QUERY OPTIMIZATION IN DATABASE SYSTEMS

A THESIS

Submitted by

TEJY K K

in partial fulfillment for the award of the degree

of

DOCTOR OF PHILOSOPHY

Department of Computer Applications
FACULTY OF COMPUTER APPLICATIONS

Dr. M.G.R.
Educational and Research Institute
University
(Declared as Deemed to be University u/s.3 of UGC Act 1956)
Maduravoyal, Chennai -600 095
(An ISO 9001: 2008 Certified Institution)

JUNE 2016
DECLARATION BY THE CANDIDATE

I declare that the thesis entitled “QUERY OPTIMIZATION IN DATABASE SYSTEMS” submitted by me for the degree of Doctor of Philosophy is a bonafide record of work carried out by me during the period from February 2009 to February 2016 under the guidance of Dr. S. K. SRIVATSA and has not formed the basis for the award of any degree, diploma, associate-ship, fellowship, titles in this or any other University or other similar institutions of higher learning and without any plagiarism.

I have also published my papers in International Journals (Scopus rated) as per list of publications in the Annexure.

[Signature of the Research Scholar]
BONAFIDE CERTIFICATE

Certified that the thesis entitled “QUERY OPTIMIZATION IN DATABASE SYSTEMS” is the bonafide work of Mrs. TEJY K K (Reg. No. CA09D001), who had carried out the research under my supervision without any plagiarism to the best of my knowledge. Certified further, that to the best of my knowledge, the work reported herein does not form part of any other thesis or dissertation on the basis of which a degree or diploma was conferred on an earlier occasion on this or any other scholar.

Signature of the Supervisor

Dr. S. K. SRIVATSA
Retired Professor
Anna University, Chennai.
ABSTRACT

Now a day's, query optimization has become a popular topic in database research. The main interest in this area of research arises because of the new trends in the usage of databases. Initially, databases were primarily used for transaction oriented processing of operative data. Today, databases are also used to facilitate reporting and analysis on consolidated, historic data. Thus, the significance of database systems is increasing day by day. This significance results in complexity in data queries as well as increasing demand of efficiency in query processing.

Query processing is actually a process of translating a query written in a high-level language into low-level data manipulation operations. Query processing is mainly concerned with execution of query or it refers to the activities involved in extracting data from a data warehouse. In query processing, one of the most critical and important step is query optimization. Query optimization refers to the process of producing an optimal execution plan for a given query, where optimality is with respect to a cost function to be minimized. The objective of query optimization is to provide minimum response time and maximum throughput. Query optimization plays a vital role in tuning overall performance of the database systems.

The great commercial success of database systems is partly due to the development of sophisticated query optimization technology, where users pose queries in a declarative way using SQL and the optimizer of the database system finds a good way to execute these queries. Query optimizer is an important system component of database system. It is the responsibility of this component to translate the user submitted query usually written in a non-procedural language into an efficient query evaluation plan which is then executed against the database. Thus, the performance of a query is critically dependent upon the ability of the query optimizer in selecting the most efficient access plan. The selection of efficient access plan is done based on the estimated cost of competing access plans. These costs are in turn based on the estimates of intermediate result size. Several techniques have been proposed in the literature to estimate query result size. Some of the techniques are statistics, histograms, sampling
and parametric techniques. Any error in the result size estimates increases the number of joins. Thus, the main operation of query optimizer includes transforming queries, estimating and generating plans. These operations have also been discussed in this thesis. The components of query optimizer such as search space, search strategy and cost model has also been discussed in this thesis.

Many related research works have been carried out in this field of query optimization. The literature describes a wide variety of optimization algorithms, techniques, methods and strategies for optimizing a query efficiently. In spite of all these techniques and methods, optimizing a query accurately is not possible. It is only because we did not focus on the information which would help us to determine the optimum method or technique best suited for the types of database we use. To determine this, we must first understand the process of query optimization and the various aspects of optimization occurring at different levels. In this thesis, a detailed investigation has been done in this area.

This study facilitates the enhancement of query performance by determining the best optimum strategy. It also describes how to build a query optimizer model that incorporates the modularity necessary for the new generation of database management systems. In this thesis, we have also done a comparative study of some popular types of optimizers such as system R optimizer, cascades optimizer, volcano optimizer, exodus optimizer and starburst optimizer. This comparative analysis helps us to understand the strength and weakness of all these optimizers. This study also helps us to determine the role of various components of optimizer in query optimization.

In this thesis, we have also analyzed the factor that plays a vital role in the enhancement of query performance. The various factors that we have analyzed are optimizer, query equivalence rules, indexes, cost estimates and dependencies implied by SQL expression. The importance and their role in query optimization have also been discussed. The main aim of this thesis is to give guidance in constructing a query optimizer that is capable of optimizing large queries in a distributed setting and capable of producing evaluation plan with the shortest execution time or response time.

In this thesis, we have proposed a query tree based method which generates dependency rules to perform query optimization. The proposed method generates the
query tree using bottom-up approach. Each object from the query is identified first and the relational objects are identified. The identified baseline attributes are placed at the leaf level and then from the input query a distinct part of query is identified. For each distinct part of the query we identify the objects or attributes of each sub query and constructed as a tree. From generated query tree, we generate dependency rules and based on generated rules we generate set of sequence of rules to be processed. For each sequence identified, we compute the query completion time and based on completion time a least processing sequence will be selected as the most efficient one. This method overcomes the problem of missing cases and helps to achieve efficiency in query performance.

This method works well for the relational database in which dependencies exist among the attributes of the table. It is not able to produce efficient results when there exist dependency among the attributes of different tables. To improve the obtained solution by this method, we propose a multi level relational mapping algorithm which helps to identify the query dependency. The use of relational map helps to identify the objects and the entity of the query. In this method, we compute the dependency measure for each of the rules being generated which helps to schedule the execution of query parts efficiently. This method results in efficient query performance and also reduces the time complexity involved in query execution. This efficiency is shown by experimental study of various complex queries.

This thesis also highlights the importance and the role played by the optimizer, index selection, equivalence rules and search strategy in the enhancement of query performance. Apart from these, we have also suggested some best practices that have to be considered in reducing the execution cost of the query. We have also discussed some of the ways in which queries can be optimized for distributed environments. The problems incurred in query processing and optimizations have been discussed in this thesis. Different kinds of search spaces and search strategies have also been discussed. This research work helps us to extend the existing optimization techniques to meet the demanding requirements of a challenging world. It also paves way to create an extensible and flexible optimizer to meet the growing demand of modern database applications.
ACKNOWLEDGEMENT

I extend my profound thanks to Mr. A. C. Shanmugam, Founder, Mr. A. C. S. Arunkumar, President, Dr. K. Meer Mustafa Hussain, Vice Chancellor, Dr. C. B. Palanivelu, Registrar, Dr. V. Cyril Raj, Dean - Engineering and Technology, Dr. A. Thirunavukkarasu, Dean - Research, Dr. T. S. Saravanan, Research coordinator and Prof. Dr. M. Ganesan, Head - Department of Mechanical Engineering of Dr. M.G.R Educational and Research Institute University for providing this wonderful opportunity to carry out my research work.

Most of all I would like to thank my esteemed research supervisor, Dr. S. K. Srivatsa, Retired Professor, Anna University, Chennai, for his incredible guidance and encouragement throughout my research work. I would also like to express my gratitude to my internal member, Dr. Viji Vinod, Head - Department of Computer Applications of Dr. M.G.R Educational and Research Institute University for her advices and moral support given to me right from the beginning to the end of my research work. I am also grateful to my subject expert, Dr. S. Ravi, Head - Department of Electronics Communication Engineering of Dr. M.G.R Educational and Research Institute University for the continuous level of feedback and amount of time he has devoted for my research work.

I also like to thank all of my friends and fellow professors at Ethiraj college. A special gratitude goes to my family members especially my mother and my husband for their unconditional support. Their affection, prayers and motivation paved way to complete my work in a full-fledged manner. My deepest gratitude goes to my creator the almighty God for all his blessings in every step of mine towards the completion of my research work. This thesis is the most significant accomplishment in my life and it would be impossible without people who supported me and believed in me.

Tejy K K
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter no.</th>
<th>Title</th>
<th>Page no.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ABSTRACT</td>
<td>iv</td>
</tr>
<tr>
<td></td>
<td>LIST OF TABLES</td>
<td>xiii</td>
</tr>
<tr>
<td></td>
<td>LIST OF FIGURES</td>
<td>xiv</td>
</tr>
<tr>
<td></td>
<td>LIST OF SYMBOLS AND ABBREVIATIONS</td>
<td>xvi</td>
</tr>
<tr>
<td>1</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Problem Overview</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1</td>
<td>Problem description</td>
<td>1</td>
</tr>
<tr>
<td>1.1.2</td>
<td>Challenges met</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Motivation for this Research</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>Terminologies</td>
<td>4</td>
</tr>
<tr>
<td>1.4</td>
<td>Contributions</td>
<td>5</td>
</tr>
<tr>
<td>1.5</td>
<td>Organization of the Thesis</td>
<td>6</td>
</tr>
<tr>
<td>1.6</td>
<td>Thesis Structure</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>RELATED RESEARCH</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>OVERVIEW OF QUERY OPTIMIZATION</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>Query Optimization Process</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>Query Optimization Techniques</td>
<td>21</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Heuristic Optimization</td>
<td>22</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Syntactical Optimization</td>
<td>23</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Cost Based Optimization</td>
<td>23</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Semantic Optimization</td>
<td>24</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Parametric Optimization</td>
<td>26</td>
</tr>
<tr>
<td>Chapter no.</td>
<td>Title</td>
<td>Page no.</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>4</td>
<td>OVERVIEW OF QUERY EXECUTION</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>4.1 Query Representation</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>4.1.1 Query tree</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>4.1.2 Query graph</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>4.2 Query Evaluation</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>4.3 Query Execution</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>4.3.1 Selection operation</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>4.3.2 Projection operation</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>4.3.3 Set operations</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>4.3.4 Join operation</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>4.3.5 Rename operation</td>
<td>36</td>
</tr>
<tr>
<td>5</td>
<td>PERFORMANCE TUNING FACTORS</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>5.1 Query Optimizer</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>5.1.1 Search space</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>5.1.2 Search strategies</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>5.1.2.1 Deterministic strategies</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>5.1.2.2 Randomized strategies</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>5.1.3 Cost model</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>5.1.4 Different types of query optimizers</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>5.1.4.1 System R optimizer</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>5.1.4.2 Exodus optimizer</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>5.1.4.3 Starburst optimizer</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>5.1.4.4 Volcano optimizer</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>5.1.4.5 Cascade optimizer</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>5.1.5 Comparative Study</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>5.2 Query Equivalence</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>5.2.1 Usage of equivalence rules</td>
<td>53</td>
</tr>
<tr>
<td>Chapter no.</td>
<td>Title</td>
<td>Page no.</td>
</tr>
<tr>
<td>------------</td>
<td>----------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>5.3</td>
<td>Indexing</td>
<td>54</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Different types of indices</td>
<td>55</td>
</tr>
<tr>
<td>5.3.1.1</td>
<td>Dense index</td>
<td>55</td>
</tr>
<tr>
<td>5.3.1.2</td>
<td>Bitmap index</td>
<td>55</td>
</tr>
<tr>
<td>5.3.1.3</td>
<td>Clustered index</td>
<td>57</td>
</tr>
<tr>
<td>5.3.1.4</td>
<td>Non-clustered index</td>
<td>57</td>
</tr>
<tr>
<td>5.3.1.5</td>
<td>Function-based index</td>
<td>58</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Guidelines for managing indices</td>
<td>59</td>
</tr>
<tr>
<td>5.4</td>
<td>Cost estimation</td>
<td>59</td>
</tr>
<tr>
<td>5.5</td>
<td>Dependencies implied by SQL expressions</td>
<td>61</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Functional dependency (FD)</td>
<td>61</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Multi-valued dependency (MD)</td>
<td>61</td>
</tr>
<tr>
<td>5.5.3</td>
<td>Join dependency (JD)</td>
<td>62</td>
</tr>
<tr>
<td>5.5.4</td>
<td>Transitive dependency (TD)</td>
<td>62</td>
</tr>
<tr>
<td>5.5.5</td>
<td>Full functional dependency (FFD)</td>
<td>63</td>
</tr>
<tr>
<td>5.5.6</td>
<td>Partial functional dependency (PFD)</td>
<td>63</td>
</tr>
<tr>
<td>6</td>
<td>PROPOSED METHOD FOR QUERY OPTIMIZATION</td>
<td>64</td>
</tr>
<tr>
<td>6.1</td>
<td>Proposed Method</td>
<td>64</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Algorithm and Running example</td>
<td>65</td>
</tr>
</tbody>
</table>
6.1.1.1 Algorithm for Pre-processing 65
6.1.1.2 Algorithm for Query tree construction 67
6.1.1.3 Algorithm for Dependency Rule Generation 68
6.1.1.4 Algorithm for Query Optimization 69
6.1.1.5 Example Illustration 70

6.2 Comparative Analysis 73
6.2.1 Linear Recursive Method 74
6.2.2 Semantic based Cache Method 74
6.2.3 Semantic-based-Combined-Similarity Method 75

6.3 Improvement on the Proposed Method 77
6.3.1 Multi level relational mapping based dependency rule generation 79
6.3.1.1 Multi level Preprocessing 80
6.3.1.2 Multi Level Relational Mapping 81
6.3.1.3 Dependency measure computation 83
6.3.1.4 Multi level dependency rule generation 84
<table>
<thead>
<tr>
<th>Chapter no.</th>
<th>Title</th>
<th>Page no.</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4</td>
<td>Structure of proposed query optimizer</td>
<td>86</td>
</tr>
<tr>
<td><strong>7</strong></td>
<td><strong>EXPERIMENTAL RESULTS</strong></td>
<td><strong>88</strong></td>
</tr>
<tr>
<td>7.1</td>
<td>Experimental setup</td>
<td>88</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Dataset description</td>
<td>88</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Query set used</td>
<td>91</td>
</tr>
<tr>
<td>7.2</td>
<td>Snapshots of the result with running example</td>
<td>93</td>
</tr>
<tr>
<td>7.3</td>
<td>Performance study</td>
<td>97</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Result study on the Improved proposed Method</td>
<td>99</td>
</tr>
<tr>
<td>7.5</td>
<td>Effectiveness of Proposed Optimizer model</td>
<td>103</td>
</tr>
<tr>
<td>7.6</td>
<td>Best practices</td>
<td>107</td>
</tr>
<tr>
<td><strong>8</strong></td>
<td><strong>CONCLUSION AND FUTURE WORK</strong></td>
<td><strong>109</strong></td>
</tr>
<tr>
<td>8.1</td>
<td>Conclusion</td>
<td>109</td>
</tr>
<tr>
<td>8.2</td>
<td>Future work</td>
<td>111</td>
</tr>
<tr>
<td><strong>REFERENCES</strong></td>
<td></td>
<td><strong>112</strong></td>
</tr>
<tr>
<td><strong>LIST OF PUBLICATIONS</strong></td>
<td></td>
<td><strong>117</strong></td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table No.</th>
<th>Title</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Cost model parameters</td>
<td>43</td>
</tr>
<tr>
<td>5.2</td>
<td>Comparative study of various optimizers</td>
<td>50</td>
</tr>
<tr>
<td>6.1</td>
<td>Notations used in preprocessing algorithm</td>
<td>66</td>
</tr>
<tr>
<td>6.2</td>
<td>Notation description for query construction Algorithm</td>
<td>68</td>
</tr>
<tr>
<td>6.3</td>
<td>Notations used in dependency rule generation algorithm</td>
<td>69</td>
</tr>
<tr>
<td>6.4</td>
<td>Notation description for query optimization Algorithm</td>
<td>70</td>
</tr>
<tr>
<td>6.5</td>
<td>Notations used in multi level preprocessing Algorithm</td>
<td>81</td>
</tr>
<tr>
<td>6.6</td>
<td>Notations used in multi level relational mapping algorithm</td>
<td>82</td>
</tr>
<tr>
<td>6.7</td>
<td>Notations used in dependency measure computing algorithm</td>
<td>84</td>
</tr>
<tr>
<td>6.8</td>
<td>Notations used in multi level dependency rule generation algorithm</td>
<td>85</td>
</tr>
<tr>
<td>7.1</td>
<td>Employee table</td>
<td>89</td>
</tr>
<tr>
<td>7.2</td>
<td>Department table</td>
<td>90</td>
</tr>
<tr>
<td>7.3</td>
<td>Jobs table</td>
<td>90</td>
</tr>
<tr>
<td>7.4</td>
<td>Jobs_History table</td>
<td>91</td>
</tr>
<tr>
<td>7.5</td>
<td>Details of evaluation parameter</td>
<td>100</td>
</tr>
<tr>
<td>7.6</td>
<td>Comparison of various query plans</td>
<td>106</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure No.</th>
<th>Title</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Structure of the thesis</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>Query representation</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>Steps in query processing</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>Query optimization process</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Query tree representation</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>Query graph representation</td>
<td>29</td>
</tr>
<tr>
<td>4.3</td>
<td>Query evaluation steps</td>
<td>31</td>
</tr>
<tr>
<td>4.4</td>
<td>Operator tree for the expression</td>
<td>32</td>
</tr>
<tr>
<td>5.1</td>
<td>Architecture of query optimizer</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>Structure of dense index</td>
<td>55</td>
</tr>
<tr>
<td>5.3</td>
<td>Bitmap index retrieval of data</td>
<td>56</td>
</tr>
<tr>
<td>5.4</td>
<td>Structure of Clustered index</td>
<td>57</td>
</tr>
<tr>
<td>5.5</td>
<td>Structure of Non-clustered index</td>
<td>58</td>
</tr>
<tr>
<td>6.1</td>
<td>Algorithm for pre-processing</td>
<td>66</td>
</tr>
<tr>
<td>6.2</td>
<td>Algorithm for query tree construction</td>
<td>67</td>
</tr>
<tr>
<td>6.3</td>
<td>Algorithm for dependency rule generation</td>
<td>68</td>
</tr>
<tr>
<td>6.4</td>
<td>Algorithm for query optimization</td>
<td>69</td>
</tr>
<tr>
<td>6.5</td>
<td>Database schema for the example query</td>
<td>71</td>
</tr>
<tr>
<td>6.6</td>
<td>Query tree for sub queries</td>
<td>72</td>
</tr>
<tr>
<td>6.7</td>
<td>Relational mapping process</td>
<td>78</td>
</tr>
<tr>
<td>6.8</td>
<td>Architecture of multi level relational</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>mapping</td>
<td></td>
</tr>
<tr>
<td>6.9</td>
<td>Algorithm for multi level preprocessing</td>
<td>81</td>
</tr>
<tr>
<td>Figure No.</td>
<td>Title</td>
<td>Page No.</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>6.10</td>
<td>Algorithm for multi level relational mapping</td>
<td>82</td>
</tr>
<tr>
<td>6.11</td>
<td>Algorithm for dependency measure</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>Computation</td>
<td></td>
</tr>
<tr>
<td>6.12</td>
<td>Algorithm for multi level dependency rule generation</td>
<td>85</td>
</tr>
<tr>
<td>6.13</td>
<td>Framework of proposed query optimizer</td>
<td>87</td>
</tr>
<tr>
<td>7.1</td>
<td>Employee database schema</td>
<td>88</td>
</tr>
<tr>
<td>7.2</td>
<td>Snapshot of query submission interface</td>
<td>93</td>
</tr>
<tr>
<td>7.3</td>
<td>Snapshot of identified query parts</td>
<td>94</td>
</tr>
<tr>
<td>7.4</td>
<td>Identified query parts</td>
<td>95</td>
</tr>
<tr>
<td>7.5</td>
<td>Snapshot of rules generated</td>
<td>95</td>
</tr>
<tr>
<td>7.6</td>
<td>Generated rules for the given query</td>
<td>96</td>
</tr>
<tr>
<td>7.7</td>
<td>Snapshot of final optimized sequence</td>
<td>96</td>
</tr>
<tr>
<td>7.8</td>
<td>Optimal sequence for query execution</td>
<td>97</td>
</tr>
<tr>
<td>7.9</td>
<td>Comparison of cost efficiency</td>
<td>98</td>
</tr>
<tr>
<td>7.10</td>
<td>Comparison of time taken for query Optimization</td>
<td>99</td>
</tr>
<tr>
<td>7.11</td>
<td>Comparison of rule generation efficiency</td>
<td>101</td>
</tr>
<tr>
<td>7.12</td>
<td>Comparison of query optimization accuracy</td>
<td>101</td>
</tr>
<tr>
<td>7.13</td>
<td>Comparison of time taken</td>
<td>102</td>
</tr>
<tr>
<td>7.14</td>
<td>Query execution plan1 for query1</td>
<td>104</td>
</tr>
<tr>
<td>7.15</td>
<td>Query execution plan2 for query1</td>
<td>104</td>
</tr>
<tr>
<td>7.16</td>
<td>Query execution plan3 for query1</td>
<td>105</td>
</tr>
<tr>
<td>7.17</td>
<td>Query execution plan4 for query1</td>
<td>106</td>
</tr>
</tbody>
</table>
# LIST OF SYMBOLS AND ABBREVIATIONS

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>Input Query</td>
</tr>
<tr>
<td>S</td>
<td>Database Schema</td>
</tr>
<tr>
<td>Os</td>
<td>Object set</td>
</tr>
<tr>
<td>Qs</td>
<td>Query set</td>
</tr>
<tr>
<td>Rms</td>
<td>Relational map set</td>
</tr>
<tr>
<td>As</td>
<td>Attribute set</td>
</tr>
<tr>
<td>RAS</td>
<td>Relational Attribute Set</td>
</tr>
<tr>
<td>Qp</td>
<td>Query part</td>
</tr>
<tr>
<td>Dm</td>
<td>Dependency measure</td>
</tr>
<tr>
<td>Qh</td>
<td>Query hierarchy</td>
</tr>
<tr>
<td>NRD</td>
<td>Number of Relational Dependent object</td>
</tr>
<tr>
<td>PQS</td>
<td>Preprocessed Query Set</td>
</tr>
<tr>
<td>QT</td>
<td>Query Tree</td>
</tr>
<tr>
<td>DRS</td>
<td>Dependency Rule Set</td>
</tr>
<tr>
<td>Seq</td>
<td>Sequence number</td>
</tr>
<tr>
<td>DR</td>
<td>Dependency Rules</td>
</tr>
<tr>
<td>NI</td>
<td>Number of Inputs</td>
</tr>
<tr>
<td>NDTR</td>
<td>Number of Data Table attributes</td>
</tr>
<tr>
<td>NL</td>
<td>Number of Locks applied</td>
</tr>
<tr>
<td>Qcp</td>
<td>Query completion probability</td>
</tr>
<tr>
<td>CDm</td>
<td>Cumulative Dependency measure</td>
</tr>
<tr>
<td>Cop</td>
<td>Conditional operator</td>
</tr>
<tr>
<td>Reo</td>
<td>Relational operator</td>
</tr>
<tr>
<td>At</td>
<td>Attributes</td>
</tr>
<tr>
<td>N</td>
<td>Number of queries</td>
</tr>
<tr>
<td>Fun</td>
<td>Functions mentioned</td>
</tr>
<tr>
<td>Rn</td>
<td>Root node</td>
</tr>
<tr>
<td>LNS</td>
<td>Leaf Node Set</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>Li</td>
<td>Leaf node</td>
</tr>
<tr>
<td>PS</td>
<td>Parent Sibling node</td>
</tr>
<tr>
<td>Noc</td>
<td>Number of attributes of child nodes in current data table</td>
</tr>
<tr>
<td>Noo</td>
<td>Number of attributes of child nodes in other data table</td>
</tr>
<tr>
<td>mdt</td>
<td>Data table containing maximum attributes</td>
</tr>
<tr>
<td>SEq</td>
<td>Final Selected sequence</td>
</tr>
<tr>
<td>Ri</td>
<td>Each rule</td>
</tr>
<tr>
<td>Qcps</td>
<td>Query completion probability set</td>
</tr>
<tr>
<td>Oi</td>
<td>Each object</td>
</tr>
<tr>
<td>U</td>
<td>Union operation</td>
</tr>
<tr>
<td>Π</td>
<td>Projection operation</td>
</tr>
<tr>
<td>σ</td>
<td>Selection operation</td>
</tr>
<tr>
<td>∩</td>
<td>Intersection operation</td>
</tr>
<tr>
<td>_</td>
<td>Set difference operation</td>
</tr>
<tr>
<td>x</td>
<td>Cartesian product</td>
</tr>
</tbody>
</table>
List of Corrections Suggested by the Examiners and Incorportations are detailed below.

<table>
<thead>
<tr>
<th>S.No</th>
<th>Examiner Comments</th>
<th>Page No.</th>
<th>Correction Carried out</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>To distil the contributions into 4-5 major contributions in Chapter 1.</td>
<td>3</td>
<td>The minor contributions are merged and distilled into 4-5 major contributions in Chapter 1.</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Difference and similarity of the Current work in the context of the prior work needs to be addressed in Chapter 2.</td>
<td>5-12</td>
<td>The difference and similarity of the current work in the context of the prior work needs has been addressed in Chapter 2.</td>
<td>8-16</td>
</tr>
<tr>
<td>3</td>
<td>Examples in figures 4.1, 4.2 and 4.4 are simplistic in Chapter 4.</td>
<td>25,26,29</td>
<td>Better examples in figures 4.1, 4.2 and 4.4 are given in Chapter 4.</td>
<td>28,29,32</td>
</tr>
<tr>
<td>4</td>
<td>To illustrate the various steps of the proposed scheme by using a running example in Chapter 6.</td>
<td>-</td>
<td>Illustration of various steps of the proposed scheme has been given by using running example in Chapter 6.</td>
<td>65-73</td>
</tr>
<tr>
<td>5</td>
<td>Mention the Data set, Query set and configuration of the machine used for the experiments in Chapter 7.</td>
<td>-</td>
<td>The data set, query sets and the configuration of the machine used for the experiments are given in Chapter 7.</td>
<td>88-92</td>
</tr>
<tr>
<td>6</td>
<td>Definition of the terms cost efficiency, rule generation efficiency and query optimization accuracy should be provided in Chapter 7.</td>
<td>-</td>
<td>Definition of the terms cost efficiency, rule generation efficiency and query optimization accuracy has been provided in Chapter 7.</td>
<td>97,100,101</td>
</tr>
<tr>
<td>7</td>
<td>Spelling and grammatical errors should be carried out.</td>
<td>-</td>
<td>Throughout the thesis Spelling and grammatical errors has been carried out.</td>
<td>1-111</td>
</tr>
</tbody>
</table>
The other things to be included are listed below

<table>
<thead>
<tr>
<th>S.No</th>
<th>Examiner Comments</th>
<th>Page No.</th>
<th>Correction Carried out</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>To include the definition of the terminologies in Chapter 1.</td>
<td>-</td>
<td>The definition of the terminologies has been included in Chapter 1.</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>To include the problem description in Chapter 1.</td>
<td>-</td>
<td>The problem description has been included in Chapter 1.</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>To specify the challenges met in persuading with the problem in Chapter 1.</td>
<td>-</td>
<td>The challenges met in persuading with the problem is specified in Chapter 1.</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>To give the insight of chapters discussed in Chapter 1.</td>
<td>-</td>
<td>The insight of chapters has been discussed in Chapter 1.</td>
<td>7</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

With the widespread applications of Database Management Systems (DBMS), users have to deal with an enormous amount of data. Therefore, it is necessary to store this information in such a way that it can be retrieved from the database in the fastest possible manner to satisfy the request from a user. Databases are most useful in representing data in an organized manner. It provides the user with the ability to acquire accurate, reliable and timely data for effective decision making process. Thus, the significance of database systems is increasing day by day. At the same time, data queries are becoming more and more complex. As a result, query optimization has emerged as the most significant factor in reducing this complexity. During the past decade, many research and development works have been carried out in the area of query optimization. Query optimization in database has gained significant importance as it helps to reduce the size, memory usage and time required for any query to be processed. The main objective of any query optimization is to determine the best strategy for executing each query. It identifies an efficient way to execute the query with less time complexity to produce better results. This process can be formally defined as transforming a query into an equivalent form which can be evaluated more efficiently.

1.1 Problem Overview

1.1.1 Problem description

In query optimization, the major role is played by the query optimizer. The main task of the query optimizer is to take a user given query and produce an query execution plan that indicates how the query should be executed. This is the crucial step which contributes in the reduction of the execution cost of the query. The problem that the
query optimizer frequently faces is, for a given user query there exists a large number of different equivalent query execution plans. Each query execution plan have its own corresponding execution cost. The cost of each query execution plan differs based on the order of evaluating each distinct part of the user given query. Thus, we require a query optimizer that is capable of estimating the best execution plan for executing the query. Many approaches have been done in this area of field. An analytical study of these existing various optimization techniques, methods and optimizers have also been discussed in this thesis. In spite of all these approaches, we are still unable to balance the amount of time spent in figuring out the best query plan and the quality of the choice.

Query optimization is a difficult task as the translation of a query in a declarative language to a query execution plan makes the whole process very complex because there are many factors to be considered. In this thesis, we have carried our detailed investigation about the factors that influence the performance of query processing. We assume that this investigation yield performance increase over the widest variety of situations. Selecting a right or wrong optimal execution plan would mean the difference between a query execution time of milliseconds. Thus, a better understanding of how the query optimizer works can help both database administrator and developer to write better queries in order to produce efficient execution plan. To facilitate this, we have provided a basic framework model for understanding the field of query processing and optimization in general.

1.1.2 Challenges met

With the emergence of various application oriented databases the complexity in queries are increasing day by day which makes the problem of determining an best query optimization plan more challenging.

The first and foremost challenge that we come across is that how to make a query optimizer explore the entire search space. However, if a query optimizer is not able to explore the entire search space then there is no way to prove that we can get an absolutely optimal plan. So we need to explore the methods to limit the search space.
Another challenge that we face in query optimization is determining accurate cost and cardinality estimation. Many existing optimizer models fail to consider all of the hardware conditions and assumptions about the environment which leads to lack in cost accuracy. The key cost metrics that have to be considered in query optimization have also been discussed in this thesis.

One more challenge that we encounter is join ordering which controls the amount of data flow between each operator in the execution plan. An optimizer must be capable of determining the best selection of join order in order to achieve the efficiency in query performance. These were the challenges that we have to overcome in building an efficient query optimizer model.

1.2 Motivation for this research

Query optimization in databases continues to be an important issue in commercial and academic fields for quite a long period of time. The importance for optimization arises from the flexibility provided by modern user interfaces to databases. The interfaces and non-procedural query languages help the users to easily specify queries effectively. At the same time, this flexibility makes computation and execution cost of the query more. To reduce the execution cost, we need to reformulate the user specified query before executing an equivalent form that is computationally more efficient. Many approaches have been discussed earlier on this, but suffer from the problem of dimension and accuracy.

To overcome this complexity issue, we propose a query tree based method which generates dependency rules to perform query optimization. In this method, for each distinct part of the query, we identify the relational objects and attributes of each sub query to construct a query tree. From the generated tree, we generate a set of sequences of query execution objects which is called dependency rule. For each sequence identified, we compute the query completion time and based on completion time, a least processing sequence will be selected as the most efficient one. This method overcomes the problem of missing cases and can produce efficient processing results. This method produces efficient results for the data available only within the same data tables.
In case of multi level queries or nested queries, we need to understand the dependency between the query parts to generate an optimal sequence for query execution. To overcome this issue, we propose an improvement to the existing method by using multi level relational mapping algorithm. The multi level relational mapping approach first splits the given input query into number of sub queries. Then for each sub query it tries to identify the relations that the query has with the help of relational mapping. The use of a relational map helps to identify the query dependency in multiple levels and generates an optimal sequence of query execution based on dependency measure. The dependency measure is computed by considering the number of relations being shared by the query part. Based on the dependency measure the dependency rules are being generated. These dependency rules sort the order of the query part and produce the final optimal sequence order for execution. This optimal sequence produces efficient results and reduces time complexity also. The main aim of this thesis is to produce an efficient query plan with the minimum execution time or response time.

1.3 Terminologies

Some of the basic terminologies used in the query optimization have been discussed in this section as follows:

A **Query** is a request for information from a database. A database is a collection of data in an organized manner. A **query plan** is an ordered set of steps used to access data in a SQL database. **Relational algebra** is a family of algebras with a well founded semantics used for modeling the data stored in relational database. A **database index** is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the data structure. **Database schema** is a way to logically group objects such as tables etc. A **query tree** is a tree data structure in which the input relations are represented as leaf nodes and the algebra operations as the internal nodes. A **leaf node** represents the input relations of the query. **Internal nodes** represent intermediate relation that is the output of applying an operation in the relational algebra. **Query graph** is a data structure that is used for internal representation of the query.
Query evaluation plan specifies how to evaluate each operation in the query. Query optimization is a process in which multiple query execution plans for satisfying a query are examined and most efficient query plan is identified for execution. The cardinality of a set is a measure of the number of elements in the set. Query optimizer is a component of database management system which attempts to determine the most efficient way to execute a query. Search space or solution space is the set of all query execution plans that represent the input query. Search strategy defines the order in which the query plans to be examined. Cartesian product creates a relation that has the combination of two different relations. Cost function not only predicts the cost of operators but also formulas to evaluate the size of the result.

1.4 Contributions

In this section, we give a summary of the main contributions of the various chapters of the thesis. The contributions of this thesis are as follows:

- We determine the factors that play a vital role in performance tuning of a database system.
- We study and analyze the role of query optimizer and equivalence rules in query optimization.
- We explore and evaluate shortcomings of various existing optimizer models and proposed a general framework for query optimizer.
- We highlight the benefits of indexing and suggest some best practices to reduce the execution cost of the query.
- We propose a query tree based method which generates dependency rules to perform query optimization and proposed a multi level relational mapping algorithm to improve the performance of query optimization by identifying the multi level query dependency.
- We have implemented the proposed techniques and present a performance study that clearly demonstrates the benefits of our approach.
1.5 Organization of the thesis

This thesis is divided into eight chapters as follows:

- Chapter 1 contains this introduction.

- Chapter 2 contains an overview of past achievement and related work done in this area of research.

- Chapter 3 discusses the basic steps involved in the query optimization process. It also describes existing query optimization techniques, algorithms and strategies.

- Chapter 4 provides general architecture of query execution, query representation and query evaluation. It also briefs the fundamental operations of relational algebra.

- Chapter 5 illustrates the various factors that play a vital role in tuning the query performance. The factors discussed in this chapter are query optimizer, query equivalence, indexing, cost estimation and dependencies implied by SQL expressions. The components of query optimizer and their role in query optimization is also discussed. Comparative study of various query optimizer models is also done in this chapter.

- Chapter 6 describes proposed query tree based method which generates dependency rules to perform query optimization. It also describes improvement on the proposed method by using multi level relational mapping algorithm.

- Chapter 7 presents the experimental study carried out for the proposed method and the improved method. The architecture of the proposed query optimizer model has also been discussed in this chapter.

- Chapter 8 concludes and describes possible extensions of this research work in future. This research work addresses the problem of facilitating the enhancement of query performance.
1.6 Thesis Structure

The flow of the research work is given in the following figure 1.1. It depicts how the research work has been carried out.

Figure 1.1 Structure of the thesis
CHAPTER 2

REVIEW OF LITERATURE

An architecture for distributed query processing was proposed by (Kosmann 2000) in addition to a series of techniques that are particularly useful for distributed database systems. It shows how techniques such as caching and replication can be used to improve the performance of queries in a distributed environment. It primarily focuses on fundamental mechanism to process queries and concentrates mainly on structured data. Here it fails to present architecture of search engines and to integrate all types of data. The issue of scalability still remains a major concern in this paper. We have addressed this issue by designing a framework for query optimizer which can incorporate the changing requirements of a growing database.

Lin (2009) describes the flow of query optimization consisting of multiple modules for distributed databases. The modules discussed were user module, system analysis module, query tree conversion module, optimizer module and order processing module. The user module in a distributed system analyzes the user query request. The system analysis module examines sentences of the query. This is followed by converting the query into its corresponding tree. This is passed to query tree conversion module which converts it into the global query tree. The global query tree is then mapped to the corresponding physical operator tree by the optimizer module. Then, the optimizer module selects a physical operator tree with lowest cost. The order processing module sends the whole process to the respective server which gives response to the user. But when the size of the corresponding table become large, the CPU processing time will be large and the memory consumption will be more which may be considered as the limitation of the proposal.

The problem of determining the optimal plan at compile time has been discussed by (Bizarro et al. 2009). An attempt to solve this problem was done by using parametric
query optimization. It suggests usage of query hints by the query optimizer to improve the plan until a best plan is picked up. It proposes building a parametric plan during execution of the query. The method was effective in determining the optimal plan, but in terms of cost it does not prove effective while executing the query infrequently.

The optimization of linear recursive queries in SQL was described by (Carlos Ordonez 2010). It describes two fundamental algorithms for SQL implementation: seminaive and direct. Five query optimizations are studied: storage and indexing, early selection, early evaluation of nonrecursive joins, pushing duplicate elimination and pushing aggregation. Experiments compare both evaluation algorithms and systematically evaluate the impact of optimizations with large input tables. Optimizations are evaluated on four types of graphs: binary trees, lists, cyclic graphs, and complete graphs, going from the best to worst case. In general, seminaive is faster than direct, except for complete graphs was proven. It mainly focuses on computing the transitive closure of a graph. The limitation of this proposed system was queries on unbalanced trees and non binary trees were not considered.

Inductive learning method for performing semantic query optimization in relational databases was proposed by (Jyoti et al. 2012). In this method rules were derived from the query processed and these rules were used for generating alternate equivalent query. The semantically equivalent queries generated are proven to be less expensive than the original query. This can be implemented in SQL using the SQL hints. These hints allow a user to implement the desired plan for the query. This method was efficient for relational database but it faced difficulty in implementing for distributed database. The issue addressed was finding difficult to retrieve the data from different location sites when there exist dependency among the attributes.

Sunitha Mahajan and Vaishali Jadhav (2012) suggest a heuristic approach for selecting the optimal evaluation plan and semi-join approach for reducing the communication cost. This framework includes the heuristic rules for finding optimal evaluation plans among various plans of a single query. The approach consists of three phases namely, local processing phase which involves all local processors, a semi-join reduction phase where a sequence of semi-joins are used to reduce the sizes of relations,
and a final processing phase in which all resulting relations are sent to the result site where the final query processing is performed. The problem incurred in this method was different approaches has to be followed for converting cyclic queries into tree queries. Lack of information about which approach to be used for which type of query was not available.

The basic components of the distributed query optimizer such as search space and search strategy was discussed by (Vinod Gangwani and Ramteke 2013). The algorithms used for deterministic and randomized strategies were also discussed in this paper. Based on the comparative study it was proposed that a combination of iterative improvement method and simulated annealing method would be well suited for obtaining the optimal plan. But this technique does not guarantee for finding the optimal plan for complex queries. This complexity issue was addressed in our optimization model.

Majid Khan and Khan (2013) review different query optimization techniques and approaches for both centralized and distributed databases. Critical analyses were made on various query optimization techniques that were used for both centralized and distributed databases. Merits, limitations and scope of various query optimization techniques and approaches have also been discussed. The problems and solutions in various approaches have also been presented. This study paper paved way for understanding the problems in the existing optimization techniques.

Kunal Jamsutkar et al. (2013) discuss the general aspects of optimization by providing better understanding of the terms query, query processing and query optimization. The role of indices and how the utilization of indices can dramatically reduce the execution time of various operators such as Select and Join have also been discussed. The complexities of various query optimization algorithms have also been discussed. This study highlights the need of developing distributed query optimization techniques that can process and optimize queries on non-structured data.

Performance analysis of optimization techniques for SQL multi query expressions over text databases in RDBMS was presented by (Swati Jain and Paras Nath Barwal 2014). It presents both static and dynamic processes of optimization as well as all the general aspects of query optimization.
They explore major principles of query optimization process with volcano query optimization. An enhancement to volcano query optimizer is proposed by adding some set of transformation rules and operators. They also elaborate various search strategy techniques including dynamic and greedy algorithms that will play a crucial role in improving the overall efficiency of relational database systems. It shows that randomized algorithms are slower than heuristics and dynamic programming for simpler queries but this is inverted for large queries. In this method also the accuracy in determining the optimal plan is not guaranteed.

Distributed Database System Query Optimization Algorithm presented (Fan Yuanyuan and Mi Xifen 2010) pertains to query optimization technology, based on a number of optimization algorithms commonly used in distributed query. They design a new algorithm to optimize the semi-connection order of the sub-query. The experimental results show that this algorithm can significantly reduce the amount of intermediate result data and thus the network communication cost can be reduced effectively. Hence, this algorithm helps to improve the optimization efficiency. But it fails to optimize the semi connection order of the sub query to further reduce the network communication cost. The issue of optimizing sub queries was addressed in our proposed method.

Query optimization in Object Oriented DBMS (Dhande and Bamnote 2015) suggests a methodology where we can handle multiple attributes of objects through direct navigation method. This method is an extension of direct navigation method of optimization. The proposed method works for complex queries which involve large number of objects and attributes. The experimental results illustrate that the proposed algorithm is efficient to handle multiple attributes. This algorithm is proven to produce optimized result. The limitation of this technique is that data manipulation rate is very low as compared to data retrieval rate.

Object query optimization through detection of independent sub queries (Nikose et al. 2012) addresses the problem of query optimization in object-oriented databases. Here authors describe a stack based approach to query languages which employs naming-scoping-binding paradigm of programming languages. The idea used is based on the
static analysis of scoping rules and binding names occurring in a query. They also present a data model that is used to deal with static query optimization. This model is efficient with queries involving independent sub queries and ineffective for dependent queries.

Selection of materialized view using query optimization in database management (Karde and Thakare 2010) describes an efficient methodology. It proposes two algorithms for handling the problem of materialized view maintenance and selection. The first algorithm is for generation and maintenance of materialized view. The tree based approach is used for creating and maintaining materialized views. The second algorithm is for node selection. This algorithm decides the nodes in the distributed environment for which materialized view should be created, updated or to be maintained. The random walk algorithm is used as base for designing the node selection algorithm. The issues incurred in this method were all of the views or queries for materialization were unexplored due to the view maintenance costs.

Importance of distributed query processing in improving the overall performance of a distributed database systems was proposed by (Garima Mahajan 2012). It introduces two important aspects namely total cost measure and response time measure for processing the query in a distributed database. From the analytical study of distributed cost model it was observed that the response time measure can be minimized by increasing the degree of parallel execution while the total cost measure can be minimized by improving the utilization of resources. This study helped us to understand that lack of adequate accurate information regarding cardinality, index availability and organization information increases complexity in distributed query processing.

Saurabh Gupta et al. (2015) describe the importance of query processor and optimizer. Author identifies the query processor and optimizer as an important component responsible for translating a user query into an efficient query evaluation program that can be executed against the database. It provides a basic framework for understanding the field of query processing and query optimization in general. In this paper, author have presented a detailed review of various query optimization techniques.
used to implement query processing and optimization in an effective manner. It also highlights the merits and critically analyzes the various query optimization techniques. It provides a framework for understanding the field of query processing and optimization in general.

Vishal Hatmode and Sonali Rangdale (2014), describe the process of SQL query optimization based on heuristic approach. It proposes a new approach of building test database to translate SQL queries into equivalent highly optimized SQL queries. Here author emphasizes on the usage of query optimizer tool for writing efficient and optimized queries. Author describes heuristic optimizer that can accept the user query and automatically generate an equivalent but highly optimized and effective query. This optimizer helps to reduce time, cost and effort. It also tries to minimize the number of accesses by reducing the number of tuples and number of columns to be selected. The ability to process the queries in a timely manner is still a problematic issue.

Corlatan et al. (2014), provides a list of SQL scenarios to serve as a quick and easy reference guide during the development phase and maintenance of the database. It discusses about Transact-SQL language that provides a wide range of techniques for updating query. Author also suggests considering not only the select queries but also other objects such as index, view or statistics for optimization. Here author also suggest the usage of stored procedure or parameterized query to provide a reusable execution plan independent of the variables used in a query. The main factors that affect the performance of a Microsoft SQL server system have also been discussed. The factors identified are missing indexes, inexact statistics, badly written queries, deadlocks, excessive fragmentation of indexes and frequent recompilation of queries. This study provided the framework to understand the factors and objects that can affect the performance of the query.

Sevinc and Cosar (2011), proposes a new genetic algorithm based query optimizer. They have done a comparative analysis of the proposed genetic algorithm performance with random and optimal algorithms. Efficiency of the proposed algorithm is shown by performing experiments on a synthetic database with replicated relations.
The results of the experiments show that the proposed method has achieved a 50% improvement over the previous GA-based genetic algorithm. This system considers only replication and not fragmentation. It does not allow to locally optimize a given query access path and assumes all queries to be irreducible.

Pund et al. (2011), introduces the basic concepts of query processing and query optimization in relational database. Here author presents how database processes a query as well as some of the algorithms and rule sets utilized to produce more efficient queries. Author also discusses about the implementation plan using join ordering to extend the capability of database engine program through the use of randomized algorithm iterative improvement method. The efficiency for larger queries was not explored.

Monjurul Alom et al. (2009), discusses distributed query optimization problem and proposed a ARRQ technique which determines how to partition fragments and where the partition fragments need to be sent for processing. It is a technique used to process a query where all the relations referenced by a query are not fragmented but distributed in different sites. The main objective of this method is to exploit parallelism as well as minimizing the quantity of inter-site data transfer. This technique provides better efficiency in terms of query processing cost when the given query references all the relations. In this paper, different methodologies used for distributed query processing and the problems of distributed query optimization has also been described. The structure and usage of Fragment and Replicate Strategy (FRS), Partition and Replicate Strategy (PRS) has also been discussed. This technique generally fragments the relations that exist only in the WHERE condition of the query.

Straube and Ozsu (1995), defines interface to an object manager where operations are executable elements of query execution plans. Parameters to the object manager interface are streams of tuples of object identifiers. The object manager can apply methods and simple predicates to the objects identified in a tuple. Two algorithms for generating such execution plans for queries expressed in an object algebra are presented. The first algorithm runs quickly but may produce inefficient plans. The second algorithm enumerates all possible execution plans and presents them in an efficient,
compact representation. The limitation of this proposed system is that it needs to
develop equivalence rules for tree of object operations.

Preethi Tiwari and Swati V Chande (2013), discusses various optimization
strategies used in query optimization. Author also proposes a new ant colony
optimization algorithm towards the optimization of distributed database queries. In this
paper, the different phases involved in distributed query processing have also been
discussed. The components of distributed query optimization have also been discussed.
It explores the usage of ant colony optimization algorithm for the queries in distributed
database especially when the size and complexity of the relations increases with the
number of parameters influencing the query. It was mainly designed to be easily adapted
to existing query optimizer that commonly use dynamic programming based algorithm.
The behavior of the system to other optimized algorithms was unexplored.

Mantu Kumar et al. (2012), proposes a cache based query optimization model. In
this model a cache is implanted between the local optimizer and local database.
Whenever a query is given to a local optimizer, the local optimizer first checks the
cache rather than fetching the data directly from the database. This method results in
saving huge amount of computation time as accessing a cache is faster than accessing
the database.

This model works on the basis of four different factors such as server distance,
server capacity, server load and current queue length to provide optimal node where
query should be executed. This method is useful only when we need to access a part of
the database. It has proved to be better option for optimizing queries in a homogeneous
database system and implementation in a heterogeneous database system has not been
explored.

Kumar P Mohan and Vaideeswaran (2012), proposes a novel method for query
optimization using heuristic based approach to evaluate the efficiency of a query search
in the database operations. In the proposed algorithm, a query is searched using the
storage file which shows an improvement with respect to the earlier query optimization
techniques. This proposed method has not been evaluated for distributed applications.
Satyanarayana et al. (2013), proposes a new dynamic query optimization algorithm which is based on greedy dynamic programming algorithm. This algorithm uses randomized strategies. This method reduces the execution cost of the queries and system resources used to some extent. This approach works well with distributed and centralized databases. It works in bottom-up way by building more complex sub-plans from simpler sub-plans until the complete plan is constructed. It exhibits dynamism in calculating query execution plans. In spite of taking great effort to accurately estimate the cost of alternative query plan, inaccuracy still exist due to the existence of dependency between the attributes. This study helped us to understand the importance of identifying the dependency prior to the generation of query plan.

Ridhi Kapoor (2013), focuses on computing and analyzing the performance of joins, semi joins and cartesian product through a comparative analysis of processing cost, communication cost and total cost in distributed database system. This analytical study on query cost estimation gives the clear view of effect of query optimization on processing cost, communication cost and total cost.

Abdelkader Hameurlain and Franck Morvan (2009), describes the evolution of query optimization methods from uniprocessor relational database systems to data grid systems through parallel, distributed and data integration systems. Here author points out a set of parameters to characterize and compare query optimization methods. It describes the mechanisms of query optimization methods with respect to the considered environments and their constraints. The general problem of query optimization has also been discussed. The limitation of this proposed system was centralized approach could not be scaled up due to the network bandwidth and latency issues.

Pankti Doshi and Vijay Raisinghani (2011), emphasis the need for developing query execution strategy or plan to minimize the cost of query processing. Here author reviews traditional optimization strategies like static and various dynamic strategies for non-autonomous distributed database system. It also analysis the suitability of these strategies for autonomous systems. However, it considers processing of all nodes in generating optimal plan which can result in high optimization cost.
CHAPTER 3

OVERVIEW OF QUERY OPTIMIZATION

Query optimization is a fundamental part of any database management system (DBMS). Efficient query optimization is one of the most important tasks of DBMS. To achieve the efficiency, we first need to examine the steps involved in query optimization process. In this section, we have discussed about the various stages that a user submitted query has to pass through during the optimization process.

3.1 Query optimization process

Query optimization is used for accessing the database in an efficient manner. It is an art of obtaining desired information in a predictable, reliable and timely manner. Formally defines query optimization as a process of transforming a query into an equivalent form which can be evaluated more efficiently. The essence of query optimization is to find an execution plan that minimizes time needed to evaluate a query. To achieve this optimization goal, we need to accomplish two main tasks. First one is to find out the best plan and the second one is to reduce the time involved in executing the query plan.

Kunal Jamsutkar et al. (2013) states a query passes through three different phases during the query processing in DBMS which are as follows:

- Parsing and translation
- Optimization
- Evaluation

Usually, user queries are submitted to DBMS as SQL queries. During the parsing and translation phase, the given query is translated into its internal form. In generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of the relations in the database and
so on. The system constructs a parse tree representation of the query, which it then translates into a relational algebra expression. For example let us consider the following SQL query:-

**Select Sno from Student where Sno='101'**

This query is then translated into either of the following relational algebra expressions as follows:-

\[ \sigma_{\text{Sno}} = '101' \left( \pi_{\text{Sno}} (\text{Student}) \right) \]

\[ \pi_{\text{Sno}} \left( \sigma_{\text{Sno}} = '101' (\text{Student}) \right) \]

After parsing and translation into relational algebra expression, the query is then transformed into a form which is usually query tree or graph that can be handled by the optimization engine. For the above example, the relational algebra expression can be represented as either query tree or query graph which are shown in the Figure 3.1.

![Figure 3.1 Query representation](image)

During the optimization phase, the optimization engine performs various analyses on the query data. It applies various rules to the internal data structures of the query to transform these structures into equivalent and efficient representation. It then generates valid evaluation plans based upon the rules applied. From the generated evaluation plans, the best evaluation plan to be executed is determined and passed onto the query execution engine.
The final phase in processing a query is the evaluation phase. During the evaluation phase, the best evaluation plan generated by the optimization engine is selected and then executed.

Avi Silbershatz et al. (2002) describes the steps involved in query processing are shown in Figure 3.2.

Deepak Sukheja and Umesh Kumar Singh (2011) states query processing and optimization process work together to execute any kind of queries. Query processing is concerned with execution of a query or refers to the activities involved in extracting data from a data warehouse. On the other hand, query optimization process deals with the efficiency of the query. It defines the execution plans, the strategy of execution of the query and chooses the best execution plan.
Kosmann (2000) states the query optimization is the process of identifying an efficient way in which the query could be executed with less time complexity to produce better results. In this process, when a query in a high level language is first submitted, it is first scanned and parsed to determine if the query consists of appropriate syntax. If the query passes the parsing checks for correct syntax, then it is converted into query tree or query graph. Here we determine different ways of representing a query which are then passed on to the query optimizer. The way by which a given query is optimized plays a vital role in the enhancement of query performance.

Sunitha Mahajan and Vaishali Jadhav (2012) states that the goal of the query optimizer is to find a reasonably efficient strategy for executing the query. It considers the possible query plans for a given input query and determines the most efficient query plan. Once the query optimizer has determined the execution plan, the code generator writes out the actual access routines to be executed. With an interactive session, the query code is interpreted and passed directly to the runtime database processor for execution. During this section, we estimate the various cost factors for executing each of the execution plans. The execution plan which results in least cost estimate is chosen as best optimal execution plans.

Alaa Aljanaby et al. (2005) in the query optimization process, user given query is first scanned, parsed and validated. The scanner identifies the language tokens such as SQL keywords, attribute names and relation names in the text of the query, whereas the parser checks the query syntax to determine whether it is formulated according to the syntax rules of the query language. An internal representation of the query is then created. A query expressed in relational algebra is usually called initial algebraic query and can be represented as a tree data structure called query tree. It presents the input relations of the query as leaf nodes of the tree and represents the relational algebra operations as internal nodes.

The next step is an optimization step that transforms the initial algebraic query using relational algebra transformation into other algebraic queries until the best one is found. A query execution plan is then founded which represented as a query tree includes information about the access method available for each relation as well as the algorithms used in computing the relational operations in the tree.
The next step is called code generator, where we generate code for the selected query execution plan. This code is then executed by the run time database processor to produce the query result. The run time database processor has the task of running the query code, whether in compiled or interpreted mode, to produce the query result. If a run time error results, an error message is generated by the run time database processor. Figure 3.3 shows the different steps of query processing.

![Figure 3.3 Query optimization process](image)

**Figure 3.3 Query optimization process**

Query optimization process becomes a complex task as query complexity increases with new applications. Significant research work has been done in developing efficient query optimization techniques for processing complex queries in a cost effective manner. Some of the popular optimization techniques have been discussed in the following section.

### 3.2 Query optimization techniques

Raghu Ramakrishnan and Johannes Gehrke (2003), query optimization is the process of selecting the most efficient query evaluation plan for a query. In this section
we describe the various query optimization techniques used to obtain efficient execution plan. The various query optimization techniques described are as follows: Heuristic optimization, Syntactical optimization, Cost based optimization and Semantic optimization.

3.2.1 Heuristic optimization

Kunal Jamsutkar et al. (2013) states that it is a rule based method of producing an efficient query execution plan. Since the query output of the standardization process is represented as a canonical query tree, each node of the tree maps directly to a relational algebraic expression. The function of heuristic query optimizer is to apply relational algebraic rules of equivalence to this expression tree and transform it into a more efficient representation. Use of relational algebraic equivalence rules ensures that no necessary information is lost during the transformation of the tree. When these steps have been accomplished, the efficiency of a query can be further improved by rearranging the remaining Select and Join operations so that they are accomplished with the least amount of system overhead.

Vishal Hatmode and Sonali Rangdale (2014), heuristics are used by the systems to reduce the number of choices that must be made in a cost based fashion. The steps involved in heuristic query optimization are as follows: initially deconstruct conjunctive selections into a sequence of single selection operations. Then move the selection operations down the query tree for the earliest possible execution. Execute first those selection and join operations that will produce the smallest relations. Later replace cartesian product operations that are followed by a selection condition by join operation. Deconstruct and move as far down the tree creating new projections where ever needed. Identify those sub trees whose operations can be pipelined and execute them using pipelining.

The central common feature of all heuristic optimization methods is that they start off with a more or less arbitrary initial solution, iteratively produce new solutions by some generation rule and evaluate these new solutions. Then eventually the best solution found during the search process is reported. In this method, the complexity depends merely on the cost for evaluating per candidate solution, on the number of
iteration and if applicable on the population size and the costs of administrating the population.

Heuristic optimization methods usually consider not only those new solutions that lead to an immediate improvement but also some of those that are knowingly inferior to the best solution found so far. Thus, the effectiveness of a heuristic optimization to produce good plan is based mainly on the effectiveness of its heuristic rules.

3.2.2 Syntactical optimization

Syntactical query optimization is an approach to query optimization that uses special properties of the query languages to transform a query into another one which has the same answer and can be processed more efficiently. Two syntactically different query expressions are said to be equivalent if their evaluation gives the same result for the same state of the knowledge base. The major goals of the syntactic query optimizer is to simplify the query expression applying transformation rules and to express the query in a convenient way.

Corlatan et al. (2014) states that syntactical optimization relies on the user's understanding of both the underlying database schema and the distribution of the data stored within the tables. All tables are joined in the original order specified by the user query. The optimizer attempts to improve the efficiency of these joins by identifying indices that are useful for data retrieval. This type of optimization can be extremely efficient when accessing data in a relatively static environment. Using syntactical optimization, indices can be created and tuned to improve the efficiency of a fixed set of queries. Problems occur with this optimization whenever the underlying data is fairly dynamic.

3.2.3 Cost based optimization

The motivation behind cost based optimization is to come up with the cheapest plans available for each SQL statement. The cheapest plan is the one that will use the least amount of resources. The main objective of cost based optimization is to estimate the cost of different equivalent query expressions and choose the execution plan with the lowest cost.
It mainly depends on two factors they are solution space and cost function. Solution space depends on the set of equivalent algebraic expressions and the cost function is equivalent to the summation of input/output cost, CPU cost and communication cost. It also depends on different distributed environments.

Dhande and Bamnote (2015), describes the steps involved in cost based optimization are as follows : parsing, transformation, implementation and plan selection based on cost estimates. Thus, in general the cost based optimization process involves generation of a number of reasonable query execution plans and then estimate the cost of each plan generated. Later execute the least cost plan based on the estimated cost.

Ridhi Kapoor (2013) states that the cost of a query plan depends on the size of the basic tables referenced as well as the size of the intermediate results. To estimate the size of the intermediate results ,we use the concept of selectivity factor. The selectivity factor roughly corresponds to the fraction of rows which are expected to satisfy a condition in the WHERE clause. A high selectivity means more rows of the table are selected and the cost is higher.

Jarke and Koch (1984), to perform cost based optimization, an optimizer needs specific information about the stored data. This information is extremely system dependent and can include information such as file size, file structure types, available primary and secondary indices and attributes selectivity. A realistic goal of a cost based optimizer is not to produce the optimal execution plan for retrieving the required data, but to provide a reasonable execution plan. A cost based query optimizer estimates the cost of query plans based on statistics maintained by the database management system and uses this information to choose a plan. The two main components of cost based query optimizer are query execution plan generator and plan cost estimator.

### 3.2.4 Semantic optimization

Shenoy and Ozsoyoglu (1989) states that a semantic query is a query pertaining to knowledge or data that is expressed purely on the basis of a common business vocabulary, without any reference to how or where the data is stored. A semantic query attempts to help a user to obtain or manipulate data in a database without knowing its detailed syntactic structure.
The term Semantic query optimization refers to the process of utilizing the integrity constraints in the optimization process. The underlying concept of semantic query optimization is that by harnessing the integrity constraints, a user's query can be transformed into a query which is syntactically different to the original but which will produce the same result for all states of the database and be more efficient to execute than the original query. Semantic query optimization results in the transformation of an input query into a semantically equivalent query. Two queries are said to be semantically equivalent if, for every state of the database they produce the same result.

Kumar P Mohan and Vaideeswaran (2012), describes the two major phases involved in semantic optimization. In the first phase, the optimizer locates applicable semantic knowledge and proposes a sequence of one or more reformulation operations that preserve the semantics of the query. It uses two forms of semantic knowledge which are semantic rules and range facts. During the second phase, it evaluates the proposed reformulations and applies the best reformulation based on a cost model of query execution.

Sree Kumar T Shenoy and Zehra Meral Ozsoyoglo (1989), a semantic query optimizer proposes reformulations depending on the applicable semantic rules. It operates on the premise that the optimizer has a basic understanding of the actual database schema. When a query is submitted, the optimizer uses its knowledge of system constraints to simplify or to ignore a particular query if it is guaranteed to return an empty result set. This technique holds great promise for providing even more improvements to query processing efficiency in future relational database systems.

The advantage of semantic optimization is that the optimizer can infer the information about intermediate data from semantic knowledge prepared prior to query execution time. It supports the extensibility of multi database systems because it minimizes the dependency on how individual sources execute a query. It helps in providing local optimization to subqueries as well as reducing unnecessary data transmission. The essential idea of semantic query optimization is to use semantic rules about data. It is not widely used in practice because it is difficult to encode useful semantic knowledge.
3.2.5 Parametric optimization

Parametric query optimization (Ioannidis 1990) assumes that a cost of a query plan depends on parameters whose values are unknown at optimization time. Such parameters can for instance represent the selectivity of query predicates that are not fully specified at optimization time but will be provided at execution time. Parametric query optimization associates each query plan with a cost function that maps from a multi-dimensional parameter space to a one-dimensional cost space.

Parametric query optimization attempts to identify at compile time several execution plans, each one of which is optimal for a subset of all possible values of the run time parameters. Parametric query optimization optimizes a query into a number of candidate plans, each optimal for some region of the parameter space. At run time, when the actual parameter values are known, the appropriate plan can be chosen. The goal of parametric query optimization is to generate all query plans that could be optimized for any of the possible parameter value combinations. This yields a set of relevant query plans. The advantage of parametric query optimization is that optimization is avoided at run time.

In parametric query optimization we associate a cost function with each query plan. This cost function describes the cost of the plan as function of multiple parameter whose values are not known at optimization time. The main aim of parametric optimization is to find a plan set that contains an optimal plan for each possible combination of parameter values. The parametric query optimization algorithm supports only one cost metric and assumes that the value domain of each parameter is known in advance.

In parametric optimization algorithm we associate query plans with cost functions instead of cost values. It associates query metric values with each query execution plan. It performs optimization over a pruned population using metric values on parameter space. Parametric query optimization heavily depends upon the randomized algorithm. The major disadvantage of parametric query optimization is that the overhead of pruning multiple plans for one query can be acceptable for centralized database systems, whereas it might not be acceptable for distributed database systems. It is due to the restriction in the size of the plan space in distributed database system.
4.1 Query representation

A query (Ramez Elmasri and Shamkant B Navathe 1994) is a language expression that describes data to be retrieved from a database. A query is a request for information from a database. Query results are generated by accessing relevant database data and manipulating it in a way that yields the requested information. Query like expressions can be used internally in a database management system, to check access rights, maintain integrity constraints and synchronize concurrent access correctly.

Queries can be represented in a number of forms. In the context of query optimization, an appropriate query representation form must fulfill the following requirements: it should be powerful enough to express a large class of queries and it should provide a well defined basis for query transformation. The internal representation may be a query tree or a query graph.

4.1.1 Query tree

At the end of query analysis phase, the high level query is transformed into some internal representation that is more suitable for processing. This internal representation is a kind of query tree. Pramanik and Vineyard (1988), a query tree is constructed using tree data structure that corresponds to the relational algebra expression. It is also known as relational algebra tree or operator tree or expression tree. It is a tree data structure in
which the input relations are represented as the leaf nodes and the relational algebra operations as the internal nodes. When the query tree is executed, first internal node operation is executed whenever its operands are available. Then the internal node is replaced by the relation resulting after the execution of the operation. The execution terminates when the root node is executed and the result of the query is produced. For example consider the following sample query:

```
SELECT P.NUMBER, P.DNUM, E.LNAME, E.ADDRESS, E.BDATE
FROM PROJECT AS P, DEPARTMENT AS D, EMPLOYEE AS E
WHERE P.DNUM = D.DNUMBER AND D.MGRSSN = E.SSN AND P.PLOCATION = 'STAFFORD';
```

A query tree representation of this query is shown in Figure 4.1. The relations Project, Employee and Department are represented by the leaf nodes P, E and D, while the relational algebra operations of the expression are represented by internal tree nodes. The symbol ∏ denotes Projection operation and represents Join operation.

![Query tree representation](image)

**Figure 4.1 Query tree representation**
The main components of query tree are as follows:

- **Root of tree** — represents result of query
- **Leaf node** — represents input relations of the query.
- **Internal nodes** — represents intermediate relation that is the output of applying an operation in the relational algebra.
- **The sequence of operations is directed from leaves to the root node.**

The query tree representation does not specify how to evaluate each operation in the query. To fully specify how to evaluate a query, each operation in the query tree is annotated with the instructions that specify the algorithm or the index to be used to evaluate the operation. The resultant tree structure is known as query evaluation plan or query execution plan or simply plan.

In general, query tree represents a specific order of operations for executing a query. It gives a good visual representation and understanding of the query in terms of the relational operations its uses. It also gives additional means for expressing query in relational algebra.

### 4.1.2 Query graph

Chiang Lee et al. (2001) it is also a data structure used for internal representation of query. The query graph representation does not specify an order on which operations to perform first. There is only a single graph corresponding to each query. Hence, a query graph corresponds to relational calculus expression. A query graph representation for the above example query is shown in Figure 4.2.

![Figure 4.2 Query graph representation](image-url)
In this representation, the relations (Project, Department and Employee) in the query are represented by relation nodes. These relation nodes are displayed as single circle. The constant values from the query selection (P.Plocation = "Stafford") are represented by constant nodes and are displayed as double circles. The Selection and Join conditions are represented by the graph edges, for example, D.MGRSSN = E.SSN and P.DNUM=D.DNUMBER in the query graph representation. Finally, the attributes to be retrieved are displayed in square brackets above each relation.

Graphs are used in query optimization (Jyoti et al. 2012) for the representation of queries or query evaluation strategies. Two classes of graphs can be distinguished which are object graphs and operator graphs. Object graphs contain the properties of the query result and are closely related to the relational calculus. Operator graphs describe an operator-controlled data flow by representing operators as nodes that are connected by edges indicating the direction of data movement.

The main components of a query graph are as follows:

- Relation nodes — represent relations by single circle
- Constant nodes — represent constant values by double circle
- Edges — represent relation and join conditions
- Square brackets — represent attributes retrieved from each relation.

4.2 Query Evaluation

Query evaluation is a final step in query processing. In this section, we present an overview of how queries are evaluated in a relational database management system. During the evaluation process the user given queries are translated into an extended form of relational algebra. Straube and Oszu (1995) states generation of query evaluation plan involves three steps which are as follows:

- Generating expressions that are logically equivalent to the given expression.
- Estimating the cost of each evaluation plan.
- Annotating the resultant expression in alternative ways to generate alternative query execution plans.
The first step is implemented by means of equivalence rules that specify how to transform an expression into a logically equivalent one. The first and third steps are interleaved in the query optimizer and second step is done in the background by collecting statistical information about the relations such as relation sizes and index depths to make a good estimate of the cost of a plan.

Sunita Mahajan and Vaishali Jadhav (2012) in general, query evaluation involves three main steps which are query compiling, query optimization and query execution. The steps involved in query evaluation process is shown in Figure 4.3.

![Figure 4.3 Query evaluation steps](image)

In query compilation step, the given SQL query is converted into logically equivalent one by using equivalence rules. The compilation process includes three
important steps that are query parsing, logical plan selection and physical plan selection. Logical plan represents different algebraic expressions and physical plan are generated from the logical plan by selecting algorithm for each operator and by selecting an execution order for these operators. In query optimization step, we estimate the result size for each of the physical plans generated and estimate the cost for each plan. Finally, we pick up the best plan for execution which is done in query execution step. A sequence of primitive operations can be used to evaluate a query which is known as query execution plan or query evaluation plan. Once the query plan is chosen, the query is evaluated with that plan and the result of the query is output.

Deepak Sukheja and Umesh Kumar Singh (2011) states that there are two ways of evaluating a query, namely, materialized evaluation and pipelined evaluation. Materialized query evaluation walks the parse or expression tree of the relation algebra operation and performs the innermost or leaf-level operations first. The intermediate result of each operation in the expression is evaluated one by one in an appropriate order and the result of each operation is materialized in a temporary relation which becomes input for subsequent operations. For example, consider this relational algebra expression:

$$\Pi_{P\text{name}, Email-id} (\sigma_{\text{Category} = "Novel"} (\text{Book} \bowtie \text{Publisher}))$$

This expression consists of 3 relational operations, Join, Select and Project. The query tree of this expression is:

![Figure 4.4. Operator tree for the expression](image)

32
In materialization approach, the lowest-level operations are evaluated first. The inputs to the lowest level operations are the relations in the database. For example, in figure, the operation $\sigma_{\text{Category}=\text{novel}}$ (Book) is evaluated first and the result is stored in a temporary relation. The temporary relation and the Publisher relation are then given as input to the next-level operation, that is, the Join operation. The Join operation then performs a Join on these two relations and stores the result in another temporary relation which is given as input to the Project operation which is at the root of the tree. The cost of materialized evaluation is the sum of the costs of the individual operations involved plus the cost of writing the intermediate results to the disk.

In the pipelined evaluation method (Karde and Thakare 2010), the operations form a queue and results are passed from one operation to another as they are calculated, rather than storing them in temporary relations. A pipeline is implemented as a separate process within the database management system. Each pipeline takes a stream of tuples from its inputs and creates a stream of tuples as its output. A buffer is created for each pair of adjacent operations to hold the tuples being passed from the first operation to the second one. Pipeline operation eliminates the cost of reading and writing temporary relations. For example, as soon as a tuple is generated from the select operation, it is immediately passed to the Join operation for processing. Similarly, tuple generated from the Join operation is passed immediately to the Projection operation for processing. The pipelined evaluation is better as it increases the query evaluation efficiency by reducing the number of temporary relations. It is much cheaper than materialization but it is not always possible.

Pankti Doshi and Vijay Raisinghani (2011) describes there are two ways of executing pipelines namely demand-driven and procedure-driven pipelines. In demand driven pipeline, the parent operation requires next tuple from its child operations, in order to output its next tuple. Since the tuples are generated lazily on demand, this technique is also known as lazy evaluation. On the other hand, in procedure-driven pipeline approach, each operations at the bottom of the pipeline produces tuples without waiting for the request from the next higher level operations. Each operations tuple then puts the output tuples in the output buffer associated with it. This output buffer is used
as input buffer by the operation at next higher level. This process continues until all the
tuples are generated. Since, the tuples are generated eagerly, this technique is also
known as eager pipelining.

4.3 Query Execution

To better understand the factors that affect query performance, first we need to
understand how the query processor executes queries. This section introduces one of the
most fundamental concept query execution. When a query is submitted to the database ,
the query optimizer evaluates possible plans for executing the query. The best possible
query execution plan is chosen finally for execution. The query execution plan tells
which operations to perform when executing the query. Queries are composed of
collection of operators. Every operator accepts one or two relation instances and returns
a relation instance. Thus, each query is composed of relational algebra expression.
Relational algebra is a procedural query language, which takes instances of relations as
input and yields instances of relations as output. It uses operators to perform queries.
The fundamental operations of relational algebra are as follows Select, Project, Set, Join
and Rename. These operations are discussed in the following section.

4.3.1 Select operation

It selects tuples that satisfy the given predicate from a relation. It is a unary
operation that denotes a subset of a relation. It is used to choose a subset of the tuples
from a relation that satisfies a selection condition, acting as a filter to retain only tuples
that fulfills a qualifying requirement. It is symbolized as σ (sigma). The notation used
for representing Select operation is σ<selection condition>(R), where σ represents Select
command, <selection condition> represents the condition for selection and (R)
represents the relation or the table from which we are making a selection of the tuples.
The two scan algorithms used to implement the selection operation are linear search
and binary search. In linear search, the systems scan each file block and test all records
to see whether they satisfy the selection condition. If the file is ordered on an attribute
and the selection condition is an equality comparison on the attribute, then we can use a
binary search to locate records that satisfy the selection.
4.3.2 Project operation

It is used to reorder, select and get rid of attributes from a table. It is used to select data of particular attributes from a single relation and discards the other columns. It is symbolized as $\prod$ (pi). The notation used for representing Project operation is $\prod<\text{attribute list}>(R)$, where $\prod$ represents Project command, $<\text{attribute list}>$ represents the attributes that we get from a relation and (R) represents the relation or table from which we want to choose the attributes. We can implement Project operation by performing projection on each tuple, which gives a relation that could have duplicate records and then removing duplicate records.

4.3.3 Set operation

The four standard Set operations are Union, Intersection, Set difference and Cartesian product. All these operations require two tables as its operands. It also requires that the operand tables must be union compatible with each other. We can implement these operations by first sorting both relations and then scanning once through each of the sorted relations to produce the result.

- **Union operation** – It performs binary union between two given relations and it is defined as $r \cup s = \{t \mid t \in r \text{ or } t \in s\}$, where $r$ and $s$ are either database relations or relation result set. It is used when we need some attributes that appear in either or both of the two relations. For a union operation to be valid the following conditions must hold. First one is $r$ and $s$ must have same number of attributes. Second one is attributes domains must be compatible. Then the last one is duplicate tuples are automatically eliminated.

- **Set difference operation** – The result of set difference operation is tuples, which are present in one relation but are not in the second relation. If $r$ and $s$ are database relations or relation result set then it is defined as $r - s$. It yields only the tuples that are in $r$ but not in $s$.

- **Intersection operation** – The result of intersection operation is tuples, which are present in both of the relations. If $r$ and $s$ are database relations or relation result set then it is defined as $r \cap s$. It yields only the tuples that are in both $r$ and $s$. 
• Cartesian product operation — It combines information of two different relations into one. It is defined as \( r \times s \), where \( r \) and \( s \) are relations and their output will be defined as \( r \times s = \{ qt \mid q \in r \text{ and } t \in s \} \). It is also referred as cross product or cross join. It creates a relation that has all the attributes of \( r \) and \( s \). It allows all the attainable combinations of tuples from \( r \) and \( s \) in the result.

4.3.4 Join operation

It is denoted by the symbol \( \Join \). It is used to compound similar tuples from two relations into single longer tuples. Every row of the first table is joined to every row of the second table. It refers to the process of calculating the optimal join order that is the order in which the necessary tables are joined, when executing the query. A join combines records from two tables based on some common information. The order of join is a key factor in controlling the amount of data flowing between each operator in the execution plan. The order in which the tables are joined determines the cost and performance of the query. There are several join operations such as index join, merge join, hash join, equi join etc. Join operation is one of the most important one in query optimization.

4.3.5 Rename operation

It is used to give a name to result or output of queries. It is symbolized by \( \rho \) (rho). It is defined as \( \rho S (B_1, B_2, \ldots, B_n) (R) \), where \( R \) is the relation or table from which the attributes are chosen. \( S \) is the new relation name. \( B_1, B_2, \ldots, B_n \) are the new renamed attributes. The rename operator returns an existing relation under a new name.
CHAPTER 5

PERFORMANCE TUNING FACTORS

Performance tuning is an ongoing process. This process requires continuous monitoring and improving database performance. One of the important tasks in database management is to improve the query performance. Thus, for improving the query performance, it is important to understand the factors that influence the performance of query optimization.

5.1 Query Optimizer

Satyanarayana et al. (2013) states that a query optimizer is an important component of database management systems, which attempts to determine the most efficient way to execute a query. The role of query optimizer is to produce query execution plans which represent an execution strategy of the query with minimum cost. It is the responsibility of the query optimizer to translate the user submitted query usually written in a non-procedural language into an efficient query evaluation plan which is then executed against the database. The optimizer considers the possible query plans for a given input query and attempts to determine which of those plans will be the most efficient.

Most query optimizer represents query plans as a tree of plan nodes. A plan node encapsulates a single operation that is required to execute the query. The nodes are arranged as a tree in which intermediate results flow from the bottom of the tree to the top. Given a relational algebra expression it is the job of the query optimizer to come up with a query evaluation plan that computes the same result as the given expression. Generally, the query optimizer cannot be accessed directly by users. Once queries are submitted to database server and parsed by the parser, then they are passed to the query optimizer where optimization occurs.
A desirable query optimizer is one where the search space includes plans that have the low cost, the costing technique is accurate and the enumeration algorithm is efficient. Pund et al. (2011), The optimizer may use several representations of a query during the life cycle of optimizing a query. The initial representation is the parse tree of the query and the final representation is an operator tree.

The overall architecture of a query optimizer is shown in Figure 5.1.

![Architecture of query optimizer](image)

**Figure 5.1. Architecture of query optimizer**

Initially, an analysis of query statement is done to detect lexical, syntactic and semantic errors. Then, internal representation of the query is done based on relational algebra. It explicitly represents the order in which operators are applied. After that
algebraic optimization is performed which eliminates the difference among different formulations of the same query. This step is usually independent of the data distribution. The output of this step is a query tree in canonical form. For the generated canonical query tree, cost based optimization is performed to select best execution plan by evaluating execution cost. The output of cost based optimization is to provide access program in executable format and the set of dependencies that specifies the conditions on which the validity of the query plan demands. Fan Yuanyuan and Mi Xifen (2010) an optimizer is a software module that performs optimization of queries on the basis of three main components: search space, search strategy and cost model.

5.1.1 Search space

The search space or solution space (Abdelkader Hameurlain and Franck Morvan 2009) is the set of all query execution plans that represent the input query. In other words, it refers to the generation of sets of all alternative and equivalent query execution plans of an input query by applying transformation rules such that they differ in the execution order of the operators. The specifications of the optimization search space are influenced by the input query and the nature of the investigated query tree. The search space for optimization depends on the set of algebraic transformations that preserve equivalence and the set of physical operators supported in an optimizer. A search space can be restricted according to the nature of the execution plans and the applied search strategy.

Taniar et al. (2004) defines a point in the solution space is one particular plan i.e. solution for the problem. A solution is described by the query tree for executing the join expression. The query tree itself is a binary tree that consists of base relations as its leaves and join operations as its inner nodes; edges denote the flow of data that takes place from the leaves of the tree to the root. The queries with a large number of join predicates make it difficult to manage associated search space which becomes too large. For this reason, many of the authors chose to eliminate bushy trees. This reduced space is called valid space. This valid space represents a significant portion of the search space which is the optimal solution. Thus, the size of the search space has significant impact on the nature of the execution plan as well as on the shape of the query tree.
5.1.2 Search strategies

Preeti Tiwari and Swati V Chande (2013) state that the search strategy explores the search space and selects the best plan. It is the central component of a query optimizer. It defines which plans are examined and in which order. We distinguish generally two classes of strategies for the query optimization. They are enumerative strategies or deterministic and randomized strategies.

5.1.2.1 Deterministic strategies

Swati Jain and Paras Nath Barwal (2014) specifies deterministic or enumerative strategies are based on the generative approach. They proceed by building plans, starting from base relations, joining one more relation at each step till the complete plans are obtained. They use the principle of dynamic programming and greedy algorithms. Shyam Padia et al. (2015) the basic dynamic programming for query optimization works in bottom-up way by building more complex sub-plans from simpler sub-plans until the complete plan is constructed. When constructing query evaluation plan through dynamic programming, equivalent partial plans are constructed and compared on some cost model. To reduce the optimization cost, partial plans that are not likely to lead to the optimal plan are discarded as soon as possible and the cheaper partial plans are retained and used to construct the full plan.

As an alternative to dynamic programming, greedy strategies have been proposed by Alaa Aljanaby et al. (2005). It builds plan using depth first search. Like dynamic programming, it consists of three phases, namely, access plan, join plan and finalize plan function in order to generate plan. Greedy strategies are the faster ones because they investigate very few processing trees. They start from the relation with the least cardinality. Different heuristics can be used in conjunction with a greedy strategy. Greedy strategies are also used to generate start solutions for randomized strategies.

A new class of enumerative strategies (Kossmann 2000) was then proposed called Iterative Dynamic Programming(IDP). The main idea of iterative dynamic programming is to apply dynamic programming several times in the process of optimizing a query.
This can be seen as a combination of dynamic programming and a greedy algorithm. It produces better plans than any other algorithm. It adapts to any optimization problem. It combines the advantage of both dynamic programming and greedy algorithm. For simple query it produces optimal plan and at the same time for complex query it produces the best plan close to the optimal.

5.1.2.2 Randomized strategies

The other class of strategy is the randomized strategies (Rosana SG Lenzelotte et al. 1993) that concentrate on searching the optimal solution around some particular points. They do not guarantee that the optimal plan is obtained but avoid the high cost of optimization in terms of memory used and time spent. Random strategies start generally with an initial execution plan which is iteratively improved by the application of a set of transformation rules. This algorithm begins by building a plan using greedy strategy and then tries to improve the plan by visiting its neighbors. A neighbor is obtained by applying a random transformation to a plan. Iterative improvement and simulated annealing are otherwise known as randomized strategies.

The iterative improvement algorithm (Swati Jain and Paras Nath Barwal 2014) starts at a random point and improves the solution by repeatedly accepting random downhill moves until it reaches a local optimization or until a stopping condition is met, at which point it returns the local minima with lowest cost. It performs several runs. Each run consists of improving one start state processing tree by applying transforms, until a local minimum is reached. In this method the number of runs is equal to the number of relations in the query.

It is a methodology based on a cyclic process of prototyping, testing, analyzing and refining a process. It starts with the known configuration of a system and proceeds by rearranging the configuration with the application of standard operations. This continues until a rearranged configuration is found that improves the cost function. Now this rearranged configuration becomes the new configuration and the process continues until no further improvement can be found.
Since this process get stuck with local optimum, it is customary to carry out the process several times starting from different randomly generated configuration and save the best result. It visits as many subspaces due to the choice of several start processing trees. It has been proved effective in centralized optimizers.

Simulated annealing (Vinod Gangwani and Ramteke 2013) is a refinement of iterative improvement. It is a probabilistic technique for approximating the global optimum of a given function in a large search space. It is often used when the search space is discrete. It accepts uphill moves with some probability, trying to avoid being caught in a high cost local minimum. The inner loop of this algorithm is called stage. Each stage is performed under a fixed value of a parameter $T$, called temperature which controls the probability of accepting uphill moves. Each stage ends when the algorithm is considered to have reached equilibrium. Then the temperature is reduced according to some function and another stage begins. The algorithm stops when the temperature is equal to zero.

From the above study, we observe that all search strategies have three common major aspects. Firstly, any search strategy needs to determine when the search should terminate. Secondly, the search strategy has to decide where to continue the search when several choices are given. Finally, the search strategy might discard certain choices thus not considering them for further exploration. Thus, search strategies are the driving force and the central component of a query optimizer.

5.1.3 Cost model

A cost model specifies the cost function expressed by arithmetic formulas that are used to estimate the cost of execution plan. Cost models are broadly used in query processing to drive the query optimization process. It helps to accurately predict the query execution time and schedule database query tasks. The main role of cost models is to estimate the time needed to run the query in a specific machine. Preeti Tiwari and Swati (2013), an optimizers cost model includes cost functions to predict the cost of operators, statistics and formulas to evaluate the size of intermediate results. The cost function can be expressed with respect to either total time or response time.
The total time is inclusive of local processing cost and communication cost. Local processing cost includes CPU time and the I/O cost. Communication cost involves time to transmit a data and the fixed time to initiate a message. Minimizing the total time implies that the utilization of resources increases thus increasing the system throughput. The response time is evaluated as the time elapsed between initiation of a query and time of receipt of the response to a query. Primarily, the cost of a query depends on the size of intermediate relation that are produced during execution and which must be transmitted over a network for a query operation. The main emphasis is on estimating the size of the intermediate relation so as to reduce the amount of data transfers and hence decrease the total cost and total time of the distributed query execution.

Alaa Aljanaby et al. (2005), the cost of a plan can be defined by the three components: total work, response time and memory consumption. Total work and response time are expressed in seconds and memory consumption in kilobytes. The first two components are used to express a trade-off between response time and throughput. The third component represents the size of memory needed to execute the plan. The cost function is a combination of the first two components. The plans which require more memory than available are discarded. A major difficulty in evaluating the cost is in assigning values to the weight of the first two components. These factors depend on the system state such as load of the system, the number of the queries submitted to the system and are ideally determined at run time.

<table>
<thead>
<tr>
<th>Card(R)</th>
<th>Number of tuples in relation R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width(R)</td>
<td>Size of one tuple of relation R</td>
</tr>
<tr>
<td>CPU</td>
<td>CPU speed</td>
</tr>
<tr>
<td>Network</td>
<td>Network speed</td>
</tr>
<tr>
<td>Packet</td>
<td>The size of packet</td>
</tr>
<tr>
<td>Send</td>
<td>The time for a send operation</td>
</tr>
<tr>
<td>Receive</td>
<td>The time for a receive operation</td>
</tr>
</tbody>
</table>
Swati Jain and Paras Nath Barwal (2014), the parameters, database schema or system parameters used in the cost model are shown in the Table 5.1. In this table, R refers to a base relation of the physical schema. Primarily, the cost of a query depends on the size of intermediate relations that are produced during execution and which must be transmitted over a network for a query operation at a different site. The emphasis is on the estimation of the size of the intermediate relations based on join orders and join methods, so as to reduce the amount of data transfers. Hence, this decreases the total cost and total time of the distributed query execution.

5.1.4 Different types of query optimizer

There has been much research work done in this field but we have considered only those works that had been a motivational factor for our research.

5.1.4.1 System R optimizer

System R optimizer, focuses on cost based optimization. It is based on dynamic programming. It attempts to obtain the best execution plan by successively joining relations using the cheapest access paths and join methods. After each join iteration the costliest plans were pruned leaving only the most eligible plan for the next phase. This optimizer works well for less than ten joins. Its cost of operations and result size are estimated using statistics. Here the optimizer uses statistics about relations and indices stored in the system catalog to estimate the cost of a query evaluation plan.

There are two parts to estimate the cost, one is estimating the cost of performing the operators and the other one is estimating the size of the result of a query block. Estimating the cost of operators requires knowledge of various parameters of the input relations such as the cardinality, number of pages and available indexes. System R defines a series of size estimation formulas which are used by the current query optimizers.

The enumeration algorithm for System-R optimizer demonstrates two important techniques namely dynamic programming and use of interesting orders. The essence of dynamic programming approach is based on the assumption that the cost model satisfies
the principle of optimality. The dynamic programming approach is significantly faster than the naive approach as it requires only $O(n^{2^n})$ plans need to be enumerated. The enumeration algorithm proceeds bottom-up. For a given query, system R identifies order of tuples that are potentially consequential to execution plans for the query. Here two plans are compared only if they represent the same expression as well as have the same interesting order. The system R framework demonstrates a simple mechanism to handle any violation of the principle of optimality. Despite the elegance of the System-R approach, the framework cannot be easily extended to incorporate other logical transformations.

The major disadvantage of system R optimizer is its space and time complexities are exponential which limits the join cardinality of a submitted query. The search space used by the optimizer is too large and only left-deep plan trees are considered.

5.1.4.2 Exodus optimizer

Graefe and Dewitt (1987) Exodus optimizer generator, was the first extensible optimizer framework using top-down optimization. It is designed to assist the database implementer in both creating a system for a new data model and in augmenting an existing system. It does not support single conceptual data model. The goal of Exodus is to build an infrastructure and tool for query optimization with minimal assumptions about the data model. Here operators and their semantics are left open thus allowing the database administrator to design and experiment with new data models.

It defines a general approach to the problem based on query algebras, data model, separation of logical and physical properties, extensive use of algebraic rules and it focuses on software modularization. Here the input data can be turned into machine code which ensures fast pattern matching. The input into the Exodus optimizer generator consists of a set of operators, a set of methods, algebraic rules for transforming the query trees and rules describing the correspondence between operators and methods. At run time each query is transformed into an operator tree by the user interface, optimized by the generated optimizer and then interpreted or transformed into a program.
The architecture of the Exodus optimizer generator enforces a modular and extensible design of the database implementers query optimizer code. In this the transformation and implementation rules are independent from one another and the property and cost functions are well defined. The main contribution of the Exodus optimizer were the optimizer generator architecture based on code generation from declarative rules, logical and physical algebras, the division of query optimizer into modular components and interface definitions for support functions to be provided by the database implementer.

However, the Exodus optimizer generator search engine was far from optimal. It required modifications for unseen algebra and their peculiarities. The organization of mesh data structure bought complexities in terms of both time and space. The random transformation expression in mesh resulted in significant overhead in reanalyzing existing plans. The ability to specify required physical properties together with the logical expression was absent in Exodus. Here a transformation is always followed immediately by a algorithm selection and cost analysis. For complex queries the cost is significantly higher because exodus optimizer and its search engine do not systematically analyze and exploit physical properties.

5.1.4.3 Starburst optimizer

Starburst optimizer, extends the System-R optimizer with an extensible and more efficient approach. It aims to bridge the gap between relational database management system and the application that would like to use them. It provides extended database support for applications, hardware architectures and devices. It provides support for adding new storage methods for tables, new types of access methods and integrity constraints. Starburst optimizer is capable of determining which table is the inner, which is the outer and which join methods to consider.

Starburst optimizer has two major components, the query language processor and the data manager. The duties of query language processor includes parsing the query, semantic analysis, choosing an execution strategy and producing the set of operations to implement the execution strategy. The services provided by the data manager includes
record management, buffer management, access path management, concurrency control and recovery. The Starburst optimizer consists of two rule based subsystems, query rewrite or query graph model and the plan optimizer. A query graph model is the internal semantic representation of a query. This optimizer eliminates redundancy and derives expressions that are easier for the plan optimizer to optimize in a cost based manner. The plan optimizer is a select-project-join optimizer consisting of a join enumerator and a plan generator. The join enumerator uses two kinds of join feasibility criteria to limit the number of joins. The plan generator uses grammar-like production rules to construct access plans for joins.

Starburst optimizer consists of two phases of optimization: the Query re-write or Query Graph Model (QGM) Optimization and the plan Optimization. In Query re-write phase of optimization rules are used to transform a QGM into another equivalent QGM. Here rules are modeled as pairs of arbitrary functions. The query re-write phase does not have the cost information available. This forces the module to either retain the alternative obtained through rule application or to use the rules in a heuristic way. In the plan optimization phase, given a QGM, an execution plan is chosen. Each plan has a relational description that corresponds to the algebraic expression. It represents an estimated cost and physical properties. The QGM optimizer eliminates redundancy and derives expressions that are easier for the plan optimizer to optimize in a cost based manner. The QGM optimizer is capable of sophisticated heuristic optimization. Thus it contributes to the efficiency of the Starburst optimizer. However, heuristics sometimes make incorrect decisions because they are based on only logical information, i.e., not based on cost estimates. Also, heuristics are hard to extend to more complicated queries containing non-relational operators.

5.1.4.4 Volcano optimizer

Volcano optimizer generator (Graefe and Mckenna 1993), provides efficient, extensible tools for query and request processing particularly for object-oriented and scientific database systems. It contributes in the improved extensibility with an efficient search engine based on dynamic programming and memorization. One major design goal of the volcano optimizer is to minimize the assumptions about the data model to be
implemented. It provides effective support for nontrivial cost models and for physical properties such as sort order. It combines dynamic programming with goal directed search and branch and bound pruning. It provides complete data model independence and more natural extensibility. It makes a distinction between logical and physical expressions.

The volcano algorithm is top-down driven. Here the cost is defined in much more general terms when compared to other optimizers. It is more extensible in particular with respect to search strategy. It performs exhaustive search for all queries with less than 1 mb of workspace. Sorting was modeled as an enforcer in volcano while it was implicit in the cost function for exodus. It permits heuristic transformation by suitable ranking and selection of move. In volcano, cost is an abstract data type for which all calculations and comparisons are performed by invoking functions provided by the optimizer implementer. One major design goal of the volcano optimizer is to minimize the assumptions about the data model to be implemented.

The volcano optimizer generator consists of two phases. In the first phase, all transformation rules are applied to create all possible logical expressions for a given query and all its sub trees. In the second phase, the actual optimization is performed by navigating within the network of equivalence classes and expressions. It applies implementation rules to obtain plans and determines the best plan. It uses two algebras called the logical and physical algebra. It generates optimizers that map an expression of the logical algebra into an expression of the physical algebra. It focuses on independent rules that ensure modularity.

The optimization goal in the volcano optimizer generator is to optimize a group or an expression with a cost limit and with required and excluded physical properties. In volcano optimizer all equivalence classes are completely expanded to contain all equivalent logical expressions before the actual optimization phase begins. It is more extensible in particular with respect to search strategy. It focuses on independent rules that ensure modularity. In Volcano optimizer, there was no guide specific search strategy. It does not facilitate even query specific rules for complex expressions.
5.1.4.5 Cascades optimizer

The Cascades optimizer, an extensible query optimization framework that resolves many short-comings of the EXODUS and Volcano optimizer generators. It achieves a substantial improvement over its predecessors in functionality, ease-of-use, and robustness without giving up extensibility and dynamic programming. It provides simple rules which requires minimal database implementer support. It provides facilities for schema and even query specific rules. It provides rule specific guidance and incremental improvement of estimated logical properties. It generates pattern that match an entire subtree rules, that map an input pattern to database supplied function.

The Cascades optimizer first copies the original query into the initial search space. The entire optimization process is then triggered by a task to optimize the top group of the initial search space, which in turn triggers optimization of smaller and smaller subgroups in the search space. Like the Volcano optimizer generator, Cascades begins the optimization process from the top group and is considered to use a top-down search strategy.

In Cascades optimizer, there is no separation into two phases. In Cascades optimizer, a group of expression is explored using transformation rules only on demand and it is explored only to create all members of the group that match a given pattern. Cascades optimizer provide a clean data structure abstraction and interface between database and optimizer.

5.1.5 Comparative study

Based on the above study, we conclude that the function of a query optimizer is to generate a set of plans, to estimate the cost of each plan, to calculate the cost of access path and to compare the cost of the plans. It then chooses the one with lowest cost. Each optimizer works with the aim of optimizing the query with least cost and execution time. In this section, we provide an analysis of various query optimizers developed for different types of queries based on some important criteria. The techniques used by each optimizer for achieving its goal have also been analyses.
Apart from these, strength and weakness of each optimizer have also been discussed in Table 5.2.

**Table 5.2 Comparative study of various optimizers**

<table>
<thead>
<tr>
<th>Criteria</th>
<th>System R</th>
<th>Exodus</th>
<th>Starburst</th>
<th>Volcano</th>
<th>Cascade</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Goal</strong></td>
<td>To provide a easy way to estimate the cost of query execution plan.</td>
<td>To build an infrastructure for query optimization</td>
<td>To build an extensible and efficient optimizer.</td>
<td>To minimize the assumptions about the data model to be implemented.</td>
<td>To provide a clean data structure abstraction and interface between database and optimizer.</td>
</tr>
<tr>
<td><strong>Focus</strong></td>
<td>Cost based optimization</td>
<td>Software modularization</td>
<td>Heuristic optimization</td>
<td>Independent rules that ensures modularity.</td>
<td>Exploring group of expression on demand.</td>
</tr>
<tr>
<td><strong>Technique Used</strong></td>
<td>Statistics and indexes</td>
<td>top-down driven</td>
<td>query rewrite , query graph model</td>
<td>top-down driven</td>
<td>dynamic programming</td>
</tr>
<tr>
<td><strong>Strength</strong></td>
<td>It demonstrates a simple mechanism to handle any violation of the principle of optimality.</td>
<td>It ensures fast pattern matching.</td>
<td>It eliminates redundancy and derive expressions that are easier to optimize.</td>
<td>Provides effective support for non-trivial cost models.</td>
<td>Provides extensible query optimization framework.</td>
</tr>
<tr>
<td><strong>Weakness</strong></td>
<td>It cannot be easily extended to incorporate logical transformations.</td>
<td>Overhead in reanalyzing existing plans.</td>
<td>Hard to extend complicated queries containing non-relational operators.</td>
<td>It always generates all equivalent logical expressions exhaustively.</td>
<td>Many features are provided only in rudimentary form.</td>
</tr>
</tbody>
</table>
From this study, we observe that each optimizer has its own way of achieving the efficiency in query optimization. Each optimizer has its unique method of reducing time complexity and overall execution cost. Each optimizer has been developed to overcome the difficulty that arose with the previous optimizers. Each optimizer had its own advantages and disadvantages. Based on this study, a comparative research work has been carried out based on some important criteria. This study helps to understand the need of a good optimizer so that enhancement of query performance can be achieved. Query optimizer plays a vital role in transforming user given query into an equivalent optimized query for execution. The goal and focus of each optimizer and the research motivation factor behind constructing each optimizer are taken into consideration for comparing the various optimizers.

We observe that the most common technique used by the optimizer are top-down driven method. The optimizer that uses top-down driven approach can use upper and lower bounds to avoid generating entire group of plans. Moreover, for the multi expressions generated top-down optimizer have the more potential to perform better than the bottom-up optimizer. The ultimate aim of each optimizer is to build an extensible and efficient optimizer.

This comparative study has helped us to understand the strength and weakness of each optimizer. It helps us to merge the positive aspects of the various optimizer and to construct an optimizer that can overcome the issues arises in each optimizer. The necessity and importance of building a uniform framework for query optimization which would be beneficial for developing an extensible and flexible optimizer to meet the growing demand of modern applications.

5.2 Query equivalence

One of the most fundamental aspects in query optimization is query equivalence. An equivalence rule (Garima Mahajan 2012) says that if the expressions of two forms are equivalent, then we can replace an expression of the first form by an expression of the second form or vice versa. The optimizer uses equivalence rules to transform expressions into other logically equivalent expressions.
The basic idea behind using equivalence rules is to find the alternate plans which are more cost efficient. We now list a number of general equivalence rules on relational-algebra expressions where we use $\theta$, $\theta_1$, $\theta_2$ and so on to denote lists of attributes and $E$, $E_1$, $E_2$ and so on to denote relational-algebra expressions.

- Conjunctive Selection operations can be deconstructed into a sequence of individual selections.
  \[ \sigma_{\theta_1 \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E)) \]

- Selection operations are commutative.
  \[ \sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E)) \]

- Only the last operation in a sequence of Projection operations are needed. The others can be omitted.
  \[ \Pi L_1(\Pi L_2(...(\Pi L_n(E))...)) = \Pi L_1(E) \]

- Selections can be combined with Cartesian products and theta joins.
  \[ \sigma_{\theta}(E_1 \times E_2) = E_1 \bowtie \theta E_2 \text{ and } \sigma_{\theta_1}(E_1 \bowtie \theta_2 E_2) = E_1 \bowtie \theta_1 \bowtie \theta_2 E_2 \]

- Theta-join operations are Commutative and Associative.
  \[ E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1 \text{ and } (E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \bowtie \theta_3} E_3 = E_1 \bowtie_{\theta_1 \bowtie \theta_3} (E_2 \bowtie_{\theta_2} E_3) \]

- Natural-join operations are Associative.
  \[ (E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3) \]

- The Selection operation distributes over the theta-join operation under the following conditions:
  \[ \sigma_{\theta_0}(E_1 \bowtie_{\theta_0} E_2) = (\sigma_{\theta_0}(E_1)) \bowtie E_2 \text{ and } \sigma_{\theta_1 \bowtie \theta_2}(E_1 \bowtie_{\theta_0} E_2) = (\sigma_{\theta_1}(E_1)) \bowtie (\sigma_{\theta_2}(E_2)) \]

- The Projection operation distributes over the theta-join operation under the following conditions:
\[
\prod_{L_1 U L_2} (E_1 \circ E_2) = (\prod_{L_1} (E_1)) \times (\prod_{L_2} (E)) \quad \text{and} \\
\prod_{L_1 U L_2} (E_1 \circ E_2) = \prod_{L_1 U L_2} ((\prod_{L_1 U L_3} (E_1)) \times (\prod_{L_2 U L_4} (E_2)))
\]

- Set operations Union and Intersection are commutative. \(E_1 \cup E_2 = E_2 \cup E_1\) and \(E_1 \cap E_2 = E_2 \cap E_1\)

- Set operations Union and Intersection are Associative.

\[(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3) \quad \text{and} \quad (E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)\]

- Selection operation distributes over the Union, intersection and Set difference operations.

\[\sigma_p (E_1 \cup E_2) = \sigma_p (E_1) \cup \sigma_p (E_2)\]

- The Projection operation distributes over the Union operation.

\[\prod_{L} (E_1 \cup E_2) = (\prod_{L} (E_1)) \cup (\prod_{L} (E_2))\]

We can use multiple equivalence rules one after the other on a query or on parts of a query. Query optimizers use minimal set of equivalence rules to reduce the number of ways an expression can be generated. A set of equivalence rules is said to be minimal if no rule can be derived from any combination of the others. We can also eliminate unneeded attributes by pushing projections based on equivalence rules. This results in reducing the size of the intermediate result. Given an expression, if any sub expression matches one side of equivalence rule, the optimizer generates a new expression where the sub expression is transformed to match the other side of the rule. This process continues until no more new expressions can be generated. This helps us to reduce space requirement significantly.

5.2.1 Usage of equivalence rules

We can utilize equivalence rules to transform the query tree as follows: break up any selection operation with conjunctive conditions into a cascade of selection operations. Move selection operations as far down the query tree as possible. Rearrange the leaf nodes of the tree, so that most restrictive selections are done first.
Most restrictive selections are the one that produces the fewest number of tuples. Combine a cartesian product with a subsequent selection operation into a Join operation, if the selection condition represents a join condition. This plays a vital role in enhancing the performance of the query in the following ways:

- It provides fast access for query processing and rule maintenance.
- It provides the ability to precomputes all chains of inferences that can occur with a query.
- It avoids delay during query processing.
- It enables query reformulation and modification so that the cost of the query execution is reduced.
- It examines the relative cost and helps to choose the least cost one.
- It helps to derive rules to operate anticipated queries.
- It helps in eliminating number of tuples that in turn reduces the cost.
- These rules can be used in systematic data analysis.
- It makes use of the domain values for attributes to recognize the redundant condition and discard the corresponding rules.

Thus equivalence rules are used to generate systematically all expressions equivalent to the given query and helps in determining which is more efficient to execute.

5.3 Indexing

Bertino and Kim (1989) describes indexing as a data structure technique used for efficiently retrieving records from the database files based on some attributes on which the indexing has been done. Indexes are used to quickly locate data without having to search every row in a database table every time a database table is accessed. Index is a performance tuning method that allows faster retrieval of records. We can use index without rewriting any queries. Indexes can be created using one or more columns of a database table. The utilization of indexes can dramatically reduce the execution time of various operations such as select and join. Let us review some of the types of index file structures and the roles they play in reducing execution time and overhead.
5.3.1 Different types of indices

There are many different types of indices (Corlatan et al. 2014) that provides easy way of retrieving data from any type of database. Here we have considered only some types of indices that had been a motivational factor for our research.

5.3.1.1 Dense index

In indexing, if there is one unique index entry corresponding to every record then it is called dense index. A dense index in databases is a file with pair of keys and pointers for every record in the data file. Every key in the file is associated with a particular pointer to a record in the sorted data file. Data file is ordered by the search key and every search key value has a separate index record. This structure requires only a single seek to find the first occurrence of a set of contiguous records with the desired search value. This makes searching faster but requires more space to store index records itself. In dense index, the number of entries in the index table is the same as the number of entries in the main table. In dense Index records contain search key value and a pointer to the actual record on the disk. Though dense index addresses quick search on any search key, the space used for index and address becomes overhead in the memory. Hence more space is consumed to store the indexes as the record size increases. Figure 5.2 shows the structure of dense index.

<table>
<thead>
<tr>
<th>Country</th>
<th>City</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>China</td>
<td>Beijing</td>
<td>3,705,386</td>
</tr>
<tr>
<td>Canada</td>
<td>Ottawa</td>
<td>3,855,081</td>
</tr>
<tr>
<td>Russia</td>
<td>Moscow</td>
<td>6,592,735</td>
</tr>
<tr>
<td>USA</td>
<td>Washington</td>
<td>3,718,691</td>
</tr>
</tbody>
</table>

Figure 5.2. Structure of dense index

5.3.1.2 Bitmap index

It is a special kind of database index that uses bitmaps. Bitmap indexes use bit arrays and answer queries by performing bitwise logical operations on these bitmaps. It includes rows that have null values, which can be useful for some type of SQL statements such as queries with aggregate function.
An example of bitmap index retrieval is shown in Figure 5.3.

Let us consider the given query to retrieve the data from the database be as SELECT * FROM Survey where Gender='Male' AND Marital status ='Single' AND Homeowner='Y'.

Bitmap indexes have traditionally been considered to work well for low-cardinality columns which have a modest number of distinct values. Bitmap indexes have significant space and performance advantage over other structures for query of such data. Their drawback is they are less efficient than the traditional B-tree indexes for columns whose data is frequently updated. They cannot be used to optimize more queries.
5.3.1.3 Clustered index

It determines the order in which the rows of the table will be stored on disk and it actually stores row level data in the leaf nodes of the index itself. It is an index where the leaf level of the index contains the actual data rows of the table. It sorts and stores the data rows of the table in order based on the clustered index key. Index is defined for one or more columns called index key. There can be only one clustered index per table and it can be combination of multiple columns.

A clustered index is an advantage when groups of data that can be clustered are frequently accessed by some queries. Here retrieval of data is much faster because the physical data stored on disk is sorted in the same order as the index. A clustered index can be a disadvantage because any time a change is made to a value of an indexed column, the subsequent possibility of re-sorting rows to maintain order is a definite performance hit. Figure 5.4 shows the structure of clustered index.

![Figure 5.4 Structure of clustered index](image)

5.3.1.4 Non-Clustered index

It does not contain the entire data row at the leaf level. It contains just the columns defined in the index and a pointer or key to the actual data row. It has a structure from the data rows. It contains the Non-clustered index key values and each key value entry has a pointer to the data row that contains the key value. It is useful for the columns that have repeated values. A non-clustered index columns always depends on the clustered columns on the database. A table can have multiple non-clustered indexes.
It is a special type of index in which the logical order of the index does not match the physical stored order of the rows on disk. The leaf node of a non-clustered index does not consist of the data pages. Instead, the leaf nodes contain index rows. Non-clustered index is generally slower than the clustered index due to bookmark lookup. Figure 5.5 shows the structure of non-clustered index.

![Figure 5.5 Structure of non-clustered index](image)

5.3.1.5 Function-based index

It computes the value of an expression that involves one or more columns and stores it in the index. The index expression can be an arithmetic expression or an expression that contain SQL function. It precomputes and stores the value of an expression. It improves the performance of queries that use the index expression.

Advantage of function based index is that it increases the number of situations where the database can perform an index range scan instead of full index scan. A function based index can be useful if a query search is based on an expression or a function. It can also help improve query performance for search conditions on Null values. The main disadvantage of function based index is that an optimizer can use this index only for cost based optimization not for rule based optimization. However, consider the following example:

```sql
CREATE INDEX Order_id on Orders ( NVL ( Shipdate, 'null' ) );
```
In this example, the function based index is created by including the null value rows. It allows null value to be indexed. The NVL function converts the null values to the string value 'null' that is stored in the index for applicable rows.

A function based index can be used to increase the performance of queries that use functions in the WHERE clause. It is helpful for user-defined ordering and decision support system applications. It allows to index the expression directly. It provides capability for case insensitive searches or sorts.

5.3.2 Guidelines for managing indices

- Create indices after inserting table data.
- Index the correct tables and columns.
- Order index columns for performance.
- Limit the number of indices for each table.
- Drop indices that are no longer needed.
- Understand deferred segment creation.
- Estimate index size and set storage parameters.
- Specify the table space for each index.
- Consider parallelizing index creation.
- Consider creating indices with no logging.
- Understand when to use unusable or invisible indices.
- Consider cost and benefits of rebuilding indices.
- Consider cost before disabling or dropping constraints.

5.3 Cost estimation

It is one of the difficult tasks in query optimization to accurately estimate the costs of alternative query plans. The cost of processing a query is expressed in terms of the total cost measure or the response time measure. The total cost measure is the sum of all cost components, whereas the response time measure is the time from the initiation of the query to the completion of a query. The cost of processing a query also includes CPU cost, I/O cost, access cost and communication cost.
The cost of a query execution plan (Monjural Alom et al. 2009) includes the following components:

**Access cost to secondary storage:** This is the cost of searching for, reading, writing data blocks of secondary storage.

**Computation cost:** This is the cost of performing in-memory operation on the data buffer during execution.

**Storage cost:** This is the cost of storing intermediate files that are generated during execution.

**Communication cost:** This is the cost of transferring the query and its result.

The cost of an operation is heavily dependent on its selectivity, that is, the properties of the input relations that form the outputs. The basic requirement in estimating the cost of an operation is to collect statistical summaries of input and output data for a given operator. Then we derive the cost for each of its operators. Once we have the costs for each of the operator nodes, the cost for the plan can be easily obtained by combining the costs of each of the operator nodes in the tree.

A fundamental technique used in cost estimation (Virk et al. 2012) is cardinality estimation. Optimizers take as input the cardinalities of tables at the leaves of a query tree and then use selectivity of operators in the tree to estimate the cardinality of the input to operators further up in the tree. To convert cardinalities to costs, optimization techniques use cost functions that estimate the cost of each execution plan. Cost functions can be expressed with respect to either the total time or the response time.

A cost function (Bizarro et al. 2009) must be capable of optimizing a given query in several dimensions. Cost function not only predicts the cost of operators but also formulas to evaluate the size of the result. The processing cost of query is evaluated in terms of the number of disk access I/O cost and CPU cost. The CPU cost is incurred when performing data operations in the main memory. The I/O cost can be minimized by using efficient buffer management techniques.

Cost estimation is an important factor that affects the performance of a query. So, it is necessary to take proper measures to accurately estimate the cost of alternative query plans.
5.5. Dependencies implied by SQL expressions

Majid Khan and Khan (2013) states that a dependency occurs in a database when information stored in the same database table uniquely determines other information stored in the same table. While determining the dependencies that hold in the final query result can be beneficial, such information can be exploited during the query optimization process for each subtree of the complete algebraic expression tree. To determine the set of dependencies that hold in an entire expression, one can simply recursively traverse the expression tree in a postfix manner and compute the dependencies that hold for a given operator once the dependencies of its inputs have been determined. The various dependencies that exist are as follows:

5.5.1 Functional dependency (FD)

It is an association between two attributes of the same relational database table. One of the attributes is called the determinant and the other attribute is called the determined. For each value of the determinant, there is associated one and only one value of the determined. If A is the determinant and B is the determined, then we say that A functionally determines B and graphically represent this as A → B. This can also be read as B is functionally dependent on A or A functionally determines B. The determinant of an functional dependency which determines all attributes of the table is super key. Consider for example, Project number (Pnumber) determines Project name (Pname) and Project location (Plocation) is shown as:

\[ \text{Pnumber} \rightarrow \{ \text{Pname}, \text{Plocation} \} \]

The determination of functional dependency is an important part of designing databases in the relational model and in database normalization. Functional dependencies are used to specify formal measures of the goodness of relational designs. It helps to determine the keys which are used to define normal forms for relations.

5.5.2 Multi-valued dependency (MD)

It occurs when two or more independent multi valued facts about the same attribute occur within the same table. It means that if in a relation R having A, B and C
as attributes, B and C are multi value facts about A, which is represented as A \( \rightarrow\!
\leftarrow \rightarrow \) B and A \( \rightarrow\!
\leftarrow \rightarrow \) C, then multi value dependency exist only if B and C are independent of each other. It is a full constraint between two sets of attributes in a relation. It requires that certain tuples be present in a relation. It is a special case of tuple generating dependency. It plays a vital role in fourth normalization form.

Multivalued dependency X \( \rightarrow\!
\leftarrow \rightarrow \rightarrow \) Y holds in a relation R, if whenever we have two tuples of R that agree on all the attributes of X, then we can swap their Y components and get two new tuples that are also in R. For example, Employee \( \rightarrow\!
\leftarrow \rightarrow \rightarrow \) Salary is read as Employee multidetermines Salary. Multivalued dependency are generalization of functional dependencies. They are used to test relations to determine whether they are legal under a given set of functional dependencies and multivalued dependencies. It helps to specify constraints on a set of relations.

5.5.3 Join dependency (JD)

It exists if a relation R is equal to the join of the projections X, Z where X, Y, Z are projections of R. A join dependency (Pramanik and Vineyard 1988) over a relation schema R[U] is an expression of the form \( \bowtie \) [x₁, ..., xₙ], where x₁ U... U xₙ = U. Then an instance I of R[U] satisfies \( \bowtie \) [x₁, ..., xₙ] if I = \( \pi \) x₁(I) ... \( \pi \) xₙ(I). In other words, an instance satisfies the join dependency if it is equal to the join of its projections on the sets of attributes x₁, ..., xₙ. It is important in connection with the decomposition technique for schema design and normalization. It is a generalization of multi-valued dependency.

5.5.4 Transitive dependency (TD)

It is a functional dependency which holds by virtue of transitivity. A transitive dependency can occur only in a relation that has three or more attributes. It occurs when a non-key attribute determines another non-key attribute. A functional dependency is said to be transitive if it is indirectly formed by two functional dependencies. This dependency helps us normalizing the data's in third normal form. Let A, B and C designate three distinct attributes in a relation. Then, A is a transitive dependency of C (A \( \rightarrow\!
\leftarrow \rightarrow \) C) if A is functionally dependent on B (A \( \rightarrow\!
\leftarrow \rightarrow \) B) and B is functionally dependent on C (B \( \rightarrow\!
\leftarrow \rightarrow \) C) but not on A.
5.5.5 Full functional dependency (FFD)

When determinant contains only a single field, the dependent is fully dependent on that determinant. This dependency is called full functional dependency (Savita et al. 2012). It is defined as attribute Y is fully functional dependent on attribute X, if it is functionally dependent on X and not functionally dependent on any proper subset of X. In other words a functionally dependent $X \rightarrow Y$ is a fully functional dependent if removal of any attribute A from X means that the dependency does not hold any more, i.e., for any attribute $A \in X$, $(X - \{A\})$ does not functionally determine Y.

5.5.6 Partial functional dependency (PFD)

It is a functional dependency where the determinant consists of key attributes, but not the entire primary key and the determined consists of non key attributes, It indicates that if A and B are attributes of a table, B is partially dependent on A if there is some attribute that can be removed from A and yet the dependency still holds. A functional dependency $X \rightarrow Y$ is a partial dependency if some attribute A can be removed from X and the dependency still holds, (i.e.) for some $A \in X$, $(X - \{A\}) \rightarrow Y$. 


CHAPTER 6

PROPOSED METHOD FOR QUERY OPTIMIZATION

Query optimization is the process of identifying an efficient way to execute the given query, so that with less time complexity we can obtain efficient results. In general, (Nikose et al. 2012) query optimization is performed by splitting the query into number of small query parts and execute them in different orders in such a way to reduce the time complexity. However, the query parts have different dependencies between them and the earlier methods do not handle this issue to reduce the time complexity and to improve the performance of query optimization.

To overcome this issue, we have proposed a query tree based method which identifies the query parts and their dependencies between them. Then, we generate dependency rules that produces a sequence for query execution. In this section we have discussed about how we generate dependency rules to perform query optimization. In this section, we have discussed about some of the methods that were taken for comparative analysis with the proposed method. Finally, we have also described a general framework for the query optimizer so that it would help in building an extensible optimizer.

6.1 Proposed method

After the analysis of various query optimization techniques and search strategies, we understand that query optimization explores different alternative query execution plans for the same query and chooses one of them as the best candidate for subsequent execution. To create different alternatives, to compare them and to select one of them in an efficient way makes query optimization complicated. However, there are many approaches that have been discussed earlier, but suffer from the problem of dimension and we could not find any efficient approach to perform query optimization. Thus, we propose a query tree based method which generates dependency rules to perform query optimization.
The proposed method generates the query tree using bottom-up approach. Each object from the query is identified first and then the relational objects are identified. The identified baseline attributes are placed at the leaf level and then from the input query, a distinct part of query is identified. For each distinct part of the query, we identify the objects or attributes of each sub query and constructed as a tree. From a generated query tree, we generate dependency rules and based on generated rules, we generate set of sequence of rules to be processed. For each sequence identified, we compute the query completion time and based on completion time, a least processing sequence will be selected as the most efficient one.

In general, the input query consists of N number of objects or databases or data sets. Each data set has its own schema and number of tables or relational objects where the original information is stored. Each relational object has number of properties or attributes which constructs the rows of a table. For any simple execution of a small query, the query execution module has to possess the schema of the relational object and has to identify which object is necessary to perform the execution of input query. The overall query execution time is the problem here and the query optimizer has to decide which part of the query has to be executed first. To overcome this problem we have proposed a method which helps us to generate an order of query execution, so that the time complexity involved in query execution can be minimized. The proposed query optimization technique has four stages, namely, Pre-processing, Query Tree Construction, Dependency Rule Generation and Query Optimization. We discuss each of the stages in detail in this chapter.

6.1.1. Algorithms and Running Example

In this section, we provide pseudo-code for the algorithms employed by the proposed optimizer model.

6.1.1.1 Algorithm for Pre-processing

An algorithm for pre-processing the given user query is shown in Figure 6.1 and the description of the notations used in the algorithm is given in Table 6.1. This algorithm gives the pseudo-code for preparing the query for optimization.
In this stage, the given user input query is split into number of small queries according to the presence of brackets, keywords, commas and functions which are identify with the help of database schema.

Input: Let input Query be represented as Q.
Output: Let the output Preprocessed Query Set be represented as PQS.

Step1: Read database Schema as S.
Step2: Parse input query Q into N sub-queries (where N represents no. of queries).

\[ PQS = \sum_{i=1}^{N} \left( \cdot \right) \in Q. \]

Step3: for each query Qi from each database schema Si

Identify Data table name Dt = \( \sum_{i=1}^{\text{size}(S)} S(i) \in Qi \)

Identify set of Attributes represented as As .

\[ As = \sum_{i=1}^{\text{size}(At(DT))} At(i) \in Qi \]

Identify the Conditional operator represented as Cop.

\[ Cop = \sum_{i=1}^{\text{size}(Cop)} (cop(i) \in Qi, cop(i),0) \]

Identify Relational operator represented as ReO.

\[ Reo = \sum_{i=1}^{\text{size}(Reo)} (Reo(i) \in Qi, Reo(i),0) \]

Identify functions mentioned Fun.

\[ Fun = \sum_{i=1}^{\text{size}(Fun)} (Fun(i) \in Qi, Fun(i),0) \]

End

Step4: Add all to Preprocessed Query Set PQS.

\[ PQS = \sum_{i=1}^{\text{size}(PQS)} (Dt, As, Cop, Reo, Fun) \]

Step5: Stop.

Figure 6.1 Algorithm for pre-processing

Table 6.1 Notations used in pre-processing algorithm

<table>
<thead>
<tr>
<th>Notations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dt</td>
<td>Data table name</td>
</tr>
<tr>
<td>At</td>
<td>Attributes of the data table</td>
</tr>
<tr>
<td>As</td>
<td>Set of attributes</td>
</tr>
</tbody>
</table>
6.1.1.2. Algorithm for Query tree construction

This algorithm illustrates the construction of query tree using the preprocessed query set. From the preprocessed query set, we identify the objects or attributes of each sub query. Based on identified sub queries, the optimizer generates root nodes which specifies the data object as the parent node. Algorithm for query tree construction is described in Figure 6.2 and the notations used in the algorithm are described in the given Table 6.2.

Input: Let the input be Preprocessed query set represented as PQS.
Output: Let the output be Query Tree represented as QT.

Step1: Create Root Node as Rn.
Step2: For each query Qi from PQS generate Attribute leafs.
Let Leaf node set denoted as LNS
\[ LNS = \prod_{i=1}^{\text{size}(\text{PQS})} \sum_{j=1}^{\text{size}(\text{PQS}(i)\text{(Att}))} \text{CreateNode(Att. Name)} \]
Create Parent Sibling node as PS and assign with integer.
Add All nodes of LNS to PS and LNS to Rn.
End.

Step3: For each leaf Li from Rn compute the value of
NOC = Identify number of attributes of child nodes in current data tables.
NOO = Identify number of attributes of child present in other data tables.
End.

Mdt = Identify data table containing maximum attributes.
Update value of Li with computed values.
Li = \{Li(Number), NOC, Mdt\}
End.

Step4: Stop.

Figure 6.2 Algorithm for query tree construction
Table 6.2 Notation description for query tree construction algorithm

<table>
<thead>
<tr>
<th>Notations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rn</td>
<td>Root node</td>
</tr>
<tr>
<td>Li</td>
<td>Each leaf node</td>
</tr>
<tr>
<td>LNS</td>
<td>Leaf node set</td>
</tr>
<tr>
<td>PS</td>
<td>Parent sibling node</td>
</tr>
<tr>
<td>Mdt</td>
<td>Data table containing maximum attributes</td>
</tr>
</tbody>
</table>

6.1.1.3 Algorithm for Dependency rule generation

This algorithm gives the pseudo-code for identifying the dependencies exist among the sub queries. Once the dependencies of its inputs have been determined, the dependency rules are generated using constructed trees and pre-processed query set. Identifying dependencies helps us to understand the relations that exist among the data's in the database schema. This understanding helps to reduce the complexity involved in retrieving data from a relational database. Algorithm description for dependency rule generation is shown in Figure 6.3 and the description of notations used in the algorithm is given in Table 6.3.

```
Input: Let the input be Preprocessed query set( PQS) and Query Tree( QT)
Output: Let the output be Dependency Rule Set (DRS).

Step1: Initialize the Dependency rule set (DRS).

Step2: For each leaf number create Sequence (Seq) randomly.
    Seq = \sum_{i=1}^{\text{size(leafs)}} \sum_{j=1}^{\text{size(QT)}} Create Pattern
    End.

Step3: For each sequence dependency rule DR are generated.
    DR = \sum \{NI, NDTR, NL\}
    End.
    DRS = DRS+DR.
    End.

Step4: Stop.
```

Figure 6.3 Algorithm for dependency rule generation
Table 6.3 Notations used in dependency rule generation algorithm

<table>
<thead>
<tr>
<th>Notations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DR</td>
<td>Dependency rules</td>
</tr>
<tr>
<td>NI</td>
<td>Number of Inputs</td>
</tr>
<tr>
<td>NDTR</td>
<td>Number of data table attributes</td>
</tr>
<tr>
<td>NL</td>
<td>Number of locks applied</td>
</tr>
</tbody>
</table>

6.1.1.4 Algorithm for Query Optimization

This algorithm gives the pseudo-code for computing the overall execution time of each dependency rule generated. This computation is based on the values of number of inputs required for each part of the query, number of data table attributes and number of locks the tables has applied for. Based on all these factors, we compute the query completion probability for each dependency rule identified. Then the most probable rule is selected for execution. Algorithm for query optimization is illustrated in Figure 6.4 and The notations used in the algorithm are described in the given Table 6.4.

Input: Let the input be Dependency Rule Set DRS.
Output: Let the output be Final resultant SEquence selected SEq.
Step1: Initialize Query Completion Probability Set QCPS.
Step2: For each rule Ri from DRS
   Compute Query completion probability Qcp.
   \[
   Qcp = N \times \log(Ri(Ni)) + (Ri(NL) \times (N \times \log(Ri(NDTR))))
   \]
   QCPS = ΣQCPS(i)+Qcp.
   End.
Step3: Choose the most probable Rule.
Step4: Return selected Dependency Rule Set DRS(i).
Step5: Stop.

Figure 6.4 Algorithm for query optimization
Table 6.4 Notation description for query optimization algorithm

<table>
<thead>
<tr>
<th>Notations</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEq</td>
<td>Final selected sequence</td>
</tr>
<tr>
<td>Qcp</td>
<td>Query completion probability</td>
</tr>
<tr>
<td>QCPS</td>
<td>Query completion probability set</td>
</tr>
<tr>
<td>Ri</td>
<td>Each generated rule</td>
</tr>
</tbody>
</table>

6.1.1.5 Example Illustration

To illustrate the performance of our implementation we use the following query as running example.

```sql
SELECT Job-id, AVG(Salary) FROM Employee GROUP BY Job-id HAVING AVG(Salary) < (SELECT MAX(AVG(Min-salary)) FROM Job WHERE Job-id IN (SELECT Job-id FROM Job-history WHERE Dept-id BETWEEN 50 AND 100) GROUP BY Job-id);
```

This query contains the following basic database:

**Employee**: { Emp-id, Fname, Lname, Email, Phno, Hire-date, Job-id, Salary }

**Job**: { Job-id, Job-title, Min-salary, Max-salary }

**Job-History**: { Job-id, Dept-id, Dept-name }

According to the proposed query optimization technique during the pre-processing stage the given input query is split into three subqueries based on the presence of brackets, commas, key words, relational expressions mentioned and functions. These three queries are nested sub query, a sub query and the outer query which are given as follows:

**Sub query 1**: SELECT Job-id, AVG(Salary) FROM Employee GROUP BY Job-id HAVING AVG(Salary) <

**Sub query 2**: SELECT MAX(AVG(Min-salary)) FROM Job WHERE Job-id IN

**Sub query 3**: SELECT Job-id FROM Job-history WHERE Dept-id BETWEEN 50 AND 100 GROUP BY Job-id
The database schema for the data tables used in the example query is shown in Figure 6.5. This database schema helps to logically group objects and depicts the relations exist among the objects.

**Figure. 6.5 Database Schema for the example query**

After generating the pre-processed query set, we identify the objects and attributes of each sub query which are as follows:

**Sub query1** : Data table -1; Data table name - Employee; Attributes used-2 ; Attribute names- Job-id, Salary ; Functions used- 1

**Sub query2** : Data table -1; Data table name - Job; Attributes used-1 ; Attribute names- Min-salary ; Functions used-2

**Sub query3** : Data table -1; Data table name - Job-History; Attributes used-1; Attribute name- Dept-id ;
Based on this information a query tree is generated using bottom-up approach. In bottom-up approach, the query solution tree is grown from the leaves to the root applying the rules from right to left. The primary advantage of this approach is that it implements a form of dynamic programming which eliminates a great deal of redundant computation. This non-procedural approach allows much more flexibility in program transformation and optimization. It is more effective in reducing the search space and thereby the time for deployment. The query tree generated for each sub query for the example query is shown in figure 6.6.

![Query tree for Sub queries](image)

**Figure 6.6. Query tree for Sub queries**

After the generation of query tree, we need to check whether any query part requires attributes contained in another data tables. We need to identify the dependencies that exist among the sub queries. This is done in the dependency rule generation stage. Based on the dependency of each query in the input query, we generate dependency rule which has number of the query and its dependency factor as a sequence. The dependency rule is generated based on the computation of total number of inputs, number of data table the attribute has and the number of constraints applied. For the example query, the dependency rule is generated as follows:

**Sub query1** : Table Used-Employee; Total no. of attributes in the table-7; Attributes used-Job-id, Salary; Functions used-1; Dependency -2; No. of constraints-1

**Dependency rule** : Employee==>7==>2==>2==>1
**Sub query2** : Table Used-Job; Total no. of attributes in the table-4; Attributes used-1; Functions used-2; Dependency -1; No. of constraints-1

**Dependency rule** : Job==4==1==1==1

**Sub query3** : Table Used-Job-History; Total no. of attributes in the table-3; Attributes used-1; Functions used-0; Dependency -0; No. of constraints-1

**Dependency rule** : Job-History==3==1==0==1

For each rule generated, we compute the overall execution time based on the values of number of inputs required for each part of query and number of data tables has the required attributes and number of locks the table has applied for. Based on all these factors, we compute the query completion probability for each dependency rule identified. The most probable rule is selected for execution. This working is done in the query optimization stage. For the example considered, the optimal sequence for execution is generated based on dependency rule are as follows:

```
SELECT Job-id FROM Job-History WHERE Dept-id BETWEEN 50 AND 100;
SELECT MAX(AVG(Min-Salary)) FROM Job WHERE Job-id IN (sub query3) GROUP BY Job-id;
SELECT Job-id, AVG(Salary) FROM Employee GROUP BY Job-id HAVING AVG(Salary) < (sub query1);
```

Thus, in general, the proposed method helps to find the optimal sequence for query execution. The dependency that exist among attributes within the data table is the main reason for the complexity involved in query execution. To find a solution for the existing complexity, we have shown a method to solve this issue and obtain the optimal sequence for query execution.

**6.2. Comparative analysis**

The proposed method produces efficient results and this has been proven by taking some of the already existing methods used for the enhancement of query optimization. Some of the methods that are taken for result study analysis are Linear Recursive method, Semantic Cache method and Combined Similarity method. These methods have been discussed so that we can understand the efficiency produced by the proposed method in query optimization.
6.2.1. Linear recursive method

This method of optimization involves rewriting queries, changing order of evaluation of relational operations and defining index structures to get equivalent non-recursive SQL queries that can be evaluated in less time. Carlos Ordonez (2010) mainly focuses on optimizing linear recursive queries in SQL with existing storage organization, indexing mechanisms and relational algebra transformation. Two classical algorithms Seminaive and direct was used to evaluate linear recursive queries. It mainly focuses on transitive closure of graph and optimizing SQL recursive view. The factors considered for performance are graph connectedness, recursion depth and data set size. An algorithm has been developed to evaluate a recursive query. Here optimization by reducing redundant intermediate results is also studied. The maintenance of recursive views through incremental insertions or deletions for the transitive closure is also discussed.

The main limitation of this algorithm is that it insists on setting a recursion depth threshold. This optimization method is applicable when a query has non-recursive join between the recursive view. In this method, each type of query is evaluated in different ways. For example early selection is used when there is a "WHERE" clause in query. Similarly, when working with Join, we use different strategies for recursive join and non-recursive join. However, when evaluating "GROUP BY" clause and the aggregate function, we require many iterations. In general, a query optimizer evaluates "SELECT/DISTINCT" with a different plan from "SELECT/GROUP BY". These SQL implementation did not use any specific data structures or internal database system features. To overcome this issues we have discussed a general procedure to be followed for optimizing the query.

6.2.2. Semantic based cache method

(Kumar P Mohan and Vaideeswaran 2012) provides semantic based cache mechanism techniques for optimizing user queries. It supports data and computation reuse, query scheduling and cache efficient utilization algorithm to improve evaluation process and to minimize the overall response time. Here the query is processed by splitting the user query into segments as selection, relation and conditional case and
processes with query matching technique. Then the query is rewritten if the query is partially related and rejected if it does not match.

In this method a set of all possible queries of a homogeneous database system is defined and stored as a semantic reference. Set of rules were defined with respect to the operators available in suitable database language preferred. Predicates are matched based on the predicate values. Here the fact table is employed in order to store the details of sub-query references. The response time is summed up as total of query shipping, total of query execution and total of result shipping. An efficient query matching algorithm is designed and implemented.

The drawback of this method is that the process is limited to only selection and projection. In case of join queries, the query is decomposed into sub-queries and then it is processed individually. For this, the dynamic hashing technique is used to get the final result. This method is effective for only homogeneous database system. When we process the query consisting of multi-operators, there arises complexity issues with the system implementation. This proposed system is not able to handle if more than one user request arises. It also finds difficult to manage the cache coherence and in-memory process during execution.

6.2.3 Semantic-based-combined-similarity-measure method

This method was developed (Saini Mayank et al. 2011) to improve the information retrieval efficiency. It proposes a new semantic based similarity measure in which each term can be phrase or a single word and the weight assigned to each term is based on its semantic importance considering each sentence. It uses this semantic similarity measure along with other standard similarity measure as Jacard and Cosine to form the semantic based-combined-similarity-measure. Semantic similarity refers to the similarity of two terms based on sameness of their meanings or their semantic contents. It uses structured knowledge extracted from the Wikipedia to compute semantic similarity. To compute semantic relatedness between two terms, firstly we extract the Wikipedia categories of each term. Then we use all these categories extracted as the child nodes of the category tree of Wikipedia.
In this Standard genetic algorithm has been used to optimize the weight given for each similarity measure. The various types of standard similarity measures used to form the semantic based-combined-similarity-measure are cosine similarity, Jaccard similarity and Euclidean distance measure which are discussed as follows:

- **cosine similarity**: When documents are represented as term vectors, the similarity of two documents corresponds to the correlation between the vectors. This is quantified as the cosine of the angle between vectors, that is the so-called cosine similarity. It is one of the most popular similarity measure applied to text documents.

- **Jaccard similarity**: It measures similarity as the intersection divided by the union of the objects. It is a similarity measure and ranges between 0 and 1. If it is 1 it means the two objects are the same and if it is 0 then it means they are completely different.

- **Euclidean Distance**: It is a standard metric for geometrical problems. It is the ordinary distance between two points and can be easily measured with a ruler in two or three dimensional space. It is widely used in clustering problems including clustering text.

The limitation of this method is the time consumption for calculating similarity between terms is more. It requires to read all the category nodes along the taxonomy until it reaches the predefined maximum path length. Similarly, the percentage of relevant information we get mainly depends on the semantic similarity matching function. Semantic similarity computing methods have to calculate the relatedness of two concepts though they don’t have the exact match.

The main usage of this similarity computing method is to compare query request to the collection of documents to obtain the semantic related document. The retrieval efficiency technique used in this method was the motivational factor for developing the query tree construction method in the proposed optimizer.
6.3 Improvement on the proposed method

The modern database maintains various schema and each has various relations. The information about any product has no limit in size and can be presented in a relational manner. For example, the information about a single human can be split into a number of categories like personal, official, financial and so on. The personal information is about the personal details about the person who has name, parent name, age and sex and so on. Further, the personal information can be split into the address and personal information. The address itself can be stored in a relational database and can be stored as a new entity. Similarly, the information about any object can be presented and stored in number of data tables where there is relation between the entities present in different tables of any database. Such relational entities stored in any database can be named as relational database.

The relational database (Kuldeep et al. 2014) helps organizing the data tuples in a more efficient manner where the retrieval also could be performed in a more strategic manner. The query produced by any user may access number of relational data tables where the result of any part of the query can become an input to the other part of the query. So in order to execute the query part, some of the other part of the query has to be completed or if two different part of the query access the same relational data table then one has to be wait till the other part of the query has to be finished. This introduces dependency in executing the query. Also, the input query can be split into number of small query parts, and the query optimization is performed according to the sequence of query execution. By executing the query parts in different sequences the time complexity will vary and to execute the query in a more efficient manner, the time complexity has to be minimized.

The proposed method generated the execution order of query to enhance the performance of query optimization by considering the dependencies among the attributes existing within a single table. This method did not consider the dependencies existing among the tables in a database. In some cases, there may exist a dependency where one table is dependent on other table records.
To overcome these issues and to improve the performance of our previous solution method, we propose a multi level relational mapping algorithm. This method identifies the objects and generates relational maps. From the relational maps, the method identifies the objects and entity of query. The use of relational map helps to identify the query dependency according to the object and to compute the dependency measure for each of the rules being produced. Finally, a subset of dependency rule is produced as a result. This method improves the performance of rule generation and improves the performance of query optimization by scheduling the execution of query parts efficiently.

A relational mapping (Kuldeep et al. 2014) is an association among entities. The mapping process shown in Figure 6.7, involves creating tables for each entity. The entity attributes should become fields of tables with their respective data types. Then we have to declare primary key for each table.

![Diagram](image)

**Figure 6.7 Relational mapping process**
A relational mapping describes how the conceptual model is to be mapped to the relational model. A conceptual model abstractly represents the database design. A relational model concretely represents the details of the database design including the relations and their attributes. The purpose of the relational model is to provide a declarative method for specifying data and queries. The main idea behind relational model is to describe a database as a collection of predicates over a finite set of predicate variables. It specifies that the tuples of a relation have no specific order and that the tuples in turn impose no order on the attributes. On the other hand, conceptual model is a model made of the composition of concepts which are used to understand or simulate a subject the model represents. It is also known as domain model.

During relational mapping we need to be concerned with two categories of object relationship. The first category is based on multiplicity and it includes three types:-

- One-to-one relationship - It is a relationship where the maximum of each of its multiplicities is one.
- One-to-Many relationship - It is also known as Many-to-One relationship. It occurs when the maximum of one multiplicity is one and the other is greater than one.
- Many-to-Many relationship - It is a relationship where the maximum of both multiplicities is greater than one.

The second category is based on directionality and it includes two types:-

- Uni-directional relationship - It exists when an object knows about the object it is related to but the other object do not know of the original object.
- Bi-directional relationship - It exists when the object on both end of the relationship know of each other.

6.3.1 Multi level relational mapping based dependency rule generation

The entire process has been presented in four stages namely Preprocessing, Relational Map Generation, Dependency Measure Computation, Multi Level Relational Mapping and dependency rule generation.
The Figure 6.8, shows the architecture of multi level relational mapping approach for dependency rule generation and its functional components.

Figure 6.8. Architecture of multi level relational mapping

The multi level relational mapping approach reads the input query and parse them to identify the relational objects first. Then the method splits the input query into number of subset queries. Then for each input query the method performs the relational mapping to identify the number of relations a single query has. Finally the method performs the multi level relational mapping to produce the dependency rule. We explain each of the stages involved in this process in detail in this section.

6.3.1.1 Multi level Preprocessing

Preprocessing is the process of preparing the input query to generate the dependency rules. First the input query is read and the method reads the schema and
identifies the set of all relational objects. Once the relational objects are identified then the method splits the input query into a number of query partitions based on the occurrence of key words from the input query. Identified relational objects and the query set will be given as a result. Algorithm for preprocessing is shown in Figure 6.9 and the notations are described in the Table 6.5.

**Input:** Let the input be Query Q and Schema S.

**Output:** Let the output be Object set Os and query set Qs.

**Start**

Identify the relational objects from the input query Q.

\[ Os = \sum_{i=1}^{size(S)} S(i) \in Q \]

Generate query set Qs.

\[ Qs = \int_{i=1}^{size(Os)} \int_{j=1}^{size(keywords)} Split(Q.\ occurrence(keyword(j))) \]

**Stop.**

**Figure 6.9 Algorithm for multi level preprocessing**

**Table 6.5 Notations used in multi level preprocessing algorithm**

<table>
<thead>
<tr>
<th>Notations</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>Input query</td>
</tr>
<tr>
<td>S</td>
<td>Database schema</td>
</tr>
<tr>
<td>Os</td>
<td>Object set</td>
</tr>
<tr>
<td>Qs</td>
<td>Query set</td>
</tr>
</tbody>
</table>

### 6.3.1.2 Multi level relational mapping

At this stage, the method reads the set of all relational objects being identified. For each object identified, the method reads the database schema which defines how the data are organized and how the relations among them are associated. Itformulates all the constraints that are to be applied on the data.
It helps to identify the relations a single object has with others. We describe the algorithm used for this process in Figure 6.10 and the notations used in Table 6.6.

**Input:** Object set Os, Schema S.

**Output:** Relational Map Set Rms.

**Start**

For each object O_i from Os

Initialize relational map Rmi.

Identify the attribute set belongs to O_i

\[ As = \sum_{i=1}^{size(O)} \text{Attr} \in S(O_i) \]

For each attribute A_i from As identify the relations.

\[ Rmi = \sum(\text{Relations} \in Rmi) \cup \text{Relation}(A_i) \]

RAS = \sum Attr \in (S(R_i))

As = \sum(A_i \in As) \cup \sum Atrr \in RAS

End

Rms= \sum(Rmi \in Rms) \cup Rmi

End

**Stop**

**Figure 6.10. Algorithm for multi level relational mapping**

**Table 6.6 Notations used in multi level relational mapping algorithm**

<table>
<thead>
<tr>
<th>Notations</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rmi</td>
<td>Relational map</td>
</tr>
<tr>
<td>O_i</td>
<td>Each object</td>
</tr>
<tr>
<td>A_i</td>
<td>Each attribute</td>
</tr>
<tr>
<td>R_i</td>
<td>Each rule</td>
</tr>
<tr>
<td>Rms</td>
<td>Relational map set</td>
</tr>
<tr>
<td>RAS</td>
<td>Relational attribute set</td>
</tr>
</tbody>
</table>
For each object, there may be a number of relations present in the database schema, the method identifies such schemas. The method does not stop at a single level, but identifies the relations iteratively in order to find the other objects with the help of relational mapping. A relational mapping helps to transform object data members to relational database fields. It helps to map an object model to relational data model. Identified relational mapping will be used in computing the dependency measure. In this algorithm we initially identify the attributes for each object and form an attribute set. Then, we identify the relations that exist among the attributes in the attribute set and construct a relational attribute set. Later with the technique of relational mapping, we identify the relations present in each part of the query and add to the relational map set.

### 6.3.1.3 Dependency measure computation

The dependency measure computation is performed using the relational maps being generated at the previous stage and the query hierarchy. A query hierarchy is built upon a parent child relationship within the same table or view. A query hierarchy helps us to retrieve records from a table by their natural relationship. It also takes into consideration the number of relational objects being shared by two different queries. The dependency measure represents the depth, the part of query depends on another part of the query. Algorithm for computing dependency measure is shown in Figure 6.11 and the notations used in the algorithm are described in Table 6.7.

**Figure 6.11. Algorithm for dependency measure computation**

<table>
<thead>
<tr>
<th>Input: Query part Qp, Query Q, Relational Map Set Rms.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: Dependency Measure Dm.</td>
</tr>
<tr>
<td>Start</td>
</tr>
<tr>
<td>For each query part other than Qp</td>
</tr>
<tr>
<td>Identify the query hierarchy Qh.</td>
</tr>
<tr>
<td>$Q_h = \int_{i=1}^{size(Rms)} \sum Qs(i).index &gt; Qs.index(Qp)$</td>
</tr>
<tr>
<td>Compute number of relational dependent objects.</td>
</tr>
<tr>
<td>$NRD = \int_{i=1}^{size(Rms)} \sum Rms(i) \in \sum Rmi(Attr)$</td>
</tr>
<tr>
<td>Compute Dependency measure $Dmi = NRD/size(Rms) * Qh$</td>
</tr>
<tr>
<td>Add to Dm.</td>
</tr>
<tr>
<td>End</td>
</tr>
<tr>
<td>Stop</td>
</tr>
</tbody>
</table>
In this algorithm, for each query part, we identify the query hierarchy. Then, with the help of query hierarchy, we compute the number of relational dependent objects. This computation helps to compute the dependency measure. The dependency measure is computed by dividing the number of relational dependent objects by the size of the relational map set. Then the result obtained from this computation is multiplied by the number of query hierarchy. The resulting value is added to compute the dependency measure. This algorithm computes the dependency measure for each part of the query and based on the value of dependency measure, the dependency rule is generated.

Table 6.7 Notations used in dependency measure computing algorithm

<table>
<thead>
<tr>
<th>Notations</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>Input Query</td>
</tr>
<tr>
<td>Qp</td>
<td>Query part</td>
</tr>
<tr>
<td>Rms</td>
<td>Relational map set</td>
</tr>
<tr>
<td>Dm</td>
<td>Dependency measure</td>
</tr>
<tr>
<td>Qh</td>
<td>Query hierarchy</td>
</tr>
<tr>
<td>NRD</td>
<td>Number of relational dependent object</td>
</tr>
</tbody>
</table>

6.3.1.4 Multi level dependency rule generation

In this stage the dependency existing in query parts are identified. The dependency existing between the query parts in multiple levels are also identified. Then the dependency rule generation is performed with the support of all the above mentioned processes. The dependency rules are generated according to the multilevel dependency. For each part of the query, the method generates the dependency rule using the other part of the query and the computed dependency measure. It then generates the different sequences for query execution. For each part of the query, the method computes the dependency measure and based on the value of all the measures, the method generates the dependency rule which can be used to execute the query submitted. In the previous method, we identified the dependencies existing in the query parts within the data tables, whereas in this method, we consider the dependencies exist in the multi levels of query parts. Here, we consider the dependencies among the attributes present in the different data tables.
The algorithm for multi level dependency rule generation is shown in Figure 6.12 and the notations used in the algorithm are described in Table 6.8.

The above discussed algorithm computes the dependency rule and based on the value of cumulative dependency measure a final sequence will be selected to perform the query execution.

**Table 6.8 Notations used in multi level dependency rule generation algorithm**

<table>
<thead>
<tr>
<th>Notations</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qs</td>
<td>Query set</td>
</tr>
<tr>
<td>Qp</td>
<td>Query part</td>
</tr>
<tr>
<td>DR</td>
<td>Dependency rule</td>
</tr>
<tr>
<td>Dm</td>
<td>Dependency measure</td>
</tr>
<tr>
<td>CDM</td>
<td>Cumulative dependency measure</td>
</tr>
</tbody>
</table>
6.4. Structure of proposed query optimizer

We propose a general procedure for optimizing a query to reduce the complexity and this is described as follows: - The first and foremost step is to find internal query representation into which user queries can easily be mapped. The second step is to apply logical transformations to the query representation that standardize the query. The next step is to map the transformed query into alternative sequences of elementary operations to generate access plan. Then, we compute the overall cost for each access plan. Finally, we choose the cheapest one and execute it. Based on this general procedure, we have designed a framework for query optimizer model which can incorporate the changing requirements of a growing database.

In this proposed model, we have four modules, namely, Creator, Transformer, Plan Generator, Evaluator and Decider. The functions of various modules are described as follows. The module Creator takes as its input the user query and represents it as a query tree structure in which leaf nodes of the tree contain nodes that access a relation and internal nodes that contain relational operators. The edges of a tree represent data flow from bottom to top i.e., from the leaves which correspond to reading data in the database to the root. Internal representation of the query namely query tree structure can be constructed by using this module. It can also be defined as a process of successively building join nodes. It includes connecting to a new node by adding a relation. A query tree is said to be complete when its root join node involves all the operand relations of the input query.

The module Transformer enables us to describe the processing tree in a syntactical way with the help of join operators. Given a relational-algebra expression, it is the job of the optimizer to come up with a query evaluation plan that computes the same result as the given expression. It is done by the module Plan Generator. In this module, the logical operators in a query tree are replaced by the physical operators that can be implemented. It generates access plan that specifies how to evaluate the query. Each plan has an execution cost associated with it.
The main aim of query optimization is to choose the most efficient way of implementing the relational-algebra operations at the lowest possible cost. It is one of the difficult tasks in query optimization to accurately estimate the costs of alternative query plans. This is done by the module Evaluator. The Figure 6.13 shows the framework of proposed query optimizer.

![Diagram of proposed query optimizer]

**Figure 6.13 Framework of proposed query optimizer**

After estimating the cost of each evaluation plan, it evaluates the cost of processing a query. The cost of processing a query is expressed in terms of the total cost measure or the response time measure. The total cost measure is the sum of all cost components. The response time measure is the time from the initiation of the query to the completion of a query. After estimating the cost for each plan with the help of cost function, it is the role of the module decider to determine which access plan to be implemented to obtain the best efficiency.
CHAPTER 7

EXPERIMENTAL RESULTS

7.1 Experimental setup

In this section, we present an experimental study to show the performance of our proposed approach.

7.1.1 Dataset Description

We used the synthetic dataset, HR employee database for the experimental study. This dataset was used as source tables. The code implementation of the optimizer framework was done using the Java Net Beans IDE (Integrated Development Environment) 7.0. Net Beans IDE is an open source project dedicated to enable software development products quickly. The employee database consisting of 2000 tuples was used as the input relation for our experiments and the schema for the database is given in the figure 7.1.

![Database Schema](image)

Figure 7.1. Employee Database Schema
The data tables used in the database are described as follows:

**Employee** : { Employee_ID, First_Name, Last_Name, Email, Phone_Number, Hire_Date, Job_ID, Salary, Manager_ID, Department_ID }

The Employee table with sample rows are shown in Table 7.1.

### Table 7.1 Employee Table

<table>
<thead>
<tr>
<th>Employee_ID</th>
<th>First_Name</th>
<th>Last_Name</th>
<th>Email</th>
<th>Phone_Number</th>
<th>Hire_Date</th>
<th>Job_ID</th>
<th>Salary</th>
<th>Manager_ID</th>
<th>Department_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Steven</td>
<td>King</td>
<td>SKING</td>
<td>515.123.45</td>
<td>6/17/1987</td>
<td>AD_PRES</td>
<td>2400</td>
<td>0</td>
<td>90</td>
</tr>
<tr>
<td>101</td>
<td>Neena</td>
<td>Kochhar</td>
<td>NKOC</td>
<td>515.123.45</td>
<td>9/21/1989</td>
<td>AD_VP</td>
<td>1700</td>
<td>100</td>
<td>90</td>
</tr>
<tr>
<td>102</td>
<td>Lex</td>
<td>De Haan</td>
<td>LDEH</td>
<td>515.123.45</td>
<td>1/13/1993</td>
<td>AD_VP</td>
<td>1700</td>
<td>100</td>
<td>90</td>
</tr>
<tr>
<td>103</td>
<td>Alexander</td>
<td>Hunold</td>
<td>AHU</td>
<td>590.423.45</td>
<td>1/3/1990</td>
<td>IT_PROG</td>
<td>9000</td>
<td>102</td>
<td>60</td>
</tr>
<tr>
<td>104</td>
<td>Bruce</td>
<td>Ernst</td>
<td>BERN</td>
<td>590.423.45</td>
<td>5/21/1991</td>
<td>IT_PROG</td>
<td>6000</td>
<td>103</td>
<td>60</td>
</tr>
<tr>
<td>105</td>
<td>David</td>
<td>Austin</td>
<td>DAUSTIN</td>
<td>590.423.45</td>
<td>6/25/1997</td>
<td>IT_PROG</td>
<td>4800</td>
<td>103</td>
<td>60</td>
</tr>
</tbody>
</table>
Department : { Department_ID, Department_Name, Manager_ID, Location_ID }

The Department table with sample rows are shown in Table 7.2.

**Table 7.2 Department Table**

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
<th>MANAGER_ID</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Finance</td>
<td>108</td>
<td>1700</td>
</tr>
<tr>
<td>90</td>
<td>Executive</td>
<td>100</td>
<td>1700</td>
</tr>
<tr>
<td>80</td>
<td>Sales</td>
<td>145</td>
<td>2500</td>
</tr>
<tr>
<td>70</td>
<td>Public Relations</td>
<td>204</td>
<td>2700</td>
</tr>
<tr>
<td>60</td>
<td>IT</td>
<td>103</td>
<td>1400</td>
</tr>
</tbody>
</table>

Jobs : { Job_ID, Job_Title, Min_Salary, Max_Salary }

The Jobs table with sample rows are given in Table 7.3.

**Table 7.3 Jobs Table**

<table>
<thead>
<tr>
<th>JOB_ID</th>
<th>JOB_TITLE</th>
<th>MIN_SALARY</th>
<th>MAX_SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD_PRES</td>
<td>President</td>
<td>20000</td>
<td>40000</td>
</tr>
<tr>
<td>AD_VP</td>
<td>Administration Vice President</td>
<td>15000</td>
<td>30000</td>
</tr>
<tr>
<td>AD_ASST</td>
<td>Administration Assistant</td>
<td>3000</td>
<td>6000</td>
</tr>
<tr>
<td>FI_MGR</td>
<td>Finance Manager</td>
<td>8200</td>
<td>16000</td>
</tr>
<tr>
<td>FI_ACCOUN NT</td>
<td>Accountant</td>
<td>4200</td>
<td>9000</td>
</tr>
</tbody>
</table>
Jobs_History : { Employee_ID, Start_Date, End_Date, Job_ID, Department_ID }

The Jobs_History table with sample rows are given in Table 7.4

### Table 7.4 Jobs_History Table

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>START_DATE</th>
<th>END_DATE</th>
<th>JOB_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>1/13/1993 12:00:00 AM</td>
<td>7/24/1998 12:00:00 AM</td>
<td>IT_PROG</td>
<td>60</td>
</tr>
<tr>
<td>101</td>
<td>9/21/1989 12:00:00 AM</td>
<td>10/27/1993 12:00:00 AM</td>
<td>AC_ACCOUN</td>
<td>110</td>
</tr>
<tr>
<td>101</td>
<td>10/28/1993 12:00:00 AM</td>
<td>3/15/1997 12:00:00 AM</td>
<td>AC_MGR</td>
<td>110</td>
</tr>
<tr>
<td>201</td>
<td>2/17/1996 12:00:00 AM</td>
<td>12/19/1999 12:00:00 AM</td>
<td>MK_REP</td>
<td>20</td>
</tr>
<tr>
<td>114</td>
<td>3/24/1998 12:00:00 AM</td>
<td>12/31/1999 12:00:00 AM</td>
<td>ST_CLERK</td>
<td>50</td>
</tr>
</tbody>
</table>

#### 7.1.2 Query Set used

For the analysis of the experimental study the following query sets were used. These query sets were tested in SQL server management studio 2008. It is a Graphical user Interface(GUI) tool included with SQL server for configuring, managing and administering all components within Microsoft SQL server.
Some of the sample queries used for testing the performance of our proposed approach are given as follows:

**Query 1:**
```
SELECT First_Name, Last_Name FROM Employee WHERE Salary >
(SELECT Salary FROM Employee WHERE Employee_ID = 163);
```
This query is used to display the first name and last name for those employees who get more salary than the employee whose id is 163.

**Query 2:**
```
SELECT Employee_ID, First_Name, Last_Name FROM Employee
WHERE Salary > (SELECT AVG(Salary) FROM Employee);
```
This query is used to display the employee id, first name and last name for all employees who earn more than the average salary.

**Query 3:**
```
SELECT e.Department_ID, e.First_Name, e.Job_ID,
d.Department_Name FROM Employee e, Department d WHERE
  e.Department_ID = d.Department_ID AND d.Department_Name = "Finance";
```
This query is used to display the department id, name, job and department name for all employees in the finance department.

**Query 4:**
```
SELECT * FROM Employee WHERE Salary BETWEEN (SELECT MIN(Salary) FROM Employee) AND 2500;
```
This query is used to display all the information of the employees whose salary is within the range of smallest salary and 2500.

**Query 5:**
```
SELECT Job_ID, AVG(Salary) FROM Employee GROUP BY
  Job_ID HAVING AVG(Salary) < (SELECT MAX(AVG(Min_Salary)) FROM Jobs
  WHERE Job_ID IN (SELECT Job_ID FROM Job_History
  WHERE Department_ID BETWEEN 50 AND 100) GROUP BY Job_ID);
```
This query is used to retrieve unique job id and there average salary from the employee table based on the constraints given.
7.2 Snapshots of the results with running example

The proposed method of query optimization technique has three stages namely Pre-Processing, Query Tree Construction, Dependency Rule Generation and Query Optimization. Each stage of the proposed method has been tested with the sample queries. The first stage of query optimization technique is Pre-processing. In this stage, the input query is split into a number of small queries according to the presence of parenthesis, keywords, commas and terms. In this stage, we identify the data tables, attributes present in the data tables, relational operators and conditional operators. We also check for the presence of any functions mentioned in the input query. Based on this information, we construct a processed query set. The snapshot for the pre-processing stage is shown in Figure 7.2.

![Figure 7.2. Snapshot of query submission interface](image)

Let us consider for example the given input query as:-

\[
\text{Select } * \text{ from Student where Sno}=\text{Select Sno from Student where mark} > 90 \text{ Union Select } * \text{ from Student where Smark} > 80.
\]

The given input query is split into a number of small queries according to the presence of keywords, parenthesis etc. Then, for each distinct query, we identify the
table used and the fields. For this given input query, the table used are Student and the field names used are Sno, mark and Smark. The expected result of this given query is to get the students who have secured marks greater than 90 and subject mark greater than 80.

The second stage of proposed query optimization technique is to construct Query Tree based on the preprocessed query set. The attributes identified in the preprocessing stage are generated as leaf nodes and for all these leaf node, the parent nodes are generated. For all these parent nodes, there will be a root node which is the root of the query tree. The snapshot of identified query parts from the submitted query is shown in Figure 7.3.

![Figure 7.3. Snapshot of identified query parts](image)

This shows that the submitted query has three sub queries which have to be optimized. It also shows the separator used in the input query. Multi complex queries are usually very difficult to resolve. Such type of queries can be optimized easily by this method. For the given query, the identified query parts are shown in Figure 7.4.
The third stage of proposed query optimization technique is dependency rule generation. Here we identify the number of data tables and their attributes. We also identify the number of locks applied and number of inputs defined by the query. It also identifies the conditions given in the input query. This information helps us to generate the dependency rule. It helps us to determine the dependency factors. Based on these dependency factor we generate the sequence order for query execution. Figure 7.5 shows the snapshot of rules generated for the given input query text. It depicts the generation of dependency rule for the given query.

Figure 7.5 Snapshot of rules generated.
For illustration, we consider the same input query then identify the sub queries. Later, for the identified query parts identify the number of attributes, data tables and conditions used. This information helps us to determine the depth of the dependency persist in the data present in the database. This helps us to generate dependency rules based on the occurrence of the query in the input query. The rules generated for the given query are shown in Figure 7.6. The numbers generated are the number of data tables and their attributes, number of locks applied and number of inputs defined by the query.

\[
\begin{align*}
\text{Student} & \Rightarrow 6 \Rightarrow 1 \Rightarrow 0 \Rightarrow 1 \\
\text{Student} & \Rightarrow 6 \Rightarrow 1 \Rightarrow 0 \Rightarrow 0 \\
\text{Student} & \Rightarrow 6 \Rightarrow 1 \Rightarrow 0 \Rightarrow 0
\end{align*}
\]

Figure 7.6 Generated rules for the given query

The final outcome of the query optimization process is shown in Figure 7.7. It shows the sequence of query part that has to be executed.

Figure 7.7. Snapshot of final optimized sequence
From the generated query execution sequences, we need to determine the optimal sequence. This optimal sequence is obtained by computing the overall execution time based on the values of number of inputs required for each part of query, number of data tables, their required attributes and number of locks the table has applied for. Based on all these factors, we compute the query completion probability for each dependency rule identified. This query completion probability help us to determine the optimum query sequence for execution.

The optimal sequence generated for query execution for the user given query is given below. This optimal sequence of query is shown in Figure 7.8. The generated query sequence helps us to reduce the time involved in query execution as well as produces best optimization cost.

```
Select Sno from Student where mark > 90
Select * from Student where Smark > 80
Select * from Student where Sno=
```

**Figure 7.8. Optimal sequence for query execution**

The optimum sequence generated indicates that we need to first execute the query part, where we need to retrieve the tuples from the data table Student satisfying the constraint. It retrieves the Sno who have secured mark greater than 90. Then from the retrieved tuples, we need to select only those Sno who have secured Subject mark (Smark) greater than 80. This method produces the best optimum result for complex queries too.

### 7.3 Performance Study

The proposed method of query optimization is tested to prove the cost efficiency and how it helps to reduce the time complexity involved in query execution. Cost efficiency is a measure that determines the cost or benefits of a project or program. This measure helps us to understand the key benefits of the project. The cost factors considered are access cost, computation cost, storage cost and communication cost.
Some of the methods taken for comparing the proposed method are linear recursive, combined similarity and semantic cache. The linear recursive method produced 60% of cost efficiency and combined similarity method produced 65% cost efficiency. The Semantic cache method produced 70% of cost efficiency whereas our proposed method produced 90% of cost efficiency.

Figure 7.9 Comparison of cost efficiency

Figure 7.9, shows the comparison of cost efficiency between different algorithms and approaches that we have considered for study. This figure shows that the proposed approach has produced more cost efficient results in terms of components of cost model we have discussed earlier. This result shows that the proposed method helps to reduce the query execution cost to a greater extend. The proposed method is then tested for efficiency in reduction of time complexity incurred in query execution. The time taken for query execution includes various factors like identifying the attributes, data tables, dependency, relations between attributes and constraints. Taking into consideration all these factors our proposed method is proven to produce the best result in reduction of time taken.
Figure 7.10 shows the comparison of time taken by different methods used for query execution. It shows clearly that the proposed approach has taken less time than other approaches. We observe from the study that a query with different types of execution operators, execution strategies and access plans gives rise to different execution costs. We also observe that joins in query increases the cost of the query and thereby it affects the query performance. Hence, it is best to minimize the number of joins to ensure optimized query performance.

### 7.4 Result Study on the Improved Proposed Method

The proposed multi level relational mapping based dependency rule generation approach for query optimization has been implemented and evaluated for its efficiency using SQL database. The method has been evaluated using various setup and has been produced efficient results on query optimization and dependency rule generation.
Table 7.5: Details of evaluation parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool Used</td>
<td>SQL</td>
</tr>
<tr>
<td>Number of relational objects used</td>
<td>200</td>
</tr>
<tr>
<td>Size of query used</td>
<td>75</td>
</tr>
<tr>
<td>Amount of Tuples</td>
<td>2 lakhs</td>
</tr>
</tbody>
</table>

The Table 7.5, shows the details of evaluation parameter being used to evaluate the proposed method and the method has been evaluated for various parameters of query optimization. With these parameters the user given input query is evaluated for the discussed methods such as linear recursive queries, combined similarity and semantic cache method. The result study shows that the proposed method produces efficient result when compared to other methods taken for consideration.

In our proposed method the given user query is read and the relational objects are identified. Then the input query is split into a number of query partition based on the occurrence of key words from the input query. The method then tries to identify the relations that a single object has with others with the help of relational mapping. Identified relational mapping will be used in computing the dependency measure. The dependency measure helps to understand the depth of the relation that a part of query has on another part of the query. Based on the value of measure, the dependency rule is generated. Once the dependency rules are generated then it is easy to generate the optimized sequence of query for execution. This yields in optimum query execution cost and reduction in time taken for query execution.

It shows that the proposed method produces better efficiency in rule generation when compared to other methods. Rule generation efficiency is a measure to compute the efficiency of dependency rule generated. This computation depends on the dependency measure for each part of the query. The dependency measure computation is performed using the query hierarchy. The rule generation efficiency for each method is discussed, analyzed and shown in Figure 7.11.
The proposed method computes the dependency measure existing in multi level attributes in different data tables using relational mapping. It helps us to achieve the optimization accuracy, as it determines the depth of the dependency existing among the attributes. Query optimization accuracy refers to the accuracy of a query optimizer which is intricately connected with a database system performance and its operational cost. The Figure 7.12, shows the comparison of query optimization accuracy produced by different methods and it shows clearly that the proposed method has produced higher accuracy than other methods.
We observe from the result that our proposed method provides accuracy in optimizing the query when compared to other methods. This accuracy is obtained by determining the depth of the dependency existing among the attributes within the data tables.

Since the main aim of query optimization is to obtain reduction in time complexity, that factor has also been tested with the proposed method. The proposed method is compared with the other methods for finding the time taken for executing the given query. As we have already seen that time taken for query execution takes into account many factors which has been discussed earlier. We have to note that our proposed method has achieved the efficiency in reduction of time complexity also. The identified dependencies among the attributes has helped us to generate the rules and which in turn helps to identify the order of query execution.

The Figure 7.13 shows the comparison of time taken by different methods and it shows clearly that the proposed method takes lesser time than other methods. We observe from the result that the proposed method takes less time for query execution when compared to other methods. The time reduction is obtained by choosing the best optimum sequence for execution.

![Figure 7.13 Comparison of time taken](image)
7.5 Effectiveness of proposed optimizer model

In this section, we have discussed about the general framework for the query optimizer. Already we have discussed about the task and goal of any query optimizer. We have also seen the vital role played by the query optimizer in query optimization. The components of query optimizer has also been analyzed. Based on this study we have framed an architecture of query optimizer model that paves way to create an extensible and flexible query optimizer to meet the various growing need in computer applications.

We have shown with a simple example the effectiveness of the proposed architecture optimizer model. We have considered the student admission database which consists of following two main tables- `stud_info`, `stud_mark` and `subject`. The `stud_info` table stores the information of the student like name, application number, community and phone number. The `stud_mark` table stores information of the mark secured by the student like subject code, subject name and mark.

We query the database to retrieve the ID, student name and marks of those students whose community='OC' and marks > 600 from the `stud_info` and `stud_mark` table. The SQL Query1 would be:

```
Select stud_id, stud_name, marks from stud_info as A, stud_mark as D where A.ID=D.stud_id and marks>190 and community='OC'.
```

The given query can be executed in different ways. So we implement different join methods and see the difference between each way of query execution. The different execution plans are considered to understand the underlying concept that query representation plays a vital role in generating the best optimum execution plan. The way we use the various query execution operators such as selection, projection, join and set difference operators has a great impact on generating the optimum execution plan.

**Plan1** - Using inner loop join on `stud_info` and `stud_mark` the query looks like:

```
Select stud_id, stud_name, marks from stud_info as A inner loop join stud_mark as D on A.ID=D.stud_id where marks>190 and community='OC'.
```

103
The Query Execution plan for plan1 is shown in Figure 7.14. In this plan, we have used inner join and non-clustered index for retrieving the tuples from the database. We observe that usage of joins in queries makes it a complex one. The complexity increases when there is usage of different types of joins to retrieve the records from the database. The complexity can be reduced, if we make minimum usage of joins in SQL queries. We observe that the time complexity also varies depending upon the usage of joins.

**Plan2** - Using Merge on stud_info and stud_mark the query looks like- Select stud_id, stud_name, marks from stud_info as A inner merge join stud_mark as D on A.ID = D.stud_id where marks > 190 and community = ‘OC’.
The query execution plan for query1 is shown in Figure 7.15. The usage of inner merge join has increased the cost incurred in query execution. It increases the time taken for scanning the table.

**Plan3** - Using right outer hash joins on stud_info and Stud_mark tables the query looks like- Select stud_id, stud_name, marks from stud_info as A right outer hash join stud_mark as D on A.ID = D. stud_id where marks > 190 and community = 'OC'.

In this query execution plan, we have used the hash join technique in retrieving the tuples from the database. So far, we saw the various join techniques in SQL queries produces variance in cost incurred and time taken for query execution. The hash join technique produces better results when compared to other join techniques. The query execution plan3 for query1 is shown in Figure 7.16.

![Figure 7.16 Query Execution Plan3 For Query1](image)

**Plan4** - Using left outer hash join on stud_info and stud_mark tables the query looks like- Select stud_id, stud_name, marks from stud_info as A left outer hash join stud_mark as D on A.ID = D.stud_id where marks > 190 and community = 'OC'.

The query execution plan4 for query1 is shown in Figure 7.17.
The cost of processing the sample query was measured with the help of SQL tool and the values obtained have been depicted in the following Table 7.6.

Table 7.6 Comparison of various query plan

<table>
<thead>
<tr>
<th>Plan No.</th>
<th>Cached Plan Size In Bytes</th>
<th>Estimated Operator Cost In Sec</th>
<th>Estimated Operator Cost in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>16 B</td>
<td>0.0015032</td>
<td>1%</td>
</tr>
<tr>
<td>P2</td>
<td>16 B</td>
<td>0.0062494</td>
<td>10%</td>
</tr>
<tr>
<td>P3</td>
<td>24 B</td>
<td>0.0041461</td>
<td>58%</td>
</tr>
<tr>
<td>P4</td>
<td>24 B</td>
<td>0.0041461</td>
<td>58%</td>
</tr>
</tbody>
</table>

As it is shown in Table 7.6 that the Plan1 not only has less cost as compared to other plans but it also had less cached plan size. So Plan1 is found to be most optimum execution plan for query 1.
We also observe from the above study that a query with different types of execution operators, execution strategies and access plan generates different execution cost. We also observe that joins in query increase the cost of the query and thereby it affects the query performance. Hence, it is best to minimize the number of joins to ensure optimized query

7.6. Best practices

Navitha Kumari (2012) have also suggested some of the practices to be followed to enhance the performance of a query. They are as follows:

- We need to restrict the queries result set by selecting only the particular columns from the table rather than all columns from a particular table.
- We should use conditions in WHERE clause carefully.
- HAVING clause in SELECT statement should be avoided.
- The number of sub query blocks within a query should be minimized.
- UNION ALL instead of UNION should be used wherever possible.
- Operators like EXISTS, IN and JOIN to be used appropriately in the query.
- Constraints to be used for selection.
- The use of view or replace view with original table should be avoided.
- Select operation should be performed as early as possible to reduce the number of tuples.
- Project operation should be used as early as possible to reduce the number of attributes.
- Select and Join operations that are most restrictive should be executed before other similar operations.
- If we need to join several tables very frequently, then we should consider create index on the joined columns.
- Unnecessary join from tables should be removed.
- Multiple sub queries should be in the most efficient order.
- User defined function should be used to keep the encapsulated code for reuse in future.
• Non-correlated sub query should be used whenever possible when dealing with large tables.

• Correlated sub query or derived tables should be used for row-by-row operation on tables.

• Indices that are not being used should be dropped. The number of executions performed should be reduced by selecting appropriate execution plan.

• The references made to I/O and CPU cost should be used to make the best decision.

• The tables should be joined if possible using the most efficient join type and in the most efficient order.

• We can make use of multiple equivalence rules to reduce the size of the intermediate result.

• We can eliminate unnecessary attributes by pushing projections based on equivalence rule.

• Minimal set of equivalence rules should be used to reduce the number of ways an expression can be generated.

• If the costs of the operators are high, then we try to parallelize more sections of the query.

• Common sub expressions should be shared so that space requirement can be reduced.

These are the some of the best practices to be followed in order to enhance the performance of the execution of a query. This would also help us to reduce the execution cost of the query and reduce the complexities involved in generating an efficient query execution plan.
CHAPTER 8

CONCLUSION AND FUTURE WORK

8.1 Conclusion

This thesis mainly focuses on facilitating the enhancement of query performance, which has been the increase in demand for many database applications. In this thesis we studied the role of query optimizer in query optimization. The primary task of the query optimizer is to transform a declarative SQL statement into a query execution plan by constructing alternative query execution plans, determining the cost of each plan and selecting the cheapest one for execution. Thus query optimizer is widely considered to be the most important component of a database management system.

In this thesis we studied the various related research work carried out in the field of query optimization. We also studied the various research problems involved in efficient query processing. We addressed several problems on answering queries when their exist relations in the database. We also studied the problem of generating efficient plans for user queries. The increase in the number of relations in the database has emerged the need for creating new SQL queries that can get required data from the database.

The problem that the optimizer faces now a days is that for a given user query there exists a large space of different equivalent query execution plans that each have a corresponding execution cost. To determine the best query execution plan among these becomes a challenging one. However, there exist many optimization algorithms techniques, methods and strategies for optimizing a query efficiently. Inspite of all these techniques and methods, optimizing a query accurately was not possible. Many of the methods find difficult in coping up with increase in complexities involved in the user queries and the relations that exist among the database.
In this thesis, we have provided a detailed study of the process of query optimization and the various aspects of optimization occurring at different levels. We have also described how to build a query optimizer model that incorporates the modularity necessary for the new generation of database management systems.

In this thesis, we have done a comparative study on the various existing optimization techniques. This study helps us to understand the necessity and importance of building a uniform framework for query optimization which would be beneficial for developing an extensible and flexible optimizer to meet the growing demand of modern applications.

We have also discussed about the structure used to represent user posed queries and the process involved in query evaluation. We have also showed how equivalence rules and indexing plays a vital role in the enhancement of query performance. Equivalence rules are used to generate systematically all expressions equivalent to the given query which is more efficient to execute. We have reviewed some of the types of index file structure and their roles in reducing execution time overhead. The guidelines for managing indexes have also been specified.

In this thesis, we have discussed about the factors that play a vital role in the performance tuning of the user query. The various factors identified are query optimizer, equivalence rules, indexing, cost estimation and dependencies. Actually cost estimation is one of the difficult task in query optimization. So, we need to take proper measures to accurately estimate the cost of alternative query plans.

Another important factor that increases complexity in retrieving data from the database are the dependencies that exist among the attributes in a database. We have also discussed about the various types of dependencies that exist among the database. Based on this study, we have proposed a query tree based method which generates dependency rules to perform query optimization. The result study of the proposed method have also been discussed by showing the comparison of cost efficiency between different algorithms and approaches. The proposed method produces efficient results and this has been proven by taking some of the already existing methods used for the
enhancement of query optimization. Some of the methods that are taken for result study analysis are linear recursive, semantic cache method and combined similarity method. The advantages and disadvantages of these methods have also been discussed.

Finally, we implemented the proposed system with the proposed architecture and optimization algorithm. The experimental results showed the capabilities and efficiency of proposed optimization algorithm.

8.2 Future Work

In this thesis the proposed method generates the execution order of query to enhance the performance of query optimization by considering the dependencies among the attributes. In this method, we only considered the dependencies among the attributes within the single table. The problem arises when there exist multi level dependencies among the attributes. This kind of issue arises when we deal with relational database. To overcome this issue, we need to improve the performance of our previous solution. For this, we propose a multilevel relational mapping algorithm. This method first identifies the relational objects and generates relation maps. The use of relational map helps to identify the query dependency and to compute dependency measure for each of the rule being produced. Based on the value of the computed dependency measure, the dependency rules are generated which in turn specifies the execution order of parts of the query to produce efficient results.

There is much additional work to be done to improve the generation of dependency rules. Here we have considered only some of the common dependencies that exist among the attributes within a given table. This deserves further exploration with various types of dependencies. Another important issue is correctness of the solutions produced by the proposed method. It is difficult to prove if the proposed method produces the optimal plan for execution just because of comparing with some of the existing methods. It requires some visualization tools or optimality verification tools to prove the accuracy of the results produced by the proposed method. We have also not taken into consideration the constraint mentioned in the user posed queries. Finally we intend to extend our study in the directions mentioned above.
REFERENCES


• Bertino, E & Kim, W 1989, ‘Indexing Techniques for queries on Nested Objects’, *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 2, pp. 196-214

• Bizarro, P, Bruno, N & De Witt, DJ 2009, ‘Progressive Parametric Query Optimization’, *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 4, pp. 582-594

• Carlos Ordonez 2010, ‘Optimization of Linear Recursive Queries in SQL’, *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 2, pp. 264-277

• Chiang Lee, Chi-Sheng Shih & Yaw-Huei Chen 2001, ‘Optimizing Large Join Queries using a Graph based approach’, *IEEE Transactions on Knowledge and Data Engineering*, vol. 13, no. 2, pp. 298-315


• Lin, X 2009, ‘Query Optimization Strategies and Implementation Based on Distributed Database’, *2nd IEEE International Conference on Computer Science and Information Technology*, August 8-11, pp. 480-484
• Majid Khan & Khan, MNA 2013, ‘Exploring Query Optimization Techniques in Relational Databases’, *International Journal of Database Theory and Application*, vol. 6, no. 3, pp. 11-21


Straube, DD & Ozsu, MT 1995, ‘Query Optimization and Execution Plan Generation in Object Oriented Data Management Systems’, *IEEE Transactions on Knowledge and Data Engineering*, vol. 7, no. 2, pp. 210-227


LIST OF PUBLICATIONS


