td>
<td>10.052</td>
<td></td>
<td></td>
</tr>
<tr>
<td>46.1</td>
<td>3.579</td>
<td>1.840</td>
<td></td>
<td></td>
</tr>
<tr>
<td>46.2</td>
<td>1.641</td>
<td>1.367</td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>4.439</td>
<td>2.699</td>
<td></td>
<td></td>
</tr>
<tr>
<td>47.1</td>
<td>3.549</td>
<td>1.871</td>
<td></td>
<td></td>
</tr>
<tr>
<td>47.2</td>
<td>1.577</td>
<td>1.437</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>4.455</td>
<td>2.504</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48.1</td>
<td>3.586</td>
<td>1.835</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48.2</td>
<td>1.633</td>
<td>1.412</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Statistical Analysis And Results

The experiment intends to use de-normalisation queries to improve performance of a database system. To test that hypothesis, the time in seconds for retrievals is obtained, both for normalised and de-normalised queries. The data are displayed in the second and third column below:

<table>
<thead>
<tr>
<th>Test</th>
<th>Normalised (x1)</th>
<th>De-normalised (x2)</th>
<th>d = x1 - x2</th>
<th>Pd square</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15.031</td>
<td>0.040</td>
<td>14.991</td>
<td>224.730081</td>
</tr>
<tr>
<td>2</td>
<td>1.558</td>
<td>0.028</td>
<td>1.530</td>
<td>2.3409</td>
</tr>
<tr>
<td>3</td>
<td>1.389</td>
<td>0.114</td>
<td>1.275</td>
<td>1.625625</td>
</tr>
<tr>
<td>4</td>
<td>6.440</td>
<td>0.040</td>
<td>6.400</td>
<td>40.96</td>
</tr>
<tr>
<td>5</td>
<td>1.596</td>
<td>0.030</td>
<td>1.566</td>
<td>2.452356</td>
</tr>
<tr>
<td>6</td>
<td>1.397</td>
<td>0.117</td>
<td>1.280</td>
<td>1.6384</td>
</tr>
<tr>
<td>7</td>
<td>2.495</td>
<td>0.030</td>
<td>2.465</td>
<td>6.076225</td>
</tr>
<tr>
<td>8</td>
<td>1.506</td>
<td>0.032</td>
<td>1.474</td>
<td>2.172676</td>
</tr>
<tr>
<td>9</td>
<td>1.398</td>
<td>0.116</td>
<td>1.282</td>
<td>1.643524</td>
</tr>
<tr>
<td>10</td>
<td>2.828</td>
<td>0.029</td>
<td>2.799</td>
<td>7.834401</td>
</tr>
<tr>
<td>11</td>
<td>1.439</td>
<td>0.025</td>
<td>1.414</td>
<td>1.999396</td>
</tr>
<tr>
<td>12</td>
<td>1.387</td>
<td>0.115</td>
<td>1.272</td>
<td>1.617984</td>
</tr>
<tr>
<td>13</td>
<td>0.847</td>
<td>0.033</td>
<td>0.814</td>
<td>0.662596</td>
</tr>
<tr>
<td>14</td>
<td>0.726</td>
<td>0.077</td>
<td>0.649</td>
<td>0.421201</td>
</tr>
<tr>
<td>15</td>
<td>0.315</td>
<td>0.093</td>
<td>0.222</td>
<td>0.049284</td>
</tr>
<tr>
<td>16</td>
<td>0.817</td>
<td>0.032</td>
<td>0.785</td>
<td>0.816225</td>
</tr>
<tr>
<td>17</td>
<td>0.696</td>
<td>0.074</td>
<td>0.622</td>
<td>0.396884</td>
</tr>
<tr>
<td>18</td>
<td>0.347</td>
<td>0.108</td>
<td>0.239</td>
<td>0.057121</td>
</tr>
<tr>
<td>19</td>
<td>0.812</td>
<td>0.036</td>
<td>0.776</td>
<td>0.602176</td>
</tr>
<tr>
<td>20</td>
<td>0.719</td>
<td>0.077</td>
<td>0.642</td>
<td>0.412164</td>
</tr>
<tr>
<td>21</td>
<td>0.341</td>
<td>0.098</td>
<td>0.245</td>
<td>0.060025</td>
</tr>
<tr>
<td>22</td>
<td>1.185</td>
<td>0.034</td>
<td>1.151</td>
<td>1.324801</td>
</tr>
<tr>
<td>23</td>
<td>0.428</td>
<td>0.035</td>
<td>0.393</td>
<td>0.154449</td>
</tr>
<tr>
<td>24</td>
<td>0.304</td>
<td>0.110</td>
<td>0.194</td>
<td>0.037636</td>
</tr>
<tr>
<td>25</td>
<td>35.839</td>
<td>0.078</td>
<td>35.761</td>
<td>1278.849121</td>
</tr>
<tr>
<td>26</td>
<td>1.696</td>
<td>0.035</td>
<td>1.661</td>
<td>2.758921</td>
</tr>
<tr>
<td>27</td>
<td>1.674</td>
<td>0.142</td>
<td>1.532</td>
<td>2.347024</td>
</tr>
<tr>
<td>28</td>
<td>13.038</td>
<td>0.065</td>
<td>12.973</td>
<td>168.298729</td>
</tr>
</tbody>
</table>

\[ \text{Sum } d^2 = 96.407 \]
\[ \text{mean } = 3.4431071 \]
\[ \text{Sum Square } = 9294.310 \]
\[ \text{Sum of } (d \text{ square}) = 1752.1299 \]
\[ \text{Critical value t 0.05} = 1.703 \]
\[ \text{Sd } = 7.25 \]
\[ t = 2.516 \]
\[ \text{Confidence interval } = 2.329 \]
The fourth column shows the difference between normalised and de-normalised retrieval time $d = x_1 - x_2$. To see if the data provide evidence to conclude that, on the average, the de-normalisation improves performance by reducing the retrieval time? A hypothesis test is performed at the 5% significance level.

To state the null and alternative hypothesis let $\mu_1$ denote the mean performance of all queries when de-normalisation is applied and $\mu_2$ denotes the mean performance of all queries when de-normalisation is not applied then the null and alternative hypothesis are:

$$H_0 : \mu_1 = \mu_2 \text{ (mean performance with de-normalisation not greater)}$$

$$H_a : \mu_1 > \mu_2 \text{ (mean performance with de-normalisation is greater)}$$

The hypothesis test is right-tailed since there is a greater than sign in the alternative hypothesis. The critical value for a right-tailed test $t_\alpha$ with $df = n-1 = 28 - 1 = 27$. thus $\alpha = 0.05$ and $t_\alpha = 1.703$ (Weiss 1993). All results shown in the above table are obtained from the following formulas:

$$D = \frac{\text{Sum}(d)}{n} = 3.443 \quad \text{..........................................................(1)}$$

$$S_d = \sqrt{\frac{n(\text{Sum}(d^2)-(\text{Sum}(d))^2)/(n*(n-1))}{\text{Sum}(d^2)/n^2)}} = 7.25 \quad \text{.........(2)}$$

$$T = \frac{d}{(S_d/sqr(n))} = 2.516 \quad \text{..........................................................(3)}$$

If the value of test statistic falls in the rejection region, reject $H_0$, otherwise do not reject $H_0$. From the above statistic we conclude that the test value of 2.516 falls in the rejection region, hence we reject $H_0$. 

209
To calculate the confidence interval for $\mu_1 - \mu_2$ the following formula is used:

$$D (+or-) (t\alpha * Sd/sqr(n))$$ ............................... (4)

Thus at 90% confidence interval for $\mu_1 - \mu_2$ is from 1.1 to 5.8 seconds. Between the mean performance of all queries when de-normalisation is used and the mean performance of all queries when de-normalisation is not used is somewhere between 1.1 and 5.8 seconds. In particular we can be 90% confident that, on the average de-normalisation reduces the retrieval time by at least 1.1 seconds for select queries.
This paper was presented at the Object Oriented Information Systems (OOIS 2000) conference in London, December 2000.

Using object-oriented databases to preserve integrity and improve performance - but at what cost?

Abdulaziz Al-kandari and Ray Dawson

Dept. of Computer Science, Loughborough University, Loughborough, Leics. LE11 3TU, UK
E-mail: A.AlKandari@Lboro.ac.uk, R.J.Dawson@Lboro.ac.uk Fax: +44 1509 211586

ABSTRACT. Two alternatives to using object-oriented databases to improve performance of complex relational databases are, firstly de-normalisation which improves performance but at a cost of additional errors and/or code to be maintained, and secondly, obtaining a bigger and faster computer though this may not give the same improvement. This paper identifies a need to properly cost the reported performance advantages of object-oriented databases and to compare these costs with alternative performance improvement methods.

KEY WORDS: Object-oriented, database, integrity, cost.

1. Improving performance of relational databases

The innovation of normalisation concepts in database systems helped to achieve such desirable features as safety, consistency and maintainability (Delobel, 1995). The DBMS guarantees reliable data definitions and manipulations when data are properly modelled. In the case of a relational database, the data must be well normalised to attain reliability. However, the normalised model for complex data usually suffer from many side effects concerning performance and response time. For this reason many working databases are often not fully normalised (Elmasri, 1994).
A problem area for normalised databases concerns many to many relationships, where an additional table (intersection entity) is required for each relationship to ensure consistency, and reduce maintenance cost but the number of table joins for data retrieval increases performance cost (Douglas, 1996).

![Diagram](Appendix E)

**Figure 1. Steps that a RDBMS application developer may take to improve performance.**

RDBMS performance improvements may be achieved through “de-normalisation” which is moving away from intersection entities (Douglas, 1996). In other words the system returns back to its improper model phase which then requires extra code to prevent any anomalies which might occur. As a result, the database system may face higher maintenance cost and would be vulnerable to errors. However, the authors determined from informal enquiries to a number of companies that companies were prepared to accept the extra development and error risk to achieve a gain performance. (See figure 1).
Another method for improving performance of a complex normalised database system is to convert it to an object-oriented system. Douglas (1996) has shown that an object-oriented database can run 7 to 100 times faster than a relational database. The advantage of this method is that proper data modelling is preserved as well as improving the overall performance. This leads to further advantages of fewer errors, easier extension and reduced maintenance costs. This is the opposite of the effect of de-normalisation.

Published papers on methodologies for converting relational to object-oriented databases assume the relational database is based on a proper, normalised data model. However, many working databases in the real world are not normalised for performance reasons or for reasons of convenience or simply because the developers did not fully understand how to normalise the database in the first place! However, this need not prevent direct conversion to an object-oriented database because, as previously reported by the authors (Al-Kandari, 1999), the conversion process will itself informally normalise the data.

2. The impact of performance on the cost of a system.

Normally, the better quality the merchandise usually means the greater the cost or value. For database systems the cost savings from improving the performance has to be measured against the cost incurred as a result of the improvement to determine the best quality solution. These improvement costs for a de-normalised system would include the cost of writing and maintaining the extra code to prevent anomalies plus the cost of additional errors incurred. For a move to an object-orientated database the costs could include the database management system, the cost of converting the data plus possible costs to train employees in object-oriented methods. Costs need to be determined over time as the de-normalisation method may give short term gains but the object-oriented system’s greater integrity and resulting reliability is likely to give longer term gains. (See figure 2.)

This research concentrates on comparing the performance of a normalised database system with complex data before and after any changes applied to it and evaluating the associated costs. The aim is to verify that a conversion to an object-oriented database gives greater cost benefit in the long term and, if possible, to determine the break-even point where it becomes the least expensive option.
The research methodology will be a mixture of performance benchmark tests and case studies. A database system will be compared in a fully normalised relational format, a de-normalised relational format and an object-oriented format. The data will be based on the style and format of real databases at a large engineering company in Derby, UK. Benchmark tests will be made using a number of database management systems. For example, non-normalised data systems will be compared with normalised systems using available databases such as Microsoft Access and MySQL. Relational data modelling will be compared with object-oriented data modelling using Java programs with systems such as POET. This benchmark data will be compared with other published data where such data is available such as provided in the report by STR (STR 1997).

![Diagram of Cost Comparison for Performance Improvement Alternatives]

Figure 2: Cost comparison for performance improvement alternatives

Both paths in figure 2 would improve the performance but the question is which will give the greater cost savings. The authors will examine case study data to provide the costs of maintaining de-normalised and object-oriented databases. Data will come from employee timesheet records for employees involved in creating and maintaining the systems. Other company data that will be analysed will be the fault reports for the database systems. The faults attributable to integrity errors resulting from table de-normalisation will be identified and the associated costs to correct the errors will be calculated. Some data will be provided by the company in Derby, the authors will then seek to verify the findings with data provided from other sources.
Finally a third, and perhaps even more significant alternative method of improving performance needs to be considered. Could the same level of improvement be achieved simply by getting a bigger and faster computer? As one industrial employee reported to us, "Once I had to make changes to 3 forms to add various triggers to get the update/insert things to work properly after the affected tables were de-normalised. The performance gain was a response below 3 seconds compared to 15 seconds. It took about 3 days of coding on the forms (working 8-hour each day!). If the database was on a slightly better machine (more RAM, disk, etc.) I wonder if there would have been any need to do the de-normalisation." Clearly this alternate must be costed and compared with the de-normalised and object-oriented database methods of improving performance.

3. Conclusion

This paper has identified two alternatives to moving to an object-oriented databases to improve performance. De-normalisation will also improve performance but at a cost of increasing the number of errors and/or increasing the code to be maintained. The second, simpler alternative is to obtain a bigger and faster computer but it is uncertain whether this can give the same improvement or cost benefits. This identifies a need to properly cost the reported advantages of speed and low maintenance of object-oriented databases and to compare these costs with the alternative methods of improving performance.

The paper outlines the methodology to be undertaken to establish the costs of creating, operating, and maintaining databases in normalised and non-normalised relational systems and also in object-oriented systems. This data will be taken from company records of employee time allocation and error reports. Some data is available from an engineering company in Derby, but the authors would like to verify that the results obtained are generally applicable. The authors are, therefore, looking for other volunteer companies to assist them in this research.

References

Appendix E


Appendix F

A Methodology to Assess the Cost of Database Performance Improvement

Abdulaziz Alkandari and Ray Dawson
Loughborough University, Loughborough, UK

Abstract

This paper shows several basic improvement methods to increase database performance. The cost of converting a relational database to an object-oriented database is compared with the cost of de-normalising complex data. A methodology is proposed for comparing these costs with the costs of other improvement methods such as hardware upgrade.

Sample data from a number of organisations in Kuwait shows that care must be taken in choosing a method of improving performance as the improvement costs may outweigh the benefits gained. The methodology described could greatly help in giving hints in choosing a certain model for each and every specific case depending on the performance desired and the approximate budget of the company.

1. Introduction

This paper considers database systems used in different Information Technology departments of governmental sectors and scientific research institutes in the State of Kuwait. The intention is to constitute a methodology to calculate and compare the cost of different improvement methods against the ongoing maintenance cost of the current and the improved database system.

For database systems, the cost savings from improving the performance has to be measured against the cost incurred as a result of the improvement to determine the best quality solution. Let us consider an existing running cost per month \(CM_{\text{exist}}\) of a database system. Let us consider the cost of improving performance is \(C_{\text{imp}}\). Let us consider the new running cost per month \(C_{\text{new}}\) of the database system. The ultimate goal is to get a better performance with lower running cost \(C_{\text{new}}\).

There is a need for a methodology to discover the gains against \(C_{\text{imp}}\) and also the difference in the new running cost and the old running cost after time \(T\).

\[ C_{\text{imp}} + CM_{\text{new}} \times T \leq CM_{\text{exist}} \times T \quad \text{...... Inequality 1} \]

This requires the break even point to be achieved in a reasonable time. The time to achieve the break even point, \(T_{\text{BEP}}\).

\[ T_{\text{BEP}} = \frac{C_{\text{imp}}}{(CM_{\text{exist}} - CM_{\text{new}})} \quad \text{...... Equation 1} \]

\(C_{\text{imp}}\) (cost of improvement) in Inequality 1 is considered part of the short-term costs and is an important factor in determining the change in the database system. \(CM_{\text{exist}}\) and \(CM_{\text{new}}\) are the long-term costs.

\(C_{\text{imp}}\) could be any of the following improvement methods:
- Conversion from legacy database system to an Object-Oriented Database System (OODB).
Appendix F

• De-normalisation of tables used in complex data queries for performance improvement.
• Use of other system, built-in improvement techniques such indexing and disk partitioning for performance improvement.
• Hardware upgrading for performance improvement.

2. Methodology for Determining Costs
The methodology has been split into four steps:

**Step 1 - Determine the cost of de-normalisation**
• Consider the costs of the de-normalisation process which will include:
  o The cost of writing the extra code to prevent anomalies.
  o The cost of maintaining the extra code to prevent anomalies.
  o The cost of slower performance of data insertion, deletion and update.
  o The cost of additional errors occurring in the data because of the anomalies missed.
• Consider the benefits of de-normalisation which will include:
  o Reduced employee time to use the system due to the increased performance of data retrieval.
• The end result will produce the overall cost of implementing and operating a relational database system with a de-normalised data structure.

**Step 2 – Determine the cost of conversion to an object-oriented database**
• Consider the costs to implement an object-oriented conversion process. Which will include:
  o The cost of the object-oriented database management system.
  o The cost of employee time to convert the data to the object oriented model.
  o The possible costs to train employees in object-oriented methods.
• Consider the benefits of conversion to the object-oriented which will include:
  o A saving of employee time in using the database due to the increased performance.
  o A possible reduction in maintenance costs if the system is modified or extended
• The end result will produce the overall cost of conversion to and operating an object-oriented database.

**Step 3 – Compare the costs from step 1 and step 2.**
Costs need to be determined over time as the de-normalisation method may give short term gains but the object-oriented system’s greater integrity and resulting reliability is likely to give longer term gains.

**Step 4 – Determine if a simple hardware upgrade would be as cost effective**
See if processor and disk storage hardware upgrades are available that will give the increase in performance required and if so, determine the cost of the upgrade and compare the costs of de-normalisation and conversion to an object-oriented system.

3. Cost Considerations
The predictions of the future costs depend on the data structure of the system, for a complex data structure we should be concerned with the following:

218
Appendix F

Short-term costs of conversion to object-oriented versus the on going cost of maintenance as Table 1 shows.

<table>
<thead>
<tr>
<th>Cost of external training agency</th>
<th>Conversion</th>
<th>De-normalisation cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost of employee time to train</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Cost of time for employees to overcome difficult cases</td>
<td>Yes</td>
<td>(Need of expertise sometimes)</td>
</tr>
</tbody>
</table>

Table 1. Short term costs of conversion versus on going cost of maintenance

Long-term savings of conversion versus the on going cost of maintenance as Table 2 shows.

<table>
<thead>
<tr>
<th>Extendibility</th>
<th>Conversion</th>
<th>De-normalisation cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reusability</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Less code</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Stability (proper model)</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 1 Long-term conversion savings versus on going maintenance costs.

There are many different costs that must be taken in consideration when trying to improve performance which were learned from the case studies such as:

- Cost of conversion to object oriented.
- Training staff for organisations that do not have OO experience.
- Cost of de-normalisation.
- Cost of data anomaly errors resulting from the improper modelling as in de-normalisation.
- Cost of hardware.
- Cost of using built-in performance improvement techniques.
Sometimes improving a database system using a specific method is compulsory e.g. the department of agricultural economics and animal science, Michigan State University [1] carried out an experiment of this advantage for the process of storing information on cows. To further extend the relational database used, the application employed had to save photographic pictures of calves for breeding purposes. The advantage of object-oriented databases in this situation cannot be gained using other alternatives such as de-normalisation of a normal relational database or replacing the current hardware with a more powerful one. It is the need to store various data types such as photographic pictures that gives a good reason to put aside the limited data type of the relational data model. The only way out for relational database is to use Binary Large Objects (BLOB) then we should calculate the cost of this improvement against conversion to object-oriented database.

4. Systems Studied in the State of Kuwait

A number of IT departments were studied in the State of Kuwait in governmental sectors and scientific research institutes. It had been hoped to gain information of the different costs incurred for a change from relational databases to object oriented databases, for de-normalisation and for hardware upgrading from a single organisation. However, this did not prove to be possible. Nevertheless some useful data was gained from each of the organisations studied which together gave a useful overall picture of the costs that may be incurred.

Zaket House is a charity organisation that uses Oracle8. It had not implemented any object oriented database development, but it had embarked on training their programmers so that they could use the object oriented tools in Oracle8. Ten programmers each attended a course supplied by an external agency. Three programmers then proceeded to an advanced course. These three programmers would then train the other programmers. The total cost for the external courses plus the time taken to attend the courses and train each other was £71,600.

The Ministry of Defense also used Oracle 8. Their systems showed that some database queries were used as frequently as 60 times an hour. The time taken for such a query was 14 seconds, but this was reduced to 2 seconds by using the indexing facilities of Oracle 8. At an assumed rate of operator salary of £40,000 this gives a cost of waiting for such queries as £9,200/year before indexing was used for each of these queries. The Ministry also used a facility within Oracle 8 for physical partitioning the tables across different discs, though logically it appears as one continuous table. This facility also gave savings as good or better than that achieved through indexing.

The Kuwait Institute for Scientific Research (KISR) had performed de-normalisation of complex queries. This they recorded as taking 320 hours for one query to perform the de-normalisation and to write the extra code to ensure the integrity of the system. This had a cost of £7,300. No record was kept of any additional errors in the system resulting from the improper modelling, but it is likely that the time spent writing the additional code prevented any significant increase in errors due to the de-normalisation. KISR also reported that hardware upgrades for one of their systems, where the functionality had remained unchanged, gave a reduction in time for a query from 5 hours in 1991 to 4 mins in 1995 and less than one second in 2000. This is an average 3 fold increase in speed every year.

There were no records in the increased performance due to de-normalisation in any organisation. However, benchmark tests carried out at Loughborough University showed that for complex queries involving table joins, an increase in speed of 10 to 100 times is possible [2].

There was also no record of the time taken to convert a relational system to an object oriented database system. This was because no organisation could be found that had carried out this improvement method in Kuwait. However, it could be assumed that it would take at least as much effort as de-normalisation. So the time of 320 hours is likely to be a lower bound for the possible time taken.

5. An Illustration of the Application of the Cost Methodology
Appendix F

To illustrate the application of the cost methodology it is useful to apply the costing to an imaginary organisation that has characteristics similar to those found at the Kuwait organisations studied. Let us assume the organization is using Oracle8 or Oracle9 already. This means no extra licensing costs will be incurred when introducing object orientation. It also means they have the capability of upgrading to an object oriented model but can be selective as to where this upgrade is applied.

The greatest benefits of object-orientation is gained when the data is complex in nature, so let us assume the organisation is only interested in a performance gain where queries on complex data are regularly performed.

Let us assume the number of development staff in the organisation is the same as that for Zaket House and that a similar training strategy is applied. Let us also assume that the complex data queries are performed on average 60 times an hour as at the Ministry of Defence, and also as at the Ministry of Defence each query takes 14 seconds before any improvements are applied. Let us assume that the de-normalisation of these complex queries would take 320 working hours as at the Kuwait Institute for Scientific Research. This effort includes that to produce additional code to preserve the integrity of the database and, because this code has been applied, the resulting number of database errors is not significantly greater than before the de-normalisation. We will assume that development staff cost the company £50,000/year and the database operator staff cost the company £40,000/year. Table 3 shows the comparison between short term improvement costs and long term improvement benefit rate for different improvement methods used in the case studies.

According to Barry Douglas [3] the improvement in performance using object orientation will normally be in the range of 7 to 100 times increase in speed. This means the cost of waiting for the query would be reduced to between £1,314 and £92 each year. Because we can be selective as to where the object-orientated model is applied, let us assume the performance benefits will be towards the end of the range giving the best performance gain. Eg. Assume the cost of the query would be £200/year giving a saving of £9,000/year on the query using the relational model.

The results of the benchmark testing show that de-normalisation can give similar improvement in performance for complex queries as that which can be achieved by conversion to the object oriented model. So let us assume the saving would also reduce the waiting time for a query to between £1,314 and £92 each year. Taking £700 as being in the middle of the range, this gives a saving of £8,500/year on the query using the relational model.

These calculations allow us to calculate the break-even points to determine how long it takes before de-normalisation or conversion to object orientation will start to give an overall saving.

<table>
<thead>
<tr>
<th>Method</th>
<th>Short Term Cost</th>
<th>Long Term Benefit Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running each complex query per year</td>
<td>--</td>
<td>£9,200/year</td>
</tr>
<tr>
<td>De-normalising a single query</td>
<td>£7,300</td>
<td>£8,500/year</td>
</tr>
<tr>
<td>Converting part of the database to object oriented</td>
<td>+ £7,300</td>
<td>£9,000/year</td>
</tr>
<tr>
<td>Training object oriented methods</td>
<td>£71,600</td>
<td>--</td>
</tr>
</tbody>
</table>

Table 3 Comparisons between Short Term Costs and Long Term Benefit Rates
For de-normalisation the initial cost of £7,300 is recovered at £8,500/year. This gives a break-even point of between 10 and 11 months.

For conversion to the object-oriented model the break even point depends on how many queries are subject to conversion. For a single query the costs would be £71,600 for training and £7,300 + for conversion, recovered at £9,000/year. This gives a break-even point would not be until at least 8 years and 9 months. However, a single query conversion is unlikely. If 10 such queries were subject to conversion to the object oriented model the break-even point would be reduced to something over 1 year and 4 months.

However, both the de-normalisation and the object oriented model conversion calculations assume no other improvements had been made. The Ministry of Defense reduced the query time to 2 seconds simply by introducing indexing. This gave a saving of £8,000/year with negligible implementation costs. It would take of the order of 15 years for de-normalisation to become less expensive than indexing and object oriented model conversion nearly 9 years to become less expensive than indexing. Clearly this makes indexing by far the best alternative.

Could the option of de-normalisation or object-orientation be used in addition to indexing? Certainly this may be possible but it is unlikely that de-normalisation or object orientation could achieve a further 7 to 100 times performance improvement if indexing had previously been introduced. However, for the sake of comparison let us assume a 100-fold increase in addition to that achieved through indexing.

Each query takes only 2 seconds after indexing is introduced giving a total waiting time cost of just over £1,300/year. The 100-fold improvement would reduce the waiting to negligible time and so give a saving of £1,300/year. This would give a break even point for de-normalisation of between 5 and 6 years and for object oriented model conversion for 10 queries a break even point of about 11 years!

This shows that if indexing is available as an option then neither de-normalisation, nor conversion to the object-oriented model is a worthwhile method of improving performance for an organisation such as the one considered here. Other improvement methods may be possible. The Ministry of Defense obtained a 27 fold increase in performance using table partitioning on the disk storage, this again would make the application of de-normilisation or object orientation not cost effective.

Furthermore, the improvement in performance has to be compared with the improvement that would naturally occur through upgrading the hardware. The experience of KISR showed that every two to three years the improved speed of hardware upgrades would give performance gains comparable with that of de-normalisation or object orientation. This will significantly reduce any long-term performance benefits of de-normalisation or conversion to object orientation.

Other organisations may have different characteristics that would make de-normalisation or conversion to object orientation more cost-effective options. This paper has shown the importance of examining the costs of each option carefully if the performance cost is an issue causing concern.

There are, of course, other reasons to convert a database to the object-oriented model. Object orientation, it is claimed, gives a system that is easier to change and extend, and Alkandari [4] showed that conversion to the object-oriented model can correct the errors of a badly modelled relational database. There may also be political reasons for converting to the object oriented model if, for example, it made the data compatible with that in other parts of the organisation. Each of these reasons must be considered on their own merit but this paper has shown that if the motivation is solely to improve performance then the organisation should proceed with caution and should carry out the cost comparison methodology described.

6. Conclusion

This paper has shown many basic improvement methods in different environments. In some cases there is no need to convert to an object-oriented system or de-normalise data, while in other cases one of the two choices could cost less with an increased performance.
Appendix F

We can conclude that the methodology described could greatly help in giving hints in choosing a certain model for each and every specific case depending on the performance desired and the approximate budget of the company.

References


