DAS Departamento de Automação e Sistemas
CTC Centro Tecnológico
UFSC Universidade Federal de Santa Catarina

OptImatch: System with Knowledge Base for Query Performance Problem Determination of Query Execution Plans

Relatório submetido à Universidade Federal de Santa Catarina

como requisito para a aprovação da disciplina:

DAS 5511: Projeto de Fim de Curso

Guilherme Fetter Damasio

Florianópolis, Março de 2017

OptImatch: System with Knowledge Base for Query Performance Problem Determination of Query Execution Plans

Guilherme Fetter Damasio

Banca Examinadora:

Piotr Mierzejewski/IBM
Orientador na Empresa

Prof. Leandro Buss Becker
Orientador no Curso

Prof. Hector Bessa Silveira
Responsável pela disciplina

Prof. Jomi Fred Hübner, Avaliador

Mateus Sant'Ana, Debatedor

Vinicius Ghizoni da Silva, Debatedor

Acknowledgements

First, I would like to thank my parents for all the love they gave me, as well as education, encouragement and support. Also, thank you for the countless times you have helped me throughout my journey at the university.

Special thanks to the Universidade Federal de Santa Catarina (UFSC) and to the entire faculty, who with their excellent structure and administration provided me with the best conditions to achieve my goals.

To my advisor, Leandro Buss Becker, for his support and mentorship during the development of this work.

To my UOIT Advisor, Jaroslaw Szlichta, for giving me the opportunity to develop this project and for the guidance and support through the execution of it.

To my IBM collaborators, Piotr Mierzejewski, Calisto Zuzarte and Vincent Corvinelli, for all the work and help in the development of this project.

I would like to thank members of the DB2 optimizer development and support teams for their feedback and guidance through the development of the OptImatch system.

Finally, to all my colleagues and friends who, in a way, contributed to the accomplishment of this work, all were very important, so I leave here registered my thanks.

Resumo Extendido

A determinação de problemas de performance de queries de banco de dados é normalmente feita analisando os planos de execução de queries, ou em inglês, *query execution plan* (QEP), além de outras propriedades de performance. Algumas ferramentas de diagnose de performance no mercado tem a capacidade de analisar queries problemáticas, porém elas estão limitadas a comparar as queries com um número limitado de padrões pré-definidos. Ainda mais, ferramentas como IBM® Optim Query Tuner® e IBM Optim Workload Tuner®[1]já fazem recomendações de aperfeiçoamento de queries para problemas já conhecidos. Mesmo sendo efetivas, tais ferramentas não proporcionam a habilidade de criar problemas padrões customizáveis devido ao fato delas não entenderem completamente a complexa estrutura dos QEPs. Conforme os dados guardados para análise aumentam, para poder analisá-los, a complexidade da query também aumenta. Fazer a análise manual de QEPs tão complexas pode ser uma tarefa demorada e que requer um grande conhecimento dos especialistas no assunto.

O presente trabalho foi desenvolvido em parceria com a IBM *Centre for Advanced Studies Toronto*. IBM é uma multinacional americana, tendo como sua principal atuação o mercado de hardware, middleware e software. A ferramenta proposta está relacionada com o sistema de banco de dados DB2 da IBM. O DB2 é um sistema de banco de dados relacionais utilizado principalmente por empresas. Devido ao grande volume de dados, o DB2 provê funcionalidades como o aperfeiçoamento de queries através da reescrita da mesma. Ao rodar uma query no DB2, o sistema produz um plano de acesso que especifica como os dados são obtidos das tabelas. Para a escolha do plano de acesso, o otimizador produz vários planos de acesso alternativos e seleciona aquele que tem o menor custo estimado de plano de execução.

Quando o otimizador falha, devido à experiência e tempo necessário, os usuários procuram as empresas para sanar o problema. Na IBM, os problemas reportados por clientes são analisados por profissionais com experiência em linguagem SQL e análise de otimizador de QEPs. Para analisar os problemas de otimização, esses profissionais utilizam de uma ferramenta do DB2 que produz um arquivo QEP. Esses arquivos são escritos em formato de texto legível para humanos, explicando o plano de acesso. Tais arquivos podem conter milhares de linhas e a análise manual do mesmo pode consumir muito tempo.

Visando sanar tais problemas, foi desenvolvida neste trabalho uma ferramenta chamada OptImatch, que tem a funcionalidade de procurar por diferentes problemas padrões dentro de um QEP e prover recomendações criadas por especialistas e salvas na base de

[1] IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml

conhecimento, abreviada por KB (*knowledge base*). Um grande problema das ferramentas existentes no mercado é a falta de uma interface que provê ao usuário uma maneira de criar seu problema padrão, pois as mesmas não conseguem impor uma estrutura para os QEPs. OptImatch, através de um sistema web, provê uma interface fácil e intuitiva para usuários criarem problemas padrões customizáveis e pesquisarem recomendações automaticamente dentro de seus QEPs.

A solução proposta é dada pela transformação do QEP para um grafo RDF e pela pesquisa dos problemas padrões utilizando o SPARQL como linguagem de query estruturada. Ao fazer upload dos QEPs, a ferramenta analisa os arquivos procurando, através de expressões regulares, as propriedades e conexões, salvando as mesmas em um arquivo RDF. Depois, através da interface do sistema, o usuário monta seu problema padrão customizado, onde o mesmo é enviado ao servidor e automaticamente transformado em uma query através da linguagem SPARQL. Tal query posteriormente será utilizada para procurar o problema customizado dentro dos arquivos RDF. Juntamente com essa pesquisa, o OptImatch também procura pelos problemas padrões pré-definidos e guardados dentro do KB. Para a criação das recomendações uma sintaxe própria foi criada. Através dessa sintaxe o usuário pode utilizar tags para indicar uma porção da query, apontando para propriedades específicas da mesma. Através dos manipuladores de tags, a ferramenta proposta substitui as tags pelos resultados obtidos, criando assim uma recomendação mais específica para cada usuário.

Mesmo que a ferramenta descrita nesse trabalho seja focada em determinação de problemas de performance de queries, a mesma pode ser aplicada para qualquer tipo de software de determinação de problemas genéricos. OptImatch necessita somente que tais softwares produzam um arquivo de diagnóstico estruturado que necessita analise futura.

Por fim, cinco experimentos são apresentados neste trabalho. O principal foco desses experimentos é analisar a efetividade da pesquisa da ferramenta em 1000 QEPs de usuários reais da IBM, avaliando a performance e escalabilidade da ferramenta em termos de quantidade de QEP, número de *low level plan operators* (LOLEPOPs) e número de recomendações. Também foi feito uma analise do novo método de pesquisa da ferramenta, comparando a mesma com o método anterior. Por último, foi realizado um estudo comparativo entre a velocidade e precisão da ferramenta proposta e os métodos de pesquisa hoje utilizados na IBM, mostrando os benefícios que o OptImatch poderá trazer a empresa.

Palavras-chave:Problema de Determinação de Performance de Queries, Web Semântica, Base de Conhecimento e Inteligência de Negócios.

Abstract

Query performance problem determination in databases is often done by analyzing query execution plans (QEPs) in addition to other properties. The majority of performance diagnostic tools help identify problematic queries. Unfortunately, most query tuning tools provide recommendations to only few known problems. Furthermore, manual analysis of QEPs requires deep knowledge in both SQL and analyzing optimizers as the complexity of the query workloads in companies increase. A tool called OptImatch was created to provide a simple way to search different problem patterns created by users in QEPs. Moreover, OptImatch matches and provides recommendations, created by experts and users, stored in a knowledge base. The proposed tool transforms the QEP into a RDF graph and automatically generate a SPARQL query from a user-based problem pattern described in the GUI by the use of handlers. The SPARQL query is then run against the RDF graph and any matching portion is returned to the user. OptImatch also searches for predefined problem patterns by scanning the knowledge base against the RDF graph, returning a recommendation for any match. In the running time, the system adapts the recommendation to the user through the handler tagging interface. An analysis of the performance and scalability of the system was performed using a real-world query workload. Also, a comparative user study was performed to evaluate the advantages of this tool in contrast to manual search as used inside IBM.

Keywords:Query Performance Problem Determination, Semantic Web, Knowledge Bases and Business Intelligence.

List of Figures

List of Tables

List of Algorithms

List of abbreviations and acronyms

BI	Business Intelligence
DA	Data Warehousing
DBA	Database Administrator
DBMS	Database Management System
GUI	Graphical User Interface
HSJOIN	Hash Join
JSON	JavaScript Object Notation
LOJ	Left Outer Join
LOLEPOP	Low Level Plan Operators
MGJOIN	Merge Join
NLJOIN	Nested Loop Join
QEP	Query Execution Plan
RDF	Resource Script Framework
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
TBSCAN	Table Scan

Contents

1 Introduction

Data, on a daily basis, is becoming more and more essential to the global economy. Industry keeps building more and more data warehousing and business intelligence to store data. This data keeps increasing as the industry continually keeps populating DAs. According to [Raph e Margy 2013], we have seen databases grow from megabytes to gigabytes to terabytes to petabytes, yet the basic challenge of DW/BI systems has remained remarkably constant. A company's data is typically used by business users in order to support decision making. The retrieval of this information is facilitated by the use of a relational query language, such as structured query language (SQL).

Given the increase of data stored, queries are becoming more and more complex in order to retrieve the needed information in a form suitable to analysis. As the complexity of queries increase and data store grow large, they are run no longer run just during work hours, but also during weekends. Moreover, analytic queries are no longer used just for batch reports, but are now an imperative part of the business operation, so the performance of these queries is that much more essential.

The creation of such complex queries is not something easily achieved, as it needs expertise in both database and SQL like query languages. Database systems are becoming more sophisticated as they can now adaptively tune the environment that they operate in. Furthermore, database administrators can analyze performance issues in a database by using general query performance problem determination tools [Zilio et al. 2004] [Alton et al. 2004]. Using these tools requires their own level of expertise to use, understand and interpret findings, but does not require deep knowledge about query execution plans (QEPs), nor expertise of an optimizer. However, they lack the customization and improvements often needed in order to be used by general end-user for the problem determination and tuning process. As a result, normally this process is done manually.

Manual search for performance issues is not just time consuming, but also requires expertise in structured query languages when analyzing big and complex queries, which are normally found in data warehouses. When facing a problem, DBAs reports it to their vendor, so experts in both SQL and analyzing optimizer QEPs can give recommendations to the user. This approach is not only difficult to scale, but also consumes a lot of effort and requires experts with deep knowledge in the matter.

Tools such as IBM® Optim Query Tuner® and IBM Optim Workload Tuner® already provide tuning recommendation for problem determination. However, they are restricted for only simple and already created problem patterns. They do not provide a way for users to create their own custom problem pattern. Furthermore, while very effective, problem determination tools do not completely understand the complex structure of QEPs, limiting the characteristics that can be searched. Sometimes users, with no experience or deep knowledge, want to search for simple questions inside the QEPs. Unfurtunately, there are no tools in the market that propose a general problem pattern search by providing both already created problem patterns and custom search of arbitrary patterns in order to make diagnosis of performance and interactive analysis.

Even for database administrators, given the complexity of QEP files, it is sometimes is difficult and time consuming to determine the solution to a problem. As there is no existing tools that allow the creation of this custom problem pattern, the experts rely on more generic searching tools such as *grep* to monitoring of files. It is easy to see that this technique is not scalable and starts to get more difficult to monitor as the number of files increase.

The tool proposed in this work, OptImatch, was developed in partnership with the IBM Center for Advanced Studies Toronto. IBM is an American multinational, having as its main activity the market for hardware, middleware and software. The proposed tool is related to the IBM DB2 relational database system. At IBM, the analysis of problem reported by DBAs is normally done over the QEP file outputted by the DB2. To analyze optimization problems, experts use the $db2exfmt$ command to produce a QEP file. These files are written in human-readable text, explaining the access plan. Such files may contain thousands of lines and manual analysis can be time consuming.

Because of these restrictions, we decided to create a tool (OptImatch) that provides a way to automatically find solutions in QEPs workloads, decreasing time, complexity, skill and expertise to search for problem patterns (list of operators and properties interconnected that describes a problem).

In order to maintain the usability and scalability of databases, commercial relational database systems such as IBM DB2®, Oracle®, and Microsoft® SQL Server® need to improve their queries. The proposed tool can make the process faster and easier to find solutions in a workload. Furthermore, there are tools that already create user queries by providing a user interface where the user is able to set some few properties. However, there is no limit of the size of these queries, normally passing 1000 lines, making it hard to analyze. Also, normally these queries consist of nesting subqueries which leads to repetitiveness of properties. In these types of scenarios, if the optimizer fails, improving performance is really time-consuming and needs deep expertise to do so manually.

With the aim to save time of experts, this kind of search was automated as much as possible. OptImatch minimizes the effort to accomplish the analysis. The tool uses abstracted artefact structure (RDF) to represent the information about the QEPs and SPARQL [Harris e Seaborne 2013] to query the information needed by the user. SPARQL is beneficial to OptImatch as it can retrieve the required and optional properties. Even more, this structured query language allows recursive search by using the property path functionality. Recursive query can search for relationships that are not directly connected, meaning that between the connections, there can exist a path with other operators. Moreover, SPARQL can search for patterns that appears multiple times in the same file and it has an efficient way to perform graph transversal analysis, decreasing time for analysis.

Even though the purpose of this work is to use OptImatch to search for query performance problem determination, it can applied for any general software problem determination. To accomplish this, the proposed tool just requires that the software outputs, automatically or dynamically, a diagnostic file that eventually requires further analysis by experts.

1.1 Contributions

The main contribution of this work is to offer a web tool that can automatically search for query performance problem determination by allowing user to create custom problem patterns and allowing experts to populate our knowledge base with pre-defined problem patterns.

This web tool was developed to transform QEP into a RDF graph format by mapping features of the QEP into a set of entities with properties and relationships between them. Even more, a web-based graphical user interface is provided for users to create custom problem patterns. OptImatch transforms the problem pattern created by the user into a SPARQL query by the use of handlers. These handlers automatically generate variables used in the SPARQL queries. A matching system was developed in order to execute the SPARQL query against the RDFs and retrieve the portions of the RDFs that match the problem pattern. Moreover, a workload of real-world IBM customer query is presented to illustrate the issues related to query problem performance.

Another contribution of this work is a knowledge base that can be populated by users and experts with predefined problem patterns and recommendations explaining how to fix a problem. OptImatch analyses the QEPs against the problem patterns stored in the knowledge base and any match is returned for the user with the recommendation. The proposed system can also find handlers in the recommendation in order to provide more specific solutions. A syntax was created aiming to provide the system a handler tagging interface. In the recommendation the expert can provide the specific problem with the query execution plan (static semantics) and by the use of the syntax, provide where the problem is and how to fix it (dynamic semantic).

Furthermore, an evaluation of the proposed tool over the real-world IBM customer problem pattern is provided. First OptImatch was tested in terms of performance and scalability in four different scenarios. Lastly, a user study was performed, showing the benefits of the tool over the manual analysis in terms of time and precision.

1.2 Manuscript Organization

The following chapters are organized as follows.

- Chapter 2 provides a description of the IBM company, how a QEP is generated as well as how the process of searching for problem patterns is done nowadays, showing the problems faced by experts, expanding the problem where a solution will be proposed. Lastly, an introduction of the other adopted technologies used in the proposed work is provided in this chapter.
- Chapter 3 explains the main functionalities of the proposed framework, describing the technologies used and how is the flow of the activities inside the tool. Also, information about data integration and the system database, such as its structure, tables and connections are presented in this chapter.
- Chapter 4 shows the developed graphical user interface, explaining how the user can use it to create a problem pattern, setting all properties and relationships. Also, this chapter introduces the concepts necessary to understand how the properties for each node are connected. Even more, an explanation of how a recommendation can be created and some extra functionalities that can be found inside OptImatch is also provided.
- Chapter 5 describes the transformation engine. The transformation engine is responsible for all transformations of data in the tool. This chapter will provide information about how the parser is made inside OpImatch, as well as how the RDF is created from the parser. Lastly, the automatically generation of the SPARQL is explained.
- Chapter 6 explains the recommendation syntax. In order to give a more specific recommendation for the user, a recommendation syntax was created, so the expert can surround the static piece with the dynamic piece in a recommendation. This dynamic piece, at running time, is replaced by the result of each QEP file. Even more, an explanation of how to use our syntax to accomplish the creation of a recommendation will be provided.
- Chapter 7 shows how OptImatch matches the pattern against the QEP workload. To achieve it, the tool run the SPARQL query auto-generated by our transformation engine against the RDF file. Also, the information returned to the user as a result is presented in this chapter. Moreover, this work shows how the proposed tool finds solutions in the knowledge base and replaces the dynamic piece of the recommendation

by the data of the query analyzed. Lastly, this chapter explains how OptImatch attaches the solutions in the result returned to the user.

- Chapter 8 provides an experimental study divided in 5 tests. First an analysis of the speed and scalability of the tool depending in the size of the workload (Number of QEPs) is provided. The next experiment measures the performance of OptImatch over the number of LOLEPOPs inside a QEP. The third one is related to the number of recommendations, so a study of how the tool performs with different number of recommendations is performed and explained. The next test is a comparative search experiment, comparing asynchronous and synchronous search. Lastly, a comparative user study was created, comparing the performance of OptImatch against experts inside IBM searching for problem patterns in query workloads.

- Chapter 9 concludes the current work, summarizing what was done and the results achieved, comparing them to the current process of search query performance problems at IBM.

2 Background

In this chapter an introduction to the technologies used along the present work (DB2, RDF, SPARQL, Jena, Dojo toolkit, D3.js and JSON) is given. Also, a background about QEPs is provided. Those already familiar with these concepts may skip this chapter and proceed to Chapter 3.

2.1 IBM Company and Related Technologies

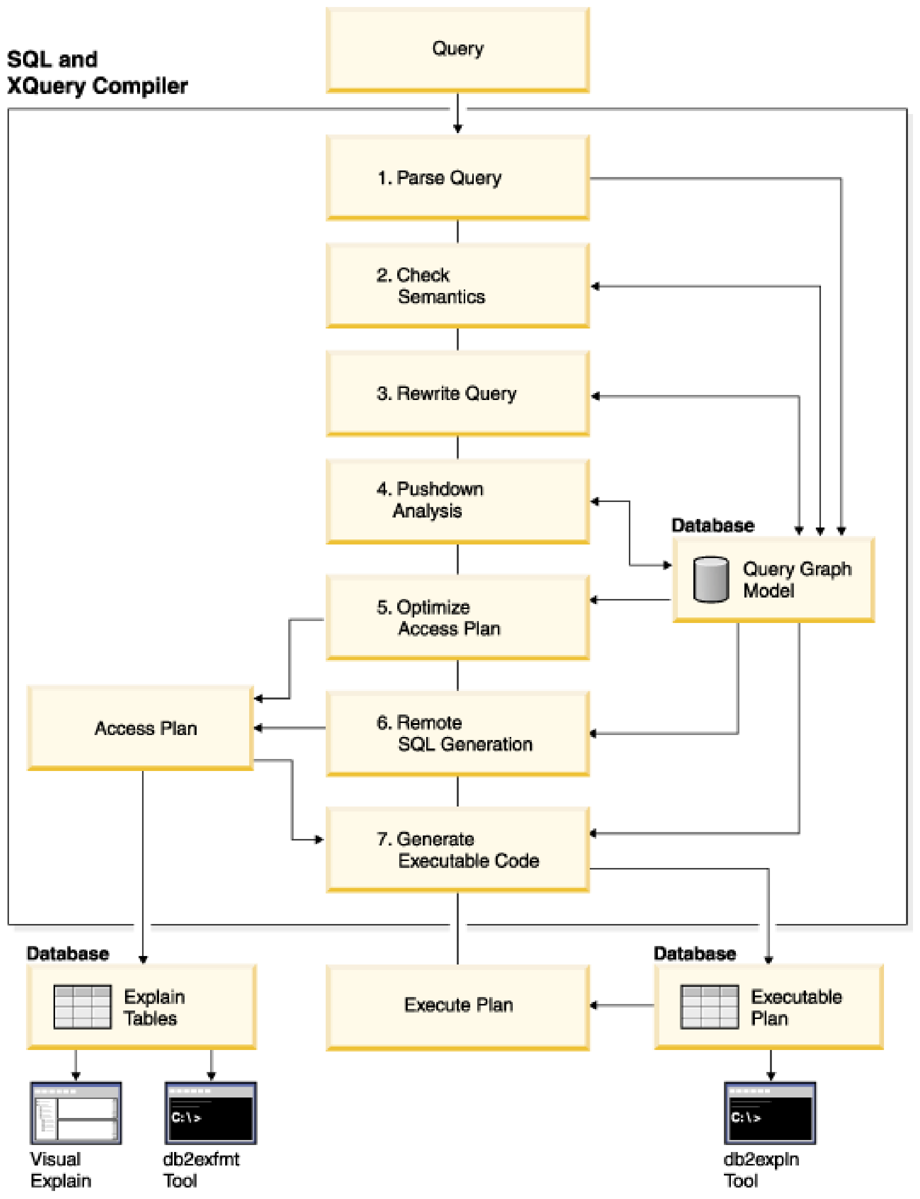
The project described in this thesis is in a partnership with IBM. IBM is an American multinational company originated in 1911 as the Computing-Tabulating-Recording Company (CRN) and was renamed International Business Machine (IBM) in 1924. The main core of the company is the development and sales of hardware, middleware and software. IBM is well known by its DB2 database and by its supercomputer Watson.

IBM DB2 is a database software mostly used in the enterprise's world. It is flexible, being possible to use it in the cloud, on-premises and hybrid. Moreover, it has massive scalability as it supports massive volumes of data. To increase performance of queries IBM DB2 provides capabilities such as tuning help and tuning queries by rewriting them. Tools such IBM Cognos® automatically generate customer queries. These queries essentially have no limit on their length, some going up thousands lines.

Figure 1 represents the SQL and XQuery compiler process of DB2. It shows the several steps required to reproduce an access plan that can be executed. According to [IBM], first the user enters the query to be tuned. With the input the compiler parses the query and analyzes it to validate the syntax. The second step is to check semantics by looking for inconsistencies among parts of the statement (e.g., making sure all relations mentioned by the query actually exist). If no inconsistencies are found, the compiler rewrites the query into a form that can be optimized more easily, storing the result in the query graph model. The next step is a pass through pushdown analysis and then a optimization access plan. With the query graph model, the optimizer generates many alternative execution plans for satisfying the query. The compiler estimates the cost of each of these plans and chooses the plan with the smallest estimated execution plan. The output of the optimizer is an access plan, and details about this access plan are captured in the explain tables. The resulting query plan, or sequence of actions the DBMS will perform to answer the query, is passed to the remote SQL generation and, lastly, goes to the execution engine, which has the responsibility for executing each of the steps in the chosen query plan. In other words, the execution engine generates an executable access plan, or section, for the query.

Sometimes, when optimizers fail, customers reach out to their vendors in order to open a problem report given the expertise and time required to do the manual analysis of the problem. At IBM, the problem reports are analyzed by experts that are well versed in both SQL and analyzing optimizer QEPs. This analysis is normally done over the QEP file outputted by the DB2. When an SQL query is executed, the DB2 optimizer tool defines the access plan to access the data. The access plan is stored in the explain tables. By using the *db2exfmt* command, the compiler reads the explain tables and outputs it in a human readable text form format, creating the aforementioned file. The QEP file details the access path, containing information about the query execution plan, like the estimate cost for a specific query, allowing the expert to analyze how the data is accessed and how the performance can be improved.

Figure 1 – SQL and XQuery compiler



Source: [IBM]

QEP files are composed by the query execution plan diagnostic information. The data in the file includes information about base objects (tables, indexes and views), operators (join, group-by, fetch) as well as costs, estimated number of rows and other characteristics about the operators. Also, it includes information about the explain instance, such as the DB2 version, database context (CPU speed, buffer pool size) and package context (SQL type, optimization level).

Some properties are identified in the QEP file as a tree diagram, while others are represented in textual blocks identified by a operator number. Figure 3 depicts a piece of the tree diagram. The three diagram is a graphical representation of the access plan, showing the order of the operators utilized to retrieve the data. Each operator of this graph is called LOLEPOP (LOw LEvel Plan OPerator). Attached to the LOLEPOP, in the tree, there is information about the estimate number of rows (estimate cardinality), cumulative total cost, cumulative I/O cost and its own ID. Figure 2 represents a LOLEPOP and its characteristics.

Figure 2 – Low Level Plan Operator (LOLEPOP)

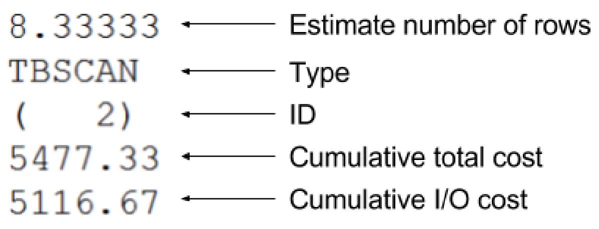
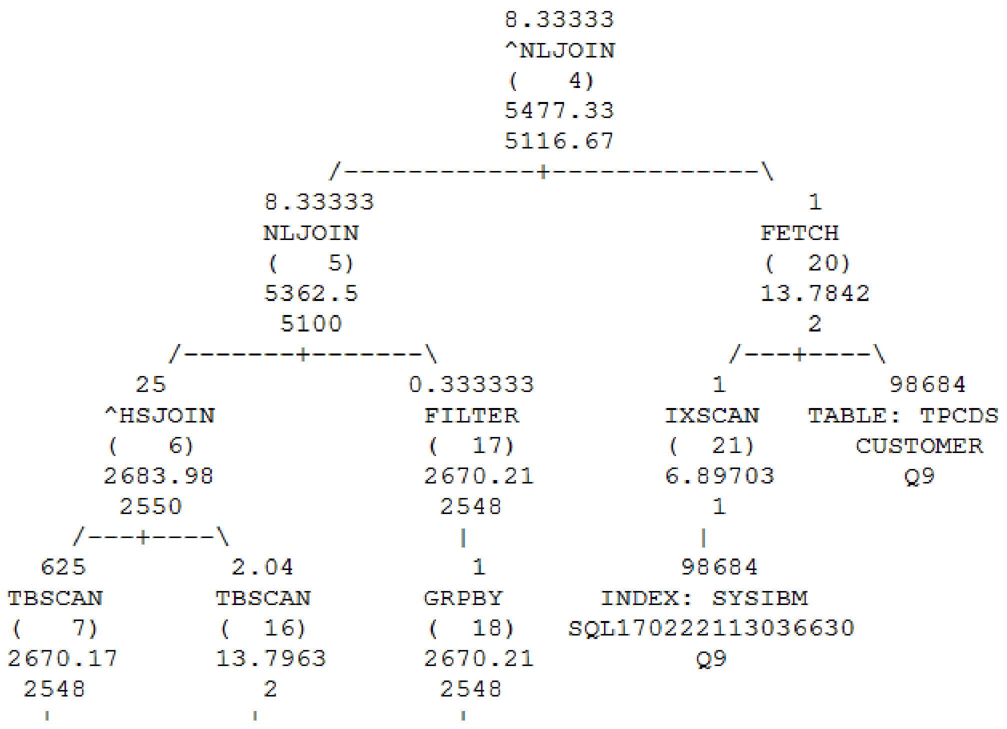


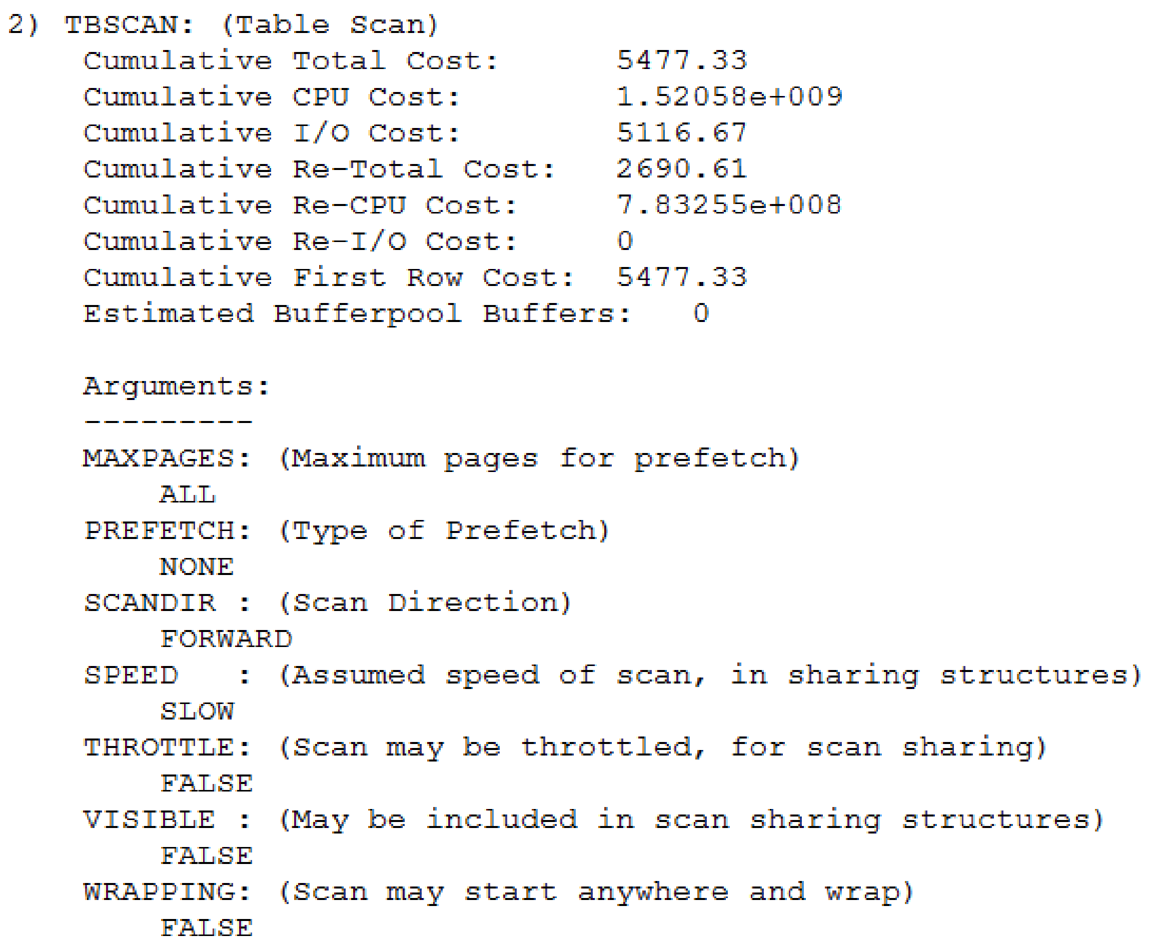
Figure 3 – Tree Diagram Portion of a QEP


```
                             8.33333
                             ^NLJOIN
                             (   4)
                             5477.33
                             5116.67
                    /------------+-------------\
               8.33333                            1
               NLJOIN                           FETCH
               (   5)                           (  20)
               5362.5                           13.7842
                5100                               2
          /-------+-------\                 /---+----\
        25               0.333333         1          98684
     ^HSJOIN              FILTER        IXSCAN    TABLE: TPCDS
     (   6)               (  17)        (  21)       CUSTOMER
     2683.98              2670.21       6.89703         Q9
      2550                 2548           1
   /---+----\               |             |
  625        2.04           1           98684
TBSCAN      TBSCAN        GRPBY      INDEX: SYSIBM
(   7)      (  16)        (  18)   SQL170222113036630
2670.17     13.7963       2670.21          Q9
 2548          2           2548
```

The textual portion of the QEP file contains the plan details, with blocks representing each LOLEPOP. Each block is identified by the ID of the LOLEPOP as well as its type. Further information detailed in the block is related to costs, arguments, inputs and output of the operator. Moreover, some properties are common between different operators (total cost, I/O cost, estimate number of rows), while others are related to a specific operator. For example, a "JOIN" operator (NLJOIN, MGJOIN, HSJOIN) has a property outer join and a "TEMP" operator has a property groups, but not vice-versa. Figure 4 depicts a piece of the block of LOLEPOP represented on Figure 2.

In the current process of the company, the analysis of QEP files is done manually with the use of search tools, such as *grep*. This task is very time consuming and it is not well scalable, as the search cannot be saved and applied to new QEPs afterwards. Depending on the complexity of the problem, manual task could take in order of hundreds hours. Furthermore, it requires a lot of resources as sometimes IBM experts need to go to the client's company in order to find what could be causing the problem. Some of these costs could be reduced by providing a way to automatically finding patterns that cause these types of errors.

Figure 4 – Plan Detail of the LOLEPOP on Figure 2



```
2) TBSCAN: (Table Scan)
     Cumulative Total Cost:        5477.33
     Cumulative CPU Cost:          1.52058e+009
     Cumulative I/O Cost:          5116.67
     Cumulative Re-Total Cost:     2690.61
     Cumulative Re-CPU Cost:       7.83255e+008
     Cumulative Re-I/O Cost:       0
     Cumulative First Row Cost:    5477.33
     Estimated Bufferpool Buffers:    0

     Arguments:
     ---------
     MAXPAGES: (Maximum pages for prefetch)
         ALL
     PREFETCH: (Type of Prefetch)
         NONE
     SCANDIR : (Scan Direction)
         FORWARD
     SPEED   : (Assumed speed of scan, in sharing structures)
         SLOW
     THROTTLE: (Scan may be throttled, for scan sharing)
         FALSE
     VISIBLE : (May be included in scan sharing structures)
         FALSE
     WRAPPING: (Scan may start anywhere and wrap)
         FALSE
```

OptImatch was created with the intuit of providing a way to search for problem patterns automatically. The tool allows experts to pre-define problem patterns and save them in a knowledge base, so customers can search for recommendations before contacting IBM and if even possibly, solve the problem themselves.

2.2 Other Adopted Technologies

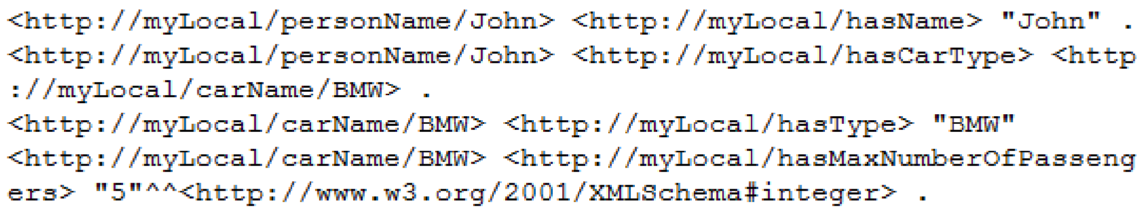
2.2.1 RDF

RDF is an acronym for Resource Description Framework (RDF). RDF is a W3C specification design originally as a metadata data model and is a standard model for data interchange on the web. The language description model allows the creation of statements about a resource either by defining its relationships with other resources or by defining its attribute. The model is composed by triples, where each triple contains subject, predicate and object. Subjects are the resources, predicates are the relationships and objects are either resources or attributes.

Subjects, predicates and objects are normally composed by a URI and a node identification. A resource can be identified by a URI or by blank nodes. According to [Cyganiak, Wood e Lanthaler 2014], blank nodes are disjoint from IRIs and literals. Otherwise, the set of possible blank nodes is arbitrary. RDF makes no reference to any internal structure of blank nodes. Concrete values are literals with data-type specification. Even a specific subject, predicate and object can be a URI or a blank node, but only the object can be literal.

RDF can be stored in different formats. N-Triples, Turtle, RDF/XML and Notation-3 are the most popular formats. Figure 5 represents an example of a N-Triples format. This example shows a person called "John" that has a car "BMW", where this car has maximum number of passengers equals to "5". The name of the person and the maximum number of passengers in the car are represented as literal, while the object of the car type is represented as a resource.

Figure 5 – N-Triples format



```
<http://myLocal/personName/John> <http://myLocal/hasName> "John" .
<http://myLocal/personName/John> <http://myLocal/hasCarType> <http
://myLocal/carName/BMW> .
<http://myLocal/carName/BMW> <http://myLocal/hasType> "BMW"
<http://myLocal/carName/BMW> <http://myLocal/hasMaxNumberOfPasseng
ers> "5"^^<http://www.w3.org/2001/XMLSchema#integer> .
```

2.2.2 SPARQL

SPARQL is a recursive acronym for SPARQL Protocol and RDF Query Language. SPARQL is an RDF query language able to retrieve data stored in RDF format. It is supported by W3C and it is a semantic query language for databases.

> SPARQL can be used to express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware. SPARQL contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions. [Harris e Seaborne 2013].

Normally SPARQL query contains a set of triple patterns that are like RDF triples. However, in the query, each of the subject, predicate and object can be a variable. A query consists of two parts: the *SELECT* clause that defines the variables to be retrieved as result and the *WHERE* clause that defines the properties to be matched. For example, a user could write a query to retrieve a list of people who have a BMW car with the maximum number of passengers equaling to 5. Figure 6 depicts this example. All variables that appear in the query are defined by the "?" symbol prefix and a variable, for example, ?*variableName* . The same variable can be referenced multiple times inside the *WHERE* clause. In this example the variable ?*name* returns the object of the *hasName* predicate. In other words, the *SELECT* statement returns the name of all people filtered in the *WHERE* clause. The ?*name* results returned are those where the resource (variable ?*x*) has a car of type ?*car*. Lastly, the ?*car* variable is filtered to be of type BMW and to have maximum number of passengers as 5.

Figure 6 – SPARQL Query Example

```
PREFIX predicate:   <http://myLocal/>
SELECT ?name
WHERE
  {
    ?x   predicate:hasName                     ?name .
    ?x   predicate:hasCarType                  ?car  .
    ?car predicate:hasType                     "BMW" .
    ?car predicate:hasMaxNumberOfPassengers 5
  }
```

2.2.3 Jena

Jena is a free and open source Java framework for building Semantic Web and Linked Data applications. It provides an programmatically environment for RDF, SPARQL and OWL.

The RDF API provided by Jena has the capability to parse RDF data from different formats and write RDF graph in various supported formats. It provides a simple way to create properties and add them to a resource. Another capability provided by Jena is the ARQ, which is a SPARQL 1.1 compliant engine. It has benefits such as manipulating easily SPARQL queries and inserting properties in the query.

2.2.4 Dojo toolkit

Dojo is a JavaScript toolkit. It provides language utilities, UI components and others functionalities to help to build a Web app. Dojo is trusted by big companies such as IBM, Cisco and Philips.

This toolkit is modular and provides asynchronous load of the resources needed by the application. By using *dojo.require* the user requires just the resources needed to accomplish that task. Also Dojo provides a Dijit UI framework. This framework contains a set of layouts, forms, widgets with built in validation, many themes, etc. Furthermore, Dojo has a DojoX library that includes extra layouts, form widgets, dataGrid and more. In other words, Dojo is a Java Toolkit that provides many features, including basic JavaScript language and helper utilities, on-demand asynchronous script loading, complete UI framework and build tools.

2.2.5 D3.js

D3.js is a JavaScript library for manipulating documents based on data. It combines powerful visualization and easy setup. D3 is bases on web standard, working on old and modern browsers. Also, D3 is open-source and it has a big community support with many examples on the internet.

According to the official D3 website [D3], D3 allows you to bind arbitrary data to a Document Object Model (DOM), and then apply data-driven transformations to the document. For example, you can use D3 to generate an HTML table from an array of numbers.

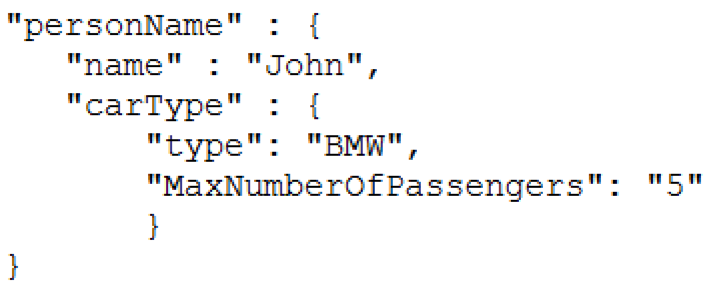
D3 provides functionalities that makes the creation of data visualization easier. For example, you can add enter, transition and exit selections to create new nodes, create the transitions and remove nodes that are no longer needed.

2.2.6 JSON

JSON is an acronym for JavaScript Object Notation. JSON standard is a lightweight data-interchange format, easy for humans to read and analyze if necessary and easy for machines to parse JSON objects. With the rise of AJAX-powered sites, it's becoming more and more important for sites to be able to load data quickly and asynchronously, or in the background without delaying page rendering.

JSON is built normally in a collection of name/value pairs. A value can also be an array of name/value pairs. Figure 7 depicts the *John* example described in Section 2.2.1. This example contains an object *personName* with a property named *name* and value *John*. The next property is named *carType* and the value is a collection of name/value pairs. The properties inside it are *type* and *MaxNumberOfPassengers*, with values *BMW* and 5, respectively.

Figure 7 – JSONExample

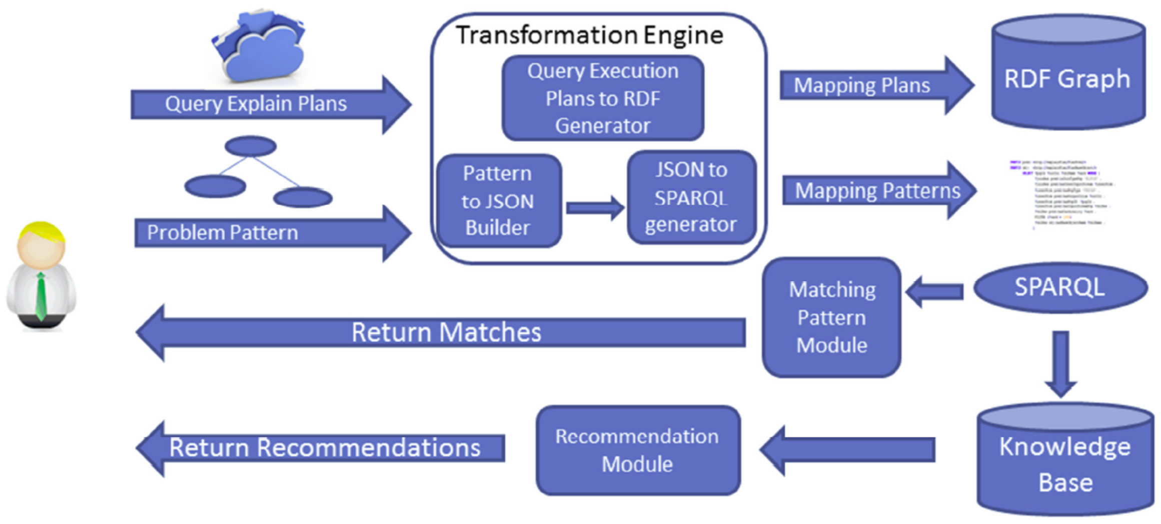


```
"personName" : {
   "name" : "John",
   "carType" : {
       "type": "BMW",
       "MaxNumberOfPassengers": "5"
       }
}
```

3 OptImatch Architecture

In this chapter an overview of the OptImatch is given, explaining the main functionalities of the system as well as how they are connected to each other. The idea of the tool is to create an easy way for normal users to analyze the QEP files uploaded to the system, by searching for an specific problem pattern that can be described at runtime by the user or recommendations previously described by experts and stored in the knowledge base. The main functionalities of the system are divided in the following parts: the transformation engine, search for problem patterns and retrieval solutions in the knowledge base. Figure 8 depicts the proposed system architecture.

Figure 8 – System Architecture



Source: Damasio, G. (2016)

The QEPs uploaded by the user are transformed into RDF graphs by the transformation engine and stored in the server. The transformation engine also transforms the problem pattern described by the user in an executable SPARQL. With this information, Optimatch runs the SPARQL query against the RDF graphs and stores the results. The tool also gets all recommendations from the knowledge base and run each query against the RDF graphs. If any recommendation is found, the recommendation module, using the handlers tagging interface, changes the handlers by the result of the query. The results are put together and sent back to the user.

QEP can be seen as a directed graph that represents the flow of operations data within a plan. Some information inside the QEP is stored in a tree structure, while others are stored in textual blocks. The link between the tree structure and the textual block is given by the id of the node. Each node contains properties related to itself and related to connections that it may have with other nodes. The transformation engine is responsible for the transformation of the QEP workload uploaded by the user to RDF labeled graphs. This transformation is done by parsing all the information contained in the QEP and then storing it in the RDF format. Apache Jena RDF API was used in the framework to create the RDF. RDF is composed of triples, where each triple contains the subject, predicate and object. The system can map each property in the QEP into a triple, where the subject is the node, the predicate is the property to be stored and the object is the value of this property. After completion, the RDF file is temporarily stored into the server until the user decides to delete or close the section on the web. RDF format was chosen because it allows the retrieval of information with the SPARQL query language and also because of the convenience, since DB2 supports RDF file format and SPARQL querying across all editions from DB2 10.1, when the RDF specific layer, DB2 RDF Store, was added to DB2. The DB2 RDF Store is optimized for graph pattern matching. Even though the information could be represented as relations and queried using SQL, RDF can be easily extended with new properties and relationships, something that is very difficult to do with a relational structure, as it would constantly need modification of the schema and tables storing the relationships.

Furthermore, the transformation engine is also responsible for transforming the problem pattern into a JSON object, sending it to the server and then on server side, transforming the JSON in a SPARQL query. The creation of the problem pattern is made in the OptImatch's graphical user interface. Dojo toolkit and d3.js JavaScript were utilized for the development of the GUI. Dojo toolkit was chosen due to its modularity and because of the asynchronous load of the resources needed by the application, keeping the application fast and maintainable. D3.js was used for the graph tree representation of the properties because of its easy setup and powerful visualization. After the creation of the problem pattern, the transformation engine transforms all information into a JSON object. For the transformation of the JSON to SPARQL, OptImatch maps each property to a valid SPARQL query statement, as well as handlers from the recommendation, as explained on Chapter 6. SPARQL was chosen given the compatibility with RDF, providing a short way to match patterns in the RDF graph. SPARQL also includes property paths, allowing the use of recursive queries, looking for connections that are not necessary direct. Moreover, the recursion in SPARQL is a lot simpler than generating the recursion in SQL.

After uploading the QEP workload, the user can then search for problem patterns and retrieve solutions in the knowledge base. To search for problem patterns the system first uses the transformation engine to auto-generate the SPARQL query from the problem pattern and then it retrieves any matched portion of the QEPs, rewriting the result in an human-readable language.

Lastly, OptImatch also allows the user to search for predefined patterns stored into the system, called recommendations. Recommendations are problem patterns defined by experts, representing a problem that can be found in a QEP. This problem pattern is linked to a recommendation that describes what the problem found in the QEP workload is. All recommendations are stored into the knowledge base, where DB2 is used to store all tables. This tool allows the user to surround the static part of the recommendation with a dynamic part. The proposed system provides a syntax where experts can make references to specific nodes, as well as sub-functions of it. At running time, OptImatch transforms the dynamic part of the recommendation into the data related to that specific file, allowing experts to make recommendations more specific for each user. The result found in the knowledge base is returned together with the problem pattern defined by the user.

3.1 Database

Database is a collection of information organized in a way that it can be easily retrieved. OptImatch makes use of the IBM DB2 database to store all necessary information, such as logs, recommendations, and problems reported by the users. OptImatch's database consists of 4 tables, where the representation and the connections are shown in Figure 9.

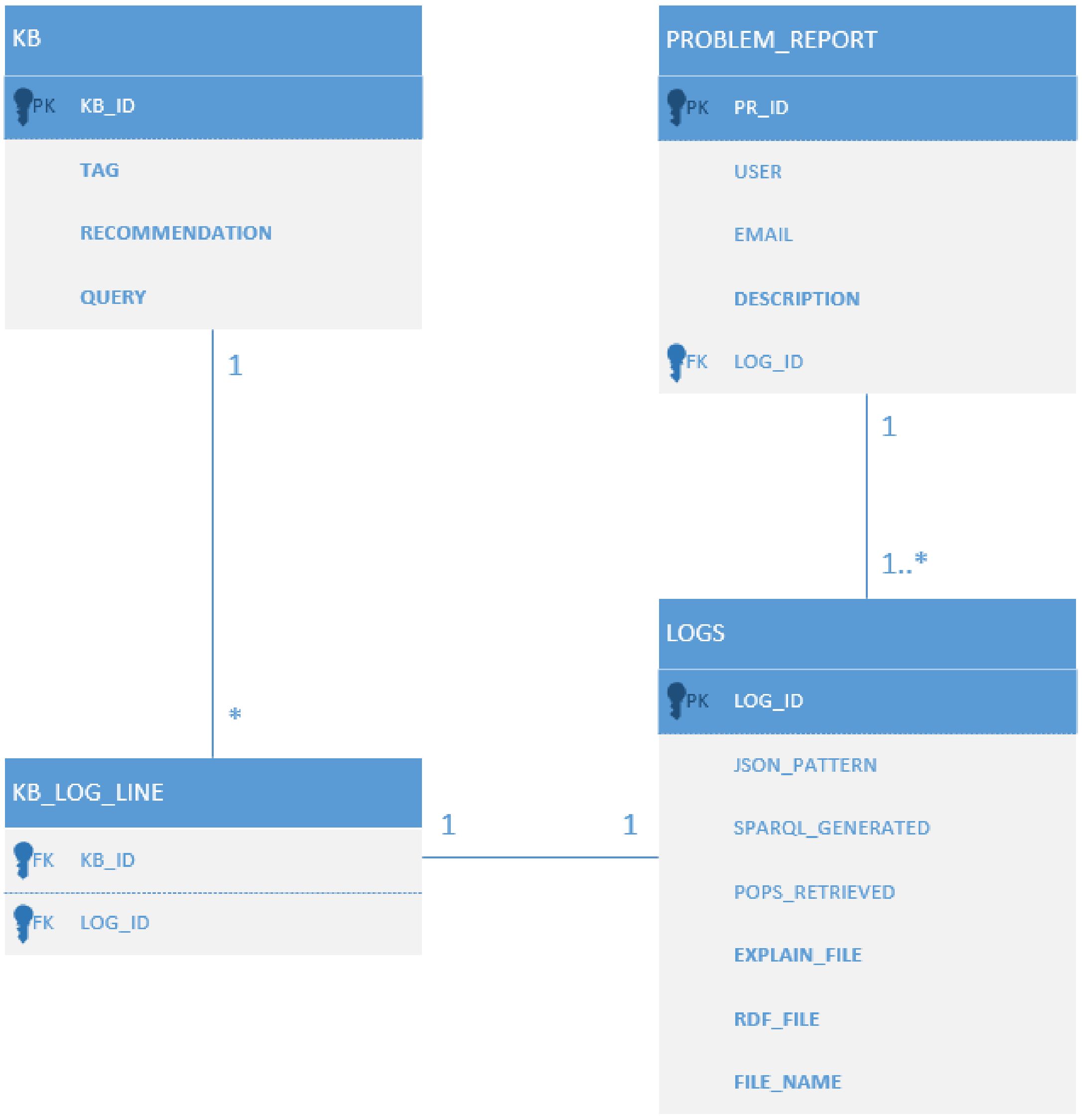
The "PROBLEM_REPORT" table is related to storing all information about any problem reported by a user. This table is used to search for bugs in the system. OptImatch stores the following information in this table, including the name and email of the user, as well as a description of the error. Lastly, if the problem is related to a file itself, OptImatch can also store the explain file.

KB table is the most important table in the system. It is here where all recommendations are stored to be further consumed by the framework in order to find any solutions for the user. This table contains the information about the tag, the recommendation with the handlers attributed to it and the already modified query to retrieve the information from the handlers if the query rewrite is necessary.

LOGS table is where any important event is stored. OptImatch keeps track of logs to make analysis of the system, seeing how effective is the system doing the tasks. OptImatch create logs when it fails or when it makes any match, for example. The information that the framework keeps track is the JSON pattern sent to the server, the SPARQL generated from it, the result found, the explain file and finally the file name.

The last table in the system is the "KB_LOG_LINE" table. This table is responsible for keeping track of which recommendation matches which file. The information stored here is the id of the "LOG" table and the id of the "KB" table, where both ids are foreign keys. A record is created every time that a match is found by OptImatch.

Figure 9 – Database Architecture

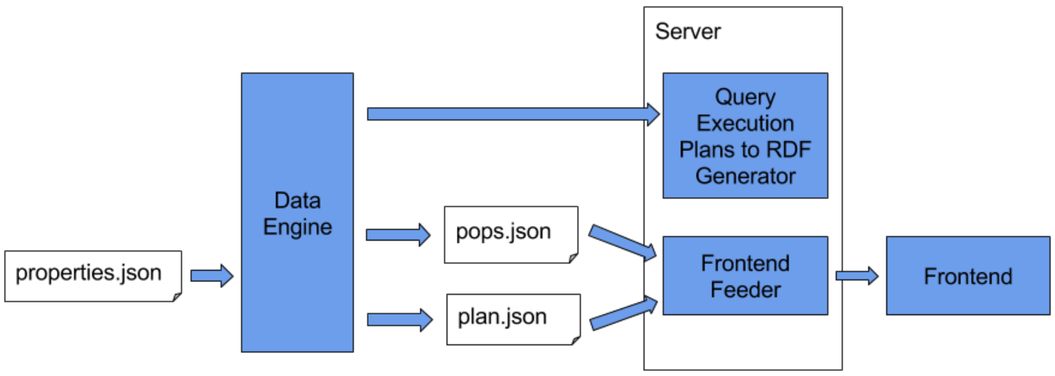


3.2 Data Integration

In the first version of this tool, giving the limited time and resources, the data was not well integrated, as some data was stored in the frontend as well as in the backend. For example, the data shown in the user side was stored statically in a javaScript file, while the information for the creation of the RDF from the QEP was stored in the code inside the server. This approach is not well scalable neither easy to change because any modification in the frontend would also require a modification in the backend, so the data could remain connected. In order to avoid this, all the data was abstracted to a folder inside the server. This folder contains all data needed for integration of the GUI information, the parser and RDF creation, ensuring the matching of properties. Figure 10 depicts how the data are integrated.

The "properties.json" is the main file, where all information is stored. It contains information about every property that OptImatch has, such as the RDF name, regex to match in the QEPs, label to show for the user, and which LOLEPOPs this property belongs to. This file is read by the server when the server is started. The engine looks through all properties in the "properties.json" and then creates two new files: "pops.json" and "plan.json".

Figure 10 – Data Integration



The "pops.json" file stores the properties related to the LOLEPOPs and it is used by the frontend. The engine only gets the information needed from the user side from the "properties.json" file. Also, the data in this file is stored in a structure that contains each LOLEPOP as an object, where the object contains all properties related to it. "plan.json" is also used in the frontend, but it is instead related to the global information about the QEPs.

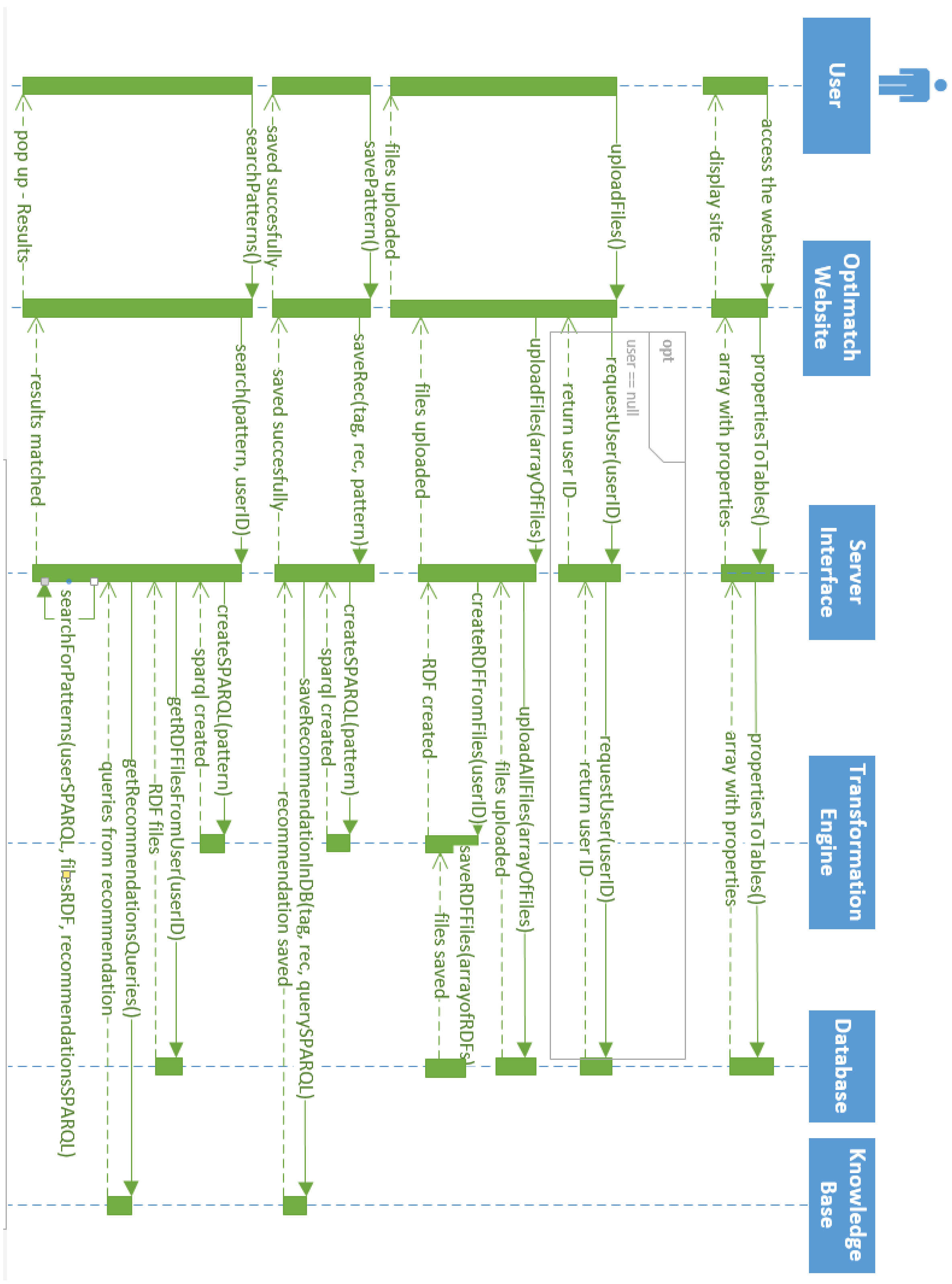
This integration makes the tool cleaner, as for any modification in the properties the code does not need to be modified. Even more, it elevates the reliability of the system because OptImatch does not contains anymore any duplicate information. As example, before the RDF name was stored separately in the user and server side, so any grammatical error could cause the system failure or misinterpretation. With this modification, when the RDF name is modified in the "properties.json" file, both user and server side are automatically updated to match the new property. Lastly, this capability allows OptImatch the ability to connect to the tool that creates the QEPs. In the matter of this project, related to the DB2, the properties could be updated by reading the explain tables from DB2 and storing all the properties in the "properties.json" files. This function would be beneficial to the tool as any modification in the DB2 properties would automatically update OptImatch properties.

For easy understanding of the system, a sequence diagram is provided on Figure 11. This image shows the main functionalities of the system and the order of the communication between the web application and the server. In the first stage, when the user access the website, the OpImatch website sends a request to the server asking for the LOLEPOPs and the properties related to them as well as the global plan properties. Once the properties are received, the website creates the tables with the information and displays the site to the user. The next step for the user is to upload the files to the server. When the user uploads the QEP workload, if a user id is not yet set to the current user, the website will request a new id from the server. On the server side, a request for a new user id is made to the database which is then returned to the website. After the id is set, the website uploads the QEP files. Once the files are received, the server then saves them in a folder for the user. Once the files are saved in the server, OptImatch uses its transformation engine to go through each QEP and create the RDF graph from it. After the process is completed, a message stating that the files were saved successfully is shown to the user.

With the files uploaded, the user can then search for the pattern and recommendations. Once the button to perform the search is clicked, OptImatch's website then sends a request to analyze the file for the current user. The website sends the id of the user as well as an array with the problem pattern described by the user in the GUI. With the request received, the server will first use the transformation engine to create the SPARQL query from the problem pattern. Next step is get the RDF files from the current user. After RDF files are received, the server get the queries of the recommendations in the knowledge base. With the possession of these three pieces of information (SPARQL query, RDF files and queries from recommendations), the server performs the search for patterns. Once finished with the task, an array with the results is returned to the website, where it is translated to a human-readable text form and displayed it to the user.

The last functionality described in the sequence diagram is the ability to save a recommendation. In order to save a recommendation, the website sends an array with the problem pattern described by the user in the GUI as well as the tag for the recommendation and the recommendation itself. When received, the server creates the SPARQL query from the pattern and then save the tag, recommendation and SPARQL query in the knowledge base. Once saved, the server returns a message stating that the recommendation was saved successfully to the website, where the website displays the message to the user.

Figure 11 – OptImatch Sequence Diagram

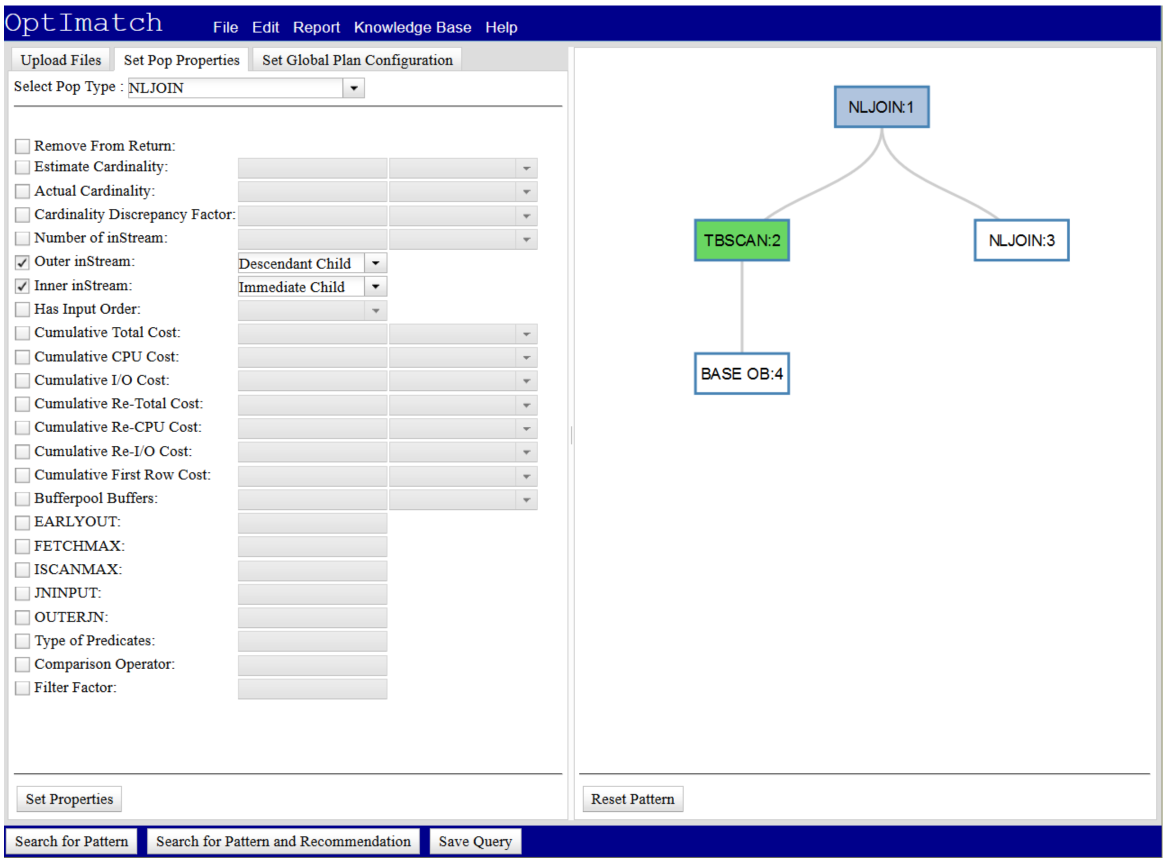


4 Graphical User Interface

The interaction between the user and an electronic device or program typically occurs by means of a graphical user interface (GUI). For this tool a web-based GUI was created, where the user can have access to all features that OptImatch offers. To access the tool, the user needs to use a browser and point to the IBM server using the correct URL.

A web-based type of GUI was chosen because it does not require any software installation and it is platform independent. When connecting to the tool, the main page is loaded, as it can be seen on Figure 12. There the user can upload all QEPs, create a problem pattern and then search for recommendations.

Figure 12 – Main Screen Graphical User Interface



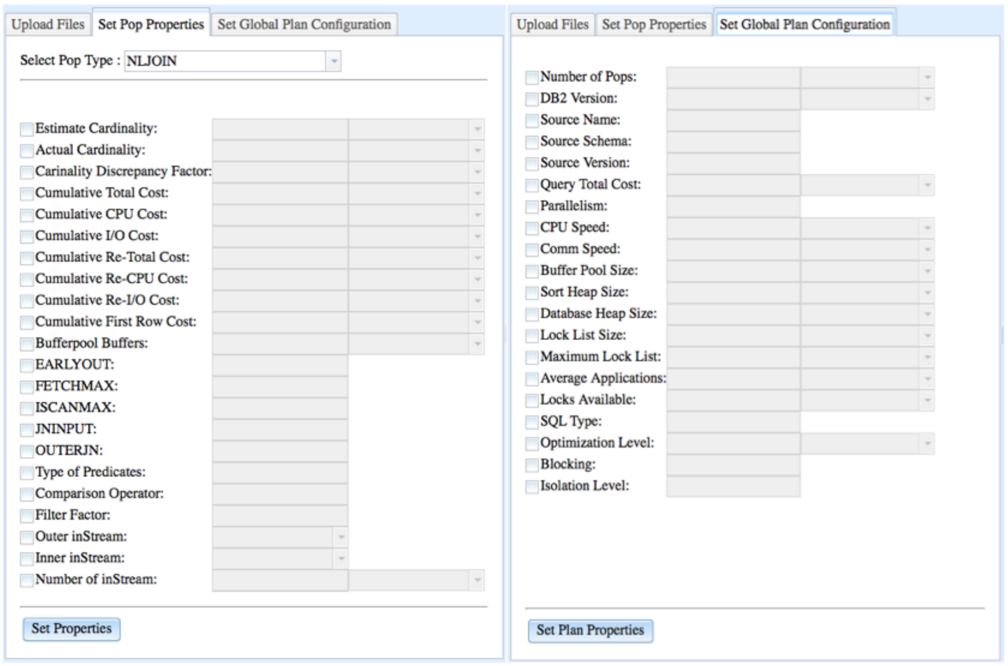
Select Pop Type : NLJOIN

- [ ] Remove From Return:
- [ ] Estimate Cardinality:
- [ ] Actual Cardinality:
- [ ] Cardinality Discrepancy Factor:
- [ ] Number of inStream:
- [✓] Outer inStream: Descendant Child
- [✓] Inner inStream: Immediate Child
- [ ] Has Input Order:
- [ ] Cumulative Total Cost:
- [ ] Cumulative CPU Cost:
- [ ] Cumulative I/O Cost:
- [ ] Cumulative Re-Total Cost:
- [ ] Cumulative Re-CPU Cost:
- [ ] Cumulative Re-I/O Cost:
- [ ] Cumulative First Row Cost:
- [ ] Bufferpool Buffers:
- [ ] EARLYOUT:
- [ ] FETCHMAX:
- [ ] ISCANMAX:
- [ ] JNINPUT:
- [ ] OUTERJN:
- [ ] Type of Predicates:
- [ ] Comparison Operator:
- [ ] Filter Factor:

4.1 Creating a Problem Pattern

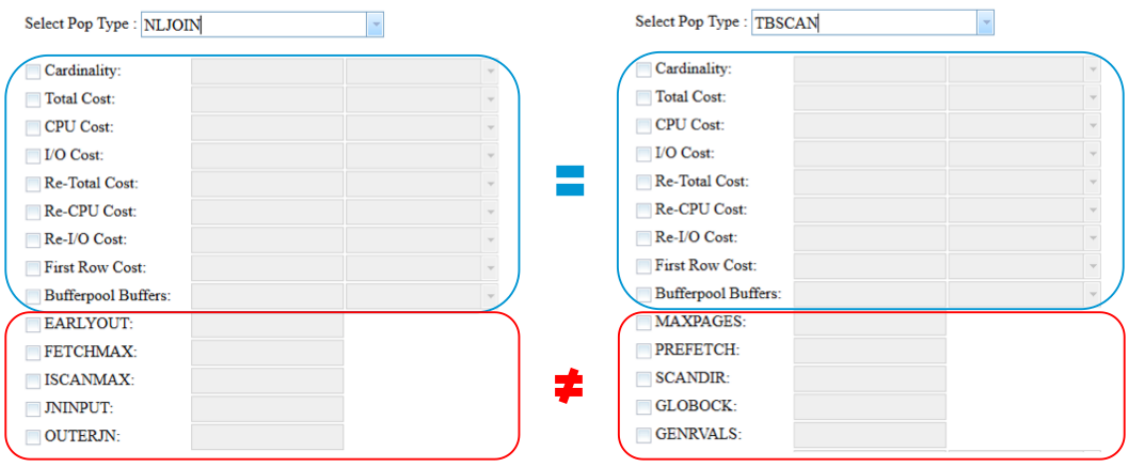
For the creation of the problem pattern, the user makes use of the "graph tree" pane to visualize and select a node. Within the problem pattern, properties can be set for the LOLEPOP as well as the Global Plan. They are separated in two tabs: "Set Pop Properties" and "Set Plan Configuration", shown in Figure 13.

Figure 13 – LOLEPOP Properties (Left) and Global Properties (Right)


Select Pop Type : NLJOIN

Inside the "Set Plan Configuration" tab the user can choose the LOLEPOP type and then the system automatically creates a table containing only properties related to this LOLEPOP, making the system more reliable as the user cannot add properties that does not belong to the LOLEPOP. Figure 14 depicts an example of different properties for different LOLEPOPs. For further configuration, the user can add properties that are not related to LOLEPOP, but are instead related to the global plan, such as DBMS instance and environment settings. Those configurations can be found inside the "Set Plan Configuration" tab.

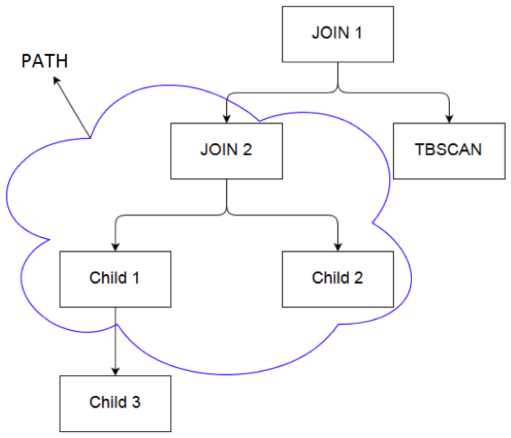
Figure 14 – Properties Differences


Select Pop Type : NLJOIN

Select Pop Type : TBSCAN

LOLEPOPs are connected to their parents as input streams. While specifying the type of input when creating a problem pattern, the user can choose between two types of relationships: immediate or descendant. Immediate child is a successor that is immediately below the current LOLELOP. Descendant is an operator that is a child but not necessarily immediately below the current LOLEPOP. This means that it could contain a path between the current LOLEPOP and the descendant child. Figure 15 shows an example of both relationships. Here, for example, "JOIN 2" is an immediate child of "JOIN 1", while "CHILD 3" is a descendant child of "JOIN 1" and the LOLEPOPs between "JOIN 1" and "CHILD 3" is the path of this connection.

Figure 15 – Immediate and Descendant Child



An example of problem pattern **(pattern A)** can be seen in Figure 16. The following properties describe the pattern: (i) LOLEPOP "pop1" is of type "NLJOIN"; (ii) "pop1" has an outer input stream of type "ANY" and cardinality greater than 1; (iii) "pop1" has an inner input stream "pop3" of type TBSCAN; (iv) "pop3" has a generic input stream of type "BASE OB" and cardinality greater than 100. Lastly, Figure 17 depicts a matched portion of a QEP by the problem pattern provided above.

Figure 16 – Problem Pattern 1

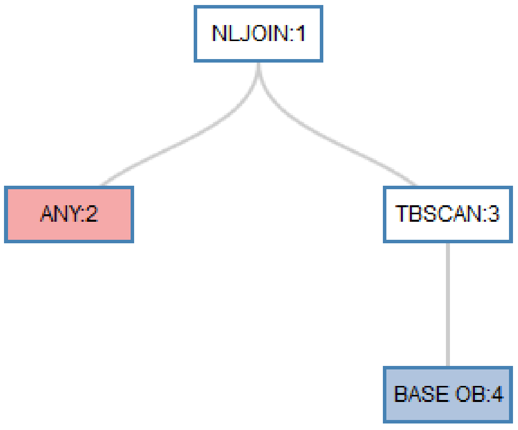
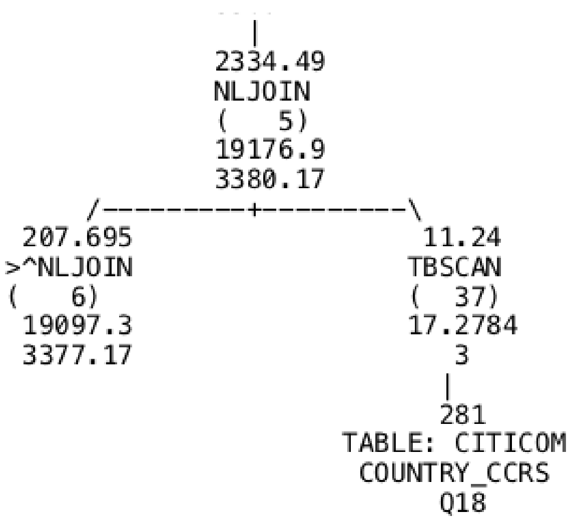


Figure 17 – Problem Pattern 1 in a QEP



```
                  |
               2334.49
               NLJOIN
               (   5)
               19176.9
               3380.17
         /---------+---------\
   207.695                   11.24
  >^NLJOIN                  TBSCAN
  (   6)                    (  37)
   19097.3                  17.2784
   3377.17                     3
                               |
                              281
                         TABLE: CITICOM
                          COUNTRY_CCRS
                              Q18
```

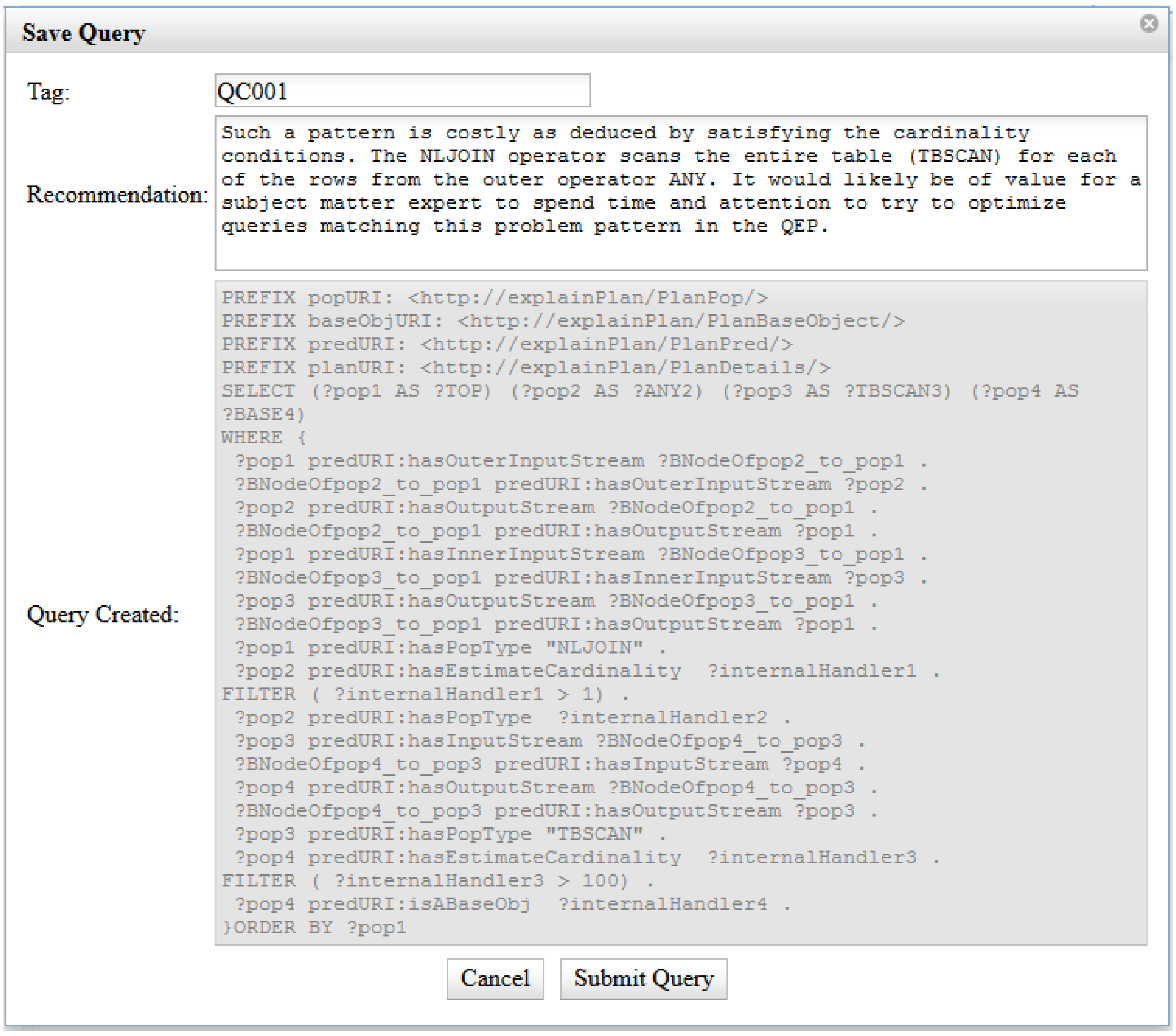
4.2 Recommendation

Recommendation is a pattern with a solution described by some expert or by the user of the system. All recommendations are stored into a "KNOWLEDGE BASE" that is used to search for problems in a QEP workload. The system automatically matches the problem patterns in the knowledge base against the QEP workload and any match is retrieved for the user in a result panel, ranking them using statistical correlation analysis. The result panel differs the problem pattern created by the user from the recommendation by dividing them inside the panel. Given the complexity of the recommendation syntax, this chapter will explain just how to save, search and delete a recommendation. Syntax will be explained in Chapter 6 as some information of the following chapters will be needed for further comprehension.

To save a recommendation the user first needs to define a problem pattern. The pattern described on Figure 16 (pattern A) can be used as example. After the pattern is created, the user clicks on the "Save Query" button and a panel will pop-up as can be seen on Figure 18. This panel contains three text boxes: TAG, recommendation and query, where TAG and recommendation are writable, while query is not. The query text box shows the query automatically created by the system, helping the build of a recommendation.

For this problem pattern an example of TAG could be "QC001", meaning that this is a problem in a QEP (Q) that is Costly (C). The number 001 is given by the fact that it is the first recommendation. As a recommendation, the user can say the following: "Such a pattern is costly as deduced by satisfying the cardinality conditions. The NLJOIN operator scans the entire table (TBSCAN) for each of the rows from the outer operator ANY. It would likely be of value for a subject matter expert to spend time and attention to try to optimize queries matching this problem pattern in the QEP".

Figure 18 – Save Query Pop-up



Tag: QC001

Recommendation: Such a pattern is costly as deduced by satisfying the cardinality conditions. The NLJOIN operator scans the entire table (TBSCAN) for each of the rows from the outer operator ANY. It would likely be of value for a subject matter expert to spend time and attention to try to optimize queries matching this problem pattern in the QEP.

Query Created:

```
PREFIX popURI: <http://explainPlan/PlanPop/>
PREFIX baseObjURI: <http://explainPlan/PlanBaseObject/>
PREFIX predURI: <http://explainPlan/PlanPred/>
PREFIX planURI: <http://explainPlan/PlanDetails/>
SELECT (?pop1 AS ?TOP) (?pop2 AS ?ANY2) (?pop3 AS ?TBSCAN3) (?pop4 AS
?BASE4)
WHERE {
 ?pop1 predURI:hasOuterInputStream ?BNodeOfpop2_to_pop1 .
 ?BNodeOfpop2_to_pop1 predURI:hasOuterInputStream ?pop2 .
 ?pop2 predURI:hasOutputStream ?BNodeOfpop2_to_pop1 .
 ?BNodeOfpop2_to_pop1 predURI:hasOutputStream ?pop1 .
 ?pop1 predURI:hasInnerInputStream ?BNodeOfpop3_to_pop1 .
 ?BNodeOfpop3_to_pop1 predURI:hasInnerInputStream ?pop3 .
 ?pop3 predURI:hasOutputStream ?BNodeOfpop3_to_pop1 .
 ?BNodeOfpop3_to_pop1 predURI:hasOutputStream ?pop1 .
 ?pop1 predURI:hasPopType "NLJOIN" .
 ?pop2 predURI:hasEstimateCardinality  ?internalHandler1 .
FILTER ( ?internalHandler1 > 1) .
 ?pop2 predURI:hasPopType  ?internalHandler2 .
 ?pop3 predURI:hasInputStream ?BNodeOfpop4_to_pop3 .
 ?BNodeOfpop4_to_pop3 predURI:hasInputStream ?pop4 .
 ?pop4 predURI:hasOutputStream ?BNodeOfpop4_to_pop3 .
 ?BNodeOfpop4_to_pop3 predURI:hasOutputStream ?pop3 .
 ?pop3 predURI:hasPopType "TBSCAN" .
 ?pop4 predURI:hasEstimateCardinality  ?internalHandler3 .
FILTER ( ?internalHandler3 > 100) .
 ?pop4 predURI:isABaseObj  ?internalHandler4 .
}ORDER BY ?pop1
```

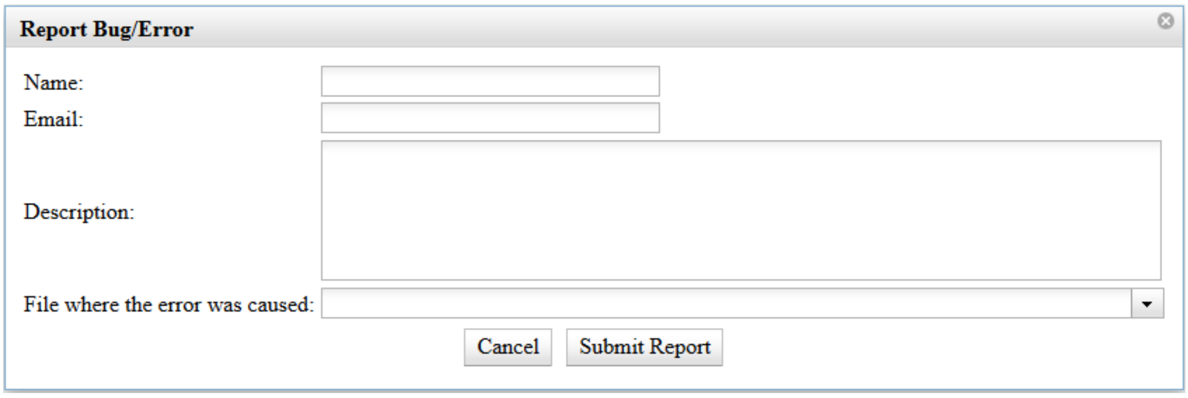
Furthermore, the system has extra capabilities related to recommendations. On the top of the main screen, the user can click on the "Knowledge Base" drop-down button to search or delete recommendations. For search capability, the user will click on "search recommendation" and the search panel will pop-up. The system will look for both TAG and Recommendation, returning all matches to the user.

For the deletion capability, the user can click on "delete recommendation" inside the drop-down menu of the button "Knowledge Base". For security reasons, the system asks for the TAG of the recommendation that the user wants to delete. Even more, the system allows just 1 deletion at a time to ensure that the user does not erroneously delete the wrong recommendation.

4.3 Extra Features

To make the user experience more enjoyable, extra features were added to OptImatch. Firstly, the functionality of saving and importing the workspace was added, so the user can stop their work and return at any later time, where all data will remain saved for further changes. Moreover, the capability of undo and redo graph changes were included. Since OptImatch still a prototype, the user can report any bug to the system through the "report bug" button inside OptImatch. The user can then describe the problem and link it to a QEP file, given that at least one was previously uploaded. Figure 19 depicts the report bug panel.

Figure 19 – Report Bug Pop-up


Report Bug/Error

Name:
Email:
Description:
File where the error was caused:

All bugs sent to the server will remain on the system database, together with some information of the current system status, so experts can analyze what could be the problem and if it is related to the system or a given file. Explanation of what is saved in the server is further explained in Section 3.1. Lastly, a help page was added to the system, where the user can find tutorials and examples. For now 6 tutorials were created: "Creating a Pattern", "Report Bug", "Saving a Recommendation in the Knowledge Base", "Searching Problem Pattern in the Uploaded Files", "Searching Recommendations" and "Uploading Files".

5 Transformation Engine

After uploading the QEP workload and creating a pattern, the user can then search for recommendations and search for a given pattern. At this stage, all information is sent to the server, where the tool will make all the analysis. For each user the server creates a unique folder where all files will be stored.

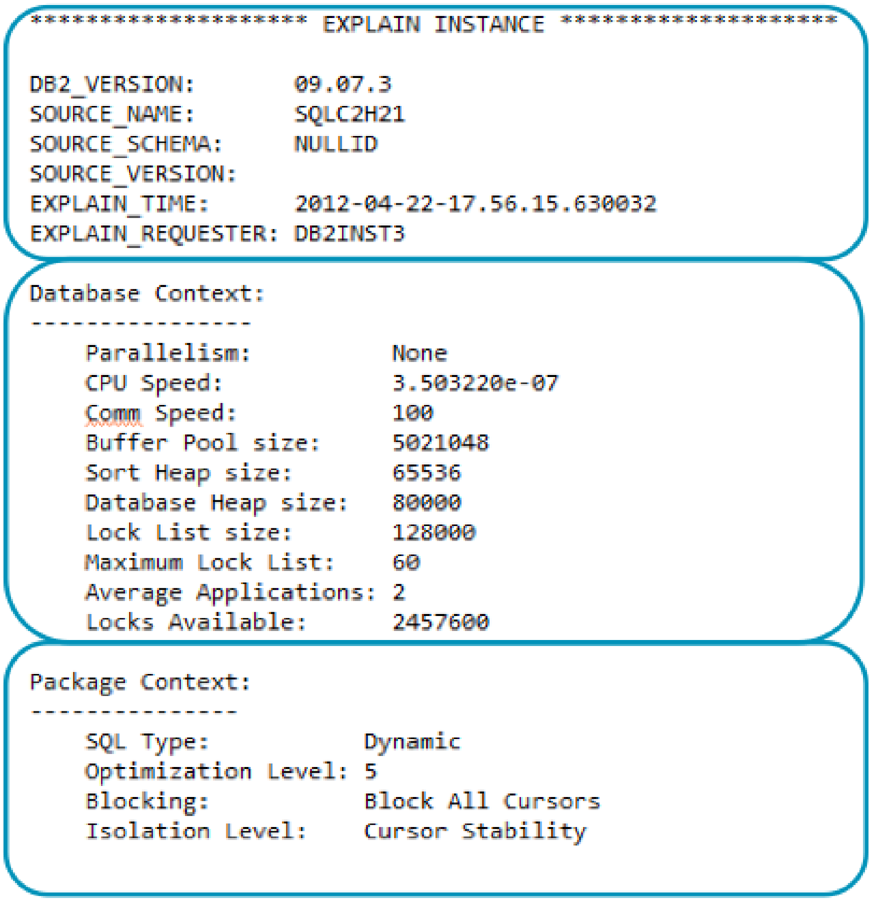
5.1 Parser

The parser is responsible for parsing each QEP, looking for predetermined properties and storing them to then later create an RDF from the given file. Before this work, the parser and the related properties were created inside the main code of the system, so any modification or addition of properties would cause a main code change, what is not something trivial. Even more, the parser was not replicable as it was attached to the code. Given these reasons, the parser was abstracted in a library.

For the creation of this library a concept of blocks was used, where the user set blocks, sub-blocks and properties related for each block. Figure 20 depicts an example of blocks that can be created in a QEP. In this example the user can set a block called "Explain Instance" with a property "DB2_Version", so when running the program the parser will find this property and will store inside the given block.

Moreover, this parser can make use of regex to gather data inside QEPs, turning this into a library that can embrace more types of data. As an example, inside "PLAN DETAILS" in the QEP, the first line of the each LOLEPOP contains both LOLEPOP id and type, such as: 1) RETURN: (Return Result), where the id is "1" and type is "RETURN". It could be difficult to gather both without the use of regex. In this example, the user can make use of regex to grab a block of text, dividing the line in two blocks (id and type) in order to gather both properties. Also, this library has functions to help the RDF creation, such as specifying if the current line is a block or a property. Also, the user can set prefix for each property, so the tool can use it to create the RDF. If no prefix was created, the tool will take the prefix from its parent block. Lastly, the parser stores all connections of properties with the blocks, so all complex diagnostic information can be stored without any loss of information.

Figure 20 – QEP in blocks



```
******************** EXPLAIN INSTANCE ********************

DB2_VERSION:       09.07.3
SOURCE_NAME:       SQLC2H21
SOURCE_SCHEMA:     NULLID
SOURCE_VERSION:
EXPLAIN_TIME:      2012-04-22-17.56.15.630032
EXPLAIN_REQUESTER: DB2INST3
```

```
Database Context:
----------------
    Parallelism:          None
    CPU Speed:            3.503220e-07
    Comm Speed:           100
    Buffer Pool size:     5021048
    Sort Heap size:       65536
    Database Heap size:   80000
    Lock List size:       128000
    Maximum Lock List:    60
    Average Applications: 2
    Locks Available:      2457600
```

```
Package Context:
---------------
    SQL Type:           Dynamic
    Optimization Level: 5
    Blocking:           Block All Cursors
    Isolation Level:    Cursor Stability
```

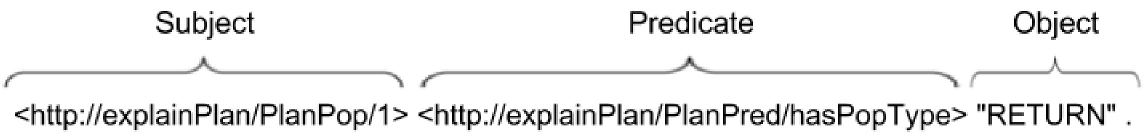
5.2 RDF Creation

After the parser definition, regarding the QEP workload the system can create the RDF with the given parser. With all blocks, properties and connections defined, the RDF creator will scan each line of the QEP looking for blocks and properties described in the parser, connecting all information.

RDF does not posses a pre-defined structure, but one can be enforced by specifying predicates, such as: hasOutputStreamPop together with hasInputStreamPop or hasInner-InputStreamPop and hasOuterInputStreamPop, using them to establish a relationship between LOLEPOPS. By these predicates the tree structure and characteristics in QEPs can be recreated. While analyzing the QEP for RDF creation, the framework models all data inside a QEP into entities with properties and relationships between them. RDF is stored as triples, each one containing subject (resource), predicate (property or relationship) and object (resource or value). Figure 21 depicts a real example of triple where the user has set the IRI (RDF URI reference) for the predicate as "http://explainPlan/PlanPred/".

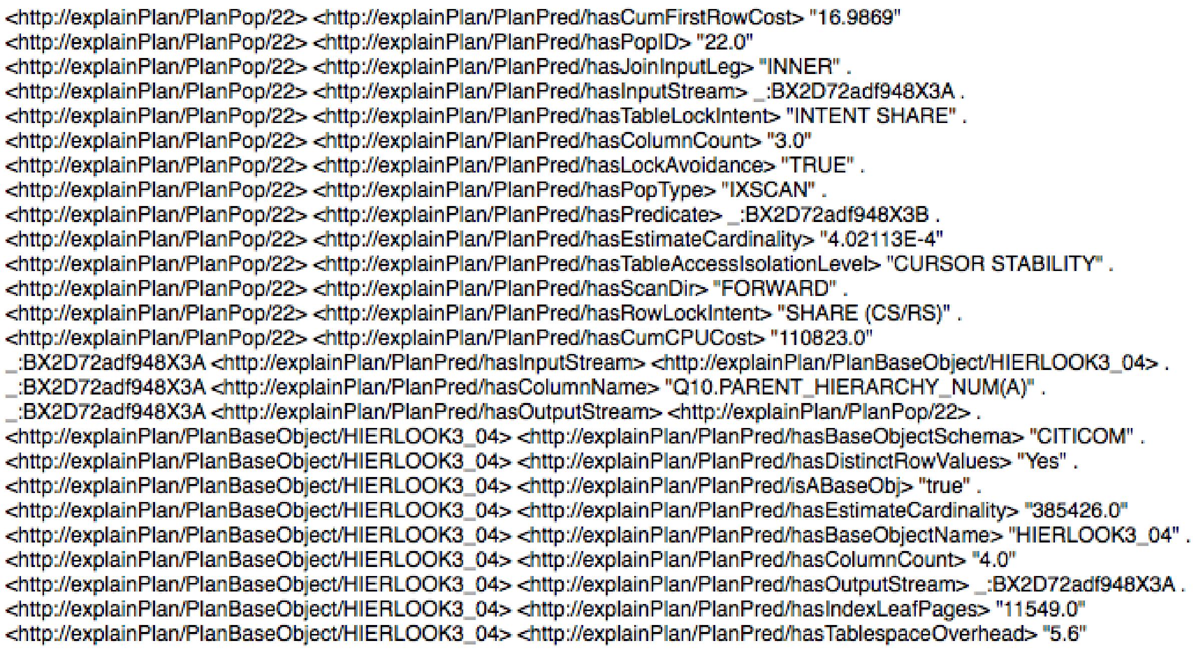
While analyzing the QEP workload, it was noticed that some properties are not linked just to a LOLEPOP, but to a relationship between a LOLEPOP and another one. For example, in the QEP, looking at the input stream of a LOLELOP, this connection

Figure 21 – RDF Triple


Subject
Predicate
Object
<http://explainPlan/PlanPop/1> <http://explainPlan/PlanPred/hasPopType> "RETURN" .

contains a property called "hasColumnName", showing all columns used in this relationship. However, a LOLEPOP can have more than one input stream. By this, it would be difficult to differ which "hasColumnName" property is related to which input stream. To solve this issue, the concept of a blank node was introduced to the tool. According to [Cyganiak, Wood e Lanthaler 2014], blank nodes are disjoint from IRIs and literals. Otherwise, the set of possible blank nodes is arbitrary. RDF makes no reference to any internal structure of blank nodes. As optImatch does not need to know the resource name of the blank node to search for data in an RDF file, the RDF capability to create arbitrary a name to the blank node was used. An example of a generated RDF can be seen on Figure 22. Lastly, the pseudocode of the RDF creation is described in Algorithm 1.

Figure 22 – Generated RDF from QEP



```
<http://explainPlan/PlanPop/22> <http://explainPlan/PlanPred/hasCumFirstRowCost> "16.9869"
<http://explainPlan/PlanPop/22> <http://explainPlan/PlanPred/hasPopID> "22.0"
<http://explainPlan/PlanPop/22> <http://explainPlan/PlanPred/hasJoinInputLeg> "INNER" .
<http://explainPlan/PlanPop/22> <http://explainPlan/PlanPred/hasInputStream> _:BX2D72adf948X3A .
<http://explainPlan/PlanPop/22> <http://explainPlan/PlanPred/hasTableLockIntent> "INTENT SHARE" .
<http://explainPlan/PlanPop/22> <http://explainPlan/PlanPred/hasColumnCount> "3.0"
<http://explainPlan/PlanPop/22> <http://explainPlan/PlanPred/hasLockAvoidance> "TRUE" .
<http://explainPlan/PlanPop/22> <http://explainPlan/PlanPred/hasPopType> "IXSCAN" .
<http://explainPlan/PlanPop/22> <http://explainPlan/PlanPred/hasPredicate> _:BX2D72adf948X3B .
<http://explainPlan/PlanPop/22> <http://explainPlan/PlanPred/hasEstimateCardinality> "4.02113E-4"
<http://explainPlan/PlanPop/22> <http://explainPlan/PlanPred/hasTableAccessIsolationLevel> "CURSOR STABILITY" .
<http://explainPlan/PlanPop/22> <http://explainPlan/PlanPred/hasScanDir> "FORWARD" .
<http://explainPlan/PlanPop/22> <http://explainPlan/PlanPred/hasRowLockIntent> "SHARE (CS/RS)" .
<http://explainPlan/PlanPop/22> <http://explainPlan/PlanPred/hasCumCPUCost> "110823.0"
_:BX2D72adf948X3A <http://explainPlan/PlanPred/hasInputStream> <http://explainPlan/PlanBaseObject/HIERLOOK3_04> .
_:BX2D72adf948X3A <http://explainPlan/PlanPred/hasColumnName> "Q10.PARENT_HIERARCHY_NUM(A)" .
_:BX2D72adf948X3A <http://explainPlan/PlanPred/hasOutputStream> <http://explainPlan/PlanPop/22> .
<http://explainPlan/PlanBaseObject/HIERLOOK3_04> <http://explainPlan/PlanPred/hasBaseObjectSchema> "CITICOM" .
<http://explainPlan/PlanBaseObject/HIERLOOK3_04> <http://explainPlan/PlanPred/hasDistinctRowValues> "Yes" .
<http://explainPlan/PlanBaseObject/HIERLOOK3_04> <http://explainPlan/PlanPred/isABaseObj> "true" .
<http://explainPlan/PlanBaseObject/HIERLOOK3_04> <http://explainPlan/PlanPred/hasEstimateCardinality> "385426.0"
<http://explainPlan/PlanBaseObject/HIERLOOK3_04> <http://explainPlan/PlanPred/hasBaseObjectName> "HIERLOOK3_04" .
<http://explainPlan/PlanBaseObject/HIERLOOK3_04> <http://explainPlan/PlanPred/hasColumnCount> "4.0"
<http://explainPlan/PlanBaseObject/HIERLOOK3_04> <http://explainPlan/PlanPred/hasOutputStream> _:BX2D72adf948X3A .
<http://explainPlan/PlanBaseObject/HIERLOOK3_04> <http://explainPlan/PlanPred/hasIndexLeafPages> "11549.0"
<http://explainPlan/PlanBaseObject/HIERLOOK3_04> <http://explainPlan/PlanPred/hasTablespaceOverhead> "5.6"
```

The pattern presented **(pattern B)** on the figure above represents the problem related to estimation of the execution cost by optimizer. This pattern is given by the following properties: (i) LOLEPOP of type index Scan (IXSCAN) or table scan (TBSCAN) (ii) has cardinality smaller than 0.001; (iii) has a generic input stream of type Base Object (BASE OB) and cardinality bigger than 100000. The recommendation in this case is to create column group statistics (CGS) on equality local predicate columns and CGS on equality join predicate columns of the Base Object.

Algorithm 1 RDF algorithm

```
Input: query execution plan files $QEPFs[\,]$
Output: QEP files represented as RDF graphs, $RDFGs[\,]$
 1: function CreateRDF($QEPFs[\,]$)
 2:     for all $qepf$ in QEPFs[ ] do
 3:         $i := 0$
 4:         $parserProperties[]$ := all properties previously created in the parser
 5:         $model$ := each property of the $qepf$ that is on the $parserProperties[]$, saving
    it in a structure pre-defined in the parser
 6:         $rdfg$ := convert $model$ into a RDF graph model with JENA RDF API
 7:         $RDFGs[\,i\,] := rdfg$
 8:         $i := i + 1$
 9:     end for
10:     return $RDFGs[\,]$
11: end function
```

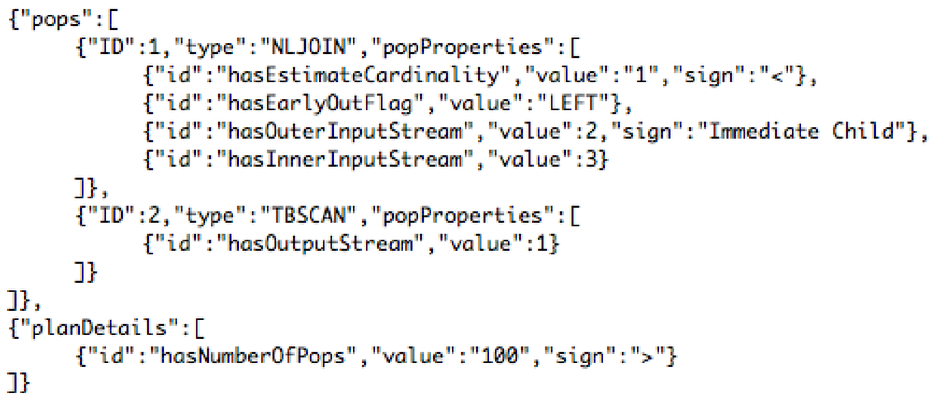
5.3 SPARQL Generator

After creation of the problem pattern, the user can search for this pattern through the QEP workload. At this time, the information is sent to the server, where the system analyzes it and automatically creates a SPARQL query. The problem pattern is transformed into a JSON object that contains an array of JSON objects, where each object describes a resource property or a relationship. This JSON object is constructed to contain a transformation of all properties defined by the user to the RDF resources and the predicates in the model used in the QEP. The transformation is made on the user's side, before it is sent to the server. The javaScript understands the human language utilized while defining the properties and automatically translating it to the predicates that will be used in the SPARQL query creation. The JSON has two main objects: *pops* and *planDetails*, where the *pops* object contains all information related to the LOLEPOPS and *planDetails* contains all information related to the global properties of a QEP.

Figure 23 represents a JSON object of a problem pattern that is: (i) LOLEPOP of type "NLJOIN" with cardinality smaller than 1; (ii) has EARLYOUT of type "LEFT"; (iii) has outer input stream of type "TBSCAN"; (iv) the QEP has more than 100 LOLEPOPs. As an example, the type "NLJOIN" is translated into a JSON object array with three objects: *id*, *type* and *popProperties*, where *id* has the value of 1, *type* has the value of "NLJOIN" and *popProperties* contains an array with all properties related to this LOLEPOP. The id attributed to each LOLEPOP is given by creation order, where the first one receives the id 1, the second number 2 and so on.

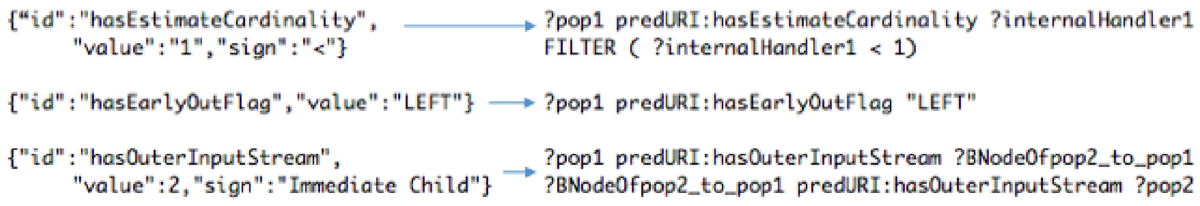
JSON standard was chosen because it is a lightweight data-interchange format, easy for humans to read and analyze if necessary and easy for machines to parse JSON objects. OptImatch uses the received JSON to auto-generate the SPARQL query, mapping each JSON object to a clause inside the WHERE statement in the query. Figure 24 depicts the JSON example above mapped to SPARQL query.

Figure 23 – Generated JSON Object



```
{"pops":[
     {"ID":1,"type":"NLJOIN","popProperties":[
          {"id":"hasEstimateCardinality","value":"1","sign":"<"},
          {"id":"hasEarlyOutFlag","value":"LEFT"},
          {"id":"hasOuterInputStream","value":2,"sign":"Immediate Child"},
          {"id":"hasInnerInputStream","value":3}
     ]},
     {"ID":2,"type":"TBSCAN","popProperties":[
          {"id":"hasOutputStream","value":1}
     ]}
]},
{"planDetails":[
     {"id":"hasNumberOfPops","value":"100","sign":">"}
]}
```

Figure 24 – JSON to SPARQL Mapping



```
{"id":"hasEstimateCardinality",                  ?pop1 predURI:hasEstimateCardinality ?internalHandler1
     "value":"1","sign":"<"}                     FILTER ( ?internalHandler1 < 1)

{"id":"hasEarlyOutFlag","value":"LEFT"}          ?pop1 predURI:hasEarlyOutFlag "LEFT"

{"id":"hasOuterInputStream",                     ?pop1 predURI:hasOuterInputStream ?BNodeOfpop2_to_pop1
     "value":2,"sign":"Immediate Child"}         ?BNodeOfpop2_to_pop1 predURI:hasOuterInputStream ?pop2
```

A query is mainly divided in two parts: the "SELECT" clause that defines the variables to be retrieved and the "WHERE" clause that defines the properties to be matched against the RDF workload selected by the user. All variables that appear in the query are defined by the "?" symbol prefix and a variable, for example, $?variableName$. The same variable can be referenced multiple times inside the "WHERE" clauses. This convention is used to define properties to resources, relationships between them and also for filtering purposes. Even more, the query accepts the use of alias in the "SELECT" statement, with the purpose of changing the column name in the results. At the SPARQL query creation the system defines the first LOLEPOP with the "TOP" alias and all other LOLEPOPs with the type plus its number, such as: "BASE5" if the LOLEPOP is a "BASE OB" and its id is 5.

To create a query, first OptImatch creates all prefixes that will be used in the query. Prefixes are used to abbreviate IRIs (RDF URI references), making the query smaller to send and more human readable. The four types of IRIs that our tool uses is: $< http : //explainPlan/PlanPop/ >$ for LOLEPOPS, $< http : //explainPlan/PlanBaseObject/ >$ for base objects, $< http : //explainPlan/PlanPred/ >$ for predicates and $< http : //explainPlan/PlanDetails/ >$ for global properties of the QEP. After the prefixes are set, the system creates the "SELECT" clause and adds a LOLEPOP to it if the user has chosen to retrieves the LOLEPOP in the result. Then, the "WHERE" clause is written by the tool, line per line, transforming each property (JSON Object) to a clause in the query.

As mentioned in [Damasio et al. 2016], our framework allows us to autogenerate SPARQL queries with a wide range of characteristics, including nesting, filtering, multiple resource mapping, and specifying property paths as well as blank nodes. Looking to facilitate the query creation, the concept of handlers was introduced. Handlers are used inside the query to automatically create a name for variables and they are generated at running time, when each property (JSON object) that needs a handler is added to the query. OptImatch makes use of three types of handlers: *result*, *internal* and *blank node*.

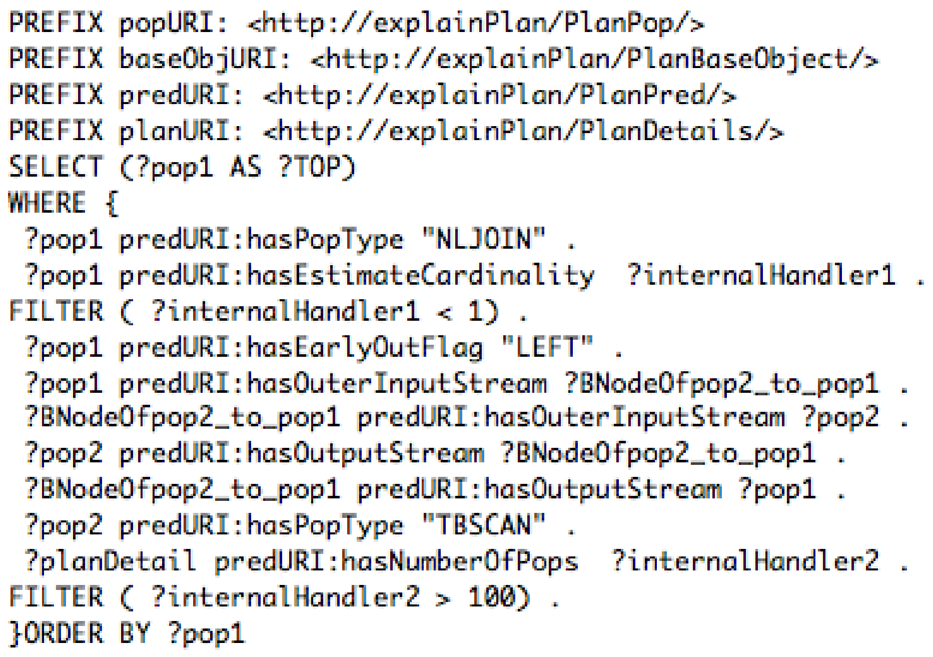
Result handlers are used for retrieval of query results. This type of handler is composed of the name "pop" plus the number of the LOLEPOP shown on the problem pattern graph tree that appears to the user. As an example, the query refers to the first LOLEPOP of the problem pattern by the handler $?pop1$, where it is used in the "SELECT" statement, returning a column with this name to the user. Additionally, the same handler is used inside the "WHERE" statement for creation of properties of the pattern. To relate the *type* property to the resource $?pop1$, the SPARQL generator module of the proposed tool adds the predicate *hasPopType* to the resource. As an example, if the type NLJOIN is selected, the clause in the query will stay in the following way: $?pop1$ *hasPopType* "NLJOIN". Furthermore, depending on the type of property, the system needs to make use of *internal handlers* to handle it.

Internal handlers are used to help properties creation, such as filtering. This handler is composed of the name *internalHandler* plus a number. The number is created in incremental order and is not attached to a specific resource. If the user wants to add to the $pop1$ the property "Estimate Cardinality" with value smaller than 1, OptImatch will then add the predicate *hasEstimateCardinality* to the resource $?pop1$ with the object $?internalHandler1$ and then it will utilized the internal handler in the filter clause: FILTER ($?internalHandler1 < 1$).

Blank node handlers are used to create relationship between LOLEPOPs. As previously mentioned in this chapter, the system makes use of blank nodes for RDF creation, so it can storage all properties correctly. As a result, the link between one LOLEPOP and another is not direct, it must pass through a blank node. OptImatch makes use of this handler to deal with the connection, for example, to create the relationship between $?pop1$ and $?pop2$, $?pop2$ being an immediate outer input stream of $?pop1$, first the system create a clause connecting $?pop1$ to the blank node handler $?BNodeOfpop2\_to\_pop1$ with the predicate *hasOuterInputStream*: $?pop1$ *hasOuterInputStream* $?BNodeOfpop2\_to\_pop1$.

Finishing the "WHERE" clause, OptImatch adds the statement "ORDER BY ?pop1", so the result returned to the user is ordered. Figure 25 depicts the full auto-generated query of the JSON on Figure 23 and Algorithm 2 represents the algorithm of the transformation of a problem pattern to a SPARQL query.

Figure 25 – Auto-generated SPARQL Query



```
PREFIX popURI: <http://explainPlan/PlanPop/>
PREFIX baseObjURI: <http://explainPlan/PlanBaseObject/>
PREFIX predURI: <http://explainPlan/PlanPred/>
PREFIX planURI: <http://explainPlan/PlanDetails/>
SELECT (?pop1 AS ?TOP)
WHERE {
 ?pop1 predURI:hasPopType "NLJOIN" .
 ?pop1 predURI:hasEstimateCardinality  ?internalHandler1 .
FILTER ( ?internalHandler1 < 1) .
 ?pop1 predURI:hasEarlyOutFlag "LEFT" .
 ?pop1 predURI:hasOuterInputStream ?BNodeOfpop2_to_pop1 .
 ?BNodeOfpop2_to_pop1 predURI:hasOuterInputStream ?pop2 .
 ?pop2 predURI:hasOutputStream ?BNodeOfpop2_to_pop1 .
 ?BNodeOfpop2_to_pop1 predURI:hasOutputStream ?pop1 .
 ?pop2 predURI:hasPopType "TBSCAN" .
 ?planDetail predURI:hasNumberOfPops  ?internalHandler2 .
FILTER ( ?internalHandler2 > 100) .
}ORDER BY ?pop1
```

Algorithm 2 Problem Pattern to SPARQL Query

Input: problem pattern *probPat*
Output: problem pattern *probPat* transformed to SPARQL Query *sparql*

```
1: function CreateQueryFromProblemPattern(probPat)
2:     probPatJsonObj[ ] := probPat translated to JSON array
3:     for all probPatJsonObj in probPatJsonObj[ ] do
4:         query := probPatJsonObj translated to a SPARQL query
5:         sparql := sparql + query
6:     end for
7:     return sparql
8: end function
```

6 Syntax for Recommendations

Recommendations are pre-defined patterns created by experts inside IBM and collaborators with the intuit of describing any problem for the user that is found in their QEP workload. Sometimes, a problem could be complex and a static text for the user may not be ideal, as the user may still not fully understand the recommendation. To avoid this problem, a handler tagging language was introduced. With the recommendation syntax, the recommendation is automatically adapted to the current QEP at running time. At this point, the SPARQL query of the problem pattern shown on Figure 16 is introduced, so all functionalities of the syntax can be covered. Figure 26 depicts this SPARQL query and for convenience also repeats Figure 16.

As previously mentioned, the queries used in the OptImatch contain aliases related to each *result handler*. For example, in Figure 26, $?pop1$ has as alias $?TOP$, $?pop2$ has as alias $?ANY2$ and $?pop4$ has as alias $?BASE4$. Aliases are used for tag recommendations to a specific *result handler*. Also, OptImatch has the capability to tag multiple *result handlers* and sets of *result handlers* in the same recommendation. Moreover, when tagging, the user can use help functions to list results, predicates, column names, table names, etc. Sometimes, when retrieving the result of a recommendation for a user, the expert may want to limit the number of results since in some complex queries or big QEPs, the number of results could be very large and only some are important to the user.

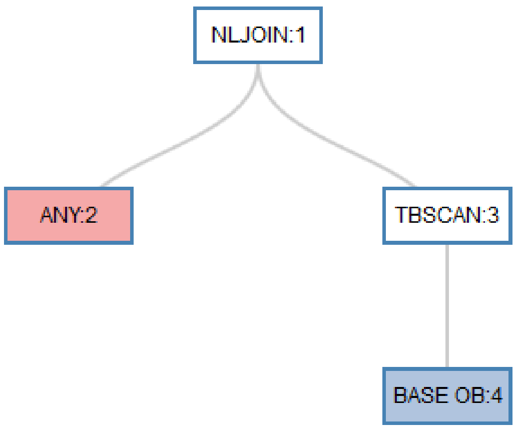
The tagging is done by surrounding the static piece of the recommendation with the dynamic part. While writing a recommendation, the user can refer to a handler by the use of "@" sign plus the name of the alias. For example, to refer to $?TOP1$, the user writes $@TOP1$. To retrieve the result of an handler, the function $listResult()$ needs to be used. For the current problem pattern, the expert may write the following recommendation:

> "Create index on table @BASE4.listResult() on input columns of the LOLEPOP @TOP.listResult()"

However, the recommendation syntax allows experts to add more information with the use of help functions. For example, instead of mentioning that the user needs to create index on columns of a given LOLEPOP, the expert can add the columns names of the LOLEPOP to the recommendation. To achieve this, the function $listColumns()$ is used. The $listColumns()$ function allows four types of inputs: $INPUT$, $OUTPUT$, $PREDICATE$ and ALL, where ALL returns input, output and predicate columns. When a recommendation is saved, OptImatch automatically goes through the recommendation and looks for $listColumns()$ functions. If any is found, the proposed tool rewrites the query to return also the columns needed for each handler with the function attached to it. For example, the recommendation above can be rewritten to the following one:

"Create index on table @BASE4.listResult() on the following columns: @TOP.listColumns("INPUT")"

Figure 26 – SPARQL Query and related problem pattern



```
PREFIX popURI: <http://explainPlan/PlanPop/>
PREFIX baseObjURI: <http://explainPlan/PlanBaseObject/>
PREFIX predURI: <http://explainPlan/PlanPred/>
PREFIX planURI: <http://explainPlan/PlanDetails/>
SELECT (?pop1 AS ?TOP) (?pop2 AS ?ANY2) (?pop3 AS ?TBSCAN3)
(?pop4 AS ?BASE4)
WHERE {
 ?pop1 predURI:hasOuterInputStream ?BNodeOfpop2_to_pop1 .
 ?BNodeOfpop2_to_pop1 predURI:hasOuterInputStream ?pop2 .
 ?pop2 predURI:hasOutputStream ?BNodeOfpop2_to_pop1 .
 ?BNodeOfpop2_to_pop1 predURI:hasOutputStream ?pop1 .
 ?pop1 predURI:hasInnerInputStream ?BNodeOfpop3_to_pop1 .
 ?BNodeOfpop3_to_pop1 predURI:hasInnerInputStream ?pop3 .
 ?pop3 predURI:hasOutputStream ?BNodeOfpop3_to_pop1 .
 ?BNodeOfpop3_to_pop1 predURI:hasOutputStream ?pop1 .
 ?pop1 predURI:hasPopType "NLJOIN" .
 ?pop2 predURI:hasEstimateCardinality  ?internalHandler1 .
FILTER ( ?internalHandler1 > 1) .
 ?pop2 predURI:hasPopType  ?internalHandler2 .
 ?pop3 predURI:hasInputStream ?BNodeOfpop4_to_pop3 .
 ?BNodeOfpop4_to_pop3 predURI:hasInputStream ?pop4 .
 ?pop4 predURI:hasOutputStream ?BNodeOfpop4_to_pop3 .
 ?BNodeOfpop4_to_pop3 predURI:hasOutputStream ?pop3 .
 ?pop3 predURI:hasPopType "TBSCAN" .
 ?pop4 predURI:hasEstimateCardinality  ?internalHandler3 .
FILTER ( ?internalHandler3 > 100) .
 ?pop4 predURI:isABaseObj  ?internalHandler4 .
}ORDER BY ?pop1
```

When saving this new recommendation, OptImatch will find the *listColumns*() function attached to the *@TOP* handler, thus the system will add the *hasColumnName* predicate to the input stream of the *?pop*1 handler in the query. The system automatically creates it by first adding an auxiliary handler *?COLUMN_IN_TOP_AUX* for the predicate *hasInputStream* of the resource *?pop*1. Then the predicate *hasColumnName* is added to the resource *?COLUMN_IN_TOP_AUX* with the object *?COLUMN_IN_TOP*. By this, *?COLUMN_IN_TOP* will retrieve all input columns. Lastly, this handler will be added in the "SELECT" statement, so the information can be returned to the user. The modified query is represented in the Figure 27, where all modifications are underlined.

Figure 27 – SPARQL Query from Figure 26 with listColumns

```
PREFIX popURI: <http://explainPlan/PlanPop/>
PREFIX baseObjURI: <http://explainPlan/PlanBaseObject/>
PREFIX predURI: <http://explainPlan/PlanPred/>
PREFIX planURI: <http://explainPlan/PlanDetails/>
SELECT (?pop1 AS ?TOP) (?pop2 AS ?ANY2) (?pop3 AS ?TBSCAN3)
(?pop4 AS ?BASE4) ?COLUMN_IN_TOP
WHERE {
 ?pop1 predURI:hasOuterInputStream ?BNodeOfpop2_to_pop1 .
 ?BNodeOfpop2_to_pop1 predURI:hasOuterInputStream ?pop2 .
 ?pop2 predURI:hasOutputStream ?BNodeOfpop2_to_pop1 .
 ?BNodeOfpop2_to_pop1 predURI:hasOutputStream ?pop1 .
 ?pop1 predURI:hasInnerInputStream ?BNodeOfpop3_to_pop1 .
 ?BNodeOfpop3_to_pop1 predURI:hasInnerInputStream ?pop3 .
 ?pop3 predURI:hasOutputStream ?BNodeOfpop3_to_pop1 .
 ?BNodeOfpop3_to_pop1 predURI:hasOutputStream ?pop1 .
 ?pop1 predURI:hasPopType "NLJOIN" .
 ?pop2 predURI:hasEstimateCardinality  ?internalHandler1 .
FILTER ( ?internalHandler1 > 1) .
 ?pop2 predURI:hasPopType  ?internalHandler2 .
 ?pop3 predURI:hasInputStream ?BNodeOfpop4_to_pop3 .
 ?BNodeOfpop4_to_pop3 predURI:hasInputStream ?pop4 .
 ?pop4 predURI:hasOutputStream ?BNodeOfpop4_to_pop3 .
 ?BNodeOfpop4_to_pop3 predURI:hasOutputStream ?pop3 .
 ?pop3 predURI:hasPopType "TBSCAN" .
 ?pop4 predURI:hasEstimateCardinality  ?internalHandler3 .
FILTER ( ?internalHandler3 > 100) .
 ?pop4 predURI:isABaseObj  ?internalHandler4 .
?pop1 predURI:hasInputStream ?COLUMN_IN_TOP_AUX .
?COLUMN_IN_TOP_AUX predURI:hasColumnName ?COLUMN_IN_TOP .
}ORDER BY ?pop1
```

Additionally, OptImatch allows grouping handlers together for retrieval. This is done by surrounding the handlers with "[ ]". For example, to retrieve the *?ANY*2 and *?BASE*4 handlers together, the user will write: [@ANY2, @BASE4]. Grouping handlers will make the system return them linked by result. For example, when running the query on Figure 26, if the answer is LOLEPOP 3 and 6 (*?ANY*2) with LOLEPOP 8 and

10 (?$BASE4$), respectively, the system will replace [@ANY2, @BASE4].listResults() by: {3,8},{6,10}. Also, all functions that can be used in a handler can also be used for group of handlers. Lastly, the user may want to limit the number of results for a pattern that appears multiple times in the same QEP. To accomplish this, the following syntax needs to be attached after the function of a handler: ":" sign plus the max number of results to be retrieved. An example would be: [@ANY2, @BASE4].listResult():1, meaning that the result will be retrieved grouped and limited by 1 result.

Algorithm 3 represents how OptImatch saves the recommendation in the knowledge base.

Algorithm 3 Save Recommendation in KB

```
Input: problem pattern probPat
       tag of recommendation tag
       suggested recommendation rec
       current knowledge base KB[]
Output: updated knowledge base KB[]

1: function SaveRecommendation(probPat, tag, rec, KB[ ])
2:     query := CreateQueryFromProblemPattern(probPat)
3:     if rec contains listColumns() function then
4:         query := updated query with return columns names handlers
5:     end if
6:     recommendation := new Recommendation(tag, rec, query)
7:     KB[ ] := recommendation
8:     return KB[ ]
9: end function
```

7 Matching Patterns and Finding Solutions in the Knowledge Base

7.1 Matching Patterns against QEP workload

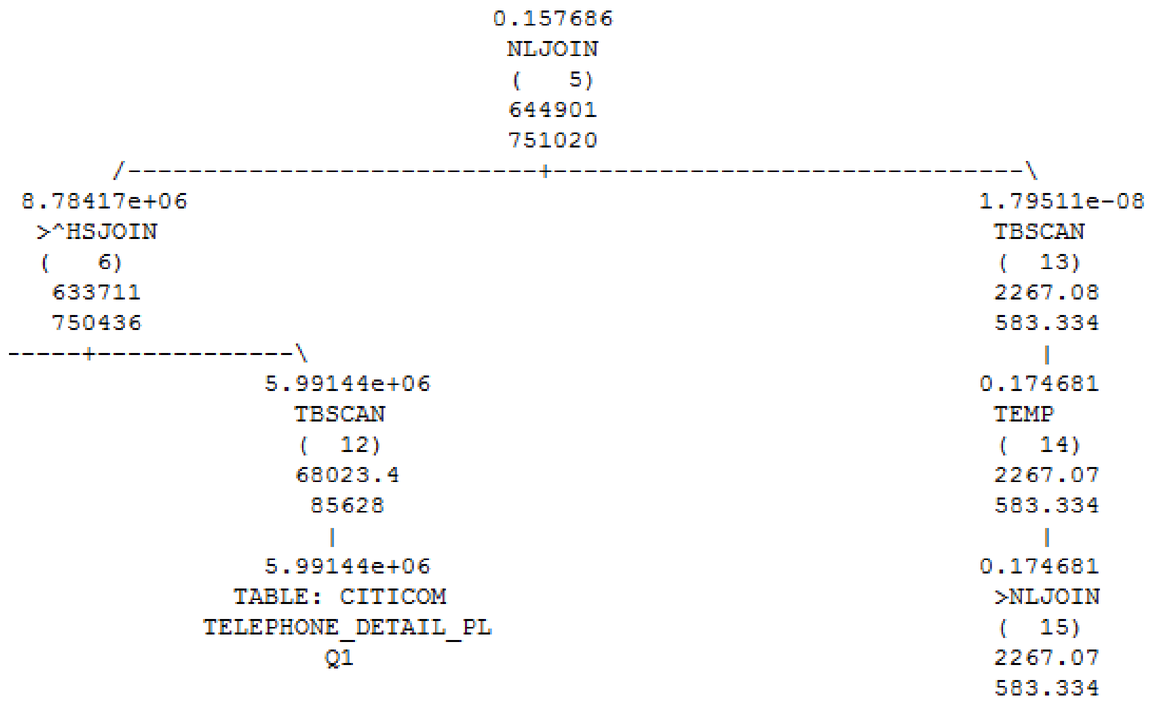
As previously mentioned, OptImatch allows dynamic analysis of ad-hoc patterns by matching problem patterns against a given QEP workloads. As explained in Chapter 5, the RDF file for a QEP is created at the time that the file is uploaded. With the RDF created, the first step to matching a pattern is to create a SPARQL query from the problem pattern described by the user. Then, with the auto-generated query, the system uses the SPARQL to search the pattern in the RDF files from the QEP workload and finally returns the result to the user.

The result is stored in a JSON object that contains an array of objects, where each one represents the result for a QEP. For better understanding the following pattern **(pattern C)** will be used: (i) LOLEPOP "pop1" is of type JOIN (which means any kind of join method, e.g NLJOIN, MSJOIN, HSJOIN); (ii) "pop1" has a descendant outer input stream "pop2" of type JOIN; (iii) "pop1" has a descendant inner input stream "pop3" of type JOIN; (iv) "pop2" is a left outer join; (v) "pop3" is a left outer join. This is a problem related to query rewrite and a possible recommendation may be: rewrite the query from the following structure (T1 LOJ T2) ... JOIN ... (T3 LOJ T4) to ((T1 LOJ T2).... JOIN ....T3) LOJ T4 as the rewritten query is more efficient. As mentioned on [Damasio et al. 2016] and [Damasio et al. 2016], this optimization is now automatically done in DB2 but was found to be a limitation in early versions of DB2. This illustrates the usefulness of the tool in database optimizer development as well as supporting clients that use previous version of the DB2 system.

The current problem pattern was found in a real user query workload, since it uses a previous version of the DB2. Moreover, this pattern is an example of a recursive query since the descendant outer and inner input stream with left outer join are not necessary immediate children of JOIN. Figure 28 depicts a portion of a QEP that contains this problem pattern, where the left outer join operator is prefixed by the > sign, as example: >NLJOIN and >HSJOIN.

In this portion of the QEP, the "pop1" is the LOLEPOP of ID equals to 5, being of type $NLJOIN$, the outer descendant child is the LOLEPOP number 6 and type of $HSJOIN$. Looking at this node the user can see the presence of the left outer join operator sign. Lastly, the inner descendant input stream is the LOLEPOP with ID number 15 and type $NLJOIN$, which also contains the > sign. When searching for descendant children, OptImatch does not require that all children have the same depth from their parent. As presented in this example, LOLEPOP 6 and 15 are descendant children from LOLEPOP 5, but in comparison to the parent node, they are not in the same depth.

Figure 28 – Portion of a QEP with problem pattern



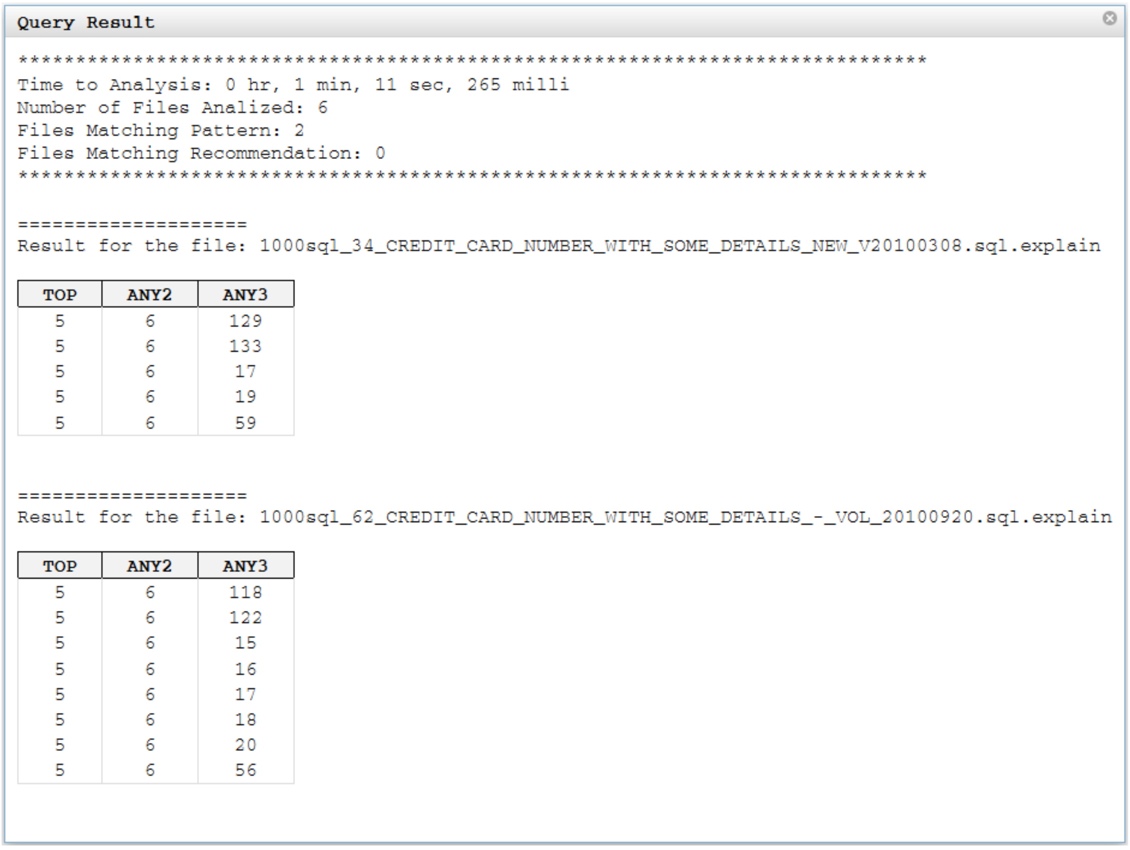
```
                                0.157686
                                 NLJOIN
                                 (   5)
                                 644901
                                 751020
       /---------------------------+-------------------------------\
 8.78417e+06                                                     1.79511e-08
  >^HSJOIN                                                        TBSCAN
  (   6)                                                          (  13)
   633711                                                         2267.08
   750436                                                         583.334
-----+-------------\                                                 |
                 5.99144e+06                                     0.174681
                   TBSCAN                                         TEMP
                   (  12)                                         (  14)
                   68023.4                                        2267.07
                    85628                                         583.334
                     |                                               |
                 5.99144e+06                                     0.174681
               TABLE: CITICOM                                     >NLJOIN
             TELEPHONE_DETAIL_PL                                  (  15)
                     Q1                                           2267.07
                                                                  583.334
```

With the problem pattern defined, once the user clicks on "Search for Pattern", the JSON object with all the properties is sent to the server and the system creates the SPARQL query from the JSON object. Then the query is execute against each RDF created from the QEP workload of the user. For each RDF, OptImatch creates an JSON Object with all results from the current QEP. Once the system finishes analysis of all the workload, a JSON Object with all results is returned to the user, where the result panel is created.

On the user's side, when OptImatch receives the answer, the JSON is translated to a human readable language. For each QEP, the framework prints the file analyzed and right below, in the result panel, it shows 2 sections: the result and recommendation section. The result section provides an example of a result panel, where it shows the result is shown to the user in a table format, with headers for each column that correspond to the name of the node in the graphical tree. Figure 29 depicts this example of result panel returned to the user. This search was based in six QEP, where 2 contains the problem pattern defined above and four does not. Lastly, along the result, OptImatch also returns a header with general information, such as analysis time, number of files analyzed and how many of those were matched against the custom problem pattern created by the user as well as how many matched any recommendation in the knowledge base.

As can be seen on this example, there is no recommendations returned to the user. All recommendations are not necessarily shown to the user, just the ones that matches the QEP file are shown by OptImatch. If none are find, the system does not provide the recommendation section to the user. An illustration of the result panel with recommendation is presented in Section 7.2, where an example is returned to the user, showing the tag as well as the recommendation itself, with the handlers already changed to the result of that specific file.

Figure 29 – Result Panel for Problem Pattern Without Recommendation



```
*******************************************************************************
Time to Analysis: 0 hr, 1 min, 11 sec, 265 milli
Number of Files Analized: 6
Files Matching Pattern: 2
Files Matching Recommendation: 0
*******************************************************************************

====================
Result for the file: 1000sql_34_CREDIT_CARD_NUMBER_WITH_SOME_DETAILS_NEW_V20100308.sql.explain
```

TOP	ANY2	ANY3
5	6	129
5	6	133
5	6	17
5	6	19
5	6	59

```
====================
Result for the file: 1000sql_62_CREDIT_CARD_NUMBER_WITH_SOME_DETAILS_-_VOL_20100920.sql.explain
```

TOP	ANY2	ANY3
5	6	118
5	6	122
5	6	15
5	6	16
5	6	17
5	6	18
5	6	20
5	6	56

The algorithm for searching for single patterns is described in Algorithm 4. It represents the whole process of finding custom problem patterns, starting from the creation of the RDF files given the QEP workload, passing through the creation of the SPARQL query until the creation of the JSON containing the result of all files analyzed.

The capability of allowing users to describe single patterns at running time adds the functionality of dynamically analyzing patterns against QEP workloads uploaded by an user in OptImatch. Even more, this tool also provides problem identification by finding solutions and recommendations inside the knowledge base, adding versatility to the tool, as described in the following section.

Algorithm 4 Search Problem Pattern

Input: query of the problem pattern *query*
RDF graph file of the QEP *rdfg*
Output: JSON object with the match *match*

```
1: function FindMatches(query, rdfg)
2:     queryResult[ ] := match abstracted problem pattern query against RDF file of the QEP rdfg
3:     if queryResult[ ] != empty then
4:         matchJsonObj := save into a JSON object the detransformation of the queryResult[ ] by relating any portion matched of the rdf back to corresponding QEP
5:         match := matchJsonObj
6:     end if
7:     return match
8: end function
```

7.2 Finding Solutions in the Knowledge Base

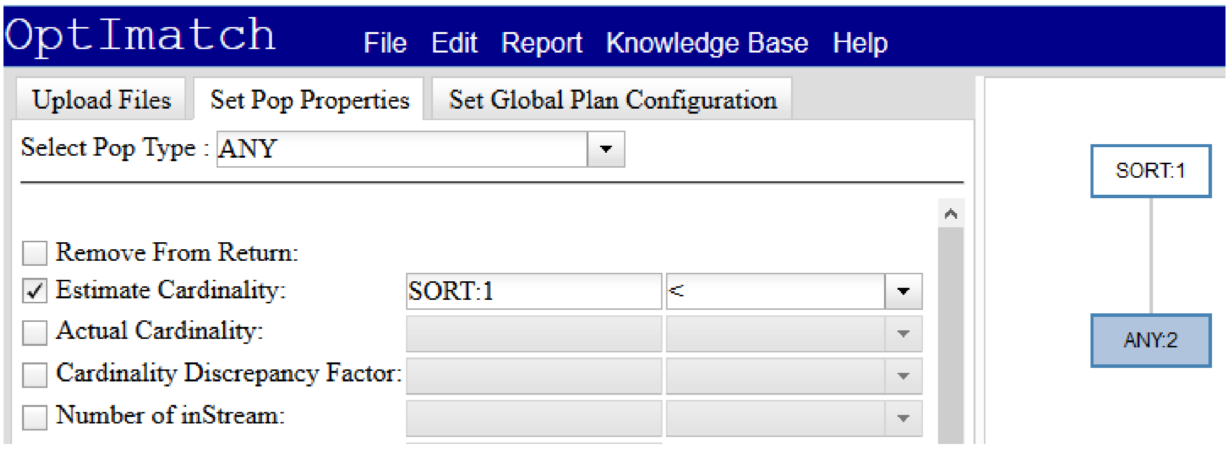
Knowledge Base is a database table containing predetermined problem patterns and recommendations for them. OptImatch has the capability of access the Knowledge Base, iterating over all problem patterns, looking for those that match a QEP. If matched, the system retrieves the recommendation and replace any handler in the recommendation by the corresponding result of the QEP, returning it to the user.

For further elaboration, the following pattern will be used: (i) LOLEPOP "pop1" is of type SORT; (ii) "pop1" has a input stream "pop2" with I/O cost less than the I/O cost of the "pop1". This is a problem related to SORT spilling and a potential recommendation for it could be to change the database memory configuration. This change is recommended in order to increase sort memory if the number of QEPs containing this pattern is large enough to benefit the performance of many queries in the workload.

To write this problem pattern, the user can utilize another capability of the system: comparison of properties between different LOLEPOPs. Figure 30 depicts the problem pattern above in the GUI. As seen on the figure, to achieve the property (ii), the value of the estimate cardinality of the "ANY:2" (pop2) LOLEPOP to be filtered is a reference for the LOLEPOP "SORT:1" (pop1). OptImatch automatically understands which property of the referenced LOLEPOP needs to be compared, so the user does not need to specify it. The auto-generated SPARQL query of this problem pattern is shown on Figure 31.

The underlined section of the query below represents the transformation of the property (ii) to the SPARQL query. To accomplish it, the system utilizes an handler ?*internalHandler*2 inside the FILTER clause and then add it as the object of the predicate "estimate cardinality" of the resource ?*pop*1.

Figure 30 – GUI With Problem Pattern



Select Pop Type : ANY

- [ ] Remove From Return:
- [✓] Estimate Cardinality: SORT:1 <
- [ ] Actual Cardinality:
- [ ] Cardinality Discrepancy Factor:
- [ ] Number of inStream:

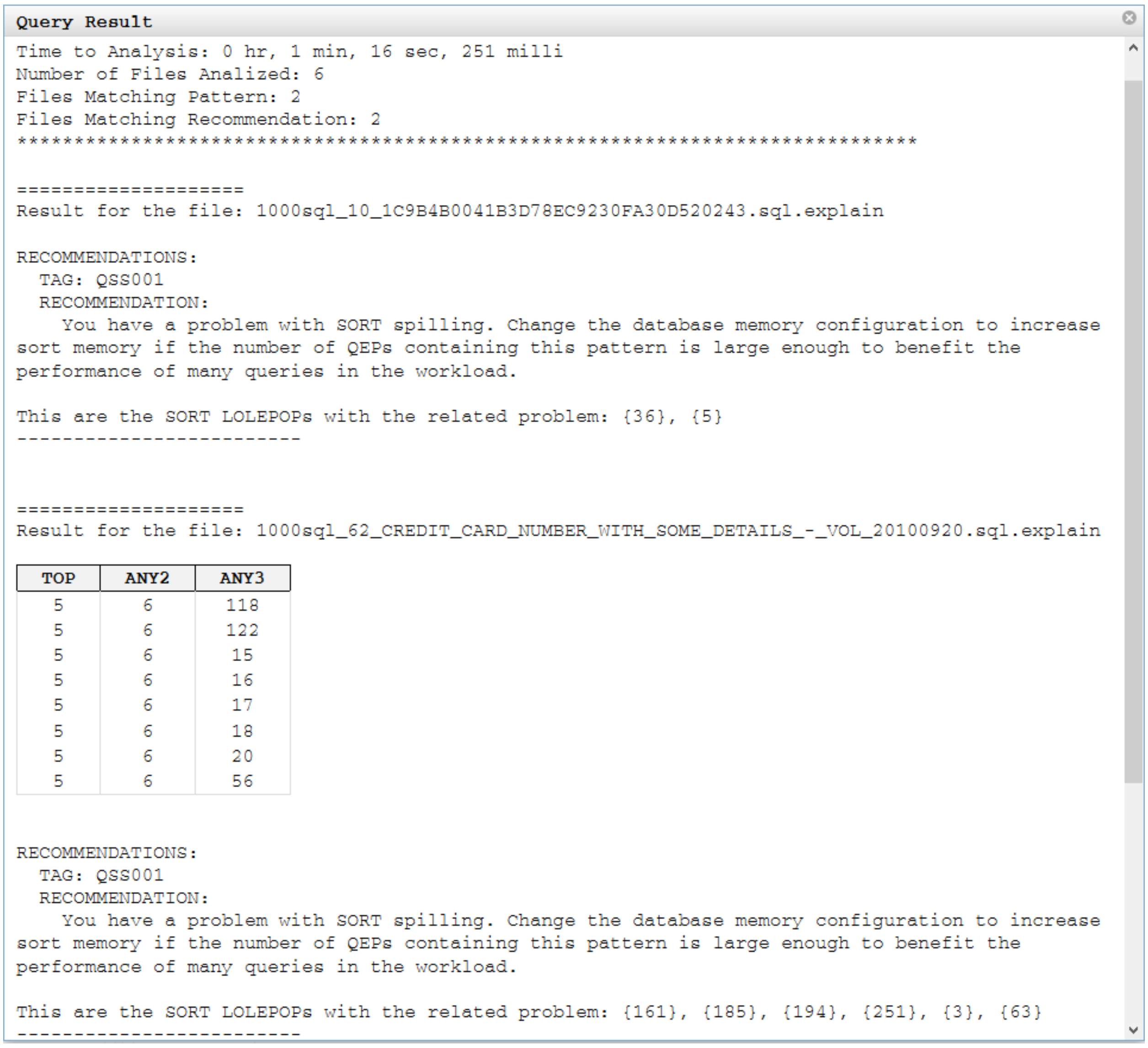
Figure 31 – SPARQL Query from Figure 30

```
PREFIX popURI: <http://explainPlan/PlanPop/>
PREFIX baseObjURI: <http://explainPlan/PlanBaseObject/>
PREFIX predURI: <http://explainPlan/PlanPred/>
PREFIX planURI: <http://explainPlan/PlanDetails/>
SELECT (?pop1 AS ?TOP) (?pop2 AS ?ANY2)
WHERE {
 ?pop1 predURI:hasInputStream ?BNodeOfpop2_to_pop1 .
 ?BNodeOfpop2_to_pop1 predURI:hasInputStream ?pop2 .
 ?pop2 predURI:hasOutputStream ?BNodeOfpop2_to_pop1 .
 ?BNodeOfpop2_to_pop1 predURI:hasOutputStream ?pop1 .
 ?pop1 predURI:hasPopType "SORT" .
 ?pop2 predURI:hasEstimateCardinality  ?internalHandler1 .
FILTER ( ?internalHandler1 <  ?internalHandler2) .
 ?pop1 predURI:hasEstimateCardinality  ?internalHandler2 .
 ?pop2 predURI:hasPopType  ?internalHandler3 .
}ORDER BY ?pop1
```

For this search the aforementioned recommendation was saved in the knowledge base and the same 6 QEPs used in Section 7.2 will be used. Also, for comparison purposes, the same custom problem pattern from last section will be utilized in the search. Figure 32 represents the result of searching for the problem pattern and for recommendations. There are also 2 files that contain the recommendation. In order to best understand how OptImatch creates the result panel, the figure displays the result panel with one file that contains only the recommendation and another with recommendation and results matched to the custom problem pattern.

For the recommendation the following description was added: "You have a problem with SORT spilling. Change the database memory configuration to increase sort memory if the number of QEPs containing this pattern is large enough to benefit the performance of many queries in the workload.". Moreover, the following statement was also included: "Here are the SORT LOLEPOPs with the related problem: @TOP.listResult()". As mentioned in Chapter 6, when retrieving the result, OptImatch automatically replaces the syntax with the result. In this case, it replaced the ID of the "TOP" LOLEPOP, that is the SORT LOLEPOP.

Figure 32 – Result Panel for Problem Pattern With Recommendation



```
Query Result
Time to Analysis: 0 hr, 1 min, 16 sec, 251 milli
Number of Files Analized: 6
Files Matching Pattern: 2
Files Matching Recommendation: 2
*******************************************************************************

====================
Result for the file: 1000sql_10_1C9B4B0041B3D78EC9230FA30D520243.sql.explain

RECOMMENDATIONS:
  TAG: QSS001
  RECOMMENDATION:
    You have a problem with SORT spilling. Change the database memory configuration to increase
sort memory if the number of QEPs containing this pattern is large enough to benefit the
performance of many queries in the workload.

This are the SORT LOLEPOPs with the related problem: {36}, {5}
-------------------------


====================
Result for the file: 1000sql_62_CREDIT_CARD_NUMBER_WITH_SOME_DETAILS_-_VOL_20100920.sql.explain
```

TOP	ANY2	ANY3
5	6	118
5	6	122
5	6	15
5	6	16
5	6	17
5	6	18
5	6	20
5	6	56

```
RECOMMENDATIONS:
  TAG: QSS001
  RECOMMENDATION:
    You have a problem with SORT spilling. Change the database memory configuration to increase
sort memory if the number of QEPs containing this pattern is large enough to benefit the
performance of many queries in the workload.

This are the SORT LOLEPOPs with the related problem: {161}, {185}, {194}, {251}, {3}, {63}
-------------------------
```

The algorithm used to search for the recommendation in the knowledge base is described in Algorithm 5. Lastly, Algorithm 6 describes the searching for both recommendations and problem pattern created by the user.

Algorithm 5 Search Recommendations in the Knowledge Base

Input: RDF graph from query execution plan file $rdfg$
current knowledge base $KB[\ ]$
Output: JSONArray containing all recommendations that matches $RECs[\ ]$

```
 1: function SearchRecommendations($rdfg$, $KB[\ ]$)
 2:     for all Recommendations $rec$ in $KB[\ ]$ do
 3:         $query$ := GetQueryFromKB($rec$)
 4:         $queryResult[\ ]$ := match abstracted problem pattern $query$ against RDF file of the QEP $rdfg$
 5:         if $queryResult[\ ]$ != empty then
 6:             $rec$ := replace any handler inside the recommendation with the current value in the $queryResult[\ ]$
 7:             $RECs$.append($rec$)
 8:         end if
 9:     end for
10:     return $RECs[\ ]$
11: end function
```

Algorithm 6 Search Problem Pattern And Recommendation

Input: problem pattern $probPat$
query execution plan files $QEPFs[\ ]$
current knowledge base $KB[\ ]$
Output: JSON object with all matches $MATCHES[\ ]$

```
 1: function SearchProblemPatternAndRecommendation($probPat$, $QEPFs[\ ]$, $KB[\ ]$)
 2:     $RDFGs[\ ]$ := CreateRDF($QEPFs[\ ]$)
 3:     $query$ := CreateQueryFromProblemPattern($probPat$)
 4:     for all $rdfg$ in $RDFGs[\ ]$ do
 5:         $QEPName$ := get the query execution plan name from the $rdfg$
 6:         $jsonProblemPattern$ := FindMatches($query$, $rdfg$)
 7:         $jsonRecommentadions$ := SearchRecommendations($rdfg$, $KB[\ ]$)
 8:         $QEPName$.append($jsonProblemPattern$).append($jsonRecommentadions$)
 9:         if $QEPName$ != empty then
10:             $MATCHES$.append($QEPName$)
11:             $QEPName$ := empty
12:         end if
13:     end for
14:     return $MATCHES[\ ]$
15: end function
```

7.3 Asynchronous Search

Previously the search was done in a synchronous way and in one time, meaning that the server received one request which asked for the result for all the QEP files uploaded by the user. This approach worked for small size of workloads. However, with the increase of the number of files, depending on the number of matches, the size of the answer started to get much larger, so the script was taking couple of minutes to prepare the answer in a human readable language text and present it. Even more, during the server analysis, the user did not receive any feedback about the number of files already analyzed, how many matches were found or the number of files with recommendations.

As seen in the Section 8.3, for 1000 QEP files OptImatch took around 50 minutes to analyze it. In other words, the user was waiting 50 minutes to received feedback from the server, without knowing the status during analysis. Aiming to improve the performance of the search and at the same time provide feedback for the user, the synchronous search was replaced by an asynchronous search.

First, when the user decides to analyze the query workload, OptImatch divides the total number of files in small buckets, each containing 20 QEP. This approach allows the proposed tool to receive feedback from the server every 20 QEPs, which also provides the necessary feedback to the user. The next step is to send asynchronous buckets to the server. OptImatch can have 6 asynchronous buckets requested at a time. After an answer is received from the server, it creates another request and stay in this loop until all files are analyzed. Another benefit of using this method is that at the same time that the server is analyzing the QEPs, OptImatch is building the result for the user with the already analyzed files. Before the result was created just receiving all results.

This new search method provides the proposed tool an overall increase in performance as well as a way to give back feedback for the user in terms of number of files analyzed, number of files that matched the custom problem pattern and number of files that matched at least one recommendation. In addition, now the user has access to the partial result of the search.

8 Experimental Study

In this chapter the experimental studies performed as well as the results obtained will be explained. The main idea of the experiments is to measure the benefits that OptImatch provides for IBM. The experiments focus on four main objectives:

- Analysis of the effectiveness of this tool by utilizing 1000 QEP files from real IBM customer query workload. (Sections 8.2 and 8.4).
- Evaluation of the performance and scalability of OptImatch against different size of workload (Section 8.2.1), number of LOLEPOPs (Section 8.2.2) and number of recommendations (Section 8.2.3).
- Performance evaluation of the asynchronous search against the old synchronous search. (Section 8.3).
- A comparative study of this tool against the manual searching task, exposing the benefit of OptImatch about time and precision. (Section 8.4).

8.1 Setup

A machine with Intel® Core^{TM} i7-56600M with 2.6GHz and 8GB of RAM memory was used to run the experiments. The three problem patterns utilized for the experiments are described below. These problem patterns were created by IBM experts, where each of them has its own recommendation (stored in the knowledge base).

- **Pattern #1 (Pattern A)** - Defined by the following properties: (i) LOLEPOP "pop1" is of type "NLJOIN"; (ii) "pop1" has an outer input stream of type "ANY" and cardinality greater than 1; (iii) "pop1" has an inner input stream "pop3" of type TBSCAN; (iv) "pop3" has a generic input stream of type "BASE OB" and cardinality greater than 100. This pattern is related to the cost of the NLJOIN to scan the entire table TBSCAN. (Section 4.1).
- **Pattern #2 (Pattern B)** - Defined by the following properties: (i) LOLEPOP of type index Scan (IXSCAN) or table scan (TBSCAN) (ii) has cardinality smaller than 0.001; (iii) has a generic input stream of type Base Object (BASE OB) and cardinality bigger than 100000. This pattern is related to statistics for better cardinality estimation, consequently changing the cost estimation. (Section 5.2).

- **Pattern #3 (Pattern C)** - Defined by the following properties: (i) LOLEPOP "pop1" is of type JOIN (which means any kind of join method, e.g. NLJOIN, MSJOIN, HSJOIN); (ii) "pop1" has a descendant outer input stream "pop2" of type JOIN; (iii) "pop1" has a descendant inner input stream "pop3" of type JOIN; (iv) "pop2" is a left outer join; (v) "pop3" is a left outer join. This pattern is related to rewriting the query. (Section 7.1).

8.2 Performance and Scalability

The first phase of the experiments was focused on the evaluation of the performance and scalability of the proposed system by testing it in three different scenarios: size of workload (number of QEP files), number of LOLEPOPs (related to the size of a single QEP) and number of recommendations.

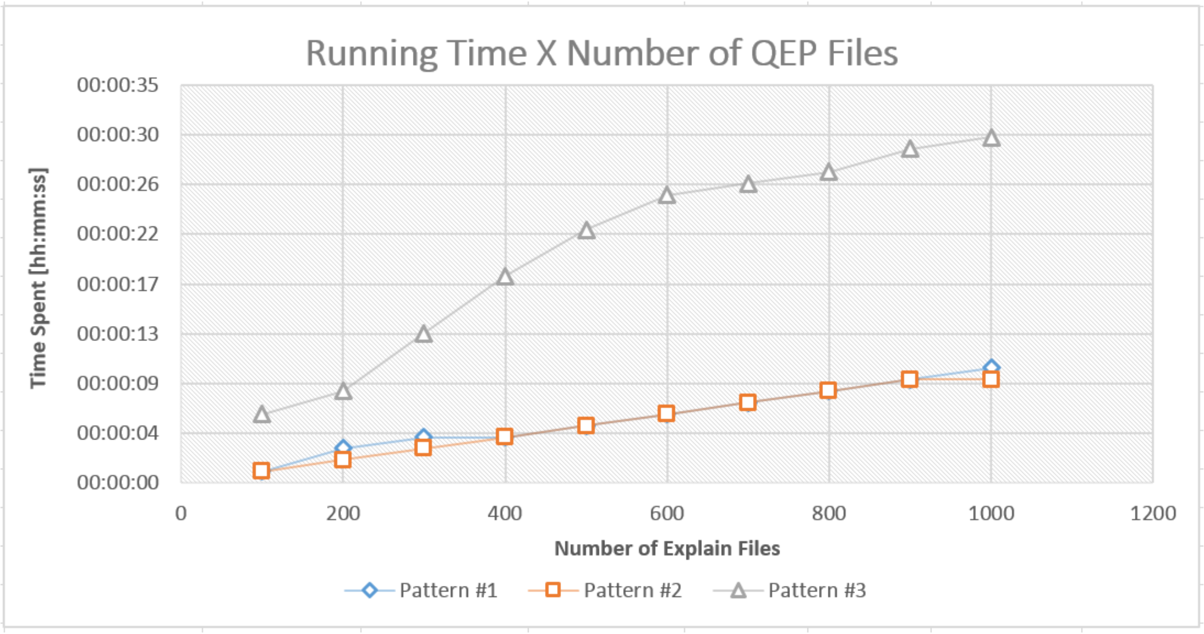
8.2.1 Size of Workload

In the first experiment, the performance of OptImatch was measured in terms of searching for portions of QEPs against various sizes of the QEP workload. The objective here is to analyze how the number of files can interfere in the performance and whether the system is scalable in term of files. To create this experiment, the real IBM query workload was divided in 10 buckets, where each one contains a different number of QEP files. The first bucket contains 100 QEPs, the second 200 QEPs and so on, adding 100 QEPs in each bucket until the bucket with 1000 QEPs is reached. In other words, the distribution of the buckets was as follows: [100, 200, ..., 1000].

The QEP files were divided randomly into buckets. This experiment was executed 10 times for each bucket and the average time (in seconds) is reported. Figure 33 depicts the result of the experiment.

As can be seen, the proposed tool has a linear performance over the number of QEP files in the workload, taking around 30 seconds to analyze 1000 QEPs. This performance makes OptImatch a real-time query problem determination search tool. Looking at pattern #1 and pattern #2, the tool takes around 10 seconds to analyze 1000 QEPs. Furthermore, a performance variation can be seen between pattern #3 and the others, where #3 takes 3 times more to be completed. This difference is given by the fact that OptImatch makes use of recursion in the pattern, searching against QEPs that normally contains more than 100 LOLEPOPs.

Figure 33 – Size of Workload Experiment

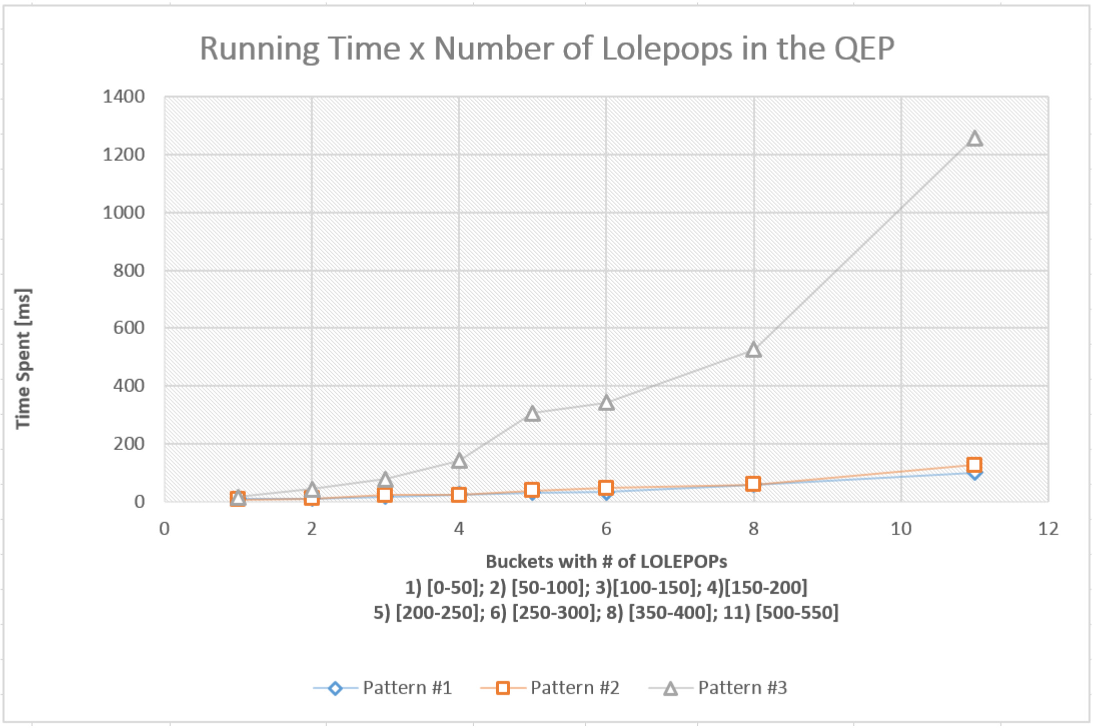


8.2.2 Number of LOLEPOPs

The second experiment is related with the number of LOLEPOPs. The number of LOLEPOPs in a QEP is tied to the size of the file, where the bigger the number, the bigger the size. The purpose of this experiment is to analyze the performance of the tool over different number of LOLEPOPs, verifying how efficient OptImatch is at searching for patterns as a function of number of LOLEPOPs. For this experiment, the real IBM query workload was divided in 11 buckets, where each one contains QEP files with different numbers of LOLEPOPs on it. The first bucket contains QEPs with number of LOLEPOPs between 0 and 50. The second one between 50 and 100 LOLEPOPs and so on, until bucket #11, that contains QEP files with number of LOLEPOPs between 500 and 550. The results of the performance for buckets #7, #9 and #10 in this test was not reported, as the query workload does not contain QEP with that number of LOLEPOPs. In other words, the distribution of the buckets is: [0-50], [50-100], [100-150], [150-200], [200-250], [250-300], [350-400] and [500-550].

The test was repeated 10 times for each bucket and the average time (in milliseconds) is reported. Figure 34 represents the result in a graph. The time for analysis increases as the number of LOLEPOPs in a QEP increases, as expected. Also, OptImatch can analyze files with big number of LOLEPOPs (between 500 and 550) in less than 200 milliseconds for patterns #1 and #2. Moreover, for these patterns, the time increases linearly. For pattern #3, when analyzing QEPs with more than 500 LOLEPOPs, the tool takes around 1.3 seconds. This extra time was expected as the query makes use of recursion.

Figure 34 – Number of LOLEPOPs Experiment

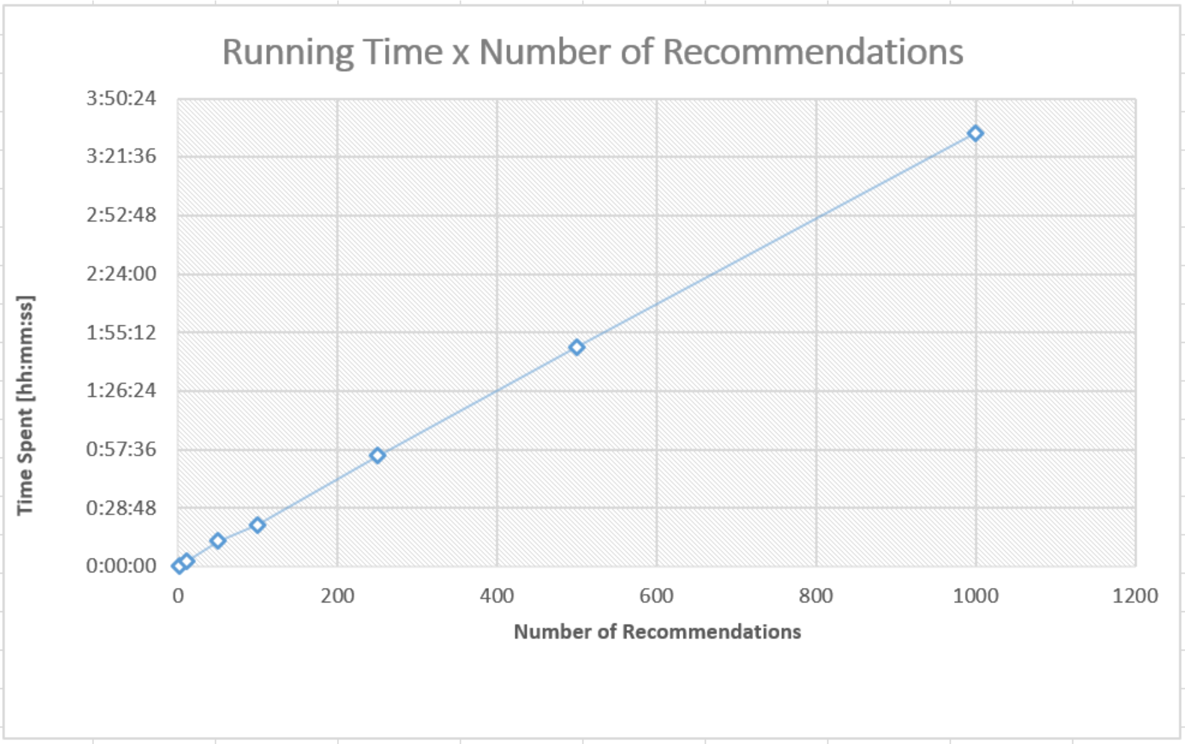


8.2.3 Number of Recommendations

The third experiment tests how the performance of the system is affected by the number of recommendations analyzed against 1000 of QEP files. This experiment was divided in six buckets, where the number or recommendations were 1, 10, 50, 100, 250 and 1000 for buckets 1, 2, 3, 4, 5 and 6, respectively. The purpose of this test is to stress the importance of the recommendations in the system. The use case simulated is described in 7.2. Instead of taking a custom problem pattern created by the user, the search of complex query workload was analyzed over the predefined problem patterns created by IBM experts. The system interacts over all the recommendations stored in the knowledge base and returns matched solutions to the user.

This experiment was repeated 5 times for each bucket and the average time (in hours:minutes:seconds) is reported. Figure 35 depicts the result in a graph. As can be seen, the time for OptImatch analyzes 1000 QEPs grows linearly with the number of recommendations in the knowledge base. This tool can search for 1000 recommendations against 1000 QEPs in approximately 3 hours and 35 minutes. In other words, OptImatch can perform 1 million searches in the given time. The results of this experiment shows that the proposed tool can be populated with recommendations by experts without losing performance.

Figure 35 – Number of Recommendations Experiment



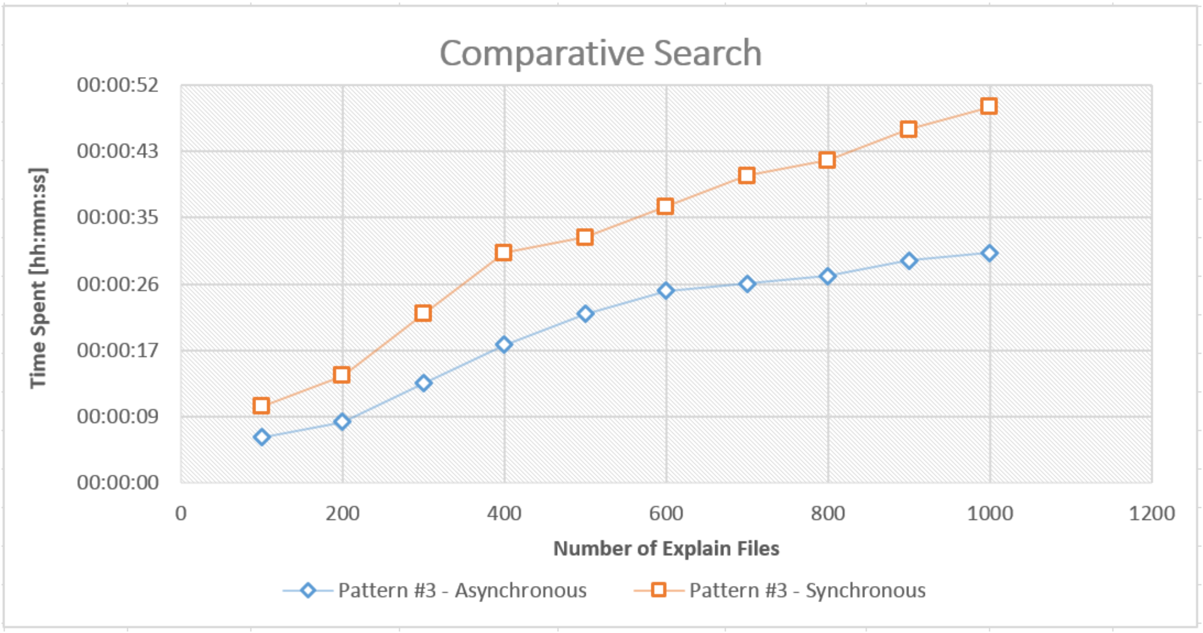
8.3 Comparative Search

The fourth experiment is a comparative study of the OptImatch's performance and involves comparing the asynchronous search versus the previous synchronous search. As explained in Section 7.3, now instead of performing a synchronous search, OptImatch performs an asynchronous search by dividing the search into buckets and sending them separately to the server. For this experiment pattern #3 (with recursion) was chosen and the same buckets from the experiment 8.2.1 were used.

The test was repeated 10 times and the average time (in seconds) is reported. Figure 36 depicts the results in a graph. Both search methods grow linearly with respect to the number of QEP files. However an expressive reduction of time to analysis can be seen in the asynchronous search. For example, to analyze 1000 QEPs, the synchronous took around 50 minutes to make the whole analysis. Asynchronously however, the search takes around 30 minutes. In other words, the asynchronous search reduced the time for search by nearly 40%.

Even more, another benefit of the new asynchronous search is that it divides the workload in smaller buckets to be analyzed by the server, so it is not just faster in analysis, it is also faster in returning the first result. In this example, the first result for the new search was returned in around 4 minutes (with 20 files), while the result for the old search (synchronous) was returned just after the full analysis (approximately 50 minutes).

Figure 36 – Comparative Search Experiment



8.4 Comparative User Study

The last experiment is a comparative study between the automatic search of this tool versus the manual search used nowadays. The purpose of this test is analyze the performance and also precision of the search. 100 different QEPs for each problem pattern were utilized for the current experiment. From the 100 QEPs, the number of files that matched the result were 15, 18 and 12 for the problem patterns A, B and C, respectively.

Three IBM experts were used for the test and the results reported are the average between them. Figure 37 represents the result of the OptImatch's automatic search versus the manual search.

Due the limited time that the experts could spend participating in this experiment, just 100 QEPs were utilized. Even for a small number of QEP files, OptImatch drastically reduces the time to search for a pattern. For searching for patterns #1 and #2 (without reduction), the manual search was around 27 minutes for #1 and 36 minutes for #2. OptImatch analyzed each one in 1 second. For the problem pattern with recursion (#3) the proposed tool reduced the time from 43 minutes to around 1 minute.

Furthermore, it is important to stress that the problem patterns used in the experiment are considered really small and without any complexity. With complex and big problem patterns the search time would be even more discrepant.

Also, beyond the performance a precision analysis was made for each of the patterns. As can be seen on Table 1, the precision for the manual search was not perfect (80% overall). Misinterpretation of information stored and formatting error while using search tools such as *grep* were common errors found in the test. For example, the value of a property inside the QEP could appear in decimal or in exponential mode, so for example, the value "0.001" could appear as "0.001" or "10e-3".

Figure 37 – Number of Recommendations Experiment

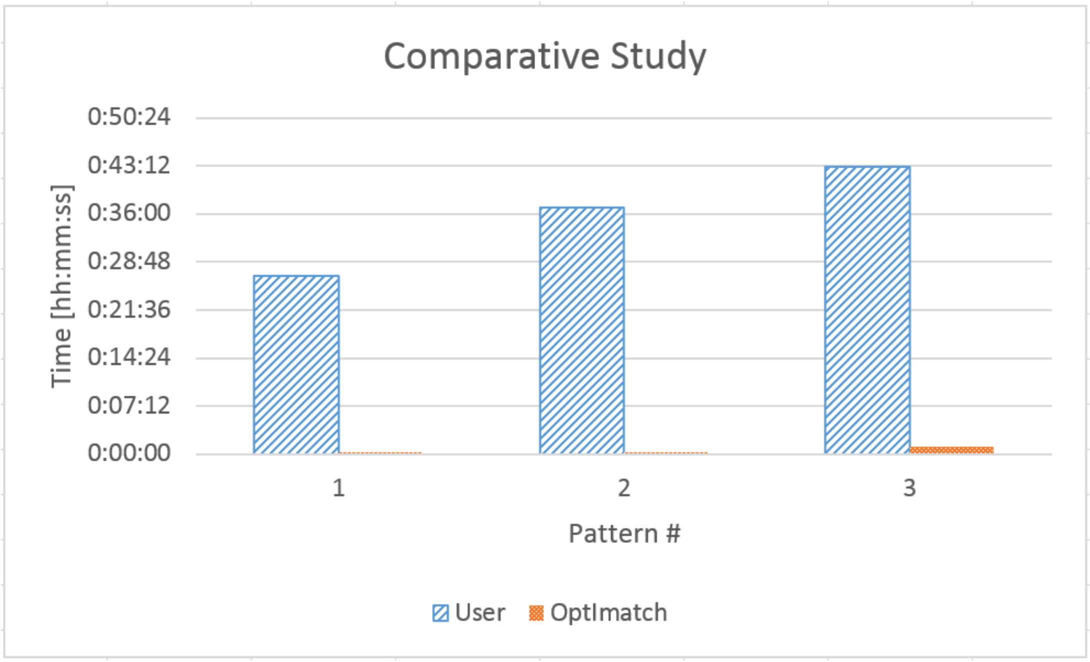


Table 1 – Precision for Manual Search

Pattern #	#1	#2	#3
Precision	88%	81%	71%

Even though IBM uses manual search nowadays, it has proven to be very time consuming and prone to human error. As OptImatch is fully automatic and it is unaffected by the different ways that a value of a property can be write, it provides 100% of precision. Even more, this tool can drastically decrease the time consumed to search problem patterns. Lastly, as OptImatch does not depend of any human intervention for searching, it can perform long searches without fatigue.

9 Conclusions

Create complex queries is not something trivial, as it needs expertise in both database and SQL like query languages. Database systems are become more sophisticated leading to systems that can completely automatically self-tune and adapt even to dynamic environments. To analyze performance issues DBAs use general problem determination tools. However, these tools need improvements as they lack of customization capacity. As a result, normally the process of searching is normally done manually or by the use of searching tools like *grep*.

The manual search for performance issues is done by analyzing QEPs. QEP files are complex and can have thousands of lines, requiring deep knowledge and expertise in optimizer as well as in SQL language. Even for DBAs sometimes is difficult and time consuming to determine the solution to a problem.

OptImatch provides an automatic search over a QEP workload, trying to find a custom problem pattern created by the user and recommendations stored in the knowledge base and provided by experts. The proposed tool makes use of RDF and SPARQL query languages to analyze QEPs in order to find query problem patterns.

Five experiments were presented in this work. The first one measured the performance of OptImatch in terms of searching for portions of QEPs against various size of QEP workload. The objective of this experiment was the analysis of how the number of files can interfere in the performance and whether the system is scalable in term of files. OptImatch showed a linear performance for this experiment, taking around 30 seconds to analyze 1000 QEPs, proving that the proposed tool can provide real-time analysis even for big QEP workloads.

The second experiment focused on the performance of OptImatch in terms of number of LOLEPOPs, where the bigger the number, the bigger the size of a QEP file. The objective was the verification of how efficient this tool can search over different number of LOLEPOPs. The performance proved to be linear for patterns without recursion.

The third experiment tested the performance of OptImatch in terms of number of recommendations analyzed against 1000 QEPs. The purpose was to understand how the number of recommendations can interfere in the OptImatch's performance. This tool proved to grow linearly with the number of recommendations in the knowledge base, analyzing 1000 recommendations over 1000 QEPs (meaning 1 million searches) in approximately 3 hours and 35 minutes.

The fourth experiment measured the performance of this tool comparing the new asynchronous search versus the previous synchronous search. The objective of this test was the analysis of the benefits of the new method. OptImatch performed 40% faster with this new search method.

The last experiment compared the automatic search of the proposed tool versus the manual search used nowadays at IBM. This experiment had as objective an analysis of the performance and the precision of this tool. OptImatch proved that it drastically reduced the time for search for problem patterns. Moreover, the manual search was prone to human error, with overall precision of 80%.

The experiments proved that OptImatch is scalable in terms of size of workload and number of LOLEPOPs. The performance of the proposed tool does not decrease as the size of the workload increases. OptImatch also proved to be able to be populated with recommendations by experts without losing performance, increasing linearly with it. Moreover, the new asynchronous search improved the performance of the tool, making it faster in around 40%. Furthermore, a comparative user study showed that the manual search of problem determination in QEPs was time-consuming and prone to human error. OptImatch drastically decreased the time to search and provided 100% precision.

OptImatch is proving to be valuable to IBM, as it can help in the IBM support of clients and database optimizer organization by providing solutions in faster way through the use of a knowledge base.

Even though this work was applied for query performance problem determination, the proposed methodology can be applied for any general software problem determination. For a future work, OptImatch can be expanded to cover other general software for problem determination. The proposed tool only requires the software to provide a structured diagnostic file that eventually requires further analysis by experts.

Bibliography

ALTON, E. et al. Recommending materialized views and indexes with ibm db2 design advisor. *Autonomic Computing, International Conference on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 00, p. 180–188, 2004.

CYGANIAK, R.; WOOD, D.; LANTHALER, M. *RDF 1.1 Concepts and Abstract Syntax*. 2014. Available at: <http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/>. Accessed: January 13th, 2017.

D3. *D3*: Data-driven documents. Available at: <https://d3js.org/>. Accessed: February 10th, 2017.

DAMASIO, G. et al. Optimatch: Semantic web system for query problem determination. In: IEEE INTERNATIONAL CONFERENCE ON DATA ENGINEERING, ICDE, 32., 2016, Helsinki, Finland. *Proceedings...* Helsinki, Finland, 2016. p. 1334–1337.

DAMASIO, G. et al. Query performance problem determination with knowledge base in semantic web system optimatch. In: INTERNATIONAL CONFERENCE ON EXTENDING DATABASE TECHNOLOGY, EDBT, 19., 2016, Bourdeaux, France. *Proceedings...* Bourdeaux, France: OpenProceedings.org, 2016. p. 515–526.

HARRIS, S.; SEABORNE, A. *SPARQL 1.1 Query Language*. 2013. Available at: <http://www.w3.org/TR/2013/REC-sparql11-query-20130321/>. Accessed: January 13th, 2017.

IBM. *IBM Knowledge Center*: The sql and xquery compiler process. Available at: <https://www.ibm.com/support/knowledgecenter/SSEPGG_10.5.0/com.ibm.db2.luw.admin.perf.doc/doc/c0005292.html>. Accessed: February 08th, 2017.

RAPH, K.; MARGY, R. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*. Third edition. [S.l.]: John Wiley & Sons, Inc., 2013.

ZILIO, D. C. et al. DB2 design advisor: Integrated automatic physical database design. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 30., 2004, Toronto, Canada. *Proceedings...* Toronto, Canada: Morgan Kaufmann, 2004. p. 1087–1097.