Performance tuning of automatically generated SQL queries

Artem Karpov
H00227718

August 2016

Dissertation submitted as part of the requirements for the award of the degree of MSc in Information Technology (Software Systems)
Declaration of Authorship

I, Artem KARPOV, confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: Artem Karpov

Date:
Abstract

A database is an organised collection of information which can be easily accessed and changed and SQL is a standard language used to manipulate the data stored in a database. Relational databases are widely used as a core part of Enterprise software applications used to satisfy the need of organisations and aimed to improve the enterprise productivity. Development of such applications is a complex and challenging process which drives developers to create tools, platforms and frameworks in order to simplify the process and satisfy short deadlines of development.

This work considers the functionality and performance of 1C:Enterprise platform - rapid enterprise database development platform. A number of experiments performed within this work demonstrate the performance of simple native queries being translated to SQL and how the nature of embedded 1C objects can be used to improve the performance of such queries.

There are a number of transformations which Query Optimizer of RDBMS can decide to apply to a particular query, all of such transformations guarantee the correctness of the result with better performance. These transformations are either cost based or heuristics based. Experiments performed within this work are heuristic transformations based on the nature of 1C objects.

The comparison of a cost of a simple query before and after “subquery unnesting” and “CASE to OR” transformations showed significantly (495 times) better performance for PostgreSQL and the same cost for MS SQL Server. For complex queries, the same transformation showed 1.03 times better performance for PostgreSQL and 98.96 times better performance for MS SQL Server.

“Pattern elimination” transformation showed 20.54 times better performance for MS SQL Server and 0.003242 time worse performance for PostgreSQL.

Finally, the “subquery unnesting”, “CASE to OR” and auxiliary “Redundant braces elimination” transformations were implemented as the functions of a Java application tool which consumes a T-SQL or PLSQL/PG query, analyses it for possible transformations and performs them following by the generation of the final transformed query.
Acknowledgements

I would like to thank my supervisor, Lilia Georgieva for her advice and support during the project. I appreciate my industrial supervisor from 1C company, Anton Shevchenko for his strong technical expertise of my investigations, recommendations, support and providing me with 1C:Enterprise licenses. And many thanks to my wife Antonina for her unbreakable faith in me and upbringing our kids during my study.
Contents

Declaration of Authorship i

Abstract ii

Acknowledgements iii

Contents iv

List of Figures vii

List of Tables ix

Contents x

Abbreviations xii

1 Introduction 1
  1.1 Motivation .................................................. 1
  1.2 Aims and Objectives ........................................... 3

2 Literature Review 4
  2.1 SQL ............................................................ 4
  2.2 1C:Enterprise ................................................ 6
    2.2.1 Overview ................................................. 6
    2.2.2 1C Programming Language ............................... 7
      2.2.2.1 Imperative 1C Script .............................. 7
      2.2.2.2 Declarative 1C-SQL ................................ 8
    2.2.3 Roles And Access Rights ................................ 9
    2.2.4 Technological Journal ................................ 9
    2.2.5 Simple Application And Indicators ................... 11
    2.2.6 Demo Application ....................................... 12
  2.3 Query Optimization Function Of RDBMS .................... 12
    2.3.1 MS SQL Server Optimizer ............................... 12
      2.3.1.1 Identification Of Costly Queries ................. 15
      2.3.1.2 Execution Plan Analysis ......................... 15
      2.3.1.3 Hints ................................................ 17
    2.3.2 PostgreSQL Optimizer .................................. 17
3 Implementation

3.1 Problem Specification ................................................. 32
3.2 Research Experiments And Evaluation ............................. 37
  3.2.1 Experiments Requirements ..................................... 37
  3.2.2 Subquery Unnesting ............................................. 39
    3.2.2.1 Experiment 1. WHERE EXISTS(subquery) To INNER JOIN . 41
    3.2.2.2 Experiment 2. WHERE Table.Column IN(subquery) To INNER JOIN ... 42
    3.2.2.3 Experiment 3. Both Transformation From Experiments 1 And 2 .................. 43
    3.2.2.4 Experiment 4. Unnesting INNER JOINs ........................ 45
    3.2.2.5 Experiment 5. Semantically Equivalent Transformation ............... 47
    3.2.2.6 Evaluation On MS SQL Server .................................. 48
    3.2.2.7 Evaluation On PostgreSQL .................................... 51
  3.2.3 Pattern Elimination. Joined Tables With The Same RLS Condition. .............. 52
    3.2.3.1 Experiment 6. Pattern Elimination ............................. 52
    3.2.3.2 Experiment 7. Pattern Elimination With Subquery Unnesting .................. 56
    3.2.3.3 Experiment 8. Subquery Unnesting Without Pattern Elimination .............. 57
  3.2.4 Predicate expression with single CASE statements at both sides .................. 58
    3.2.4.1 Experiment 9. Transforming predicate expression with two single CASE Es To Semi Join and Anti Semi Join conditions .................. 58
    3.2.4.2 Experiment 10. Case To OR With Subquery Unnesting ....................... 60
    3.2.4.3 Experiment 11. Case To OR With Subquery Unnesting and pattern elimination .................. 61
  3.2.5 Combinations of transformations ................................ 61
  3.2.6 Summary On Experiments ...................................... 64
3.3 Common SQL grammar .............................................. 65
3.4 Java Application Design And Development .......................... 66
3.5 Recommendations For Further Activity ........................................ 70

4 Conclusion ....................................................................................... 72

A Sources and experiments .................................................................. 74
A.1 Subquery Unnesting Transformation. PostgreSQL .......................... 74
  A.1.1 WHERE EXISTS(subquery) To INNER JOIN .............................. 76
  A.1.2 WHERE Table.Column IN(subquery) To INNER JOIN ............... 78
  A.1.3 Both Transformation From Experiments 1 And 2 ....................... 79
  A.1.4 Unnesting INNER JOINs ..................................................... 81
  A.1.5 Semantically Equivalent Transformation .................................. 83
A.2 Pattern Elimination Transformations. MS SQL Server ..................... 85
  A.2.1 Pattern elimination ............................................................. 85
A.3 Transformations. Oracle database. .............................................. 91
  A.3.1 OR expansion ....................................................................... 91
  A.3.2 Simple view merging ........................................................... 92
  A.3.3 Outer Join View Merging ....................................................... 92

B Development of a simple 1C application ......................................... 94
B.1 Development ................................................................................. 94
B.2 Data Generation ............................................................................ 104
B.3 Reading The Data .......................................................................... 106

C Project management .......................................................................... 111
  C.1 Project Plan ................................................................................ 111
    C.1.1 Project Stakeholders ......................................................... 111
    C.1.2 Project Deliverables And Tasks ........................................ 112
    C.1.3 Gantt Chart of the Project ................................................ 113
  C.2 Professional, Legal, Ethical And Social Issues ............................. 114
    C.2.1 Professional Issues ........................................................... 114
    C.2.2 Legal Issues ........................................................................ 114
    C.2.3 Ethical And Social Issues ................................................... 115
  C.3 Requirement Analysis ............................................................... 115
    C.3.1 Functional Requirements .................................................... 115
    C.3.2 Non-Functional Requirements ............................................ 115
    C.3.3 System and Software Requirements .................................... 116

Bibliography ....................................................................................... 117
List of Figures

2.1 Root causes ......................................................... 5
2.2 Sequence ........................................................... 13
2.3 Left-deep and bushy trees ........................................... 13
2.4 Sample MS SQL Server execution plan (graphical representation) ..... 16
2.5 Sample MS SQL Server execution plan (XML representation) .......... 16
2.6 Sample execution plan PostgreSQL .................................. 18
2.7 ANTLR data flow diagram ............................................ 29
2.8 ANTLR Parse tree .................................................... 30
2.9 Lexer and Parser interconnection .................................... 31
2.10 Parse tree walker .................................................... 31
3.1 The sequence of JOINs for the query from Experiment 1 ............... 49
3.2 Default sequence of JOINs for the initial query ........................ 49
3.3 Optimised sequence of JOINs ......................................... 49
3.4 The difference between subquery 1 and subquery 2 ..................... 54
3.5 UML sequence diagram of the transform method of the Controller class . 68
3.6 The graphical user interface .......................................... 69
3.7 UML class diagram ................................................... 70
B.1 1C:Enterprise start .................................................. 94
B.2 Create database ...................................................... 95
B.3 Database source ..................................................... 95
B.4 Application architecture ............................................... 96
B.5 Application parameters .............................................. 97
B.6 Application properties ............................................... 97
B.7 Database creation completed ........................................ 98
B.8 1C:Enterprise Designer ............................................... 98
B.9 1C Designer metadata tree ......................................... 99
B.10 New catalogue - Customers .......................................... 100
B.11 New catalogue - Companies ......................................... 100
B.12 New document Sales ............................................... 101
B.13 New document attribute - “Company” ................................ 101
B.14 New document attribute “Customer” ................................ 102
B.15 Build the app ....................................................... 102
B.16 Accept changes ..................................................... 103
B.17 Simple application ready in 1 minute ................................ 103
B.18 list of simple application’s tables within MS SQL Server ............. 104
B.19 Creation of external data processor .................................. 105
List of Figures

B.20 Random data generator .......................................................... 105
B.21 Source code 1C Script ............................................................ 106
B.22 Execute external data processor .............................................. 107
B.23 Data is created ................................................................. 107
B.24 Simple 1C-SQL query ........................................................... 108
B.25 Technological log for executed 1C-SQL query ......................... 108
B.26 The result of the execution ................................................... 109
B.27 Record level security configuration ........................................ 109
B.28 Simple 1C-SQL query with RLS ............................................ 110
B.29 Enlarged query ................................................................. 110
C.1 Gantt Chart of the Project ..................................................... 113
List of Tables

2.1 Possible join orders for left-deep and bushy trees ...................... 14
3.1 The cost of the execution ............................................ 37
3.2 Metadata classes of 1C application and their corresponding tables ... 38
3.3 The set of the objects required for RLS .............................. 38
3.4 The content of data sets .............................................. 39
3.5 The cost of the execution on MS SQL Server .......................... 48
3.6 The cost of the execution on PostgreSQL .............................. 51
3.7 The cost of the “pattern elimination” transformation on MS SQL Server 55
3.8 The cost of the “pattern elimination” transformation on PostgreSQL ... 56
3.9 The cost of the “pattern elimination” transformation .................. 56
3.10 The cost of the combination of “pattern elimination” and “subquery unnesting” transformations. ............................. 57
3.11 The cost of the “subquery unnesting” transformation. ................ 57
3.12 The cost of the “CASE to OR” transformation. ....................... 60
3.13 The cost of the combination of “CASE to OR” and “subquery unnesting” transformations. ............................. 60
3.14 The cost of the combination of “CASE to OR”, “subquery unnesting” and “pattern elimination” transformations. ............................. 61
3.15 The cost of the execution of different combinations of transformations “Pattern elimination” (PE), “Subquery unnesting” (SU) and “CASE to OR” (CAS) ........................................ 62
3.16 The list of source files of T-SQL queries ............................. 63
## Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Configuration file of the Technological journal. logcfg.xml</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Example of technological journal log file. 16060813.log</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Sample query</td>
<td>15</td>
</tr>
<tr>
<td>2.4 FORCE ORDER hint example</td>
<td>17</td>
</tr>
<tr>
<td>2.5 LEMMA 1. $Q_1$ and $Q_2$ are equivalent</td>
<td>26</td>
</tr>
<tr>
<td>2.6 LEMMA 2. $Q_3$ and $Q_3$ are equivalent</td>
<td>26</td>
</tr>
<tr>
<td>2.7 LEMMA 3. $Q_5$ and $Q_6$ are equivalent</td>
<td>27</td>
</tr>
<tr>
<td>2.8 Parse-tree graphical representation of a sample SQL statement</td>
<td>29</td>
</tr>
<tr>
<td>3.1 Initial 1C-SQL query. RLS switched off.</td>
<td>33</td>
</tr>
<tr>
<td>3.2 Corresponding T-SQL query. RLS switched off.</td>
<td>33</td>
</tr>
<tr>
<td>3.3 Initial 1C-SQL query. RLS feature switched on.</td>
<td>33</td>
</tr>
<tr>
<td>3.4 Final T-SQL query. RLS feature switched on.</td>
<td>34</td>
</tr>
<tr>
<td>3.5 Subquery unnesting transformation on MS SQL Server. Initial SQL statement</td>
<td>40</td>
</tr>
<tr>
<td>3.6 Subquery unnesting transformation on MS SQL Server. IO output statistics</td>
<td>40</td>
</tr>
<tr>
<td>3.7 Subquery unnesting on MS SQL Server. Experiment 1. WHERE EXISTS(subquery) to INNER JOIN</td>
<td>41</td>
</tr>
<tr>
<td>3.8 Subquery unnesting on MS SQL Server. Experiment 1. IO statistic.</td>
<td>42</td>
</tr>
<tr>
<td>3.9 Subquery unnesting on MS SQL Server. Experiment 2. WHERE table.column IN(subquery) to INNER JOIN</td>
<td>42</td>
</tr>
<tr>
<td>3.10 Subquery unnesting on MS SQL Server. Experiment 2. IO statistic.</td>
<td>43</td>
</tr>
<tr>
<td>3.11 Subquery unnesting on MS SQL Server. Experiment 3. Both transformation from Experiments 1 and 2</td>
<td>44</td>
</tr>
<tr>
<td>3.12 Subquery unnesting on MS SQL Server. Experiment 3. IO statistic.</td>
<td>44</td>
</tr>
<tr>
<td>3.13 Subquery unnesting on MS SQL Server. Experiment 4. Unnesting INNER JOINs.</td>
<td>46</td>
</tr>
<tr>
<td>3.14 Subquery unnesting on MS SQL Server. Experiment 4. IO statistic.</td>
<td>46</td>
</tr>
<tr>
<td>3.15 Subquery unnesting on MS SQL Server. Experiment 5. Semantically equivalent transformation</td>
<td>47</td>
</tr>
<tr>
<td>3.16 Subquery unnesting on MS SQL Server. Experiment 5. IO statistic.</td>
<td>47</td>
</tr>
<tr>
<td>3.17 Subquery unnesting on MS SQL Server. Experiment 5*. Hints</td>
<td>50</td>
</tr>
<tr>
<td>3.18 Subquery unnesting on MS SQL Server. Experiment 5*. IO statistic.</td>
<td>50</td>
</tr>
<tr>
<td>3.19 Initial 1C-SQL query (pattern recognition)</td>
<td>52</td>
</tr>
<tr>
<td>3.20 Simplified initial query (pattern recognition)</td>
<td>53</td>
</tr>
<tr>
<td>3.21 initial CASE statement</td>
<td>58</td>
</tr>
<tr>
<td>3.22 predicate with two CASES to OR</td>
<td>59</td>
</tr>
<tr>
<td>3.23 New keywords to support PLSQL/PG</td>
<td>65</td>
</tr>
<tr>
<td>3.24 Grammar rule of constant expression of T-SQL</td>
<td>65</td>
</tr>
</tbody>
</table>
List of Listings

3.25 Grammar rule of constant expression of T-SQL and PLSQL/PG ........ 65
A.1 Subquery unnesting transformation on PostgreSQL. Initial SQL statement. 74
A.2 Subquery unnesting transformation on PostgreSQL. The initial query EXPLAIN(ANALYZE) output. 75
A.3 Subquery unnesting on PostgreSQL. WHERE EXISTS(subquery) to INNER JOIN. 76
A.4 Subquery unnesting (WHERE EXISTS(subquery) To INNER JOIN) on PostgreSQL. EXPLAIN(ANALYZE) statistic. 76
A.5 Subquery unnesting on PostgreSQL. WHERE table.column IN(subquery) to INNER JOIN. 78
A.6 Subquery unnesting (WHERE Table.Column IN(subquery) To INNER JOIN) on PostgreSQL. EXPLAIN(ANALYZE) statistic. 78
A.7 Subquery unnesting on PostgreSQL. Both transformation from Experiments 1 and 2. 79
A.8 Subquery unnesting (Both Transformation From Experiments 1 And 2) on PostgreSQL Server. EXPLAIN(ANALYZE) statistic. 80
A.9 Subquery unnesting on PostgreSQL. Unnesting INNER JOINs. 81
A.10 Subquery unnesting (Unnesting INNER JOINs) on PostgreSQL. EXPLAIN(ANALYZE) statistic. 82
A.11 Subquery unnesting on PostgreSQL. Semantically equivalent transformation. 83
A.12 Subquery unnesting (Semantically Equivalent Transformation) on PostgreSQL. EXPLAIN(ANALYZE) statistic. 83
A.13 Pattern elimination on MS SQL Server. Representation of the common subquery. 85
A.14 Pattern elimination on MS SQL Server. Representation of the initial query indicating the common subquery. 86
A.15 Before OR expansion Index Access Path can not be used .................. 91
A.16 After OR expansion Index Access Path enabled ....................... 91
A.17 Before OR expansion. Cartesian product .................................. 91
A.18 After OR expansion to avoid Cartesian product ......................... 92
A.19 Before simple view merging ............................................... 92
A.20 After simple view merging ............................................... 92
A.21 View merging is not possible ........................................... 92
A.22 Before outer view merging ............................................... 93
A.23 After outer view merging ............................................... 93
### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
</tr>
<tr>
<td>ANTLR</td>
<td>ANother Tool for Language Recognition</td>
</tr>
<tr>
<td>BSC</td>
<td>British Computer Society</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environement</td>
</tr>
<tr>
<td>LALR</td>
<td>Look-Ahead Left to Right</td>
</tr>
<tr>
<td>LLC</td>
<td>Limited Liability Company</td>
</tr>
<tr>
<td>MIT</td>
<td>Massachusetts Institute of Technology</td>
</tr>
<tr>
<td>MVC</td>
<td>Model View Controller</td>
</tr>
<tr>
<td>PL/SQL</td>
<td>Procedural Language Structured Query Language</td>
</tr>
<tr>
<td>PL/SQL PG</td>
<td>Procedural Language Structured Query Language PostGres</td>
</tr>
<tr>
<td>RDBM</td>
<td>Relational DataBase Management</td>
</tr>
<tr>
<td>RLS</td>
<td>Record Level Security</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>SQL PL</td>
<td>Structured Query Language Procedural Language</td>
</tr>
<tr>
<td>T-SQL</td>
<td>Transact Structured Query Language</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

A wide range of business applications make use of RDBMS for example accounting and taxation, enterprise management, customer relation management and material requirement planning systems. Development of enterprise management systems which comprise almost all operations in a company is complex and challenging. In order to simplify the process of constructing such systems, developers create frameworks, macro languages or special platforms that make the process of development simpler and faster, providing a new level of abstraction. There are a lot of different platforms build on top of RDBMS such as BFC (Microsoft SQL Server, Oracle, IBM DB2, Sybase, MySQL), Clarion (MS SQL Server, Sybase SQL Anywhere, Oracle), CodeCharge Studio (Microsoft SQL Server, Oracle, DB2, MySQL, and Microsoft Access), DataFlex (Microsoft SQL Server, IBM DB2 Universal Database, Pervasive.SQL (Btrieve), and any ODBC-enabled data source (Oracle, MySQL, Excel, etc.)), 1C:Enterprise (MS SQL Server, Oracle database, IBM DB2, PostgreSQL, native database). This work considers 1C:Enterprise platform as a platform which translates native 1C-SQL queries to T-SQL, PL/SQL PG, SQL PL depending on selected RDBMS.

1C-SQL operates with native 1C objects, where each object has the corresponding table within the database. These tables are being created and modified automatically by
1C:Enterprise each time when a developer changes the metadata of the corresponding object. Table names and fields are generated automatically by 1C:Enterprise, for example for the object `Catalog.Companies` the platform will create the table `dbo._Reference12`.

1C:Enterprise supports user defined types, where the type of particular field of an object can refer to another object. This makes possible to get access to the field of field of an object. This significantly simplifies the process of writing 1C-SQL queries and instead of using JOINs developers can simply get the property of a field using dot operator, for example, `Document.Sales.Company.Description`. 1C:Enterprise platform will translate this statement to the LEFT OUTER JOIN of `Documents.Sales` and `Contracts.Companies`.

Another feature of 1C:Enterprise is record level security (RLS). The goal of RLS is also to simplify the process of development and maintenance of entire 1C application. RLS stores predicate within different roles to restrict access to a table for the user or groups of users granted with one or multiple roles. 1C:Enterprise will automatically add all of these preconfigured predicates to each table inside any 1C-SQL query while translating it to SQL.

There are other features of the platform that modify the structure of the initial 1C-SQL query during the translation process. All of these features could make the final SQL query more complex, containing redundant subqueries, multiple nested joins and complex predicates combined with RLS.

These simplifications aim to make the process of creating, maintaining and supporting of any 1C application as easier as possible, automating all routine operations, implementing native scripting language and performing optimisations on low levels inside the platform itself or using RDBMS optimisation features.

If developer will be asked to write SQL query with the same requirements, returning absolutely the same result then SQL query could be more efficient and effective than generated by the platform. This drives the author to discover the area of optimisation the structure of queries automatically generated by 1C:Enterprise.
1.2 Aims and Objectives

The goal of this work is to discover means of reducing the cost of the execution of SQL queries translated from 1C-SQL by 1C:Enterprise platform. This goal is approached through the following objectives:

- Discovering how particular features of the 1C:Enterprise intricate the final SQL statement and affect the cost of its execution on MS SQL Server and PostgreSQL. This is done through performing experiments on a sample 1C application. Each experiment starts with writing a 1C-SQL query, extracting the SQL query generated by 1C:Enterprise and analysing the structure of the final query.

- Understanding the root causes of performance degradation and providing means of transforming the structure of SQL query in order to boost the performance up. This is done by rebuilding the structure of the initial query and(or) applying relational algebra to transform the initial query following by the comparison of the cost of the execution of the pairs on different amount of data sets.

- Developing Java application - SQL performance translator which analyses the structure of an SQL query, define the necessity of discovered transformations, apply them and generate a new query which performs better on considered RDBMSs.
Chapter 2

Literature Review

An SQL query is a request for data from a database. SQL is a declarative language and there are many equivalent ways of constructing the query statement (Churcher, 2008) which will return the same result and this is developer’s responsibility to generate efficient and effective queries. This work considers queries generated by 1C:Enterprise platform which has its own simple 1C-SQL. 1C Enterprise translates each 1C-SQL query to SQL, passes this query to RDBMS underneath which executes it and returns the resulting data back to the 1C application for further processing.

This work is not an attempt to improve the query optimisation feature of RDBMS, but the way of how to help it to execute the query better through the transformation of the initial query and providing the optimizer with better conditions.

2.1 SQL

Structured query language (SQL) was invented by Donald D. Chamberlin at IBM in the early 1970s. Inspired by the concepts described by Edgar F. Codd, Donald D. Chamberlin and Raymond Boyce Oracle corporation developed their own relational database management system (RDBMS). The first commercial release – Oracle v2 was introduced in June 1979. In 1983 IBM presented their RDBMS – DB2. Microsoft corporation took part in the competition in 1989, developing the first version of MS SQL Server. Development of an open-source SQL management system was started at 1982 as an Ingres
project at the University of California, Berkeley and in 1996 this system became known as PostgreSQL.

SQL is a standard of the American National Standard Institute (since 1986) and of the International Organization for Standardization (since 1987) despite that there are a lot of SQL dialects, “For example MS SQL Server specific version of the SQL is called T-SQL, Oracle version of SQL is called PL/SQL, MS Access version of SQL is called JET SQL, etc.” (Inc, 2015)

![Figure 2.1: Root cause for the most recent performance problems.](image)

SQL is a special-purpose programming language designed to manipulate data stored in RDBMS. Development of SQL statements, especially SQL queries accompanied by the risk of performance issues, when simple and elegant SQL query performs inefficiently wasting CPU time and producing a lot of input-output operations (IOO). Rebuilding the structure of SQL query could improve the performance and save time and subsequently money.

SQL as a language offers mechanisms to manipulate (store, retrieve, update, delete, create) data within one or a number of database tables. “Almost all modern Relational Database Management Systems like MS SQL Server, Microsoft Access, MSDE, Oracle,

---

1(Randal, 2011)
Chapter 2. Literature review

DB2, Sybase, MySQL, Postgres, and Informix use SQL as the standard database language.” (Inc, 2015). According to survey Figure 2.1 – “Common causes of performance problems” (Randal, 2011) T-SQL code is the top root cause for the performance problems. Paul S. Randal – CEO of SQLskills.com sums up – “No matter how powerful the hardware and how much memory or IOPS capacity you have. One of the most common causes of poorly performing T-SQL code and indexing strategy is that developers do not write and test the code in an environment that simulates a true production environment and workload.”

2.2 1C:Enterprise

2.2.1 Overview

“1C:Enterprise is a universal cloud and on-premise system of programs for automating a company’s financial and wider operational activities.” (1C, 2016a)

There are two major modes of launching the system, Designer mode, and Enterprise mode. The Designer mode used by administrators and developers during configuration and development stage to define the structure of objects, their graphical user interfaces, algorithms, roles and manage the list of users. Before starting 1C:Designer it required to create an empty application which will store all metadata and user data in a database. The process of creation is fairly simple, the platform will automatically create an empty database inside selected RDBMS system and empty tables for each object created in the design stage. Every time when developers make changes in 1C:Designer the platform makes changes in a database. There are no needs to do anything within the database directly and it is not recommended to do it. The only thing required to do directly are database maintenance operations (backup, restore, shrink tables). Enterprise mode used by users of the application to book records into the database, generating of various reports and operations configured in design mode. (1C, 2016a)

The architecture of a 1C application is multi-tier and depends on the purpose of the application created and business requirements. The application which uses a native database is 2-tier, the client tier and the database tier. Applications with other types of

\[^2\]Microsoft’s and Sybase’s proprietary extension to SQL
databases (MS SQL Server, Oracle database, Postgres database or IBM DB2) are 3 or 4 tier, client tier, application server tier, database server tier and web server tier. Developers do not care about database management system underneath, the script language is independent of database tier selection, the platform translates 1C-SQL statements to SQL supported by selected RDBMS (T-SQL, PL/SQL, PL/pgSQL) in a background. There are also different ways of launching the application, thin, thick and web clients. Web access to the application can be configured for MS Internet Information Server or Apache, 1C:Enterprise supports both of them, graphical user interfaces configured in 1C:Designer have the same shape and behaviour for all types of clients and 1C:Enterprise generates views automatically based on configured rules. (1C, 2016c)

The following sections describe briefly the features of 1C:Enterprise required to understand the problem of discovering performance issues and implementing the tool to resolve them.

### 2.2.2 1C Programming Language

1C:Enterprise launched in design mode or simply 1C:Designer is IDE to develop specific configuration algorithms in program modules written in the 1C:Enterprise script. 1C:Enterprise script is an imperative script language which has among others a special object - Query. The Query object is used by the system to generate and execute database queries and provide means for developers to write 1C-SQL declarative statements to manipulate the data stored in a database.

#### 2.2.2.1 Imperative 1C Script

The 1C:Enterprise script is a domain-specific programming language designed by developers to developers and used to specify algorithms of a particular application. “It has many common features with other languages, such as Pascal, JavaScript, Basic. However, it is not a direct analogue of the listed languages.” (Radchenko and Khrustaleva, 2011).

Configuration development stage allows developers configure the application, create objects with restricted special data types such as Catalog, Document, AccumulationRegister, Enumeration, Report, etc, and write script code to define the behaviour of the objects.
Chapter 2. Literature review

There are no possibilities to define other user defined types. Types of variables are not fixed and depend on its value. There are arrays, structures, maps and other universal data collections. 1C script has some object oriented features and relatively simple to use, there are access rules for properties, methods of objects. IDE has all tools to create objects, write and debug scripts, compile the entire application, it also provides means to write unit tests and its own version control system. (1C, 2016d)

2.2.2.2 Declarative 1C-SQL

1C:Enterprise query language (1C-SQL) is based on SQL but at the same time it contains a number of enhancements, which are oriented on minimising of development effort. The crucial feature of the query language that it provides read-only access to the data. Essential features of 1C-SQL are addressing subordinate fields through the point, access to nested tables, automatic ordering, multidimensional and multilevel generation of totals, support of virtual tables, standard SQL operations, temporary tables and query packages. (Radchenko and Khrustaleva, 2011).

Each object defined in 1C:Enterprise Designer has its corresponding database table(s). Each object has its real table, and some of the objects have virtual tables as well. Real tables are stored in the database, the name of the table and its properties within the database have different names, given by the system itself rather than by developer. Virtual tables are based on real tables and are generated during the execution process. (Khrustaleva, 2013) This is the functionality of the platform to hide database implementation to simplify the process of creation of applications and to support different RDBMS.

1C-SQL has 80 keywords in total. Some of that key phrases have the same names and meaning as ANSI SQL analogues, such as SELECT, FROM, WHERE, UNION, but there are a few unique key phrases, such as ALLOWED, UNDEFINED, TYPE, PRESENTATION. Each 1C-SQL statement written in 1C: Enterprise Designer the platform translates to SQL statement for selected RDBMS. It will be T-SQL for MS SQL Server, PL/SQL for Oracle, PL/psSQL for PostgreSQL and so on. 1C:Enterprise platform support different RDBMSs and does not provide developers with the ability to manipulate data directly. (1C, 2016e) This approach has its pros and cons.
2.2.3 Roles And Access Rights

Database management systems like MS SQL Server Management Studio provide means to restrict access to particular tables, for users or groups of users. Such functionality is insufficient for real applications, and business dictates requirements to grant users more strict permissions, where users can read or modify only part of the table in conformity with a special condition. 1C:Enterprise designer provides developers with advanced administration tools that can resolve this issue. Developers and administrators of the system can configure such conditions using 1C-SQL during the configuration stage. The set of such conditions is called record level security or row level security (RLS). In essence, each 1C-SQL statement during translation process becomes enriched by RLS conditions. (1C, 2016f,g)

2.2.4 Technological Journal

Each 1C-SQL query written within the 1C:Designer the platform translates to SQL and passes to database engine for execution. The resulting amount of data is sent to 1C application back for further manipulations and/or display for users. 1C:Enterprise does this task in the background but provides developers with the ability to analyse this information. The Technological journal is a configurable log file with a special format. To be able to analyse the source code of T-SQL statement being sent to the database engine the technological journal configuration file should be created, configured and placed to the installation directory. This file specifies what kind of information 1C:Enterprise should collect and put into the log file, the full path to the destination folder for logs. This folder should be accessible to the owner of the 1C:application server processes.

Possible structure of the file looks as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="http://v8.1c.ru/v8/tech-log">
  <dump create="true" location="S:\srvinfo\TJ\Dump" type="0" prntscrn="false"/>
  <log location="S:\srvinfo\TJ\Log" history="1">
    <event>
      <eq property="name" value="dbmssql"/>
      <like property="sql" value="%dbo._document229%"/>
    </event>
    <property name="sql">
      <event>
        <eq property="name" value="dbmssql"/>
      </event>
    </property>
  </log>
</config>
```
Listing 2.1: Configuration file of the Technological journal. logcfg.xml

This configuration file should be placed to 1C:Enterprise platform folder to instructs the platform collecting all events “dbsql” where the property of an event contains the table name `dbo._document229`.

\(<eq\ property="name" value="dbmssql"/>\) property tells to collect all events being sent to the MS SQL Server database. The log file itself contains full source code of each T-SQL statement with the list of parameters and value pairs.

\(<like\ property="sql" value="%dbo._document229%"/>\) filter drives 1C:Enterprise to collect the events that contain the text `%dbo._document229%` where `dbo._document229` is the name of the table within the SQL Server database, in this instance the corresponding object within the 1C application is the `Document.SalesProductsServices`.

Each time when users open the list of these documents or execute any other request to read the data from this object a new event will be added to the technological journal.

The following text is a sample content of the log file.

```sql
33:10.692000-32000, DBMSSQL, 4, Sql = 'SELECT TOP 45 CASE WHEN T1._Fld6093 = 0x01 THEN CASE WHEN T1._Marked = 0x01 THEN 10.0 WHEN (T1._Posted = 0x00) THEN 9.0 ELSE 8.0 END ELSE CASE WHEN T1._Marked = 0x01 THEN 2.0 WHEN T1._Posted = 0x01 THEN 1.0 ELSE 0.0 END END, T1._IDRRef, T1._Marked, T1._Date_Time, T1._Number, T1._Posted, T1._Fld6066RRef, T1._Fld6067RRef, T1._Fld6091RRef, T1._Fld6068RRef, T1._Fld6070RRef, T1._Fld6074RRef, T1._Fld6090, T1._Fld6092, CASE WHEN (T2._Fld25110_TYPE = 0x08 AND T2._Fld25110_RTRef = 0x00005D13 AND T2._Fld25110_RRRef = ?) THEN 0x01 ELSE 0x00 END, T3._Fld15401RRef, CASE WHEN T4._Fld22601_TYPE IS NULL AND T4._Fld22601_RTRef IS NULL AND T4._Fld22601_RRRef IS NULL THEN 0x00 ELSE 0x01 END,'``
2.2.5 Simple Application And Indicators

Development of a simple 1C application will help to understand how 1C:Enterprise translates 1C-SQL to T-SQL, how RLS enlarges the query, what are virtual tables and what is the relationship between 1C objects and tables within MS SQL Server. The full description and step by step instruction of development explained in Appendix B. The 1C application has the following 1C objects Catalog.Companies, Catalog.Customers, Document.Sales, one superuser’s role and the role which grants restricted access for reading data from catalogs and the document. The restricted amount of data will be stored within the object InformationRegister.Companies_Permitted. Other modifications required for experiments described in Chapter 3.
2.2.6 Demo Application

Real enterprise applications have a more complex set of objects, more parameters within roles to define the record level security and more complex queries to read data from the database. 1C demo application for accounting represents enough complexity to perform the analysis. So some of the queries are generated by 1C:Enterprise platform from two installations of the 1C demo application for MS SQL Server and Postgres SQL.

2.3 Query Optimization Function Of RDBMS

Before starting optimising automatically generated queries it is required to understand how to write SQL query for SQL Server that performs well and it is important to know how RDBMS itself optimises queries. “...a better understanding of how the Query Optimizer works can help both database administrators and developers to write better queries and to provide the Query Optimizer with the information it needs to produce efficient execution plans.” (Nevarez, 2011) The following sections describe the core functionality of the optimizer of Microsoft SQL Server and PostgreSQL.

2.3.1 MS SQL Server Optimizer

SQL as a declarative language allows constructing SQL query which tells RDBMS what data it is asking but not how to get it from the file system. Optimizer answers the last question. The whole process of the query execution depicted in Figure 2.2.

The result of “Parsing and Binding” is a tree of logical operations (reading the table, making the join and so on). The tree then passed to “Query optimisation” to generate a number of possible execution plans (the set of physical operations like index searching or table scan). In this phase, Optimizer estimates the cost of these plans and defines the best one. The last step is to execute the query with selected execution plan and return the result.

The process of selecting the best plan is the most complex task. Depending on the complexity of an initial query, Query Optimizer could devise thousands or even millions

(Nevarez, 2011)
of different execution plans. This means that Query optimizer could neglect the best plan if it takes too much time to find it, instead, the optimizer selects the execution plan which is close to the optimal. According to (Nevarez, 2011) “Cost estimation models are inherently inexact” and the problem of query optimisation still faces some technical challenges for example MS SQL optimizer assumes that it reads data from the disc, even the data is in memory which results in the wrong estimation of the cost.

The next challenging task for optimizer appears with joins. The number of possible execution plans for join ordering depends on the number of tables and the shape of the query tree. Figure 2.3 shows two different shapes of join trees, linear left-deep tree and arbitrary shaped bushy tree which include both left and right-deep trees. Table 2.3.1 presented by (Nevarez, 2011) shows how the number of possible combinations for join orders depends on the number of tables, for different join tree shapes. The number of permutations of tables for left-deep tree is calculated as $n!$ and for a bushy tree as $\frac{(2n-2)!}{(n-1)!}$. Evaluation of all possible execution plans for all possible join orders of such queries would take too much time, for example, for 6 table join Query Optimizer

\[\text{(Nevarez, 2011)}\]
would take 30240 possible join orders, and for each of such order, it should find the best execution plan based on an estimation of the cost for each of such plans. As a result, Query Optimizer does not analyse all these possible plan combinations and “uses heuristics, such as considering the shape of the query tree, to help it narrow down the search space” (Nevarez, 2011).

<table>
<thead>
<tr>
<th>Tables</th>
<th>Left-Deep Trees</th>
<th>Bushy Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>24</td>
<td>120</td>
</tr>
<tr>
<td>5</td>
<td>120</td>
<td>1680</td>
</tr>
<tr>
<td>6</td>
<td>720</td>
<td>30240</td>
</tr>
<tr>
<td>7</td>
<td>5040</td>
<td>665280</td>
</tr>
<tr>
<td>8</td>
<td>40320</td>
<td>17297280</td>
</tr>
<tr>
<td>9</td>
<td>362880</td>
<td>518918400</td>
</tr>
<tr>
<td>10</td>
<td>3628800</td>
<td>17643225600</td>
</tr>
<tr>
<td>11</td>
<td>39916800</td>
<td>670442572800</td>
</tr>
<tr>
<td>12</td>
<td>479001600</td>
<td>28158588057600</td>
</tr>
</tbody>
</table>

Table 2.1: Possible join orders for left-deep and bushy trees

“Transformation rules” are used by Query Optimizer to explore possible execution plans for a query. These rules are based on relational algebra. Each rule consists of pattern and substitute. “The main types of transformation rules include simplification, exploration and implementation rules.” (Nevarez, 2011) The most interesting for this work are simplification and exploration rules. Simplification rules generate a simpler logical tree and exploration rules based on relational algebra (commutativity and associativity rules). At the moment of writing his book (Nevarez, 2011) SQL Server had 377 transformation rules, at the moment of writing this work the number of possible transformation rules has increased to 394.  

\footnote{The total number of transformation rules obtained by executing select * from sys.dm_exec_query_transformation_stats on MS SQL Server 2014.}
2.3.1.1 Identification Of Costly Queries

The measure of cost involves estimation of CPU time, a number of input-output operations and an amount of memory required to get data. These depend on the cardinality of the data in tables which is a number of unique values in a column. RDBMS calculates cardinality by collecting statistics. So it is vital to have actual statistics to help Optimizer to select an effective plan. Calculation of the cost for possible execution plans has its own cost, therefore RDBMS stores them in the cache and uses cached plans next time and skips the phase of optimisation.

The following metrics are good starting point to understand where a query costs: cpu_time, logical_reads, and physical_reads. “The higher the number of logical reads for a query, the higher the possible stress on the disk could be. An excessive number of logical pages also increases load on the CPU…” (Fritchey, 2014).

SET STATISTICS TIME ON is T-SQL commands to get information about time spent on parsing, compiling and executing a query. SET STATISTICS IO ON command displays the number of scan counts, logical and physical reads and CPU time.

2.3.1.2 Execution Plan Analysis

SQL Server displays a query execution plan in two main forms, graphical representation, and XML. Graphical representation of an execution plan for the query 2.3 displayed in Figure 2.4 and XML representation in Figure 2.5.

```sql
SELECT T1._IDRRef, T2.Description_ FROM dbo._Document12 T1 LEFT OUTER JOIN (SELECT T3._IDRRef AS IDRRef, T3._Description COLLATE Latin1_General_CI_AS AS Description_ FROM dbo._Reference11 T3 INNER JOIN dbo._InfoRg35 T4 ON T3._IDRRef = T4._Fld38RRef) T2 ON T1._Fld13RRef = T2.IDRRef INNER JOIN dbo._InfoRg35 T7 ON T1._Fld13RRef = T7._Fld38RRef
```

Listing 2.3: Sample query
The execution plan shows two flows of information where each flow split into next two flows. Each node represents an operator with relative estimated cost. The thickness of connecting arrows indicates an amount of rows transferred from one node to another.

Physical operators are usually most right operators or the first operators if read from right to left. The execution plan on Figure 2.4 contains four physical operators retrieving data from _Document12, _InfoRg33, and _Reference11. In this example, all of them retrieve data from indexes (another option is to get data directly from table or heap). There are two index operations seek and scan, and “... index seek is usually the most efficient way of accessing a small number of rows.” (Fritchey, 2014). There could be sorting physical operation.
The next operator after physical retriever operator is joining operator. This will be a hash match join, merge join or nested loop join. According to (Fritchey, 2014) “For small result sets, a nested loop join is usually the preferred join technique.”

### 2.3.1.3 Hints

Hints are sort of prescriptions for controlling the execution plan selection. They disable transformation rules and restrict the search space and should be used with caution. Within this work, the author uses hints for troubleshooting purposes. There are three types of hints which affect plan selection such as query hints, join hints and table hints (Nevarez, 2011).

**FORCE ORDER** hint is a query hint which tells the Query Optimizer to preserve the sequence of join as SQL query indicates.

```sql
SELECT *
FROM A,B,C
LEFT OUTER JOIN (SELECT *
    FROM D
    INNER JOIN E
    ON somepredicate
    ON nextpredicate
WHERE anotherpredicate
OPTION (FORCE ORDER)
```

Listing 2.4: FORCE ORDER hint example

### 2.3.2 PostgreSQL Optimizer

(Wiles, 2016) describes the process of execution “The first step to learning how to tune your PostgreSQL database is to understand the life cycle of a query. Here are the steps of a query:

- Transmission of query string to database back-end
- Parsing of query string
- Planning of query to optimise retrieval of data
• Retrieval of data from hardware

• Transmission of results to client

The first step is establishing a connection with PostgreSQL server, transmitting an SQL query and switching to waiting for the result. The server performs syntax checking of the query and parses it into a query tree. Then Rewrite system of the server picks the transformation rules up and apply rule bodies, rebuilding the query tree. (PostgreSQL, 2016) Planner/optimizer takes transformed tree and devises a query plan based on the cost estimation. The executor of the server recursively execute operations described in the plan and delivers the result to the client application.

2.3.2.1 Execution Plan Analysis And Cost Calculation

PostgreSQL allows to explain an SQL statement before its execution and analyse what actually happened after this using EXPLAIN and ANALYSE keywords. When PostgreSQL optimizer estimates the number of rows in output it may make mistakes. The difference between estimated and actual numbers represents the common source of query execution mistakes. (Smith, 2010).

“The easiest tool for targeting query performance problems is use of the EXPLAIN and EXPLAIN (ANALYZE) commands. EXPLAIN by itself will give you just an idea of how the planner intends to execute the query without running it. Adding the ANALYZE argument, as in EXPLAIN (ANALYZE), will execute the query and give you a comparative analysis of expected versus actual” (Obe and Hsu, 2014) EXPLAIN (ANALYZE) command shows the actual execution plan selected for the query with the estimated and actual cost, rows, width and loops for each node within the plan. Possible output for the command displayed on Figure 2.6.

![Figure 2.6: Sample execution plan PostgreSQL.](image-url)
According to (Smith, 2010) “Some nodes, such as ones doing joins, execute more than once. In that case, the loops value will be larger than one, and the actual time and row values shown will be per loop, not total. You’ll have to multiply by the number of loops in order to get a true total”

2.3.2.2 Hints

There were a lot of discussions whether hints are good or not for PostgreSQL and the final decision is no and for the moment of writing the last version has now such functionality and there are no plans to implement it. From another side, there are still a number of people who do not agree with this and they have implemented the extension for PostgreSQL which has such functionality. The source code and documentation of this module published in http://sourceforge.jp/projects/pghintplan/. This external module works only on Linux and does not support latest versions of PostgreSQL. Since that, the usage of this module within this work is rejected.

2.3.3 Oracle Optimizer

Oracle Optimizer makes three main decisions during optimisation an SQL statement such as access method to data, join method (nested loops joins, hash join, or other) and join order. (Consulting, 2014) urged to start SQL tuning from server kernel, then configure the Optimizer statistic, parameters and Oracle instance, finishing with SQL statement tuning.

Since this work is focused on SQL query optimisation the rest of steps are out of this work besides the last phase - “Rewrite SQL”. A thorough consideration of Oracle Optimizer done by (Lewis, 2005). Lewis considered three versions of Oracle database, such as 8i, 9i and 10gR2, explaining the behaviour of Oracle Optimizer for considered versions in different scenarios. This section briefly summarises these scenarios starting from explaining how the Optimizer works.

The sequence of operations performed by Oracle to execute a query is as follows: At first, Oracle does syntax checking, verifying whether the language elements are sequenced correctly to form valid statements. Next time Oracle evaluates the meaning of the code, this is called a semantic analysis. All references to database objects and host variables
are thoroughly analysed by Oracle to make sure they are valid. Oracle also checks permissions.

The cost of an SQL statement in Oracle database is the result which is calculated by the Oracle Cost-Based Optimiser (CBO). This result represents the best estimation of time to execute SQL statement.

\[
\text{“the CPU costing model:” } \frac{\#SRDs \times \text{sreadtim} + \#MRDs \times \text{mreadtim} + \#\text{CPUCycles}}{\text{cpuspeed}}
\]

“where #SRDs - number of single block reads; #MRDs - number of multi block reads; #CPUCycles - number of CPU Cycles; sreadtim - single block read time mreadtim - multi block read time cpuspeed” (Lewis, 2005)

Depending on the structure of SQL statement, environment, database model, indexes and content of database CBO could rebuild initial SQL statement and(or) devise the execution plan which Oracle considers the best for this statement. From version to version Oracle improves the quality and expand the functionality of CBO fixing wrong decisions made in the early version and introducing new features of optimisation.

CBO does many different things to execute SQL statement effectively. Consideration of such scenarios and issues involved in this process is crucial for developing SQL to SQL performance translator.

In the latest version of Oracle database, Oracle corporation introduced Adaptive Query Optimisation (AQO). “Adaptive Query Optimisation is a set of capabilities that enable the optimizer to make run-time adjustments to execution plans and discover additional information that can lead to better statistics. This new approach is extremely helpful when existing statistics are not sufficient to generate an optimal plan. There are two distinct aspects in Adaptive Query Optimisation, adaptive plans, which focuses on improving the initial execution of a query and adaptive statistics, which provide additional information to improve subsequent executions.” (Corporation, 2016) This approach fully described in Oracle documentation (Paper, 2013) is not a new method of optimisation but a new feature of Oracle which allows CBO to improve the efficiency of execution plans.
2.3.3.1 Table Scan

A table scan is a scan performed by database system where each row is sequentially read and the value is verified for some conditions. Table scan generally appears to be the slowest method of scanning a table since it generates a lot of input-output reads. In Oracle, the cost of sequential scan “is largely the cost of assumed multiblock reads” (Lewis, 2005). Issues involved are table scan for partitioned tables and tables “where the number of leaf blocks can be much smaller than the number of blocks below the high water mark” (Lewis, 2005).

2.3.3.2 Single table selectivity

Correctly calculated cardinality is a crucial point to get correct order of joins and for optimal selection of indexes. Cardinality displays the number of unique values in a column of a database table. Low cardinality indicates a high number of duplicate values. The value of cardinality is used by the Optimiser to find a better execution plan. Selectivity is closely related to cardinality and used in connection with indexes. The following formula explains the value of selectivity. 

\[
\text{SelectivityOfIndex} = \frac{\text{cardinality}}{\text{numberOfRecords} \times 100}\%
\]

For tables where a number of unique values in a column is small, Oracle Optimizer for such tables incorrectly calculates cardinality when in-lists values are in predicates, and this inaccuracy becomes less with increasing the number of unique values in the column. The wrong cardinality is calculated by Oracle Optimizer when in-list contains a value which is out of bound or mixed, or has null values in list and binding variables. (Lewis, 2005)

Predicate \((> 8 \text{ and} \leq 8)\) is not the same as true for the Optimizer. This could cause performance issues. Multiple predicates increase inaccuracy. Ranges and in-list values in predicates also potentially could cause performance issues. (Lewis, 2005)

2.3.3.3 B-Tree Access

The term B-tree as a self-balancing tree data structure is a generalisation of a binary tree. The main purpose of this data structure is to increase the speed of searching a
value, minimising the number of reads. This structure allows data manipulations in logarithmic time.

\[ \text{CostOfBTree} = blevel + \text{ceiling(leaf_blocks*effective_index_selectivity)} + \text{ceiling(clustering_factor*effective_index_selectivity)} \] (Lewis, 2005, Wolfgang, 2003) where blevel - the depth of the index, leaf_blocks - the number of leaf blocks to visit, clustering_factor - an number of table blocks to visit, ceiling function depending on the version of Oracle round the value up or to the closest value.

### 2.3.3.4 The Clustering Factor

Clustering factor is very important for the calculation of the cost of index range scans. Oracle provide the administration with the ability to adjust the value of clustering factor. This could be done to ensure that the Optimizer has the correct information to generate the execution plan.

### 2.3.3.5 Query Transformation

Before starting the process of optimisation Oracle does the query transformation. The main purpose of rewriting the query is to provide the Optimizer with more options to optimise. The Optimizer compares the cost of execution of both queries, before and after transformation. This makes the process of optimisation slower (Lewis, 2005).

According to Oracle documentation, Oracle 12c does the number of transformations such as OR Expansion, View Merging, Predicate Pushing, Subquery Unnesting, Query Rewrite with Materialized Views, Star Transformation, In-Memory Aggregation, Table Expansion, Join Factorization. (Oracle, 2016)

With **OR expansion** Oracle rewrites disjunctive\(^6\) queries into the query with UNION ALL operation. There are a number of reasons to make this transformation. The first is to give a chance to the Optimizer to use indexes and (or) partition pruning. The second reason is to make the Optimizer a chance to consider other join methods and avoid Cartesian product\(^7\), which is too expensive. An example of avoiding Cartesian product for the disjunction in WHERE clause using OR expansion is in Appendix A.3.1.

---

\(^6\)queries that contain OR clauses

\(^7\)Cartesian product the result of a query which returns all the rows in all the tables listed in the query
OR expansion is applicable in cases when the disjunctive predicates are conditions for different tables. For disjunctive predicates filtering one table or in-list predicates Oracle uses BITMAP and FULL index scan, there are no reasons for OR expansion.

SimpleView merging\(^8\) moves the child view merging tables to the parent query block. The resulting query with all tables in one query block becomes easier for the Optimizer, because for the columns from view there are no indexes. An example of simple view merging is in Appendix A.3.2. In cases when nested view contains Group by, Distinct, Outer-join, Spreadsheet clause, Connect by, Set operators or Aggregation, when it is on the right side of a semi-join or anti-join, when the view includes subqueries and when the outer query block contains PL/SQL functions optimiser do not do automatic view merging. Merging such views require additional verification.

More complex views containing outer joins with a table in outer block becomes more dangerous for this transformations since the result could be not the same as initial query returns. There is “one significant restriction on views on the left of an outer join: each table from the outer query block can be outer-joined to at most one underlying table of the view”. (Allison, 2011) See Appendix A.3.3

The View is not the same as a table, it does not have any indexes. Because of this queries with views could be slow. Join predicate push-down transformation leaves the view as a separate block, but moves join condition down to view from the outer query block in order to make the view correlated. This transformation allows using indexes. According to (Colgan, 2011) “the join predicate push down transformation is not always optimal” and Oracle makes cost estimation comparing the cost with and without this transformation.

Query unnesting transforms the query with nested query to the equivalent join. Nested queries are estimated by the Optimizer separately, unnesting allows the Optimizer to estimate the cost of the entire unnested query as one. This transformation could be done automatically by the Optimizer only in the case when after this transformation oracle returns the same data. (Colgan, 2010)

\(^8\)simple view merging refers to simple-project-join expressions which exclude union and difference operations, only Cartesian product; natural join; or theta-join. For theta-joins, the join condition must again be a conjunction of equalities
**Table expansion** allows the Optimizer to generate a plan based on indexes with the read-mostly portion of the data. This transformation applicable for partitioned tables and based on analysis of indexes created for partitions. If an initial query requires data from partitions without indexes, such query will be transformed to UNION ALL for partitions with indexes and without.

**Join factorisation** is transformation which is opposite to Query unnesting, this transformation define common part in UNION ALL moves it to parent block which helps to avoid multiple reading one common table. (Hong, 2011)

### 2.3.3.6 Other SQL Statement Tuning Cases

(Consulting, 2014, Date, 2009) proposed tips for writing more efficient queries, such as “use minus instead of EXISTS subqueries, use SQL analytic functions, re-write NOT EXISTS and NOT EXISTS subqueries as outer joins, leave column names alone, avoid the use of NOT IN or HAVING, avoid the LIKE predicate, never mix data types, use decode and case, avoid possibly nondeterministic expressions, don’t use the comparison operators ”<”, ”<=”, “>”, and ”>=” on rows of degree greater than one, never write code that relies on left to right column positioning, avoid duplicates, be careful over the use of COUNT; in particular, don’t use it where EXISTS would be more logically correct, don’t use ALL or ANY comparisons”

### 2.4 Related Works

Query optimisation has been researched extensively during past decades. Subquery execution strategies, nested subqueries, partitioned tables, materialised views, nonaggregate subqueries, special methods of optimisations, different languages like XQuery (Seshadri et al., 1996), magic sets and nested relational algebra has been discovered to improve the performance of the query execution on relational databases.

“Proper choice of subquery execution strategies becomes even more essential today with the increase in decision support systems and automatically generated SQL, e.g., with ad-hoc reporting tools.” (Elhemali et al., 2007) This research explains the full range of MS SQL Server’s functionality aimed to optimise the execution of subqueries. They have
defined their own operator Apply and discussed how MS SQL generated alternative query plans and how they influence the performance. “Proper choice of execution strategy from forward or reverse lookup plans to set-oriented processing is essential for satisfactory performance” (Elhemali et al., 2007)

Another approach of nested query evaluation has been proposed by (Cao and Badia, 2007). This approach is based on their own version of nested relational algebra and can be used as part of RDBMS and on top of it. They proved that this approach is “efficient and uniform approach that treats all nested queries with any type of linking operators and any level of nesting in a uniform manner.” (Cao and Badia, 2007)

(Ghionna et al., 2011) presented their own Optimizer which could be used on top of existing RDBMS. This optimizer combines cost based estimation with structural decomposition methods and provides “support to optimizing SQL queries with arbitrary output variables, aggregate operators, ORDER BY statements, and nested queries.” (Ghionna et al., 2011) (Trummer and Koch, 2014) discovered an issue of classical query optimization and introduced “the Multi-Objective Parametric Query Optimization (MPQ) problem where query plans are compared according to multiple cost metrics and the cost of a given plan according to a given metric is modeled as a function that depends on multiple parameters” Finally they presented an algorithm for this problem which guarantees the coverage of all relevant query plans.

Partitioned tables were implemented as a way to enhance the performance and reduce the total cost of ownership for storing large amounts of data. Partitioning divides huge tables into chunks for finer granularity. (Herodotou et al., 2011) discovered this feature and developed “novel partition-aware optimization techniques to generate efficient plans for SQL queries over partitioned tables” which “can generate plans that are an order of magnitude better than plans generated.” (Herodotou et al., 2011)

(Kwanglat and Natwichai, 2011) discovered an effect of query transformations to exact cardinality computing proposed by (Chaudhuri et al., 2009) for Optimizer testing. Since computation of exact cardinality is costly optimizers use cardinality estimation which could be inaccurate. A number of different transformations were applied to check if exact cardinality benefits from particular transformation. As the result, the subquery unnesting transformation showed positive effect during evaluation using TPC-H⁹. The

⁹Transaction Processing Performance Council. Benchmark H
rest of transformations, such as “Group-by View Merging”, “Join Factorization” and “Join Predicate Pushdown” were not as effective as subquery unnesting. The result of this research is not fully finished and does not show that exact cardinality computation is better than cardinality estimation, but showed that subquery unnesting transformation is worthy.

Join operations are associative and commutative, the table relation from Figure 2.3 algebraically represented as Join(Join(A,B),Join(C,D)) is equivalent to Join(A, Join(B, Join(C,D))) or Join(B, Join(C, Join(D,A))) etc. The first representation is the bushy representation which “require materialization of intermediate relations. While bushy trees may result in cheaper query plan, they expand the cost of enumerating the search space considerably” (Chaudhuri, 1998). Such permutations are not applicable for outer joins, but “when the join predicate is between (R,S) and the outer-join predicate is between (S,T), the following identity holds: Join(R, S LOJ T) = Join (R,S) LOJ T. If the above associative rule can be repeatedly applied, we obtain an equivalent expression where evaluation of the “block of joins” precedes the “block of outer-joins”. Subsequently, the joins may be freely reordered among themselves.” (Chaudhuri, 1998)

(Kim, 1982) has discovered 3 lemmas for merging nested subqueries.

Let Q1 be

\[
\begin{align*}
\text{SELECT } & R_i, C_k \\
\text{FROM } & R_i, R_j \\
\text{WHERE } & R_i.Ch = R_j.Cm
\end{align*}
\]

And let Q2 be

\[
\begin{align*}
\text{SELECT } & R_i.C_k \\
\text{FROM } & R, \\
\text{WHERE } & R_i.Ch \text{ IS IN } (\text{SELECT } R_j.C_m \\
\text{FROM } & R_j)
\end{align*}
\]

Listing 2.5: LEMMA 1. Q1 and Q2 are equivalent

Let Q3 be

\[
\begin{align*}
\text{SELECT } & R_i, C_k \\
\text{FROM } & R_i \\
\text{WHERE } & R_i.Ch = (\text{SELECT AGG}(R_j.C_m) \\
\text{FROM } & R_j \\
\text{WHERE } & R_j.Ch = R_i.Cp)
\end{align*}
\]
Further, let Q4 be

```sql
SELECT Ri.Ck
FROM Ri
WHERE Ri.Ch = (SELECT Rt.C2
FROM Rt
WHERE Rt.C1 = Ri.Cp);
```

where Rt(C1, C2) is obtained by

```sql
Rt(C1, C2) = (SELECT Rj.Ch
FROM Rj
GROUP BY Rj.Ch);
```

---

**Listing 2.6:** LEMMA 2. $Q_3$ and $Q_3$ are equivalent

Let Q5 be the following query.

```sql
SELECT Ri.Ck
FROM Ri
WHERE (SELECT Rj.Ch
FROM Rj
WHERE Rj.Ch = Ri.Cp)
OP
(SELECT Rk.Cm,
FROM Rk);
```

Further, let Q6 be

```sql
SELECT Ri.Ck
FROM Ri
WHERE Ri.Cp = (SELECT C1
FROM Rt);
```

where Rt is obtained as

```sql
Rt(C1) = (SELECT Rj.Cr
FROM Rj RX
WHERE (SELECT Rj.Ch
FROM Rj RY
WHERE RY.Ch = RX.Ch)
OP
(SELECT Rk.Cm,
FROM Rk);
```

---

**Listing 2.7:** LEMMA 3. $Q_5$ and $Q_6$ are equivalent
The associativity and commutativity properties of JOIN operation, especially of INNER JOIN operation are important for subquery unnesting transformation.

2.5 Language Recognition

A language is a way of communication when messages are composed of sentences consisting of words, which in turn made up of symbols sequences. Such words in computer science are called tokens and in contrast with natural languages they cannot be broken down any further. In this instance, SQL language has an infinite number of tokens but the combinations of such tokens would probably have infinite set of sentences. The fixed and structured sequence of words in a sentence describes the “meaning” or semantic of the sentence. To be able to understand the meaning of an SQL statement we have to define the rules of the language. The set of such rules is called grammar of the language. According to (Grune and Jacobs, 2008) “a generative grammar is an exact, fixed-size recipe for constructing the sentences in the language”. The formal definition made by (Grune and Jacobs, 2008) define generative grammar as 4-tuple \((V_N, V_T, R, S)\) such that

1. \(V_N\) and \(V_T\) are finite sets of symbols,
2. \(V_N \cap V_T = 0\),
3. \(R\) is a set of pairs \((P, Q)\) such that
   - \(3a\) \(P \in (V_N \cup V_T)^+\) and
   - \(3b\) \(Q \in (V_N \cup V_T)^*\),
4. \(S \in V_N\).

where \(V_N\) is the set of non-terminals, \(V_T\) - the set of terminals, \(R\) is the set of rules, \(S\) - is the start symbol non-terminal.

Language parsing itself is a complex subject with a lot of different parsing techniques which are described by (Grune and Jacobs, 2008). They believe that “basically almost all parsing is done by top-down search with left-recursion protection” and all of such techniques could be unified.
2.5.1 ANTLR4

ANTLR v4 is a powerful parser generator which according to (Terence, 2013) used by Twitter, Hadoop, Lex Machina, Oracle. The latter uses ANTLR4 within the SQL Developer IDE and its migration tools. “From a formal language description called a grammar, ANTLR generates a parser for that language that can automatically build parse tree, which are data structures representing how a grammar matches the input. ANTLR also automatically generates tree walker that you can use to visit the nodes of those trees to execute application-specific code.” (Terence, 2013). ANTLR is written in Java and the latest version at the moment of this research is 4.5.3 and in essence, this tool converts grammar into the set of Java classes such as `BaseListener`, `Lexer`, `Parser`, `Visitor`, `CharStream`, `Token`, `ParseTree` and others. The exact name of these classes depends on the name of the grammar file. This set of classes is a program which recognises the input written in language which follows grammar rules described in the grammar file.

2.5.1.1 Grammar

ANTLR Lexer is a program which analyses input and combines characters into words or symbols (tokens). ANTLR Parser is another program which recognises the structure of a sentence made of tokens. ANTLR Parser uses tree representation of sentences. The data flow of a language recognition displayed in Figure 2.7.

![Figure 2.7: Data flow of a language recognition](image)

ANTLR4 tool is jar archive which has also a special graphical tool to display the parse tree. For example, the parsing tree of the SQL statement 2.8 displayed in Figure 2.8.

```sql
SELECT * FROM a
```

(10) (Terence, 2013)
2.5.1.2 Development Of SQL Application Using Parse Tree

Development of SQL interpreter and translator requires the execution of some appropriate code for each word of the input SQL statement. ANTLR Parse-Tree data structure is the easiest way to do that. The diagram on Figure 2.9 shows how Parser and Lexer are interconnected by TokenStream class. “The diagram shows that leaf (token) nodes in the parse tree are containers that point at tokens in the token stream.” (Terence, 2013)

Parse-tree listener is a Java interface that responds to rule entry and exit phrase recognition events. These events are triggered by parse-tree walker in a top-down sequence. ANTLR generates the Listener for the grammar file with enter and exit methods for each rule. The rule Assign displayed in Figure 2.10 has methods enterAssign() and exitAssign() which are called in a depth-first sequence.
Figure 2.9: ANTLR Lexer and parser interconnection\textsuperscript{11}

Figure 2.10: Parse tree walker\textsuperscript{12}
Chapter 3

Implementation

3.1 Problem Specification

1C:Enterprise translates native 1C-SQL statements to SQL syntax and passes this query to RDBMS engine for execution. The process of translation contains an enrichment of the query with RLS condition if the former is switched on. RLS conditions simplify the process of scripting but 1C:Enterprise so developers do not care about record level security. Instead 1C:Enterprise implicitly changes each table with subquery which has RLS condition, inside. From other side 1C:Enterprise does not optimise the final query, even when RLS conditions are the same or have deeply nested statements.

In a real business 1C application, the number of objects(and corresponding tables) such as catalogues, documents, information and accumulation registers could be around 1000-2000 and the number of roles 100-200 items. Depending on the functional goals of the application queries there could be dozens of tables used within particular 1C-SQL and each of that tables will be enriched with RLS condition taken from corresponding roles.

For example 1C:Accounting - the best seller 1C application in Russia has 290 Catalog, 212 Document, 453 InformationRegister and 76 AccumulationRegister subclasses. The total number of different roles are 132. The number of lines of 1C script within the restriction template “ByValues” of the role “ReadAccountingData” equals 2679. This restriction template assembly the RLS condition and apply it to relevant SQL statement. For example, the following simple 1C-SQL statement enhanced with RLS and translated to T-SQL becomes more complex, bigger.
A simple 1C-SQL query displayed in Listing 3.1 reads all Sales documents and displays the UUID of the document and Contract description.

```sql
SELECT Sales.Ref,
       Sales.Contracts.Description
FROM Document.SalesProductsServices AS Sales

Listing 3.1: Initial 1C-SQL query. RLS switched off.
```

1C:Enterprise translates this query into T-SQL query displayed in Listing 3.2, where `dbo._Document229` and `dbo._Reference41` are the corresponding tables to the classes `Document.Sales` and `Catalog.Contracts`. Table `dbo._Reference41` is automatically joined to the table `dbo._Document229` by 1C:Enterprise to retrieve the description of the contract. Join condition and "WHERE" condition also strengthened by `T2._Fld521 = 0` and `T1._Fld521 = 0` to control the security between data areas. In this example, the functional option which controls "RLS" is switched OFF.

```sql
SELECT T1._IDRRef,
       T2._Description
FROM dbo._Document229 T1
LEFT OUTER JOIN dbo._Reference41 T2
ON T1._Fld6071RRef = T2._IDRRef
   AND T2._Fld521 = 0
WHERE T1._Fld521 = 0

Listing 3.2: Corresponding T-SQL query. RLS switched off.
```

After switching “RLS” functional option ON it is required to modify the initial 1C-SQL query by adding “ALLOWED"keyword. Strictly speaking, it is better to add this option to each query. It does not affect the execution of the query with RLS switched ON, the absence of this keyword with RLS switched ON will result in system alert that user does not have access to read data. Modified 1C-SQL query displayed in Listing 3.3

```sql
SELECT ALLOWED
       Sales.Ref,
       Sales.Contract.Description
FROM Document.SalesProductsServices AS Sales

Listing 3.3: Initial 1C-SQL query. RLS feature switched on.
```
Chapter 3. Implementation

The result of translation of this query to T-SQL displayed in Listing 3.4. The platform generates this query when the user with the role “ReadAccountingData” executes the initial 1C-SQL query when RLS is switched ON. The system checks what are the default and configured restrictions for the user. In this particular example, the user has permissions to read `Document.SalesProductsServices` and `Catalog.Contracts` which belong to one particular Company. The administrator of the application can easily change the rule, by adding a few more companies or changing the rule to all except the companies from the list.

```sql
SELECT T1._IDRRef, T2.Description_
FROM dbo._Document229 T1
LEFT OUTER JOIN (SELECT T3._Description COLLATE Cyrillic_General_CI_AS AS Description_, T3._Fld846RRef AS Fld846RRef, T3._IDRRef AS IDRRef, T3._Folder AS Folder_, T3._Fld521 AS Fld521_
FROM dbo._Reference41 T3
WHERE EXISTS(SELECT 1
FROM dbo._Reference71 T4
WHERE T4._Fld1170RRef = T3._Fld846RRef
AND T4._Fld1170RRef <> 0x00000000000000000000000000000000
OR T4._IDRRef = T3._Fld846RRef
AND T4._Fld521 = 0
AND T3._Folder = 0x00 OR EXISTS(SELECT 0x01 AS Q_001_F_000_
FROM dbo._Reference11872 T5
INNER JOIN dbo._Reference37 T6
ON T5._Fld11877 = 'Catalog.Contracts'
AND EXISTS(SELECT 0x01 AS Q_004_F_000_
FROM dbo._InfoRg15405 T7
WHERE T7._Fld521 = 0
AND T7._Fld15406RRef = T5._IDRRef
AND T7._Fld15407RRef = T6._IDRRef)
AND T6._IDRRef IN(SELECT T8._Reference37_IDRRef AS Q_005_F_000RRef
FROM dbo._Reference37_VT821 T8
INNER JOIN dbo._InfoRg9004 T9
ON T9._Fld9006_TYPE = 0x08
AND T9._Fld9006_RTRef = 0x00000005E

AND T8._Reference37_IDRRef AS Q_005_F_000RRef
FROM dbo._Reference37_VT821 T8
INNER JOIN dbo._InfoRg9004 T9
ON T9._Fld9006_TYPE = 0x08
AND T9._Fld9006_RTRef = 0x00000005E
```

AND T9._F1d9006_RRRef = 0

WHERE T8._F1d521 = 0
  AND T9._F1d521 = 0

) WHERE T6._F1d521 = 0

AND CASE
  WHEN T4._IDRRef IS NULL THEN 0x01
  ELSE CASE
    WHEN T4._IDRRef IS NOT NULL THEN 0x08 END
  END <> 0x01

AND CASE
  WHEN EXISTS (SELECT
    0x01 AS Q_002_F_000_
    FROM dbo._InfoRg20259 T10
    WHERE T10._F1d521 = 0
      AND T10._F1d20260RRef = T6.
  _IDRRef
      AND T10._F1d20261_TYPE = CASE
        WHEN T4._IDRRef IS NOT NULL THEN
          0x08 END
      AND T10._F1d20261_RTRef = CASE
        WHEN T4._IDRRef IS NOT NULL THEN
          0x00000047 END
      AND T10._F1d20261_RRRef = T4._IDRRef
  ) THEN 0x01
  ELSE 0x00 END = CASE
    WHEN EXISTS (SELECT
      0x01 AS Q_003_F_000_
      FROM dbo._InfoRg20266 T11
      WHERE T11._F1d521 = 0
        AND T11._F1d20267RRef = T6.
  _IDRRef
        AND T11._F1d20268_TYPE = CASE
          WHEN T4._IDRRef IS NOT NULL THEN
            0x08 END
        AND T11._F1d20268_RTRef = CASE
          WHEN T4._IDRRef IS NOT NULL THEN
            0x00000047 END
        AND T11._F1d20269 = 0x00
    ) THEN 0x01
    ELSE 0x00 END
  )
  ELSE 0x00 END
  )
AND T3._Fld521 = 0) T2
ON T1._Fld6071RRef = T2._IDRRef
WHERE T1._Fld521 = 0
    AND EXISTS(SELECT
        0x01 AS Q_001_F_000_
      FROM dbo._Reference11872 T12
      INNER JOIN dbo._Reference37 T13
      ON EXISTS(SELECT
            0x01 AS Q_004_F_000_
          FROM dbo._InfoRg15405 T14
          WHERE T14._Fld521 = 0
            AND T14._Fld15406RRef = T12._IDRRef
            AND T14._Fld15407RRef = T13._IDRRef
      )
      AND T13._IDRRef IN(SELECT
            T15._Reference37_IDRRef AS Q_005_F_000RRef
          FROM dbo._Reference37_VT821 T15
          INNER JOIN dbo._InfoRg9004 T16
          ON T16._Fld9006_TYPE = 0x08
            AND T16._Fld9006_RTRef = 0x0000005E
            AND T16._Fld9005_TYPE = T15._Fld823_TYPE
            AND T16._Fld9005_RTRef = T15._Fld823_RTRef
            AND T16._Fld9005_RRRef = T15._Fld823_RRRef
          WHERE T15._Fld521 = 0
            AND T16._Fld521 = 0
      )
      AND T12._Fld11877 = 'Document.SalesProductsServices'
WHERE T13._Fld521 = 0
    AND CASE
      WHEN EXISTS(SELECT
            0x01 AS Q_002_F_000_
          FROM dbo._InfoRg20259 T17
          WHERE T17._Fld521 = 0
            AND T17._Fld20260RRef = T13._IDRRef
            AND T17._Fld20261_TYPE = 0x08
            AND T17._Fld20261_RTRef = 0x00000047
            AND T17._Fld20261_RRRef = T1._Fld6067RRef
          ) THEN 0x01
      ELSE 0x00
      END = CASE
      WHEN EXISTS(SELECT
            0x01 AS Q_003_F_000_
          FROM dbo._InfoRg20266 T18
          WHERE T18._Fld521 = 0
            AND T18._Fld20267RRef = T13._IDRRef
            AND T18._Fld20268_TYPE = 0x08
            AND T18._Fld20268_RTRef = 0x00000047
            AND T18._Fld20268_RRRef = 0x00000047
          )
      THEN 0x01
      ELSE 0x00
      END = CASE

LISTING 3.4: Final T-SQL query. RLS feature switched on.

This query still reads data from two tables - Document.SalesProductsServices and Catalog.Contracts but RLS inflate the query with conditions which read the data from RLS configuration tables and the total number of tables becomes 18 and complexity of the query increased.

Before starting experiments it is required to define the measure of the cost of the query execution. The simplest way of comparing the cost of the execution within the MS SQL Server is to collect the IO statistics and compare the total number of logical reads for an initial and transformed query (see explanation in Section 2.3.1.1). “SET STATISTICS IO ON;” before the execution of the query will give an order for Execution Engine to display input-output statistics. “SET STATISTICS IO OFF” will switch it off.

Table 3.1 represents the cost of the execution of the initial 1C-SQL query comparing the RLS switched on and off.

<table>
<thead>
<tr>
<th>Description</th>
<th>The number of rows returned</th>
<th>The of logical reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLS off</td>
<td>3027</td>
<td>2</td>
</tr>
<tr>
<td>RLS on</td>
<td>250</td>
<td>72291</td>
</tr>
</tbody>
</table>

Table 3.1: The cost of the execution

This table does not represent the problem but gives the clear understanding of the cost comparison logic.

3.2 Research Experiments And Evaluation

3.2.1 Experiments Requirements

Table 3.2 shows relations between 15 1C metadata classes defined within the 1C application introduced in Section 2.2.6 and database tables corresponding to each metadata
class. These objects used to perform pattern elimination experiments explained later in Section 3.2.3.

<table>
<thead>
<tr>
<th>1C Object child of the Document class</th>
<th>DB Table name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document.CessionAssets</td>
<td>dbo._Document207</td>
</tr>
<tr>
<td>Document.BillFactIssued</td>
<td>dbo._Document248</td>
</tr>
<tr>
<td>Document.ContractorSettlements</td>
<td>dbo._Document161</td>
</tr>
<tr>
<td>Document.SalesCorrection</td>
<td>dbo._Document10970</td>
</tr>
<tr>
<td>Document.ProductionServicesProvisionAct</td>
<td>dbo._Document141</td>
</tr>
<tr>
<td>Document.ProductsReturnedFromCustomer</td>
<td>dbo._Document149</td>
</tr>
<tr>
<td>Document.SalesReportFromCommissionAgent</td>
<td>dbo._Document190</td>
</tr>
<tr>
<td>Document.SalesReportToCommitent</td>
<td>dbo._Document191</td>
</tr>
<tr>
<td>Document.CessionIntangibleAssets</td>
<td>dbo._Document205</td>
</tr>
<tr>
<td>Document.ProcessingServicesSales</td>
<td>dbo._Document230</td>
</tr>
<tr>
<td>Document.SalesProductsServices</td>
<td>dbo._Document229</td>
</tr>
<tr>
<td>Document.ProductsReturnedToProvider</td>
<td>dbo._Document150</td>
</tr>
<tr>
<td>Document.VATCalculationRecording</td>
<td>dbo._Document188</td>
</tr>
<tr>
<td>Document.VATDeductionRecording</td>
<td>dbo._Document189</td>
</tr>
<tr>
<td>Document.ShippedProductsSales</td>
<td>dbo._Document228</td>
</tr>
</tbody>
</table>

Table 3.2: Metadata classes of 1C application and their corresponding tables

The set of objects within the demo application in which RLS stores its rules, configured by the administrator of the database represented in Table 3.3. The table shows relations between 1C objects of RLS data set, their corresponding database tables and the total number of items for each table. This data set used to perform subquery unnesting experiments (see below in Section 3.2.2).

<table>
<thead>
<tr>
<th>1C Object</th>
<th>DB table names</th>
<th>Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>InformationRegister.TablesOfAccessGroups</td>
<td>dbo._InfoRg15405</td>
<td>7548</td>
</tr>
<tr>
<td>InformationRegister.UserGroupComposition</td>
<td>dbo._InfoRg9004</td>
<td>16</td>
</tr>
<tr>
<td>Catalog.MetadataObjectID</td>
<td>dbo._Reference11872</td>
<td>2469</td>
</tr>
<tr>
<td>Catalog.AccessGroups</td>
<td>dbo._Reference37</td>
<td>11</td>
</tr>
</tbody>
</table>
Table 3.3: The set of the objects required for RLS

Table 3.4 describes the content of 8 data sets required to perform pattern elimination experiments explained below in Section 3.2.3. Dataset 0 is the initial state of the demo application. Dataset 1 is an expansion of the dataset 0 with 500 of new 1C objects Document.SalesProductsServices and 500 of new objects Document.SalesCorrection, where each object Document.SalesProductsServices has one corresponding object Document.SalesCorrection generated on the basis of former object. The next dataset contains additional 500 objects of the class Document.SalesProductsServices and 500 of the class Document.SalesCorrection and so on. Doc.10970 is the shortened form of dbo._Document10970 and so forth.

<table>
<thead>
<tr>
<th>Dataset index</th>
<th>Doc.10970</th>
<th>Doc.229</th>
<th>Doc.230</th>
<th>Doc.205</th>
<th>Doc.207</th>
<th>Doc.191</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset 0</td>
<td>4</td>
<td>26</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Dataset 1</td>
<td>501</td>
<td>500</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Dataset 2</td>
<td>1001</td>
<td>1000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Dataset 3</td>
<td>5001</td>
<td>5000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Dataset 4</td>
<td>10001</td>
<td>10000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Dataset 5</td>
<td>15001</td>
<td>10000</td>
<td>5000</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Dataset 6</td>
<td>20001</td>
<td>10000</td>
<td>5000</td>
<td>5000</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Dataset 7</td>
<td>25001</td>
<td>10000</td>
<td>5000</td>
<td>5000</td>
<td>5000</td>
<td>4</td>
</tr>
<tr>
<td>Dataset 8</td>
<td>65001</td>
<td>50000</td>
<td>5000</td>
<td>5000</td>
<td>5000</td>
<td>4</td>
</tr>
<tr>
<td>Dataset 9</td>
<td>115001</td>
<td>50000</td>
<td>5000</td>
<td>5000</td>
<td>5000</td>
<td>50000</td>
</tr>
</tbody>
</table>

Table 3.4: The content of data sets

3.2.2 Subquery Unnesting

There are a number of possible semantic query transformations which could be applied to SQL query without affecting the number of rows returned. This experiment focused on transforming IN (subquery) and EXISTS(subquery) to INNER JOIN and Unnesting INNER JOINs which may appear on intermediate steps of transformation.

The initial query is displayed in the Listing 3.5. This query is part of the query previously introduced in the Listing 3.4. The initial query has EXISTS(subquery), IN
Chapter 3. Implementation

(subquery) and INNER JOIN as well. The first step is to change EXISTS(subquery) to INNER JOIN, the second change IN(subquery) to INNER JOIN and the last two steps are semantically equivalent forms of INNER JOIN. This query collects all RLS conditions configured by the administrator of the system. Within this query, the value 0x8B6F0011D85708FF11DECD1905C14CB0 is GUID of a user who executes this query in this instance this user is an accountant with the restricted role to read data from object Catalog.Contracts where the company is in permitted list. This particular query checks if the user has access to read the entire object.

Listing 3.5: Subquery unnesting transformation on MS SQL Server. Initial SQL statement.

Table 3.6 shows the input-output statistics of physical operations sequentially performed by MS SQL Server Engine to retrieve the data from the database during the execution process. Each line describes the number of scan counts, logical and physical reads for the operation and other statistics information.

(1 row(s) affected)
Table `_InfoRg15405`.
Scan count 0, logical reads 2, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table `_Reference11872`.
Scan count 1, logical reads 2, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table `_Reference37`.
Scan count 0, logical reads 2, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table `_InfoRg9004`.
Scan count 0, logical reads 10, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table `_Reference37_VT821`.
Scan count 1, logical reads 2, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

**Listing 3.6:** Subquery unnesting transformation on MS SQL Server. IO output statistics.

### 3.2.2.1 Experiment 1. WHERE EXISTS(subquery) To INNER JOIN

`EXISTS(subquery)` can not be directly changed to `INNER JOIN`, because of possible duplicates in the source table of the subquery which will result in Cartesian product. The `DISTINCT` keyword will solve this issue by removing duplicate items from the result. The final transformed query where `ON EXISTS(subquery)` changed to `INNER JOIN` and the resulting IO statistics displayed in the Listings 3.7 and 3.8.

```sql
SELECT 0x01 AS Q_001_F_000_ FROM dbo._Reference11872 T5 INNER JOIN dbo._Reference37 T6 ON T5._Fld11877 = 'Catalog.Contracts'
   AND T6._IDRRef IN(SELECT T8._Reference37_IDRRef AS Q_005_F_000RRef FROM dbo._Reference37_VT821 T8 INNER JOIN dbo._InfoRg9004 T9 ON T9._Fld9006_TYPE = 0x08
                       AND T9._Fld9006_RTRef = 0x0000005E
                       AND T9._Fld9006_RRRef = 0x886F0011D85708FF11DECD1905C14CB0
                       AND T9._Fld9005_TYPE = T8._Fld823_TYPE
                       AND T9._Fld9005_RTRef = T8._Fld823_RTRef
                       AND T9._Fld9005_RRRef = T8._Fld823_RRRef
                       WHERE T8._Fld521 = 0
                       AND T9._Fld521 = 0)
INNER JOIN(SELECT DISTINCT
Listing 3.7: Subquery unnesting on MS SQL Server. Experiment 1. WHERE EXISTS(subquery) to INNER JOIN.

Listing 3.8: Subquery unnesting on MS SQL Server. Experiment 1. IO statistic.

3.2.2.2 Experiment 2. WHERE Table.Column IN(subquery) To INNER JOIN

This transformation changes the structure of the initial query (Listing 3.5) by transforming WHERE T6._IDRRef IN(subquery) to INNER JOIN. Possible duplicate rows of subquery are eliminated using the DISTINCT keyword. The Listing 3.9 is the source code of transformed query and the Listing 3.10 - input-output statistics of its execution.
FROM dbo._InfoRg15405 T7
WHERE T7._Fld521 = 0
    AND T7._Fld15406RRef = T5._IDRRef
    AND T7._Fld15407RRef = T6._IDRRef

INNER JOIN (  
    SELECT DISTINCT  
        T8._Reference37_IDRRef AS Q_005_F_000RRef  
    FROM dbo._Reference37_VT821 T8  
    INNER JOIN dbo._InfoRg9004 T9  
    ON T9._Fld9006_TYPE = 0x08  
        AND T9._Fld9006_RTRef = 0x00000005E  
        AND T9._Fld9006_RRRef = 0x8B6F0011D85708FF11DECD1905C14CB0  
        AND T9._Fld9005_TYPE = T8._Fld823_TYPE  
        AND T9._Fld9005_RTRef = T8._Fld823_RTRef  
        AND T9._Fld9005_RRRef = T8._Fld823_RRRef  
    WHERE T8._Fld521 = 0  
        AND T9._Fld4521 = 0) T800  
ON T6._IDRRef = T800.Q_005_F_000RRef  
WHERE T6._Fld521 = 0

(1 row(s) affected)

Table '_InfoRg15405'. Scan count 0, logical reads 2, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table '_Reference11872'. Scan count 1, logical reads 2, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table '_Reference37'. Scan count 0, logical reads 2, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table '_InfoRg9004'. Scan count 0, logical reads 10, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table '_Reference37_VT821'. Scan count 1, logical reads 2, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Listing 3.10: Subquery unnesting on MS SQL Server. Experiment 2. IO statistic.

3.2.2.3 Experiment 3. Both Transformation From Experiments 1 And 2

This transformation is the combination of the first 3.2.2.1 and the second 3.2.2.2 transformations applied to the initial query (Listing 3.5). The result of this transformation...
displayed in the Listing 3.11 and the result of the execution in the Listing 3.12

```sql
SELECT
  0x01 AS Q_001_F_000_
FROM dbo._Reference11872 T5
INNER JOIN dbo._Reference37 T6
ON T5._Fld11877 = 'Catalog.Contracts'
INNER JOIN (SELECT DISTINCT
  T8._Reference37_IDRRef AS Q_005_F_000RRef
  FROM dbo._Reference37_VT821 T8
  INNER JOIN dbo._InfoRg9004 T9
  ON T9._Fld9006_TYPE = 0x08
  AND T9._Fld9006_RTRef = 0x0000005E
  AND T9._Fld9006_RRRef = 0x8B6F0011D85708FF11DECD1905C14CB0
  AND T9._Fld9005_TYPE = T8._Fld823_TYPE
  AND T9._Fld9005_RTRef = T8._Fld823_RTRef
  AND T9._Fld9005_RRRef = T8._Fld823_RRRef
  WHERE T8._Fld521 = 0
  AND T9._Fld521 = 0) T800
ON T6._IDRRef = T800.Q_005_F_000RRef
INNER JOIN (SELECT DISTINCT
  0x01 AS Q_004_F_000_
, T7._Fld15406RRef , T7._Fld15407RRef
  FROM dbo._InfoRg15405 T7
  WHERE T7._Fld521 = 0) T700
ON T700._Fld15406RRef = T5._IDRRef
AND T700._Fld15407RRef = T6._IDRRef
WHERE T6._Fld521 = 0
```

Listing 3.11: Subquery unnesting on MS SQL Server. Experiment 3. Both transformation from Experiments 1 and 2.

(1 row(s) affected)
Table '_Reference37'. Scan count 0, logical reads 2, physical reads 0, read-ahead
reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0, read-ahead
reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead
reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
Table '_InfoRg15405'. Scan count 1, logical reads 2, physical reads 0, read-ahead
reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
Table '_Reference11872'. Scan count 1, logical reads 2, physical reads 0, read-ahead
reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead
reads 0.
3.2.2.4 Experiment 4. Unnesting INNER JOINs

The resulting query of the Transformation 3 (Listing 3.11) has two subqueries with DISTINCT keyword. This keyword being added guarantee the same result as the initial query when subquery has duplicate rows for the set of columns included into connection predicate. More thorough analysis of the first subquery which represents the set of objects of the class InformationRegister.TablesOfAccessGroups or table dbo._InfoRg15405 shows that this class has only two dimensions Table and AccessGroup. These dimensions are in essence columns _Fld15406RRef and _Fld15407RRef of the table dbo._InfoRg15405 and both of these fields are used within the join condition. Information register class of 1C:Enterprise does not allow instantiation of the objects where the combination of values of all dimensions has duplicate combinations. This feature allows removing of the keyword DISTINCT out of the first subquery.

The second subquery reads data from two tables joined with INNER JOIN, where left table dbo._Reference37_VT821 corresponds to the class Catalog.AccessGroups.Users and dbo._InfoRg9004 to InformationRegister.UserGroupComposition class which has two dimensions UserGroup and User and both of them are used within the join condition (columns _Fld9006 and _Fld9005). The Catalog class has the property _IDRRef which is the unique identification number of any object of this class. These features of the classes allow removing of the keyword DISTINCT from the second subquery.

Both of these deductions allow removing of the DISTINCT keyword the query contains only INNER JOIN operations without any restrictions for subquery unnesting transformation.

Finally, the INNER JOIN operation is commutative and associative, which means we can unnest INNER JOINs without any risk to get the wrong result. The Listing 3.13 displays
how the initial query looks after this transformation and the Listing 3.14 displays the input-output statistics of its execution on MS SQL Server.

```sql
SELECT 0x01 AS Q_001_F_000_ FROM dbo._Reference11872 T5
INNER JOIN dbo._Reference37 T6
ON T5._Fld11877 = 'Catalog.Contracts'
INNER JOIN dbo._InfoRg15405 T7
ON T7._Fld521 = 0
    AND T7._Fld15406RRef = T5._IDRRef
    AND T7._Fld15407RRef = T6._IDRRef
INNER JOIN dbo._Reference37_VT821 T8
ON T6._IDRRef = T8._Reference37_IDRRef
INNER JOIN dbo._InfoRg9004 T9
ON T9._Fld9006_TYPE = 0x08
    AND T9._Fld9006_RTRef = 0x0000005E
    AND T9._Fld9006_RRRef = 0xB6F0011DB8708FB11DECD1905C14CB0
    AND T9._Fld9005_TYPE = T8._Fld823_TYPE
    AND T9._Fld9005_RTRef = T8._Fld823_RTRef
    AND T9._Fld9005_RRRef = T8._Fld823_RRRef
WHERE T6._Fld521 = 0
    AND T8._Fld521 = 0
    AND T9._Fld521 = 0
```


(1 row(s) affected)

Table '_Reference37'. Scan count 0, logical reads 2, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table '_InfoRg9004'. Scan count 0, logical reads 6, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table '_Reference37_VT821'. Scan count 9, logical reads 18, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table '_InfoRg15405'. Scan count 1, logical reads 2, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Table '_Reference11872'. Scan count 1, logical reads 2, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Listing 3.14: Subquery unnesting on MS SQL Server. Experiment 4. IO statistic.
3.2.2.5 Experiment 5. Semantically Equivalent Transformation

This experiment changes the structure of the transformed query from the Experiment 4 (Section 3.2.2.4) to another, semantically equivalent form and executes it to calculate the cost. The Listing 3.15 displays the new form and the Listing 3.16 the cost.

```
SELECT 0x01 AS Q_001_F_000_
FROM dbo._Reference11872 T5, dbo._Reference37 T6, dbo._InfoRg15405 T7, dbo._Reference37_VT821 T8, dbo._InfoRg9004 T9
WHERE T5._Fld11877 = 'Catalog.Contracts'
AND T6._Fld521 = 0
AND T7._Fld521 = 0
AND T8._Fld521 = 0
AND T9._Fld521 = 0
AND T6._IDRRef = T8._Reference37_IDRRef
AND T7._Fld15406RRef = T5._IDRRef
AND T7._Fld15407RRef = T6._IDRRef
AND T9._Fld9006_TYPE = 0x08
AND T9._Fld9006_RTRef = 0x00000005E
AND T9._Fld9006_RRRef = 0xB6F9011D85708FF11DECD1905C14CB0
AND T9._Fld9005_TYPE = T8._Fld823_TYPE
AND T9._Fld9005_RTRef = T8._Fld823_RTRef
AND T9._Fld9005_RRRef = T8._Fld823_RRRef
```

Listing 3.15: Subquery unnesting on MS SQL Server. Experiment 5. Semantically equivalent transformation.

(1 row(s) affected)
Table '_Reference37'. Scan count 0, logical reads 2, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
Table '_InfoRg9004'. Scan count 0, logical reads 6, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
Table '_Reference37_VT821'. Scan count 9, logical reads 18, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
Table '_InfoRg15405'. Scan count 1, logical reads 2, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
Table '_Reference11872'. Scan count 1, logical reads 2, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Listing 3.16: Subquery unnesting on MS SQL Server. Experiment 5. IO statistic.
3.2.2.6 Evaluation On MS SQL Server

A final comparison of the cost of the initial query with the cost of each resulting query from transformations 1-5 displayed in Table 3.5. All queries were executed on MS SQL Server directly from MS SQL Server Enterprise Management studio. The set of data which shows the total number of rows in tables used in queries is displayed in Table 3.3.

<table>
<thead>
<tr>
<th>Description</th>
<th>Rows returned</th>
<th>Logical reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial SQL</td>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td>Exp. 1. EXISTS to INNER JOIN</td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td>Exp. 2. IN to INNER JOIN</td>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td>Exp. 3. IN and EXISTS to INNER JOINs</td>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td>Exp. 4. unnesting INNER JOIN</td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>Exp. 5. semantic equivalent</td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>Exp. 5*. semantic equivalent with HINT</td>
<td>1</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 3.5: The cost of the execution on MS SQL Server

The comparison of the cost of the queries for MS SQL Server shows that transformations from Experiments 1, 4 and 5 tend to increase the cost, the rest of Experiments have the same cost as the cost of the initial query. The cost for the query from Experiment 1 became more expensive since Optimizer can not find a better sequence of joins, than the sequence displayed in Figure 3.1. This happens because of DISTINCT keyword for the subquery which is not semantically correct to unnest and therefore the sequence of joining of the tables is not possible. Experiments 4 and 5 shows the worse performance because the Query Optimiser made mistakes while looking for the best sequence of joining.

The default sequence of joins for the initial query according to the SQL statement displayed in Figure 3.2. The shape of this tree is the bushy tree which has 1680 of permutations of tables (Section 2.3.1).

Since that the Query optimizer has transformed the query to the left-deep representation and decided to perform joins in the sequence displayed in Figure 3.3 as it thinks is the best.
Chapter 3. *Implementation*

**Figure 3.1:** The sequence of JOINs for the query from Experiment 1

**Figure 3.2:** Default sequence of JOINs for the initial query

**Figure 3.3:** Optimised sequence of JOINs
As soon as Query Optimizer does not analyse all possible join combinations it might not find the best execution plan. For example, the cost of the query from Experiment 5 (section 3.2.2.5) decreased from 30 to 18 after using hint \texttt{OPTION (FORCE ORDER)} (see Listing 3.17) which instructed the Query Optimizer to join the tables in the exact order indicated in the query. As the result of executing the query with this hint, the cost of the query has returned to the same cost (Listing 3.18) as the cost of the initial query.

\begin{verbatim}
SELECT 0x01 AS Q_001_F_000_
FROM dbo._Reference37_VT821 T8, dbo._InfoRg9004 T9, dbo._Reference37 T6, dbo.
    ➔ _Reference11872 T5, dbo._InfoRg15405 T7
WHERE T6._Fld11877 = 'Catalog.Contracts'
    AND T6._Fld521 = 0
    AND T7._Fld521 = 0
    AND T8._Fld521 = 0
    AND T9._Fld521 = 0
    AND T6._IDRRef = T8._Reference37_IDRRef
    AND T7._Fld15406RRef = T5._IDRRef
    AND T7._Fld15407RRef = T6._IDRRef
    AND T9._Fld9006_TYPE = 0x08
    AND T9._Fld9006_RTRef = 0x0000005E
    AND T9._Fld9006_RRRRef = 0x8B6F0011D85708FF11DECD1905C14CB0
    AND T9._Fld9005_TYPE = T8._Fld823_TYPE
    AND T9._Fld9005_RTRef = T8._Fld823_RTRef
    AND T9._Fld9005_RRRRef = T8._Fld823_RRRRef
OPTION (FORCE ORDER)
\end{verbatim}

Listing 3.17: Subquery unnesting on MS SQL Server. Experiment 5*. Hints

(1 row(s) affected)
Table 'InfoRg15405'. Scan count 0, logical reads 2, physical reads 0, read-ahead
\rightarrow reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads
\rightarrow 0.
Table 'Reference11872'. Scan count 1, logical reads 2, physical reads 0, read-
\rightarrow ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead
\rightarrow reads 0.
Table 'Reference37'. Scan count 0, logical reads 2, physical reads 0, read-ahead
\rightarrow reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads
\rightarrow 0.
Table 'InfoRg9004'. Scan count 0, logical reads 10, physical reads 0, read-ahead
\rightarrow reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads
\rightarrow 0.
Table 'Reference37_VT821'. Scan count 1, logical reads 2, physical reads 0, read
\rightarrow ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead
\rightarrow reads 0.
Listing 3.18: Subquery unnesting on MS SQL Server. Experiment 5*. IO statistic.

### 3.2.2.7 Evaluation On PostgreSQL

The same SQL statements to those performed in Experiments 1-5 and their corresponding output executed on PostgreSQL are shown in Appendix A in Table A.1. Table 3.6 displays the cost of the queries executed on PostgreSQL with the same datasets used in experiments 1-5.

<table>
<thead>
<tr>
<th>Description</th>
<th>Rows returned</th>
<th>Actual rows * loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial SQL</td>
<td>1</td>
<td>8424</td>
</tr>
<tr>
<td>Exp. 1. EXISTS to INNER JOIN</td>
<td>1</td>
<td>22676</td>
</tr>
<tr>
<td>Exp. 2. IN to INNER JOIN</td>
<td>1</td>
<td>8417</td>
</tr>
<tr>
<td>Exp. 3. IN and EXISTS to INNER JOIN</td>
<td>1</td>
<td>23509</td>
</tr>
<tr>
<td>Exp. 4. unnesting INNER JOIN</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>Exp. 5. semantic equivalent</td>
<td>1</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 3.6: The cost of the execution on PostgreSQL

PostgreSQL query optimizer shows more diversity between costs for the experimental queries’ transformations. Changing EXISTS to INNER JOIN with DISTINCT keyword increase the cost from 8424 to 22676, surprisingly the cost of the query after changing IN to INNER JOIN with DISTINCT keyword reduce the cost from 8424 to 8417 which is not material anyway. The cost of both transformations is also higher than the cost of the initial query. The cost of the query after removing the DISTINCT keyword and subquery unnesting shows significantly better performance, almost 500 times better than the cost of the initial query. The same cost appears for the semantically equivalent query.

There could be another result after changing the sequence of joining tables, but PostgreSQL officially does not support hints which make this experiment difficult to implement. The author has left it for further experiments, out of this work.
3.2.3 Pattern Elimination. Joined Tables With The Same RLS Condition.

SQL queries generated by 1C:Enterprise enhanced by RLS contain slightly similar sub-queries. Elimination of such redundancy could improve the performance by providing database optimizer with better conditions. The following experiments aimed to:

- analyse the cost of the query with one complex implementation of RLS
- detect similarity patterns and eliminate them
- compare the number of rows returned by physical operations of SQL optimizer before and after such elimination.

3.2.3.1 Experiment 6. Pattern Elimination

This experiment is the analysis of an SQL query generated by 1C:Enterprise from the following 1C-SQL query:

```sql
SELECT ALLOWED
    SalesCorrection .Ref ,
    SalesCorrection . SalesDocument . Ref
FROM
    Document . SalesCorrection AS SalesCorrection
```

Listing 3.19: Initial 1C-SQL query (pattern recognition)

The document `SalesCorrection` is the special purpose document used to correct mistakes made within previously booked Sales documents and which are closed to direct modification because of accounting rules. There are 15 different types of Sales Documents. This means that the property `SalesDocument` of the class `Document.Sales` could refer to an object of one of 15 different types and the platform will translate it to T-SQL query, where all of that 15 documents will be joined to the `SalesCorrection` document with LEFT OUTER JOIN and RLS will apply complex conditions to each of 16 tables including left table. The resulting translated by 1C:Enterprise platform T-SQL query becomes very large (source code of the query is attached to this document in two files. For MS SQL Server - MSinit.sql \(^1\) and for PostgreSQL - PGinit.sql).

\(^1\) Acrobat Reader displays the list of attached files in Attachment Navigation Pane (View → Show/Hide → Navigation Panes → Attachments)
It is noticeable that the query joins `dbo._Document10970` with 15 subqueries, and `WHERE` conditions look the same. Comparison of the first two subqueries displayed in Figure 3.4. In order to highlight differences, all aliases of the same tables are harmonised. The comparison of this two subqueries shows the four differences. The first is the name of the column of the table TX in the select list. Each of these 16 documents has property `Company` and 1C:Enterprise named the corresponding column with different names. For example, the `Company` property of the class `Document.CessionAssets` corresponds to the column `_Fld5131RRef` and of the class `Document.BillFactIssued` to the column `_Fld6718RRef`. The second difference is the table TX itself, for the class `Document.CessionAssets` it is `dbo._Document207` and `dbo._Document248` for the class `Document.BillFactIssued`. The next difference is filter value from `INNER JOIN` condition like `T11872._Fld11877 = 'Document.CessionAssets'` and the last difference if the `Company` property as part of `WHERE` from the subquery statement of table `dbo._InfoRg20259`. Table 3.2 shows the relation between object names and table names for the rest of 16 documents.

Thereby the query could be transformed to a new form where the common part is moved from subqueries out to one combined `WHERE` condition with case statement. This particular transformation is based on the nature of the correction document, were each correction document always belongs to the same Company as the Sales document it is correcting. This constraint is vital since RLS checks whether the user has access to read documents which are booked on behalf of the company which is in the list of allowed companies.

Representation of entire initial query where common subquery displayed as subquery $N$, where $N \in (1, 2, \cdots, 14, 15)$ displayed in Appendix A Listing A.14.

A part of the initial query where correction document is joined with only one sales document is as follows.

```sql
SELECT T1._IDRRef,
FROM dbo._Document10970 T1
LEFT OUTER JOIN (SELECT T3._Fld5131RRef AS Fld5131RRef,
T3._IDRRef AS IDRRef,
T3._Fld521 AS Fld521_
FROM dbo._Document207 T3
WHERE EXISTS(subquery 1) AND T3._Fld521 = 0) T2
```
Figure 3.4: The difference between subquery 1 and subquery 2

ON T1._Fld11376_TYPE = 0x08 AND T1._Fld11376_RTRef = 0x000000CF AND T1.
  _Fld11376_RRRef = T2.IDRRef
WHERE (T1._Fld521 = 0) AND (EXISTS (subquery 16))

Listing 3.20: Simplified initial query (pattern recognition)

Left table T1 and the table T3 joined with LEFT OUTER JOIN. At a glance, we can not move the common predicate EXISTS(CommonSubquery) from the WHERE condition of T3 to WHERE condition of T1, because it could change the final result by removing rows from T1 which do not have a corresponding value in T3 which satisfy the condition EXISTS(CommonSubquery). From another side, we see the same common subquery already exists in WHERE condition for the table T1. Moreover, it is known from the specification that RLS checks whether the user has access to read the document by checking that the company is in the list of allowed companies, which are the same for the sales
document and its correction. These decisions guarantee the absence of rows within the table T1 which do not have the corresponding row in table T3 and this transformation is correct.

The column name TX._Fld5131RRef, the table name dbo._Document207 and the value 'Document.CessionAssets' depend on the value T1._Fld11376_RTRef. Therefore all of the subqueries 1-15 could be moved to one common pattern subquery displayed in Appendix A Listing A.13.

The final transformed query where all pattern subqueries are combined into one common part attached as file for MS SQL Server MSPE.sql and PGPE.sql for PostgreSQL.

Table 3.7 displays the cost of the execution of the initial query in comparison with the cost of transformed query where transformed query performs 20.57 times better.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Rows</th>
<th>Logical reads</th>
<th>Cost of 1 row</th>
<th>Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Initial</td>
<td>Transformed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>dataset 0</td>
<td>10723</td>
<td>46</td>
<td>46</td>
<td>0</td>
</tr>
<tr>
<td>dataset 1</td>
<td>11220</td>
<td>30182</td>
<td>10800</td>
<td>2.69</td>
</tr>
<tr>
<td>dataset 2</td>
<td>12220</td>
<td>60518</td>
<td>21754</td>
<td>4.95</td>
</tr>
<tr>
<td>dataset 3</td>
<td>20220</td>
<td>301824</td>
<td>723</td>
<td>14.93</td>
</tr>
<tr>
<td>dataset 4</td>
<td>30220</td>
<td>604812</td>
<td>2570</td>
<td>20.01</td>
</tr>
<tr>
<td>dataset 5</td>
<td>40219</td>
<td>907757</td>
<td>3341</td>
<td>22.57</td>
</tr>
<tr>
<td>dataset 6</td>
<td>50219</td>
<td>3302</td>
<td>2719</td>
<td>0.07</td>
</tr>
<tr>
<td>dataset 7</td>
<td>60219</td>
<td>3971</td>
<td>3494</td>
<td>0.07</td>
</tr>
<tr>
<td>dataset 8</td>
<td>140219</td>
<td>7359</td>
<td>8779</td>
<td>0.05</td>
</tr>
<tr>
<td>dataset 9</td>
<td>240215</td>
<td>12248</td>
<td>13957</td>
<td>0.05</td>
</tr>
<tr>
<td>average</td>
<td>61569</td>
<td>193202</td>
<td>6818</td>
<td>6.54</td>
</tr>
</tbody>
</table>

Table 3.7: The cost of the “pattern elimination” transformation on MS SQL Server.

The same initial 1C-SQL query executed on the same 1C application but installed on PostgreSQL were translated by 1C:Enterprise to an PLSQL/PG query. The cost of the initial PLSQL/PG query and transformed PLSQL/PG queries are compared in Table 3.8.
Table 3.8: The cost of the “pattern elimination” transformation on PostgreSQL.

The cost of the rest experimental transformations displayed without data for all datasets.
For example, the cost of the execution of the initial and transformed queries for the “pattern elimination” transformation executed on MS SQL Server and PostgreSQL displayed in Table 3.9.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Rows</th>
<th>Logical reads</th>
<th>Cost of 1 row</th>
<th>Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D = B/A</td>
</tr>
<tr>
<td>data set 0</td>
<td>10723</td>
<td>8929</td>
<td>30717</td>
<td>0.83</td>
</tr>
<tr>
<td>data set 1</td>
<td>11220</td>
<td>20588</td>
<td>3677936</td>
<td>1.83</td>
</tr>
<tr>
<td>data set 2</td>
<td>12220</td>
<td>33157</td>
<td>7344738</td>
<td>2.71</td>
</tr>
<tr>
<td>data set 3</td>
<td>20220</td>
<td>130156</td>
<td>36575715</td>
<td>6.44</td>
</tr>
<tr>
<td>data set 4</td>
<td>30220</td>
<td>251422</td>
<td>73090340</td>
<td>8.32</td>
</tr>
<tr>
<td>data set 5</td>
<td>40219</td>
<td>358943</td>
<td>109614559</td>
<td>8.92</td>
</tr>
<tr>
<td>data set 6</td>
<td>50219</td>
<td>455676</td>
<td>145990488</td>
<td>9.07</td>
</tr>
<tr>
<td>data set 7</td>
<td>60219</td>
<td>591976</td>
<td>182457188</td>
<td>9.83</td>
</tr>
<tr>
<td>data set 8</td>
<td>140219</td>
<td>1461769</td>
<td>473853111</td>
<td>10.42</td>
</tr>
<tr>
<td>dataset 9</td>
<td>140219</td>
<td>2579707</td>
<td>836507955</td>
<td>18.4</td>
</tr>
<tr>
<td>average</td>
<td>61569</td>
<td>589232</td>
<td>186914275</td>
<td>6.91</td>
</tr>
</tbody>
</table>

Table 3.9: The cost of the “pattern elimination” transformation

3.2.3.2 Experiment 7. Pattern Elimination With Subquery Unnesting

This experiment is a combination of “pattern elimination” and “subquery unnesting” transformations. “Pattern elimination” transformation explained in Experiment 6 (Section 3.2.3.1). The “subquery unnesting” transformation explained in Experiment 5.
Finally transformed queries are attached as files MSPESU.sql and PGPESU.sql. The cost of transformed query compared to the cost of the initial query displayed in Table 3.10.

\[
\begin{array}{|c|c|c|c|c|c|c|}
\hline
\text{RDBMS} & \text{Rows} & \text{Logical reads} & \text{Cost of 1 row} & \text{Quotient} \\
& & \text{Initial} & \text{Transformed} & \text{Initial} & \text{Transformed} & \\
\hline
\text{MSSQL} & 61569 & 193202 & 3641.5 & 6.54 & 0.05 & \textbf{125.55} \\
\text{PostgreSQL} & 61569 & 589232 & 1032585 & 6.91 & 11.50 & 0.6 \\
\hline
\end{array}
\]

Table 3.10: The cost of the combination of “pattern elimination” and “subquery unnesting” transformations.

3.2.3.3 Experiment 8. Subquery Unnesting Without Pattern Elimination

This experiment does the “subquery unnesting” transformation from transformation Experiment 5 (Section 3.2.2.5) to each subquery without “pattern elimination”. Transformed queries for MS SQL Server and PostgreSQL are attached as files MSSU.sql and PGSU.sql. The cost of transformed query compared to the cost of the initial query displayed in Table 3.11.

\[
\begin{array}{|c|c|c|c|c|c|c|}
\hline
\text{RDBMS} & \text{Rows} & \text{Logical reads} & \text{Cost of 1 row} & \text{Quotient} \\
& & \text{Initial} & \text{Transformed} & \text{Initial} & \text{Transformed} & \\
\hline
\text{MSSQL} & 61569 & 193202 & 24664.2 & 6.54 & 1.8 & 3.62 \\
\text{PostgreSQL} & 61569 & 589232 & 580770 & 6.91 & 6.56 & 1.05 \\
\hline
\end{array}
\]

Table 3.11: The cost of the “subquery unnesting” transformation.
3.2.4 Predicate expression with single CASE statements at both sides

3.2.4.1 Experiment 9. Transforming predicate expression with two single CASEs To Semi Join and Anti Semi Join conditions

This experiment aims to decrease the cost of the query (MSinit.sql) analysing the CASE statement (Listing 3.21).

```
CASE
  WHEN EXISTS(SELECT 0x01 AS Q_002_F_000_
            FROM dbo._InfoRg20259 T9
            WHERE T9._Fld521 = 0
              AND T9._Fld20260RRef = T5._IDRRef
              AND T9._Fld20261_TYPE = 0x08
              AND T9._Fld20261_RTRef = 0x00000047
              AND T9._Fld20261_RRRef = T3._Fld5131RRef
           ) THEN 0x01
  ELSE 0x00 END

= CASE
  WHEN EXISTS(SELECT 0x01 AS Q_003_F_000_
            FROM dbo._InfoRg20266 T10
            WHERE T10._Fld521 = 0
              AND T10._Fld20269 = 0x00
              AND T10._Fld20267RRef = T5._IDRRef
              AND T10._Fld20268_TYPE = 0x08
              AND T10._Fld20268_RTRef = 0x00000047
           ) THEN 0x01
  ELSE 0x00 END

Listing 3.21: initial CASE statement
```

This predicate is part of WHERE condition and is an expression which compares two values, each of these values is a single CASE statement with semi join statement - EXISTS(subquery) condition. This predicate returns TRUE if both of EXISTS(subquery) statements return TRUE or FALSE. TRUE value can be obtained after executing EXISTS(subquery1) AND EXISTS(subquery2). FALSE value can be returned by executing NOT EXISTS(subquery1) AND NOT EXISTS(subquery2). Finally the entire predicate could be transformed into EXISTS(subquery1) AND EXISTS(subquery2) OR
(NOT EXISTS(subquery1) AND NOT EXISTS(subquery2)) where comparison expression with two Semi Join is changed into search condition with two Semi Join and two Anti Semi Join displayed in Listing 3.22.

Listing 3.22: predicate with two CASES to OR

The cost of the execution of the initial and transformed queries performed on MS SQL Server (MSinit.sql, MSCAS.sql) and PostgreSQL (PGinit.sql, PGCAS.sql) displayed in
Table 3.12. This transformation shows 2.81 times better performance on MS SQL Server and does not make effect on PostgreSQL. For further simplicity this transformation called “CASE to OR” transformation.

<table>
<thead>
<tr>
<th>RDBMS</th>
<th>Rows</th>
<th>Logical reads</th>
<th>Cost of 1 row</th>
<th>Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D = B/A</td>
<td>E = C/A</td>
</tr>
<tr>
<td>MSSQL</td>
<td>61569</td>
<td>193202</td>
<td>3490.5</td>
<td>6.54</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>61569</td>
<td>589232</td>
<td>586455</td>
<td>6.91</td>
</tr>
</tbody>
</table>

Table 3.12: The cost of the “CASE to OR” transformation.

3.2.4.2 Experiment 10. Case To OR With Subquery Unnesting

This experiment is a combination of “CASE to OR” and “subquery unnesting” transformations. This experiment shows that the combination of the two transformation decrease the cost of retrieving of one row from 6.54 to 0.07 for MS SQL Server and from 6.91 to 5.68 for PostgreSQL which is 98.96 and 1.22 times better. The source code of transformed queries for considered RDBMS is attached as files MSSUCAS.sql and PGSUCAS.sql.

<table>
<thead>
<tr>
<th>RDBMS</th>
<th>Rows</th>
<th>Logical reads</th>
<th>Cost of 1 row</th>
<th>Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D = B/A</td>
<td>E = C/A</td>
</tr>
<tr>
<td>MSSQL</td>
<td>61569</td>
<td>193202</td>
<td>3490.5</td>
<td>6.54</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>61569</td>
<td>589232</td>
<td>586455</td>
<td>6.91</td>
</tr>
</tbody>
</table>

Table 3.13: The cost of the combination of “CASE to OR” and “subquery unnesting” transformations.
3.2.4.3 Experiment 11. Case To OR With Subquery Unnesting and pattern elimination

This experiment is a combination of “Pattern elimination”, “subquery unnesting” and “CASE to OR” transformations. The calculation of the cost of the initial and transformed queries shows 132.2 times better performance on MS SQL and due to inefficiency of a single “pattern elimination” transformation the combination of the three transformations on PostgreSQL remains bad and inefficient. The source code of transformed queries for considered RDBMS is attached as files MSPESUCAS.sql and PGPESUCAS.sql.

<table>
<thead>
<tr>
<th>RDBMS</th>
<th>Rows</th>
<th>Logical reads</th>
<th>Cost of 1 row</th>
<th>Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Initial</td>
<td>Transformed</td>
<td>Initial</td>
</tr>
<tr>
<td>MSSQL</td>
<td>61569</td>
<td>193202</td>
<td>3532.9</td>
<td>6.54</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>61569</td>
<td>589232</td>
<td>1035198</td>
<td>6.91</td>
</tr>
</tbody>
</table>

Table 3.14: The cost of the combination of “CASE to OR”, “subquery unnesting” and “pattern elimination” transformations.

3.2.5 Combinations of transformations

There were proposed three different transformations such as “subquery unnesting” (SU), “pattern elimination” (PE) and “CASE to OR” (CASE). The final experiment aimed to compare the cost of all of these transformations with the cost of the initial query in the following seven combinations:

- single “pattern elimination” transformation (PE)
- combination of “pattern elimination” and “subquery unnesting” transformations (PESU)
- combination of “pattern elimination”, “subquery unnesting” and “CASE to OR” transformations (PESUCAS)
- combination of “pattern elimination” and “CASE to OR” transformations (PECAS)
- single “subquery unnesting” transformation (SU)
• combination of “subquery unnesting” and “CASE to OR” transformations (SU-CAS)

• single “CASE to OR” transformation (CAS)

Each column in Table 3.15 represents whether the cost of transformed query is better (value is more than one) then the cost of the initial query or not (the value is less than one). The cost of each query is an average cost executed for 9 datasets. The first row contains the cost comparison for MS SQL Server (MSSQL) and the second for PostgreSQL.

<table>
<thead>
<tr>
<th>RDBMS</th>
<th>PE</th>
<th>PESU</th>
<th>PESUCAS</th>
<th>PECAS</th>
<th>SU</th>
<th>SUCAS</th>
<th>CAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSSQL</td>
<td>20.54</td>
<td>125.55</td>
<td>132.3</td>
<td>34.63</td>
<td>3.62</td>
<td>98.96</td>
<td>2.81</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>0</td>
<td>0.6</td>
<td>0.6</td>
<td>0</td>
<td>1.05</td>
<td>1.03</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Table 3.15: The cost of the execution of different combinations of transformations “Pattern elimination” (PE), “Subquery unnesting” (SU) and “CASE to OR” (CAS)

Files with the source code of an initial SQL query and a number of transformed queries are attached as separate files. The whole list of files is displayed in Table 3.16. Evaluation.xlsx file contains description of datasets used to make an evaluation of the cost of all queries described in Table 3.16 and two separate sheets “MSSQLServer” and “PostgreSQL” with the massages returned by Query Optimisers of these RDBMS after executing of the SQL queries on all 9 datasets.
<table>
<thead>
<tr>
<th>T-SQL</th>
<th>PLSQL/PG</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSinit.sql</td>
<td>PGinit.sql</td>
<td>an initial query</td>
</tr>
<tr>
<td>MSRB.sql</td>
<td>PGRB.sql</td>
<td>“redundant braces elimination” transformation</td>
</tr>
<tr>
<td>MSPE.sql</td>
<td>PGPE.sql</td>
<td>“pattern elimination” transformation</td>
</tr>
<tr>
<td>MSSU.sql</td>
<td>PGSU.sql</td>
<td>“subquery unnesting” transformation</td>
</tr>
<tr>
<td>MSCAS.sql</td>
<td>PGCAS.sql</td>
<td>“CASE to OR” transformation</td>
</tr>
<tr>
<td>MSPESU.sql</td>
<td>PGPESU.sql</td>
<td>“pattern elimination” and “subquery unnesting” transformations in combination</td>
</tr>
<tr>
<td>MSPECAS.sql</td>
<td>PGPECAS.sql</td>
<td>“pattern elimination” and “CASE to OR” transformations in combination</td>
</tr>
<tr>
<td>MSSUCAS.sql</td>
<td>PGSUCAS.sql</td>
<td>“subquery unnesting” and “CASE to OR” transformations in combination</td>
</tr>
</tbody>
</table>
| MSPESUCAS.sql | PGPESUCAS.sql | “pattern elimination”, “subquery unnest-  
ing” and “CASE to OR” transformations in combination |

Table 3.16: The list of source files of T-SQL queries

All possible combinations of discovered transformations showed better average performance for MS SQL Server, where the most valuable excellence showed the combination of all three transformations when the average cost of transformed query 132.3 times better then the average cost of the initial query. “Pattern elimination” transformation became the most valuable single transformation 20.54 times better then the cost of the initial query and “CASE to OR” showed the least cost improvement, 2.81 times better then the cost of the initial query. And each combined transformation showed an increasing positive effect. Evaluation of the same combinations for PostgreSQL showed
“pattern elimination” transformation as the most undesirable transformation and none of the rest transformations improved or enhanced it. The only single transformation which improved the performance was “subquery unnesting” transformation 1.05 times better cost. “CAS to OR” transformation showed almost the same cost and the combination of this transformation with “subquery unnesting” transformation showed 1.03 better performance.

### 3.2.6 Summary On Experiments

Analysis of the results of experiments performed in this work showed that even MS SQL Server has 377 different transformations on board which are either heuristics or cost based it still could not apply them for automatically generated queries because of insufficient information. For example the nature of information register objects of 1C:Enterprise, when all combinations of fields which represent dimensions of the register are unique is not discoverable by the Query optimizer. 1C:Enterprise from other side has all information about it but also do not transform the query even this transformation could remove restrictions for unnesting subqueries and result in better estimation of cardinality and selectivity of a table and give way for other transformations.

“Subquery unnesting” transformation being applied decrease the number of subqueries transforming them to join operation which gives more options for the Query Optimizer to find a better execution plan, however, Query Optimizer still makes mistakes, which has been shown by providing the Optimizer with hints.

PostgreSQL is not as smart as MS SQL Server showed that single “subquery unnesting” and the combination of “subquery unnesting” with “CASE to OR” transformation improves the performance. These conclusions made these transformation reasonable for implementing in an automatic manner.

“Pattern elimination” transformation is more difficult to implement and did not demonstrate better performance for both considered RDBMS, since that this transformation was not implemented in an automatic manner within this work.
3.3 Common SQL grammar

This work considers T-SQL and PLSQL/PG which are the SQL procedural extensions of Microsoft and Postgres. Grammar for T-SQL language is created by ANTLR community and is free to use under MIT licence (https://github.com/antlr/grammars-v4). This grammar file has been modified to support PLSQL/PG. The first difference between these extensions is representation of primitive expressions. For example a constant of an arbitrary type in PLSQL/PG can be entered using 'string'::type or boolean value. T-SQL define constants simpler as a sting. This require modification of the Lexer part of ANTLR grammar by adding new keywords TYPEPG, TRUEPG and FALSEPG (Listing 3.23).

```
TYPEPG : ':: mvarchar'
   | ':: bytea';
TRUEPG : TRUE;
FALSEPG : FALSE;
```

**LISTING 3.23:** New keywords to support PLSQL/PG

The rule “constant” of the original tsq1.g4 grammar file (Listing 3.24) modified by adding alternatives such as optional specification of a type to string alternative of the constant and two new alternatives TRUEPG and FALSEPG (Listing 3.25)

```
constant :
   STRING
   | BINARY
   | number
   | sign? (REAL | FLOAT)
   | sign? '$' (DECIMAL | FLOAT)
   ;

```

**LISTING 3.24:** Grammar rule of constant expression of T-SQL

```
constant :
   STRING TYPEPG?// string, datetime or uniqueidentifier TYPEPG added to support PLSQL/PG
   | BINARY
   | TRUEPG//added to support PLSQL/PG
   | FALSEPG//added to support PLSQL/PG
   | number
   | sign? (REAL | FLOAT)
```

**LISTING 3.25:** Grammar rule of constant expression of T-SQL
3.4 Java Application Design And Development

A Java application, an SQL to SQL translator is a tool which is designed and developed to demonstrate “sunquery unnesting” and “CASE to OR” transformations. This application aimed to analyse SQL queries which are generated by 1C:Enterprise platform and should be used carefully with SQL statements of other origin since this translator will unnest subqueries reading data from tables, where the table name contains the string `_InfoRg` or `_Reference` and could produce a Cartesian product of joined tables for such queries which does not happen with 1C:Enterprise’s queries. The data stored in rows of such tables are in essence objects of the classes `InformationRegister` and `Catalog`.

The application performs transformations discovered and explained in the experiments. The “subquery unnesting” transformation applicable where the subquery is part of a semi-join condition. This semi-join condition could be either `EXISTS(subquery)` or `table.column IN(subquery)`. “Subquery unnesting” transformation is applicable for queries containing a predicate which satisfy to the one of the following groups of requirements

**Group 1. EXISTS**

- an initial query contains a predicate `EXISTS(subquery)`
- table name of the predicate subquery starts from `_InfoRg`
- the predicate is part of an INNER JOIN condition and does not have OR neighbours and NOT prefix

**Group 2. IN**

- an initial query contains a predicate `table.column IN(subquery)`
- table source name of the predicate subquery starts from `_Reference`
• the predicate is part of an INNER JOIN condition and does not have \texttt{OR} neighbours and \texttt{NOT} prefix

• the subquery has one INNER JOIN part where the source table name starts from \_\texttt{InfoRg}

• column name equals \_\texttt{IDRRef}

The “CASE to OR” transformation is more strict and is applicable only in case when

• an initial SQL query contains \texttt{case\_expression} = \texttt{case\_expression} predicate(s)

• each \texttt{case\_expression} is in form of
\begin{verbatim}
CASE WHEN search\_condition THEN expression ELSE expression END
\end{verbatim}

• each \texttt{case\_expression} contains only one \texttt{WHERE search\_condition}

where \texttt{search\_condition} consists of a predicate in form of a semi join condition \texttt{EXISTS}(subquery)

In addition, this application performs “redundant braces elimination” transformation which removes unnecessary braces. This elimination is essential before applying the “subquery unnesting” and “CASE to OR” transformations, but can be used independently. The last function of the application is generation, formatting and displaying of transformed query.

The transformation process starts from reading the source code of an initial query provided by a user. The next step, when the user presses the “Go” button the application analyses the text and tries to interpret the language and generate a parse tree of the query. Depending on transformation option selected by the user the program either eliminates redundant braces or performs one or both of the “subquery unnesting” and “CASE to OR” transformations. Then the application traverses the parse tree from the top to the down and from the left to the right. The ANTLR4 walker traverses the tree and fire enter and exit events for each node. It analyses the tree when it enters a node, collecting all required information storing this information in intermediate data structures and modifying the part of the tree which contains already visited nodes when it exits the node. The walker is also used to generate the source code of the transformed query into human readable form. It traverses transformed parse tree in the same manner.
Figure 3.5: UML sequence diagram of the transform method of the Controller class
calling exit listeners for each node and generating the text of the query, adding spaces, newlines and indents when required. The whole process of transformation is depicted using UML sequence diagram displayed in Figure 3.5.

“Redundant braces elimination” transformation does not affect the performance of the query but is mandatory for subquery unnesting transformation, it simplifies the parse tree removing unnecessary nodes giving a program to identify whether this transformation is possible. The query, which is generated by 1C:Enterprise is difficult to read and usually contains a lot of redundant braces and this transformation makes the query easier for understanding.

The application designed in acceptance with system requirements explained in Appendix C.3. Model-View-Controller architectural pattern used to implement the application. The core part of this pattern is a Model which captures the behaviour of the application independent of the user interface. It manages the input data, intermediate data structures, logic and transformation rules. A view is a graphical user interface which provides the user with an ability to write or paste the source if the query into a text area, select options, initiate the process of transformation and see the final query.

Figure 3.6: The graphical user interface
The view also shows the statistics of the transformations - the number of transformations performed. Controller accepts user’s input and passes it to the Model and/or the View. The View is displayed in Figure 3.6.

As soon as each syntactical component of an SQL query has its own class the total number of classes equals 158, which is hardly possible to display using UML class diagram. Since that, the class diagram on Figure 3.7 comprises the core classes only.

![Figure 3.7: UML class diagram](image)

### 3.5 Recommendations For Further Activity

1C:Enterprise along with MS SQL Server and PostgreSQL supports IBM DB2 and Oracle database systems and has many other technological and functional features which could affect the performance. Installation of Oracle database requires installation of all last updates which require official subscription. Without latest patches this RDBMS does not work properly with 1C:Enterprise. IBM DB2 RDBMS has been left for further
research due to insufficient hardware capacity, insufficient time required for installation and performing research experiments.

Areas of further research:

- Evaluation of transformations on IBM DB2 and Oracle databases
- Evaluation of “subquery unnesting” with hints on PostgreSQL
- Analysis of other classes of 1C metadata, such as accumulation and calculation registers
- Comparison of the performance between supported RDBMS
- Generalise considered transformation
- Evolving the grammar rule to support the rest of PLSQL/PG features
Chapter 4

Conclusion

Rapid database development platforms are widely used to design, develop and evolve business applications. This approach can save time during configuration stage but from another side could result in long-lasting query execution. The whole process of database creation and configuration is simple and quick. 1C:Enterprise automatically does a lot for the developer who creates the application and it is not required to have a deep knowledge of databases, relational theory or database modelling. There is no need to spend a time to design the application since 1C:Enterprise generates all interfaces by default which could be redesigned later if necessary. Record level security feature simplifies the process of managing user’s permissions on row level, providing mechanisms to write complex SQL-like conditions to minimise the risk of reading and modifying secret or confidential data.

From other side 1C:Enterprise platform assembles an SQL query from chunks without a thorough analysis of the final SQL query and belongs to the power of Query Optimiser of RDBMS underneath. Query optimizers perform a multitude of query transformations which are cost-based or heuristic-based. Heuristic-based transformations do not depend on data and Query Optimizer apply them every time when possible. These transformations rely on relational algebra and the properties of SQL operators and functions. Cost-based transformations depend on selectivity and cardinality of data and require the prior gathering of data statistics and performing the comparison between multiple possible execution plans and strategies. For some queries, this calculation could take more time that the execution itself. Therefore Query Optimizers does not do an exhausted
calculation and use estimations instead to minimise the time of execution plan selection. Modern enterprise RDMBS have a lot of cost-based transformations on board.

Having that this work is focused on performing transformations based on the properties of 1C:Enterprise classes such as *Catalog*, *Document* and *InformationRegister*. Tables which are corresponding to subclasses of these classes have more constraints, for example, *InformationRegister* has dimensions and resources or values which are stored in intersections of dimensions. The *Catalog* and *Document* have the filed *Reference* which is unique for each object. This information means that “subquery unnesting” heuristic transformation can be applied in more cases that it is semantically possible.

The experiments performed in this work showed 3.62 times better performance for “sub-query unnesting” transformation on MS SQL Server and 1.05 better performance on PostgreSQL. “CASE to OR” in combination with “subquery unnesting” transformation showed 98.96 times better performance on MS SQL Server and 1.03 times better performance on PostgreSQL. Anton Shevchenko, leading developer of 1C:Enterprise platform referenced this work “Artem showed deep understanding of concepts based on real world problems. The company will consider to use the results of the work in ongoing and future projects.”

Java application tool has been created to demonstrate both of these transformations. This tool analyses an initial SQL query either T-SQL from MS SQL Server or PLSQL/PG from PostgreSQL and recognises whether these transformations are possible. Then this application transforms the query and displays the final resulting query with statistics about type and number of transformations performed.
Appendix A

Sources and experiments

A.1 Subquery Unnesting Transformation. PostgreSQL

```sql
EXPLAIN (ANALYZE) SELECT
  TRUE AS Q_001_F_000_
FROM _Reference11872 T5
INNER JOIN _Reference37 T6
ON T5._F1d1877 = 'Catalog.Contracts'::mvarchar
  AND EXISTS(SELECT
      TRUE AS Q_004_F_000_
    FROM _InfoRg15405 T7
    WHERE T7._F1d521 = 0
      AND T7._F1d15406RRef = T5._IDRRef
      AND T7._F1d15407RRef = T6._IDRRef
    )
  AND T6._IDRRef IN(SELECT
    T8._Reference37_IDRRef AS Q_005_F_000RRef
  FROM _Reference37_VT821 T8
  INNER JOIN _InfoRg9004 T9
  ON T9._F1d9006_TYPE = '\010'::bytea
    AND T9._F1d9006_RTRef = '\000\000\000\000'::bytea
    AND T9._F1d9006_RRRRef = '\213o\000\021\330W'::bytea
WHERE T9._F1d521 = 0
  AND T9._F1d521 = 0)
WHERE T6._F1d521 = 0
```

Listing A.1: Subquery unnesting transformation on PostgreSQL. Initial SQL statement.
"Nested Loop (cost=3.99..241.66 rows=1 width=0) (actual time=2.253..6.096 rows=1 loops=1)"

→ Hash Join (cost=3.71..241.23 rows=1 width=20) (actual time=0.066..3.388 rows=863 loops=1)"
  Hash Cond: (t7._fld15407rref = t6._idrref)"
  → Seq Scan on _inforg15405 t7 (cost=0.00..202.35 rows=7548 width=40) (actual time=0.009..2.468 rows=7548 loops=1)
  Filter: (_fld521 = 0::numeric)"
  → Hash (cost=3.70..3.70 rows=1 width=20) (actual time=0.066..3.388 rows=863 loops=1)"
  Buckets: 1024 Batches: 1 Memory Usage: 1kB"
  → Hash Semi Join (cost=2.52..3.70 rows=1 width=20) (actual time=0.047..0.048 rows=1 loops=1)"
  Hash Cond: (t6._idrref = t8._reference37_idrref)"
  → Seq Scan on _reference37 t6 (cost=0.00..1.14 rows=11 width=20) (actual time=0.004..0.007 rows=11 loops=1)
  Filter: (_fld521 = 0::numeric)"
  → Hash (cost=2.51..2.51 rows=1 width=20) (actual time=0.036..0.036 rows=1 loops=1)
  Buckets: 1024 Batches: 1 Memory Usage: 1kB"
  → Hash Join (cost=1.15..2.51 rows=1 width=20) (actual time=0.033..0.035 rows=1 loops=1)
  Hash Cond: ((t9._fld9005_type = t8._fld823_type) AND (t9._fld9005_rtref = t8._fld823_rtref) AND (t9._fld9005_rrref = t8._fld823_rrref))"
  → Seq Scan on _inforg9004 t9 (cost=0.00..1.32 rows=2 width=33) (actual time=0.006..0.012 rows=2 loops=1)
  Filter: ((_fld521 = 0::numeric) AND (_fld9006_type = '000'::bytea) AND (_fld9006_rref = '0000000'::bytea) AND (_fld9006_rref = '21300002133000301'::bytea) AND (_fld9006_rref = 'L260'::bytea))"
  Rows Removed by Filter: 14"
  → Hash (cost=1.06..1.06 rows=5 width=53) (actual time=0.017..0.017 rows=5 loops=1)
  Buckets: 1024 Batches: 1 Memory Usage: 1kB"
  → Index Scan using _reference11872_pkey on _reference11872 t5 (cost=0.28..0.42 rows=1 width=20) (actual time=0.003..0.003 rows=0 loops=863)
  Index Cond: (_idrref = t7._fld15406rref)"
  Filter: (_fld11872 = 'Catalog.Contracts'::mvarchar)"
  Rows Removed by Filter: 1"
"Planning time: 1.895 ms"
"Execution time: 6.191 ms"
Listing A.2: Subquery unnesting transformation on PostgreSQL. The initial query EXPLAIN(ANALYZE) output.

### A.1.1 WHERE EXISTS(subquery) To INNER JOIN

```sql
EXPLAIN (ANALYZE) SELECT
TRUE AS Q_001_F_000_
FROM _Reference11872 T5
INNER JOIN (SELECT DISTINCT T7._Fld15406RRef AS _Fld15406RRef,
T6._IDRRef as _IDRRef,
T6._Fld521 as _Fld521
FROM _Reference37 T6
INNER JOIN _InfoRg15405 T7 ON(
  T7._Idrref = 0
  AND T7._Fld15407RRef = T6._IDRRef)) T600
ON T600._Fld15406RRef = T5._IDRRef
AND T5._Fld11877 = 'Catalog.Contracts'::mvarchar
WHERE T600._IDRRef IN( SELECT
  T8._Reference37_IDRRef AS Q_005_F_000RRef
FROM _Reference37_VT821 T8
INNER JOIN _InfoRg9004 T9
ON T9._Fld9006_TYPE = '\010':bytea
  AND T9._Fld9006_RTRef = '\000\000\000\330\W':bytea
  AND T9._Fld9005_TYPE = T8._Fld823_TYPE
  AND T9._Fld9005_RTRef = T8._Fld823_RTRef
  AND T9._Fld9005_RRRef = T8._Fld823_RRRef
WHERE T8._Fld521 = 0
  AND T9._Fld521 = 0)
AND T600._Fld521 = 0
```

Listing A.3: Subquery unnesting on PostgreSQL. WHERE EXISTS(subquery) to INNER JOIN.

"Hash Semi Join (cost=374.98..554.30 rows=2 width=0) (actual time=8.419..8.953
rows=1 loops=1)"

- Hash Cond: (t6._idrref = t8._reference37_idrref)"

- "Hash Join (cost=372.46..551.75 rows=3 width=20) (actual time=7.536..8.904
rows=9 loops=1)"

- Hash Cond: (t7._fld15406rref = t5._idrref)"

- "HashAggregate (cost=364.02..439.50 rows=7548 width=43) (actual time
  =7.231..8.092 rows=7548 loops=1)"

- Group Key: t7._fld15406rref, t6._idrref, t6._fld521"
Appendix A. Sources and experiments

Listing A.4: Subquery unnesting (WHERE EXISTS(subquery) TO INNER JOIN) on PostgreSQL. EXPLAIN(ANALYZE) statistic.
A.1.2 WHERE Table.Column IN(subquery) To INNER JOIN

EXPLAIN (ANALYZE) SELECT TRUE AS Q_001_F_000 FROM _Reference11872 T5 INNER JOIN _Reference37 T6 ON T5._Fld11877 = 'Catalog.Contracts'::mvarchar AND EXISTS (SELECT TRUE AS Q_004_F_000 FROM _InfoRg15405 T7 WHERE T7._Fld521 = 0 AND T7._Fld15406RRef = T5._IDRRef AND T7._Fld15407RRef = T6._IDRRef ) INNER JOIN ( SELECT DISTINCT T8._Reference37_IDRRef AS Q_005_F_000RRef, T8._Fld521 AS _Fld521 FROM _Reference37_VT821 T8 INNER JOIN _InfoRg9004 T9 ON T9._Fld9006_TYPE = '\010 '::bytea AND T9._Fld9006_RTRef = '\000\000\000^ '::bytea AND T9._Fld9005_TYPE = T8._Fld823_TYPE AND T9._Fld9005_RTRef = T8._Fld823_RTRRef AND T9._Fld9005_RRRef = T8._Fld823_RRRef AND T9._Fld521 = 0 ) T800 ON T800._Fld521 = 0 AND T6._IDRRef = T800.Q_005_F_000RRef WHERE T6._Fld521 = 0

Listing A.5: Subquery unnesting on PostgreSQL. WHERE table.column IN(subquery) to INNER JOIN.

"Nested Loop (cost=2.83..243.30 rows=1 width=0) (actual time=2.469..5.951 rows=1 loops=1)"
"  Join Filter: (t7._fld15407rref = t6._idrref)"
"  Rows Removed by Join Filter: 10"
"  -> Nested Loop (cost=2.83..242.02 rows=1 width=40) (actual time=2.442..5.924 rows=1 loops=1)"
"    -> Hash Join (cost=2.55..241.59 rows=1 width=60) (actual time=0.100..3.117 rows=863 loops=1)
    "      Hash Cond: (t7._fld9006_rtrref = t800.q_005_f_000rref)
    "    -> Seq Scan on _inforg15405 t7 (cost=0.00..202.35 rows=7548 width=40) (actual time=0.009..2.213 rows=7548 loops=1)
"      Filter: (_fld521 = 0::numeric)"
"    -> Hash (cost=2.53..2.53 rows=1 width=20) (actual time =0.083..0.083 rows=1 loops=1)"
Appendix A. Sources and experiments

79

"Buckets: 1024  Batches: 1  Memory Usage: 1kB"

-> Subquery Scan on t800  (cost=2.51..2.53 rows=1 width=20)

  (actual time=0.082..0.083 rows=1 loops=1)

-> HashAggregate  (cost=2.51..2.52 rows=1 width=23)

  (actual time=0.081..0.081 rows=1 loops=1)

  Group Key: t8.._reference37_idrref, t8.._fld521

-> Hash Join  (cost=1.15..2.51 rows=1 width=23)

  (actual time=0.077..0.078 rows=1 loops=1)

  Hash Cond: (t9.._fld9005_type = t8.._fld823_type)

  AND (t9.._fld9005_rtrref = t8.._fld823_rtrref)

  AND (t9.._fld9005_rrrref = t8.._fld823_rrrref)

  -> Seq Scan on _inforg9004 t9  (cost=0.00..1.32 rows=2 width=33)

    (actual time=0.055..0.059 rows=2 loops=1)

    Filter: ((_fld9006_type = '010 '::bytea)

    AND (_fld9006_rtrref = '000\000\000':bytea)

    AND (_fld9006_rrrref =

    '2130\000\021\330\021\336\031\301\260':bytea)

    AND (_fld521 = 0::numeric))

  Rows Removed by Filter: 14

-> Hash  (cost=1.06..1.06 rows=5 width=5)

  (actual time=0.009..0.009 rows=5 loops=1)

  Buckets: 1024  Batches: 1  Memory Usage: 1kB"

  -> Seq Scan on _reference37_vt821 t8

  (cost=0.00..1.06 rows=5 width=56)

  (actual time=0.006..0.009 rows=5 loops=1)

  Filter: (_fld521 = 0::numeric)

  "Planning time: 1.674 ms"

  "Execution time: 6.050 ms"

Listing A.6: Subquery unnesting (WHERE Table.Column IN(subquery) To INNER JOIN) on PostgreSQL. EXPLAIN(ANALYZE) statistic.

A.1.3 Both Transformation From Experiments 1 And 2

EXPLAIN (ANALYZE) SELECT TRUE AS Q_001_F_000_
FROM _Reference11872 T5
Appendix A. Sources and experiments

INNER JOIN (SELECT DISTINCT T7._Fld15406RRef AS _Fld15406RRef, T6._IDRRef as _IDRRef, T6._Fld521 as _Fld521 FROM _Reference37 T6 INNER JOIN _InfoRg15405 T7 ON(T7._Fld15406RRef = T6._IDRRef)) T600 ON T600._IDRRef = T5._IDRRef AND T5._Fld11877 = 'Catalog.Contracts':mvvarchar

INNER JOIN (SELECT DISTINCT T8._Reference37_IDRRef AS Q_005_F_000RRef, T8._Fld521 AS _Fld521 FROM _Reference37_VT821 T8 INNER JOIN _InfoRg9004 T9 ON T9._Fld9006_TYPE = '010':bytea AND T9._Fld9006_RTRef = '00000000':bytea AND T9._Fld9006_RRRef = '213000021330W' \\010\377\021\336\315\031\005\301L\260':bytea AND T9._Fld9005_TYPE = T8._Fld823_TYPE AND T9._Fld9005_RTRef = T8._Fld823_RTRef AND T9._Fld9005_RRRef = T8._Fld823_RRRef AND T9._Fld521 = 0) T800 ON T800._Fld521 = 0 AND T600._IDRRef = T800.Q_005_F_000RRef WHERE T600._Fld521 = 0

Listing A.7: Subquery unnesting on PostgreSQL. Both transformation from Experiments 1 and 2.

"Nested Loop (cost=366.97..555.11 rows=1 width=0) (actual time=11.070..11.783 rows=1 loops=1))"
  * Join Filter: (t7._fld15406rref = t5._idrref)
  * Rows Removed by Join Filter: 862"  
  * Index Only Scan using _refer11872_byfield11890_sr on _reference11872 t5 (cost=0.41..8.42 rows=1 width=20) (actual time=0.319..0.323 rows=1 loops=1))"
  * Index Cond: (_fld11877 = 'Catalog.Contracts':mvvarchar)
  * Heap Fetched: 1"  
  * Hash Join (cost=366.57..546.21 rows=38 width=20) (actual time=9.483..11.380 rows=863 loops=1))"
  * Hash Cond: (t6._idrref = t800.Q_005_F_000rref)
  * -> HashAggregate (cost=364.02..439.50 rows=7548 width=43) (actual time=9.422..10.495 rows=7548 loops=1))"
  * Group Key: t7._fld15406rref, t6._idrref, t6._fld521"
  * -> Hash Join (cost=1.27..307.41 rows=7548 width=43) (actual time=0.031..5.507 rows=7548 loops=1))"
    * Hash Cond: (t7._fld15407rref = t6._idrref)
    * -> Seq Scan on _inforg15405 t7 (cost=0.00..202.35 rows
      =7548 width=40) (actual time=0.013..3.038 rows=7548 loops=1)"
Appendix A. Sources and experiments

81

"Filter: (_fld521 = 0::numeric)"

-> Hash (cost=1.14..1.14 rows=11 width=23) (actual time
  ↪ =0.013..0.013 rows=11 loops=1)"

  Buckets: 1024 Batches: 1 Memory Usage: 1kB"

-> Seq Scan on _reference37 t6 (cost=0.00..1.14 rows
  ↪ =11 width=23) (actual time=0.006..0.010 rows=11 loops=1)"

  Filter: (_fld521 = 0::numeric)"

-> Hash (cost=2.53..2.53 rows=1 width=20) (actual time=0.043..0.043
  ↪ rows=1 loops=1)"

  Buckets: 1024 Batches: 1 Memory Usage: 1kB"

-> Subquery Scan on t800 (cost=2.51..2.53 rows=1 width=20) (""
  ↪ actual time=0.042..0.042 rows=1 loops=1)"

-> HashAggregate (cost=2.51..2.52 rows=1 width=20) (actual
  ↪ time=0.042..0.042 rows=1 loops=1)"

  Group Key: t8._reference37_idrref , t8._fld521"

-> Hash Join (cost=1.15..2.51 rows=1 width=23) (actual
  ↪ time=0.034..0.036 rows=1 loops=1)"

  Hash Cond: ((t9._fld9005_type = t8._fld823_type)
  ↪ AND (t9._fld9005_rtref = t8._fld823_rtref) AND (t9._fld9005_rrref = t8.
  ↪ _fld823_rrref)))"

-> Seq Scan on _inforg9004 t9 (cost=0.00..1.32
  ↪ rows=2 width=33) (actual time=0.015..0.019 rows=2 loops=1)"

  Filter: ([_fld9006_type = '010'::bytea)
  ↪ AND (_fld9006_rtref = '000\000\000'::bytea) AND (_fld9006_rrref = '213o
  ↪ \000\021\330\010\377\021\336\315\031\005\301\L260'::bytea) AND (_fld521 =
  ↪ 0::numeric))"

  Rows Removed by Filter: 14"

-> Hash (cost=1.06..1.06 rows=5 width=56) (actual
  ↪ time=0.009..0.009 rows=5 loops=1)"

  Buckets: 1024 Batches: 1 Memory Usage: 1kB"

-> Seq Scan on _reference37_vt821 t8 (cost=0.00..1.06 rows=5 width=56) (actual
  ↪ time=0.003..0.005 rows=5 loops=1)"

  Filter: (_fld521 = 0::numeric)"

"Planning time: 3.770 ms"
"Execution time: 12.239 ms"

Listing A.8: Subquery unnesting (Both Transformation From Experiments 1 And 2)
on PostgreSQL Server. EXPLAIN(ANALYZE) statistic.

A.1.4 Unnesting INNER JOINs

EXPLAIN (ANALYZE) SELECT
TRUE AS Q_001_F_000_
FROM _Reference11872 T5
INNER JOIN _Reference37 T6
INNER JOIN _InfoRg15405 T7 ON
T7._Fld521 = 0
AND T7._Fld15407RRef = T6._IDRRef
ON T7._Fld15406RRef = T5._IDRRef
AND T5._Fld11877 = 'Catalog.Contracts':::mvarchar
INNER JOIN _Reference37_VT821 T8
INNER JOIN _InfoRg9004 T9
ON T9._Fld9006_TYPE = '10':::bytea
AND T9._Fld9006_RTRef = '000\000\000\000':::bytea
AND T9._Fld9006_RRRef = '210\000\021\330\W':::bytea
⇒ \010\377\021\336\315\031\005\301\L\260':::bytea
AND T9._Fld9005_TYPE = T8._Fld823_TYPE
AND T9._Fld9005_RTRef = T8._Fld823_RTRef
AND T9._Fld9005_RRRef = T8._Fld823_RRRef
AND T9._Fld521 = 0
ON T7._Fld521 = 0
AND T6._IDRRef = T8._Reference37_IDRRef
WHERE T8._Fld521 = 0

Listing A.9: Subquery unnesting on PostgreSQL. Unnesting INNER JOINs.
Appendix A. Sources and experiments

83

"Index Cond: (_fld11877 = 'Catalog.Contracts'::mvarchar)"
"Heap Fetches: 1"

"-> Index Only Scan using _infor15405_bydims15414_rr on _inforg15405 t7 (cost=0.28..8.31 rows=1 width=40) (actual time =8.225..8.226 rows=1 loops=1)"

"Index Cond: ((_fld521 = 0:: numeric) AND (_fld15407rref = t8._reference37_idrref) AND (_fld15406rref = t5._idrref))"
"Heap Fetches: 1"

"-> Index Only Scan using _referenc37_byfield837_rr on _reference37 t6 (cost =0.14..0.23 rows=1 width=20) (actual time =0.750..0.751 rows=1 loops=1)"

"Index Cond: (_idrref = t7._fld15407rref)"
"Heap Fetches: 1"

"Planning time: 4.284 ms"
"Execution time: 9.639 ms"

Listing A.10: Subquery unnesting (Unnesting INNER JOINs) on PostgreSQL.

EXPLAIN(ANALYZE) statistic.

A.1.5 Semantically Equivalent Transformation

EXPLAIN (ANALYZE) SELECT TRUE AS Q_001_F_000_ FROM _Reference11872 T5 , _Reference37 T6 , _InfoRg15405 T7 , _Reference37_VT821 T8 , _InfoRg9004 T9 WHERE T5._Fld11877 = 'Catalog.Contracts'::mvarchar AND T6._Fld521 = 0 AND T7._Fld521 = 0 AND T8._Fld521 = 0 AND T9._Fld521 = 0 AND T6._IDRRef = T8._Reference37_IDRRef AND T7._Fld15406RRef = T5._IDRRef AND T7._Fld15407RRef = T6._IDRRef AND T9._Fld9006_TYPE = '\010'::bytea AND T9._Fld9006_RTRef = '\000\000\000\000'::bytea AND T9._Fld9006_RRRef = '\213\000\021\330\010\377\021\336\315\031\005\301L\260'::bytea AND T9._Fld9005_TYPE = T8._Fld823_TYPE AND T9._Fld9005_RTRef = T8._Fld823_RTRef AND T9._Fld9005_RRRef = T8._Fld823_RRRef

Listing A.11: Subquery unnesting on PostgreSQL. Semantically equivalent transformation.

"Nested Loop (cost=1.97..19.43 rows=1 width=0) (actual time =0.148..0.154 rows=1 loops=1)"
"Join Filter: (t8. _reference37_idrref = t6. _idrref)"
"-> Nested Loop (cost=1.84..19.26 rows=1 width=40) (actual time=0.135..0.139
(rows=1 loops=1)"
"  -> Hash Join (cost=1.15..2.51 rows=1 width=20) (actual time
=0.040..0.043 rows=1 loops=1)"

"-> Seq Scan on _inforg9004 t9 (cost=0.00..1.32 rows=2 width=33)
(rows=1 loops=1)"
"  Filter: (( _fld521 = 0:: numeric ) AND ( _fld9006_type = '010 ':: bytea ) AND ( _fld9006_rtref = '000\000\000\000':: bytea ) AND ( _fld9006_rrref = '213o\000\021\330\010\377\021\336\315\031\005\301L
\260 ':: bytea ))"
"  Rows Removed by Filter: 14"

"  -> Hash (cost=1.06..1.06 rows=5 width=53) (actual time=0.010..0.010 rows=5 loops=1)"
"  Buckets: 1024 Batches: 1 Memory Usage: 1kB"

"  -> Seq Scan on _reference37_vt821 t8 (cost=0.00..1.06 rows=5 width=53)
(rows=1 loops=1)"
"  Filter: (_fld521 = 0::numeric)"

"  -> Nested Loop (cost=0.69..16.74 rows=1 width=20) (actual time
=0.093..0.094 rows=1 loops=1)"

"  -> Index Only Scan using _refer11872_byfield11890_sr on _reference11872 t5 (cost=0.41..8.42 rows=1 width=20) (actual time
=0.028..0.029 rows=1 loops=1)"
"  Index Cond: (_fld11877 = 'Catalog.Contracts ':: mvarchar)"
"  Heap Fetches: 1"

"  -> Index Only Scan using _infor15405_bydims15414_rr on _inforg15405 t7 (cost=0.28..8.31 rows=1 width=40) (actual time
=0.013..0.014 rows=1 loops=1)"
"  Index Cond: ((_fld521 = 0::numeric) AND (_fld15407rref = t8. _reference37_idrref) AND (_fld15406rref = t5. _idrref))"
"  Heap Fetches: 1"

"  -> Index Only Scan using _reference37hpk on _reference37 t6 (cost=0.14..0.16 rows=1 width=20) (actual time=0.006..0.006 rows=1 loops=1)"
"  Index Cond: ((_fld521 = 0::numeric) AND (_idrref = t7. _fld15407rref))"
"  Heap Fetches: 1"

"Planning time: 3.055 ms"
"Execution time: 0.340 ms"

Listing A.12: Subquery unnesting (Semantically Equivalent Transformation) on PostgreSQL EXPLAIN(ANALYZE) statistic.
Appendix A. Sources and experiments

A.2 Pattern Elimination Transformations. MS SQL Server

A.2.1 Pattern elimination

```sql
SELECT
  TX._IDRRef AS IDRRef,
  TX._Fld11376Ref AS Fld11376Ref,
  -- depend on the type of the field _Fld11376_TYPE of the table dbo._Document10970 T1
  TX._Fld521 AS Fld521
FROM dbo._Document229 TX
  -- depend on the type of the field _Fld11376_TYPE of the table dbo._Document10970 T1
WHERE EXISTS (SELECT
  0x01 AS Q_001_F_000_
FROM dbo._Reference11872 T11872
  INNER JOIN dbo._Reference37 T37
ON T37._IDRRef IN (SELECT
  T821._Reference37_IDRRef AS Q_005_F_000RRef
FROM dbo._Reference37_VT821 T821
  INNER JOIN dbo._InfoRg9004 T9004
ON T9004._Fld9006_TYPE = 0x08
AND T9004._Fld9006_RTRef = 0x0000005E
AND T9004._Fld9006_RRRef = 0
  AND T9004._Fld9005_TYPE = T821._Fld823_TYPE
AND T9004._Fld9005_RTRef = T821._Fld823_RTRef
AND T9004._Fld9005_RRRef = T821._Fld823_RRRef
WHERE T821._Fld521 = 0
  AND T9004._Fld521 = 0)
AND T11872._Fld11877 = 'Document.DocumentName'
  -- depend on the type of the field _Fld11376_TYPE of the table dbo._Document10970 T1
AND EXISTS (SELECT
  0x01 AS Q_004_F_000_
FROM dbo._InfoRg15405 T15405
WHERE T15405._Fld15406RRef = T11872._IDRRef
AND T15405._Fld15407RRef = T37._IDRRef
WHERE T37._Fld521 = 0
AND CASE
  WHEN EXISTS (SELECT
  0x01 AS Q_002_F_000_
FROM dbo._InfoRg20259 T20259
WHERE T20259._Fld20260RTRef = T37._IDRRef
AND T20259._Fld20261_TYPE = 0x08
AND T20259._Fld20261_RRRef = 0x00000047
  )
  AND T11872._Fld11877 = 'Document.DocumentName'
  -- depend on the type of the field _Fld11376_TYPE of the table dbo._Document10970 T1
```

AND T20259._Fld20261_RRRef = TX._Fld6067RRef
)
THEN 0x01
ELSE 0x00
END = CASE
WHEN EXISTS(SELECT
0x01 AS Q_003_F_000_
FROM dbo._InfoRg20266 T20266
WHERE T20266._Fld521 = 0
AND T20266._Fld20267RRef = T37._IDRRef
AND T20266._Fld20268_TYPE = 0x08
AND T20266._Fld20268_RTRRef = 0x00000047
AND T20266._Fld20269 = 0x00
)
THEN 0x01
ELSE 0x00
END
AND 0x08 <> 0x01)

Listing A.13: Pattern elimination on MS SQL Server. Representation of the common subquery.

SELECT -- removed redundant braces
T1._IDRRef,
CASE
WHEN T1._Fld11376_TYPE = 0x08
  AND T1._Fld11376_RTRRef = 0x000000CF THEN CASE
  WHEN T2._IDRRef IS NOT NULL THEN 0x000000CF END
WHEN T1._Fld11376_TYPE = 0x08
  AND T1._Fld11376_RTRRef = 0x000000F8 THEN CASE
  WHEN T11._IDRRef IS NOT NULL THEN 0x000000F8 END
WHEN T1._Fld11376_TYPE = 0x08
  AND T1._Fld11376_RTRRef = 0x000000A1 THEN CASE
  WHEN T20._IDRRef IS NOT NULL THEN 0x000000A1 END
WHEN T1._Fld11376_TYPE = 0x08
  AND T1._Fld11376_RTRRef = 0x000002ADA THEN CASE
  WHEN T29._IDRRef IS NOT NULL THEN 0x000002ADA END
WHEN T1._Fld11376_TYPE = 0x08
  AND T1._Fld11376_RTRRef = 0x0000008D THEN CASE
  WHEN T38._IDRRef IS NOT NULL THEN 0x0000008D END
WHEN T1._Fld11376_TYPE = 0x08
  AND T1._Fld11376_RTRRef = 0x00000095 THEN CASE
  WHEN T47._IDRRef IS NOT NULL THEN 0x00000095 END
WHEN T1._Fld11376_TYPE = 0x08
  AND T1._Fld11376_RTRRef = 0x000000BE THEN CASE
  WHEN T56._IDRRef IS NOT NULL THEN 0x000000BE END
WHEN T1._Fld11376_TYPE = 0x08
  AND T1._Fld11376_RTRRef = 0x000000BF THEN CASE
  WHEN T65._IDRRef IS NOT NULL THEN 0x000000BF END
WHEN T1._Fld11376_TYPE = 0x08
  AND T1._Fld11376_RTRRef = 0x000000CD THEN CASE
WHEN T74.IDRRef IS NOT NULL THEN 0x000000CD END
WHEN T1._F1d11376_TYPE = 0x08
    AND T1._F1d11376_RTRef = 0x000000E6 THEN CASE
    WHEN T83.IDRRef IS NOT NULL THEN 0x000000E6 END
WHEN T1._F1d11376_TYPE = 0x08
    AND T1._F1d11376_RTRef = 0x000000E5 THEN CASE
    WHEN T92.IDRRef IS NOT NULL THEN 0x000000E5 END
WHEN T1._F1d11376_TYPE = 0x08
    AND T1._F1d11376_RTRef = 0x00000096 THEN CASE
    WHEN T101.IDRRef IS NOT NULL THEN 0x00000096 END
WHEN T1._F1d11376_TYPE = 0x08
    AND T1._F1d11376_RTRef = 0x000000BC THEN CASE
    WHEN T110.IDRRef IS NOT NULL THEN 0x000000BC END
WHEN T1._F1d11376_TYPE = 0x08
    AND T1._F1d11376_RTRef = 0x000000BD THEN CASE
    WHEN T119.IDRRef IS NOT NULL THEN 0x000000BD END
WHEN T1._F1d11376_TYPE = 0x08
    AND T1._F1d11376_RTRef = 0x000000E4 THEN CASE
    WHEN T128.IDRRef IS NOT NULL THEN 0x000000E4 END
ELSE CAST ( NULL AS BINARY(4) ) END,
CASE
WHEN T1._F1d11376_TYPE = 0x08
    AND T1._F1d11376_RTRef = 0x000000CF THEN T2.IDRRef
WHEN T1._F1d11376_TYPE = 0x08
    AND T1._F1d11376_RTRef = 0x000000F8 THEN T11.IDRRef
WHEN T1._F1d11376_TYPE = 0x08
    AND T1._F1d11376_RTRef = 0x000000A1 THEN T20.IDRRef
WHEN T1._F1d11376_TYPE = 0x08
    AND T1._F1d11376_RTRef = 0x000002ADA THEN T29.IDRRef
WHEN T1._F1d11376_TYPE = 0x08
    AND T1._F1d11376_RTRef = 0x0000008D THEN T38.IDRRef
WHEN T1._F1d11376_TYPE = 0x08
    AND T1._F1d11376_RTRef = 0x00000095 THEN T47.IDRRef
WHEN T1._F1d11376_TYPE = 0x08
    AND T1._F1d11376_RTRef = 0x000000BE THEN T56.IDRRef
WHEN T1._F1d11376_TYPE = 0x08
    AND T1._F1d11376_RTRef = 0x000000BF THEN T65.IDRRef
WHEN T1._F1d11376_TYPE = 0x08
    AND T1._F1d11376_RTRef = 0x000000CD THEN T74.IDRRef
WHEN T1._F1d11376_TYPE = 0x08
    AND T1._F1d11376_RTRef = 0x000000E6 THEN T83.IDRRef
WHEN T1._F1d11376_TYPE = 0x08
    AND T1._F1d11376_RTRef = 0x000000E5 THEN T92.IDRRef
WHEN T1._F1d11376_TYPE = 0x08
    AND T1._F1d11376_RTRef = 0x00000096 THEN T101.IDRRef
WHEN T1._F1d11376_TYPE = 0x08
    AND T1._F1d11376_RTRef = 0x000000BC THEN T110.IDRRef
Appendix A. Sources and experiments

AND T1._Fld11376_RTRef = 0x000000BD THEN T119.IDRRef
WHEN T1._Fld11376_TYPE = 0x08
AND T1._Fld11376_RTRef = 0x000000E4 THEN T128.IDRRef
ELSE CAST ( NULL AS BINARY (16) ) END
FROM dbo._Document10970 T1
LEFT OUTER JOIN (SELECT
    T3._Fld5131RRef AS Fld5131RRef,
    T3._IDRRef AS IDRRef,
    T3._Fld521 AS Fld521_
FROM dbo._Document207 T3
WHERE EXISTS(subquery 1)
    AND T3._Fld521 = 0) T2
ON T1._Fld11376_TYPE = 0x08
AND T1._Fld11376_RTRef = 0x000000CF
AND T1._Fld11376_RRRef = T2.IDRRef
LEFT OUTER JOIN (SELECT
    T12._IDRRef AS IDRRef,
    T12._Fld6718RRef AS Fld6718RRef,
    T12._Fld521 AS Fld521_
FROM dbo._Document248 T12
WHERE EXISTS(subquery 2)
    AND T12._Fld521 = 0) T11
ON T1._Fld11376_TYPE = 0x08
AND T1._Fld11376_RTRef = 0x000000F8
AND T1._Fld11376_RRRef = T11.IDRRef
LEFT OUTER JOIN (SELECT
    T21._Fld521 AS Fld521_,
    T21._IDRRef AS IDRRef,
    T21._Fld3620RRef AS Fld3620RRef
FROM dbo._Document161 T21
WHERE EXISTS(subquery 3)
    AND T21._Fld521 = 0) T20
ON T1._Fld11376_TYPE = 0x08
AND T1._Fld11376_RTRef = 0x000000A1
AND T1._Fld11376_RRRef = T20.IDRRef
LEFT OUTER JOIN (SELECT
    T30._IDRRef AS IDRRef,
    T30._Fld11374RRef AS Fld11374RRef,
    T30._Fld521 AS Fld521_
FROM dbo._Document10970 T30
WHERE EXISTS(subquery 4)
    AND T30._Fld521 = 0) T29
ON T1._Fld11376_TYPE = 0x08
AND T1._Fld11376_RTRef = 0x000002ADA
AND T1._Fld11376_RRRef = T29.IDRRef
LEFT OUTER JOIN (SELECT
    T39._IDRRef AS IDRRef,
    T39._Fld2535RRef AS Fld2535RRef,
T39._Fld521 AS Fld521_
FROM dbo._Document141 T39
WHERE EXISTS(subquery 5)
   AND T39._Fld521 = 0) T38
ON T1._Fld11376_TYPE = 0x08
   AND T1._Fld11376_RTRef = 0x0000008D
   AND T1._Fld11376_RRRef = T38.IDRRef
LEFT OUTER JOIN (SELECT
   T48._IDRRef AS IDRRef,
   T48._Fld521 AS Fld521_,
   T48._Fld3284RRef AS Fld3284RRef
FROM dbo._Document149 T48
WHERE EXISTS(subquery 6)
   AND T48._Fld521 = 0) T47
ON T1._Fld11376_TYPE = 0x08
   AND T1._Fld11376_RTRef = 0x00000095
   AND T1._Fld11376_RRRef = T47.IDRRef
LEFT OUTER JOIN (SELECT
   T57._IDRRef AS IDRRef,
   T57._Fld4445RRef AS Fld4445RRef,
   T57._Fld521 AS Fld521_
FROM dbo._Document190 T57
WHERE EXISTS(subquery 7)
   AND T57._Fld521 = 0) T56
ON T1._Fld11376_TYPE = 0x08
   AND T1._Fld11376_RTRef = 0x000000BE
   AND T1._Fld11376_RRRef = T56.IDRRef
LEFT OUTER JOIN (SELECT
   T66._IDRRef AS IDRRef,
   T66._Fld4540RRef AS Fld4540RRef,
   T66._Fld521 AS Fld521_
FROM dbo._Document191 T66
WHERE EXISTS(subquery 8)
   AND T66._Fld521 = 0) T65
ON T1._Fld11376_TYPE = 0x08
   AND T1._Fld11376_RTRef = 0x000000BF
   AND T1._Fld11376_RRRef = T65.IDRRef
LEFT OUTER JOIN (SELECT
   T75._IDRRef AS IDRRef,
   T75._Fld5091RRef AS Fld5091RRef,
   T75._Fld521 AS Fld521_
FROM dbo._Document205 T75
WHERE EXISTS(subquery 9)
   AND T75._Fld521 = 0) T74
ON T1._Fld11376_TYPE = 0x08
   AND T1._Fld11376_RTRef = 0x000000CD
   AND T1._Fld11376_RRRef = T74.IDRRef
LEFT OUTER JOIN (SELECT
Appendix A. Sources and experiments

T84._Fld521 AS Fld521_,
T84._IDRRef AS IDRRef,
T84._Fld6158RRef AS Fld6158RRef
FROM dbo._Document230 T84
WHERE EXISTS(subquery 10)
   AND T84._Fld521 = 0) T83
ON T1._Fld11376_TYPE = 0x08
   AND T1._Fld11376_RTRef = 0x000000E6
   AND T1._Fld11376_RRRef = T83.IDRRef
LEFT OUTER JOIN (SELECT
   T93._IDRRef AS IDRRef,
   T93._Fld6067RRef AS Fld6067RRef,
   T93._Fld521 AS Fld521_
FROM dbo._Document229 T93
WHERE EXISTS(subquery 11)
   AND T93._Fld521 = 0) T92
ON T1._Fld11376_TYPE = 0x08
   AND T1._Fld11376_RTRef = 0x000000E5
   AND T1._Fld11376_RRRef = T92.IDRRef
LEFT OUTER JOIN (SELECT
   T102._IDRRef AS IDRRef,
   T102._Fld3340RRef AS Fld3340RRef,
   T102._Fld521 AS Fld521_
FROM dbo._Document150 T102
WHERE EXISTS(subquery 12)
   AND T102._Fld521 = 0) T101
ON T1._Fld11376_TYPE = 0x08
   AND T1._Fld11376_RTRef = 0x00000096
   AND T1._Fld11376_RRRef = T101.IDRRef
LEFT OUTER JOIN (SELECT
   T111._Fld521 AS Fld521_,
   T111._Fld4333RRef AS Fld4333RRef,
   T111._IDRRef AS IDRRef
FROM dbo._Document188 T111
WHERE EXISTS(subquery 13)
   AND T111._Fld521 = 0) T110
ON T1._Fld11376_TYPE = 0x08
   AND T1._Fld11376_RTRef = 0x000000BD
   AND T1._Fld11376_RRRef = T110.IDRRef
LEFT OUTER JOIN (SELECT
   T120._IDRRef AS IDRRef,
   T120._Fld4385RRef AS Fld4385RRef,
   T120._Fld521 AS Fld521_
FROM dbo._Document189 T120
WHERE EXISTS(subquery 14)
   AND T120._Fld521 = 0) T119
ON T1._Fld11376_TYPE = 0x08
   AND T1._Fld11376_RTRef = 0x000000BD
   AND T1._Fld11376_RRRef = T110.IDRRef
AND T1._Fld11376_RRRef = T119.IDRRef
LEFT OUTER JOIN (SELECT
  T129._Fld6047RRef AS Fld6047RRef,
  T129._IDRRef AS IDRRef,
  T129._Fld521 AS Fld521_
FROM dbo._Document228 T129
WHERE EXISTS (subquery 15)
  AND T129._Fld521 = 0) T128
ON T1._Fld11376_TYPE = 0x08
  AND T1._Fld11376_RTRef = 0x000000E4
  AND T1._Fld11376_RRRef = T128.IDRRef
WHERE T1._Fld521 = 0
  AND EXISTS (subquery 16)

Listing A.14: Pattern elimination on MS SQL Server. Representation of the initial query indicating the common subquery.

A.3 Transformations. Oracle database.

A.3.1 OR expansion

Select *
From products
Where prod_category = 'Photo' or prod_subcategory = 'Camera Media';

Listing A.15: Before OR expansion Index Access Path can not be used

Select *
From products
Where prod_subcategory = 'Camera Media'
UNION ALL
Select *
From products
Where prod_category = 'Photo'
And lnnvl (prod_subcategory = 'Camera Media');

Listing A.16: After OR expansion Index Access Path enabled

Select *
From promotions pm, products pt
Where pm.promo_id = pt.prod_id or
  pm.promo_name = pt.prod_name;

Listing A.17: Before OR expansion. Cartesian product
Select *
From promotions pm, products pt
Where pm.promo_name = pt.prod_name
UNION ALL
Select *
From promotions pm, products pt
Where pm.promo_id = pt.prod_id
And LNNVL(pm.promo_name = pt.prod_name);

Listing A.18: After OR expansion to avoid Cartesian product

A.3.2 Simple view merging

```sql
select e.first_name, e.last_name, dept_locs_v.street_address, dept_locs_v.postal_code
from employees e,
     (select d.department_id, d.department_name, l.street_address, l.postal_code
     from departments d, locations l
     where d.location_id = l.location_id) dept_locs_v
where dept_locs_v.department_id = e.department_id
and e.last_name = 'Smith';
```

Listing A.19: Before simple view merging

```sql
select e.first_name, e.last_name, l.street_address, l.postal_code
from employees e, departments d, locations l
where d.location_id = l.location_id
and d.department_id = e.department_id
and e.last_name = 'Smith';
```

Listing A.20: After simple view merging

A.3.3 Outer Join View Merging

```sql
select e1.first_name||' '||e1.last_name emp_name, dept_managers_v.manager_name, dept_managers_v.department_name
from employees e1,
     (select e2.manager_id, e2.first_name||' '||e2.last_name as manager_name, d.department_id, d.department_name
     from departments d, employees e2
     where d.manager_id = e2.employee_id) dept_managers_v
where dept_managers_v.department_id = e1.department_id(+)
and dept_managers_v.manager_id = e1.manager_id(+);
```

Listing A.21: View merging is not possible
Appendix A. Sources and experiments

Listing A.22: Before outer view merging

```sql
select e1.first_name || ' ' || e1.last_name emp_name, dept_managers_v.manager_name, dept_managers_v.department_name
from employees e1,
     (select e2.manager_id, e2.first_name || ' ' || e2.last_name as manager_name, d.department_id, d.department_name
      from departments d, employees e2
      where d.manager_id = e2.employee_id) dept_managers_v
where dept_managers_v.department_id = e1.department_id (+);
```

Listing A.23: After outer view merging

```sql
select e1.first_name || ' ' || e1.last_name emp_name,
     e2.first_name || ' ' || e2.last_name as manager_name,
     d.department_name
from employees e1, employees e2, departments d
where d.manager_id = e2.employee_id
and d.department_id = e1.department_id (+);
```
Appendix B

Development of a simple 1C application

B.1 Development

Development of a simple 1C application is required to understand the main features of 1C:Enterprise platform and especially discover the process of generation of SQL queries. The aim of the application developed in this chapter is to analyse the how particular features of the platform affect query performance. This chapter mainly is step-by-step instruction.

The first step is to launch 1C:Enterprise client and start the creation wizard by pressing “Add” button. (Figure B.1)
The next step is to specify creation of a new infobase or attachment of an existing one to the user’s infobase list. To create a new infobase select “Create infobase”. (Figure B.2) and press “Next” button. Infobase is a special term which indicate the set of configuration files and data stored in a database.

![Create database](image)

**Figure B.2:** Create database

Each 1C application can be created from a template or from scratch. Such templates usually contain tens or hundreds of objects and thousands of lines of script code to automate the set of business operations. To start developing the application from scratch select “Create an infobase without a configuration to develop a new configuration or restore previously dumped infobase” and press “Next” button. (Figure B.3)

![Database source](image)

**Figure B.3:** Database source
The next step is to give a name for the application, this name is just a description of the application preferred by the user and can be changed later at any time. “This computer or LAN computer” option tells the wizard to create database in native format without application server tier. Select “1C:Enterprise server” to be able to use external RDBMS in the next step. (see Figure B.4)

![Add infobase/group](image)

**Figure B.4: Application architecture**

The window displayed in Figure B.5 provide the user with ability to specify the name of the 1C:Enterprise application server, the name of the application to be created, the type of RDBMS it should use, the name of the database server, database user name and password. In case when the database user credentials include permissions to create databases within the database server 1C:Enterprise will create the database itself, otherwise we have to create the database manually. After creating the database 1C:Enterprise will create new application within the 1C:Enterprise application server and bind it to the database. For better performance and scalability it is better to use different independent servers for 1C:Enterprise Server, RDBMS and Web server. This installation build on virtual machine with MS Windows Server 2012 R2 to be able to use MS SQL Server. 1C:Enterprise supports Linux operation system as a platform for 1C:Enterprise application server or platform for RDBMS like Oracle or PostgreSQL. As soon as all fields are filled press “Next” button.

The final step is preconfigure startup options such as connection speed, initialization parameters, type of the client and the version of 1C:Enterprise platform to be used. Left
all fields as is and press “Finish” button. The process of creation of new application will take a few seconds. (Figure B.6)

As the result 1C:Enterprise will create a new empty application which consists from infobase within the application server and the database within RDBMS. The latter has a few tables to store the metadata of the application, the list if users and other system information. (Figure B.7) The application can now be started or configured. Press “1C:Enterprise” to open default GUI of the application which has almost nothing and require configuration. The process of configuring the application is a creation of objects and writing scripts within 1C:Enterprise Designer. 1C:Designer is IDE for developing,
debugging, team development and administering. To start configuring restart the application in 1C:Designer mode.

![Database creation completed]

**Figure B.7:** Database creation completed

1C:Designer interface displayed in Figure B.8. It has a metadata tree on left side. Creation of all objects should be done here.

![1C:Enterprise Designer]

**Figure B.8:** 1C:Enterprise Designer

In order to generate a simple SQL statement we have to create a few objects write a small piece of code and add some data to the database. Let’s start from creating of
Appendix B. Development of a simple 1C application

two Catalogs, the first to store the list of Companies and the second to store the list of Customers. Then create a Document which represents one business operation - sale of services to the customer on behalf of a company, assuming all companies are affiliated or represent a group of allied companies. Then we will write an external data processor to generate some random data and perform a simple query, and the query enlarged with record level security conditions.

There are a limited number of objects that can be created, Catalogs to store conditionally permanent information, Documents to store the result of a business operation where each operation has a time stamp and the code of operation. Report object used to configure complex reports. The rest of objects are not interesting for this work. At first we will create catalogs. Press right mouse button on Catalogs item in a metadata tree and then press “Add” button or simply press “Add” button from menu. (Figure B.9)

![Figure B.9: 1C Designer metadata tree](image)

The object creation wizard will guide the process. Give a new catalogue the name “Customers” and press “Close” button. This will create a new object “Catalog.Customers” with default properties “Code” and “Description”. (Figure B.10)
Create a new catalogue “Companies” in the same way. (see Figure B.11)

Select the “Documents” item in a metadata tree, press right button and select add to start document creation wizard. Create a new document with name “Sales”. (Figure B.12)

Select tab “Data” and add a new property “Company”, give it a type \texttt{CatalogRef.Companies}. (Figure B.13)

Add the second property “Customer” - type \texttt{CatalogRef.Customers}. (Figure B.14)
Appendix B. Development of a simple 1C application

Figure B.12: New document Sales

Figure B.13: New document attribute - “Company”
Now we can build the application, the process of compiling and building the application contains generation of a corresponding tables to each of new objects within the database. Press “F5” button or select “Start debugging” from main menu “Debug”. (Figure B.15)

1C:Enterprise mode of application appears. The platform has automatically generated the GUI for all objects, including the form to display and edit of one object and the
Appendix B. Development of a simple 1C application

Figure B.16: Accept changes

form to display the list of objects. Now we can add new items to catalogs and register a new Sales documents. (Figure B.17)

Figure B.17: Simple application ready in 1 minute
B.2 Data Generation

MS SQL Server Management Studio is a special program to manage and maintain databases. Previously generated database `v83Sample` contains tables `dbo._Document12`, `dbo._Reference10`, `dbo._Reference11`. These tables were generated to store objects of their corresponding classes such as `Document.Sales`, `Catalog.Customers`, `Catalog.Companies`. 1C:Enterprise generated these tables with clustered and non-clustered indexes. All of these tables are empty at this moment (Figure B.18).

![List of simple application’s tables within MS SQL Server](image)

Figure B.18: list of simple application’s tables within MS SQL Server

A simple script written in 1C imperative language generates 100 companies, 100 customers and 100 documents. Since 1C:Enterprise platform is interpreter it has ability to execute script without compilation it into machine language program. We can create an external utility which is called external data processor, with button “Generate” and procedure which will be executed when the user clicks the button. The same external data processor we will use to read the data. Creation wizard of external data processor
Appendix B. Development of a simple 1C application

can be called from the main menu within the 1C:DESigner - “File”, “New”. (Figure B.19)

![Figure B.19: Creation of external data processor](image)

Give the external data processor a name, right on a forms item and select “Add” to create a new form. Add a new form command “Generate”, select , drag this command and drop to the left side to generate the button. (Figure B.20)

![Figure B.20: Random data generator](image)

Return to “Generate” command and add the action, when ask create command handler “Create on client and a procedures on server”, switch to “Module” tab and add the following script. (Figure B.21) Press “Ctrl +S” to save changes into file.

Open the application in 1C:Enterprise mode, Open external data processor from file and press the button “Generate”. (Figure B.22)
Now we can check that tables are not empty using MS SQL Server Management Studio (see Figure B.23)

### B.3 Reading The Data

The next button we will configure within the external data processor is “Select”. The server procedure script displayed in Figure B.24

Before we press the “Select” button we have to configure technological journal. Configuration file “logcfg.xml” should be placed to the folder C:\ProgramFiles\1cv8\8.3.7.1917\bin\conf. Existence of this configuration file will give instruction to the platform to log all T-SQL commands being sent to the database engine for execution. Now we can press “Select” button to execute the 1C-SQL query. 1C:Enterprise will save T-SQL code to the file C:\Log\rphost_2788\16061412.log. The content of the log file displayed in Figure B.25

```csharp
&AtServer
Procedure GenerateAtServer()
    for i = 0 to 100 do
        newCompany = Catalogs.Companies.CreateItem();
        newCompany.Description = "Company " + i;
        newCompany.Write();
        newCustomer = Catalogs.Customers.CreateItem();
        newCustomer.Description = "Customer " + i;
        newCustomer.Write();
        newSale = Documents.Sales.CreateDocument();
        newSale.Date = CurrentDate();
        newSale.Company = newCompany.Ref;
        newSale.Customer = newCustomer.Ref;
        newSale.Write(DocumentWriteMode.Write);
    EndDo
EndProcedure
&AtClient
Procedure Generate(Command)
    GenerateAtServer();
EndProcedure
```
Figure B.22: Execute external data processor

Figure B.23: Data is created
Appendix B. Development of a simple 1C application

Figure B.24: Simple 1C-SQL query

![SQL query code]

We can execute the query directly within the SSMS and see the result. (Figure B.26)

The last task is to configure record level security restrictions. This require creation of a new object “Role” within the “Roles” item inside the metadata tree in 1C:Designer. Creation of this object provide developers with ability to specify RLS restrictions for objects. The Figure B.27 displays an example of restricted “Read” permission for the object Document.Sales. The example condition restricts access for users with “Restricted” role to read Sales documents where the code of Customer equals “000000001” or “000000002”. (Figure B.27)

Now, if we will start the application on behalf of user with restricted access and press “Select” button from inside external data processor 1C:Enterprise will generate another SQL query which is displayed in Figure B.28

Differences between these SQL queries without RLS and with RLS are displayed in Figure B.29
Appendix B. Development of a simple 1C application

Figure B.26: The result of the execution

Figure B.27: Record level security configuration
Appendix B. Development of a simple 1C application

Figure B.28: Simple 1C-SQL query with RLS

Figure B.29: Enlarged query
Appendix C

Project management

C.1 Project Plan

C.1.1 Project Stakeholders

The following is the list of stakeholders of the project.

- **1C Developers** will find this work very useful to understand how 1C:Enterprise platform translates 1C-SQL to SQL and what are the main reasons of why the execution of 1C-SQL becomes slow

- **Developers of platforms** that are able to automatically generate SQL queries also could find the result of this work valuable for the improvement of functionality and efficiency of their platform

- **The author** of this work expect to discover the area of SQL query performance to deepen his knowledge and develop strong base for the further job
C.1.2 Project Deliverables And Tasks

The whole scope of the project partitioned into chunks of tasks, representing the work break down structure of the project

1. Analysis and Planning
   1.1. Literature review
      1.1.1. RDBMS optimisation
         1.1.1.1. Microsoft SQL Optimiser
         1.1.1.2. Oracle cost based optimiser
         1.1.1.3. PostgreSQL Optimiser
      1.1.2. SQL parsing
      1.1.3. platforms which generates SQL queries

2. Experiments
   2.1. Development and execution of 1C-SQL queries
   2.2. Extracting corresponding SQL query from technological journal
   2.3. Manual transformation using RDBMS environment
   2.4. Execution of transformed query directly on RDBMS
   2.5. Collecting the data about the cost of the execution

3. Design
   3.1. Lexical grammar for SQL
   3.2. SQL parser and translator

4. Implementation
   4.1. Lexical grammar for SQL
   4.2. SQL parser and translator

5. Testing and Evaluation
   5.1. testing environment
      5.1.1. Virtual machine installation
Appendix C. Project management

5.1.2. Oracle database installation

5.1.3. MS SQL Server installation

5.1.4. PostgreSQL Server installation

5.2. query execution and calculation of the cost

5.3. comparison and evaluation

C.1.3 Gantt Chart of the Project

Figure C.1: Gantt Chart of the Project
C.2 Professional, Legal, Ethical And Social Issues

C.2.1 Professional Issues

Information technology (Software systems) as a programme of education of Heriot-Watt University is accredited by Chartered Institute of IT (BCS). This indicates the quality of the programme and presume students to follow BCS Code of Conduct.

ANTLR4 library developed by Terence Parr under BSD license and described in his book (Terence, 2013). The author decided to implement the application under the same license, providing users with user manual and well-commented code and publish the project https://github.com/ArKarpov/queryOptimiser.

This will make this application accessible for everyone interested in SQL query performance tuning and gives them a chance to make changes in the application for their purposes. SQL query optimisation area continue improving. RDBMS such as Oracle and MS SQL Server implement new features from version to version. This could make functionality of the application out of date.

The author will develop the application in respect of BSC Code of Conduct.

C.2.2 Legal Issues

To develop the application the author planning to use freely accessible software to generate SQL queries, store and manage the data with trial or open-source RDBMS.

This work is inspired by rapid development with 1C:Enterprise platform. 1C LLC company provide developers with free training version of 1C:Enterprise platform This requires registration on the web site 1c-dn.com. In this work the author used the latest version of the platform to generate the set of different SQL queries. Licences for the platform are gently given to the author by 1C company to perform this research. MS SQL Express, PostgreSQL and Oracle database are also freely available to download (in more details see section C.3.

ANTLR4 library as an open-source parser library used under the “BSD license” without any modifications.
C.2.3 Ethical And Social Issues

This work will not be involved in children or young people under 18, vulnerable people (elderly, physically or mentally ill, people with learning difficulties, in care, bereaved, prisoners, others), participants who do not understand English or have low functional literacy. There will be no risks or hazards to the researcher.

C.3 Requirement Analysis

C.3.1 Functional Requirements

Input for the program should be a text file or a text containing the source of the initial T-SQL query generated by 1C:Enterprise platform. Output is a T-SQL query - the resulting query after subquery unnesting transformation explained in Section 3.2.2. The program should:

- allow the user to specify what type of subquery unnesting transformation should be analysed
- allow the user to specify whether deletion of redundant braces should be done
- recognise if the initial query can be transformed with specified transformation
- remove redundant braces if necessary
- perform specified transformations and display the resulting query
- display the statistics of how many times selected transformations has been performed
- transform the structure of the query without affecting an amount of requested data

C.3.2 Non-Functional Requirements

- written on Java and packed into executable jar file
- have user friendly interface
C.3.3 System and Software Requirements

1C:Enterprise platform is 3 tier application with 1C:Enterprise application server tier, Database Server tier and client tier. The platform supports the following RDBMS: 1C, MS SQL Server, PostgreSQL, IBM DB2 and Oracle Database. "1C:Enterprise 8 system of programs is intended for automation of everyday enterprise activities: various business tasks of economic and management activity, such as management accounting, business accounting, HRM, CRM, SRM, MRP, etc. 1C:Enterprise 8 system consists of two parts: an integrated framework (aka “1C:Enterprise 8 platform”) and a set of applied solutions created and executed in the framework.” (1C, 2016b)

The latest version of 1C Enterprise platform at the moment of writing this work was 8.3.7.1917. Training version was available at 1c-dn.com. The author had professional version downloaded from users.v8.1c.ru, which allowed installation of the client onto OS X operation system and did not have limitations built-in into the training version. 1C Enterprise platform application server was installed on MS Windows 2012 R2 virtual machine. 1C Enterprise platform client was installed on MAC OS X El Capitan version 10.11.3 and on the virtual machine.

Microsoft SQL Server 2014 and PostgreSQL 9.4.2-1.1C database management systems was installed on the same virtual machine with MS Windows Server 2012 R2 on board.

PostgreSQL with all patches required available at releases.1c.ru.

MS SQL Server 2014 was downloaded from microsoft.com and was available for 180 day standard evaluation period.

Virtual machine was created with Oracle VM VirtualBox 5.0 virtualbox.org.

Windows 2012 R2 operation system downloaded from microsoft.com. This gave 180 days period to try which was enough for research.

To create Java application tool the author selected Java programming language version 1.8 and ANTLR4 tool 4.5.3 library antlr.org and Eclipse IDE for Java Developers version Mars.2 Release (4.5.2).
Bibliography

1C, L. (2016a). 1c enterprise. Available at: http://1c-dn.com/1c_enterprise/what_is_1c_enterprise/.


1C, L. (2016d). 1c script. Available at: http://1c-dn.com/1c_enterprise/1c_enterprise_script/.


1C, L. (2016f). The system of access rights. Available at: http://1c-dn.com/1c_enterprise/access_rights/.


Date, C. J. (2009). *SQL and Relational Theory: How to Write Accurate SQL Code.* O’REILLY.


