Inferencing Relational Database Tuning Actions with OnDBTuning Ontology

Luciana de Sá Silva Perciliano [1], Veronica dos Santos[1], Fernanda Baião [2], Edward Hermann Haeusler [1], Sérgio Lifschitz[1], Ana Carolina Almeida [3]

[1]Departamento de Informática, PUC-Rio
[2] Departamento de Engenharia Industrial, PUC-Rio
[3] Instituto de Matemática e Estatística, UERJ

{lperciliano, vdsantos, hermann,sergio}@inf.puc-rio.br

fbaiao@puc-rio.br, ana.almeida@ime.uerj.br

Abstract. *OnDBTuning is a relational database (automatic) tuning ontology. Ontologies are software artifacts that represent specific domain knowledge and can infer new knowledge. However, most cases involve only a formal and static description of concepts. Moreover, as database tuning involves many rules-of-thumb and black-box algorithms, it becomes challenging to describe these inference procedures. This research work first presents the OnDBTuning ontology solution focusing on the inference of tuning actions. Next, we provide an actual implementation using SPARQL Inferencing Notation (SPIN). Finally, we discuss a practical evaluation for index recommendation.*

1. Introduction

Database tuning actions are essential to improve applications performance by reducing database response time and increasing throughput. Database tuning involves many rules-of-thumb and black-box algorithms empirically defined towards specific Relational Database Management Systems (RDBMSs) vendors and particular releases. To mention a frequent and straightforward situation, a DBA should check if the referenced columns (from the WHERE clause of a SQL statement) are indexed, and, if not, the DBA may create the respective indexes to avoid a full table scan [Shasha and Bonnet 2002].

It is possible to find a set of tuning heuristics followed by Database Administrators (DBAs) using their own scripts or the available (either proprietary or open-source) tuning tools. These heuristics encompass decisions upon creating and maintaining access structures such as indexes and materialized views or even data replication, partitioning, query rewriting strategies, among others.

This tuning knowledge may be modeled using ontologies, which - from a computational perspective - are artifacts capable of formally and explicitly representing a shared conceptualization from a domain of discourse [Gruber 1993]. Definition of classes (concepts), relations (object properties), attributes (data properties), and instances represents the universe of discourse. Such knowledge representation makes knowledge-based systems able to infer new knowledge through semantic reasoning. However, most cases involve only a formal and static description of concepts.

This paper proposes OnDBTuning, an ontology that represents knowledge about the relational databases (automatic) tuning domain. It enables semantic reasoning for

performance improvements within a database system, focusing on the inference and suggestion of possible tuning actions. Our solution considers SPARQL Inference Notation (SPIN) for defining heuristic rules. We provide an actual implementation and practical evaluation on top of PostgreSQL DBMS.

The semantic support for the implementation of tuning heuristics provides DBAs with a transparent methodology for tuning databases [de Almeida 2013, Almeida et al. 2019]. Since heuristics may automate the user's assistance in achieving a goal, OnDBTuning heuristics help DBAs in their task to obtain successful results in most cases with the advantage of being aware of the rationale followed by the ontology.

The remainder of this paper is organized as follows: in Section 2 we present theoretical background from database tuning domain and ontologies, including related works. Section 3 describes the OnDBTuning ontology focusing on the inference of tuning actions for index recommendation and present (Section 4) evaluation results considering a subset of TPC-H queries. Section 5 concludes and discusses some current and future works.

2. Background

Database tuning is an activity that aims to reduce transactions response time and increase database throughput. Since database performance is crucial for applications, tuning has become essential. Typically, it is composed of a set of actions proposed by a specialist, also known as a database administrator (DBA), based on previous experience or support of advisor tools, after monitoring the system looking for contentions and bottlenecks [Shasha and Bonnet 2002].

Tuning comprises adjustments at the database level of configurations, parameters, and physical design to promote efficient use of existing resources without changing physical machine components. It can be performed automatically (also called database self-tuning) or manually. One way to adjust the physical design is to create an auxiliary access structure such as an index.

Indexes are access structures that speed up data retrieval [Shasha and Bonnet 2002]. The database optimizer can use the index to fetch only the rows that satisfy range or lookup filter conditions of SQL statements avoiding full table scan. These can be classified as real (actual) or hypothetical. Real indexes are indexes that physically exist in a database. In contrast, hypothetical indexes are candidate indexes that exist only in the database's metadata with no physical space consumption [de Almeida 2013]. What-if analysis, implemented by some RDBMS, simulates how query execution plans benefit from indexing based on hypothetical indexes.

(Self)Tuning techniques for relational database systems have been studied for decades, particularly for index selection These strategies are modeled as heuristics to explicitly and precisely define DBA's knowledge. We may cite the Automatic Index Reconstruction Heuristic (HRAI) [Morelli et al. 2012], which has rules that suggest index automatic elimination or reconstruction. Other works [Valentin et al. 2000] proposed heuristics for choosing candidate indexes that best match each SQL statement of a given workload.

Ontologies, RDF and SPIN

An ontology is represented as a tuple $O = \langle C, R, I, A \rangle$, where: i) C represents the set of concepts of a domain; ii) R is the set of relationships - or associations - between these concepts; iii) I is the set of instances of classes and iv) A is a set of domain axioms used to model restrictions and rules on the objects of the domain [Staab and Studer 2010]. These constructors can be modeled using RDF/RDF-S and OWL languages.

RDF[1] is a data model specified by the W3C as a directed, labeled graph data format and is at its core a collection of triples. RDF-S (or RDF-Schema) is a specification that allows describing ontologies through the definition of classes, properties, and their relationships using few elements. The OWL language (Ontology Web Language) is a language developed based on the RDF/RDF-S specifications and recommended by the W3C. It is a widely used language for formally representing ontologies.

The Modeling languages mentioned above are useful to model classes, properties, and relationships between these concepts, i.e., they are used to model the static structure of knowledge, including axiomatic definitions of data structures. Other languages are necessary to describe objects behavior and inferencing. Inferencing can be defined as the ability to create new knowledge, that is, to generate new triples, based on the current knowledge (existing triples) using a Rule Engine. SWRL[2] cover this scope through combining OWL with horn logic rules. SWRL has been a popular choice for rule-based systems built on top of ontologies. It is supported by widespread tools such as ontology editors, rule engines, and ontology reasoners. However, it has never become a W3C recommendation [Bassiliades 2020].

SPARQL[3] is a declarative language (a SQL-like style) designed to manipulate RDF triples. The query language is primarily intended for pattern (subgraph) matching rather than path traversal. SPARQL supports four query types for (sub)graph pattern matching: SELECT (project out specific variables and expressions), CONSTRUCT (construct RDF triples/graphs), ASK (ask whether or not there are any matches, the result is either "true" or "false".), and DESCRIBE (describe the resources matched by the given variables retrieving basic node/edge adjacency). Particularly, the CONSTRUCT query form can be used to map subject, predicates, and objects of stored triples to domain ontology's classes, relations, and attributes or infer new triples. SPARQL is the standard language to query graph data represented as RDF triples. It is one of the three core standards of the Semantic Web. The other two are OWL and RDF itself.

Almost all programming languages have an API to SPARQL to operate queries on RDF triple stores (3Stores). There are also specialized 3Stores, AllegroGraph[4], for example, that perform better than these APIs. They have state-of-the-art inference engines. Among these expert ontology inference engines, we can cite SPIN. The main idea behind the design of SPIN is to relate ontology classes with queries in SPARQL such that we obtain constraints and rules formalizing the class members' behavior. As such that, SPIN can define inference rules (from that infer that) on top of SPARQL queries. Hence, SPIN is the natural choice to be the inference engine we need to express in a rule-based way

[1] https://www.w3.org/TR/rdf11-concepts/
[2] https://www.w3.org/Submission/SWRL/
[3] https://www.w3.org/TR/sparql11-query/
[4] https://allegrograph.com/

the OnDBtunning represented knowledge. In the sequel, we provide a brief explanation of SPIN.

According to [Bassiliades 2020], SPIN is a de-facto industry standard to represent rules and constraints on the Semantic Web, built based on the SPARQL query language, commonly used for querying and processing Linked Open Data. SPIN specification[5] defines a vocabulary as a collection of RDF properties to represent SPARQL queries as RDF triples. SPIN Vocabulary includes constructors such as spin:rule, spin:constraint, and spin:constructor, enabling queries, restrictions, and rules definition and storage attached to classes. For example, spin:constructor can be used by modelers to automatically record instance creation timestamp as a property value through a SPARQL CONSTRUCT statement. A spin:constraint using a SPARQL ASK statement allows to specify conditions that all class instances and their subclasses must fulfill.

An inference rule can be defined through the use of SPARQL CONSTRUCT in the spin:rule, where the WHERE clause defines the sub-graph conditions (IF) to be satisfied (TRUE) and the CONSTRUCT builds the new triples (THEN). In Figures 1a and Figure 1b we can see an example of an inference rule that creates a new triple of type *tuning:QueryStatement* (Figure 1a - line 1 to 3) if what is defined in the WHERE (Figure 1a - line 4 to 8) is true. Figure 1b shows the RDF representation of the SPARQL CONSTRUCT statement shown in figure 1a.

(a) SPARQL Statement **(b)** RDF representation

Figure 1. spin:rule Example

Some works found in the literature made use of ontology reasoning to infer new knowledge. Among them, we can cite [Doulaverakis et al. 2016], from the medical field, used an Ontology of Clinical Practice Guidelines (CPGs) and rules in SPIN. Another

[5]https://www.w3.org/Submission/spin-modeling/

work is [Promkot et al. 2019], which uses a traditional herbal medicine ontology to infer, through SWRL rules, recommendations for traditional herbal medicines in a similar way as a specialist would do. With regard to machine learning tuning approaches, proposals [Aken et al. 2021] and [Zhang et al. 2021] aim at efficiency in parameter adjustments without explaining how to achieve this goal. In this research work, we investigate how an ontology like OnDBTuning may be helpful with a reasoning process and rule-based approach that suggests possible tuning actions for the DBAs with their explainability.

3. OnDBTuning Inferencing

OnDBTuning[6], the Database Tuning Ontology, was initially proposed in [de Almeida 2013] and defines concepts, properties, and relationships used in the database tuning domain. The current research work brings a declarative and well-founded solution, as originally suggested, to add semantics to self-(or autonomic) tuning activities. Figure 2 illustrates a fragment of the OnDBTuning. Among these concepts and relationships, we can highlight *tuning:DMLStatement* that represents any data manipulation language (DML) statement that access and manipulate data from tables of a relational database. Every DML statement from the input workload corresponds to an asserted instance of this concept. An instance of query type, that is, if the DML statement begins with "SELECT" (figure 1a - line 7), is inferred as *tuning:QueryStatement* (figure 1a - line 2). Other concepts will be instantiated through parsing the DML statement such as *tuning:WhereClause* (figure 4), which contains the components that constitute the WHERE clause and *tuning:ReferencedTable* that relates the DML statement with the accessed tables from the physical schema through its FROM clause.

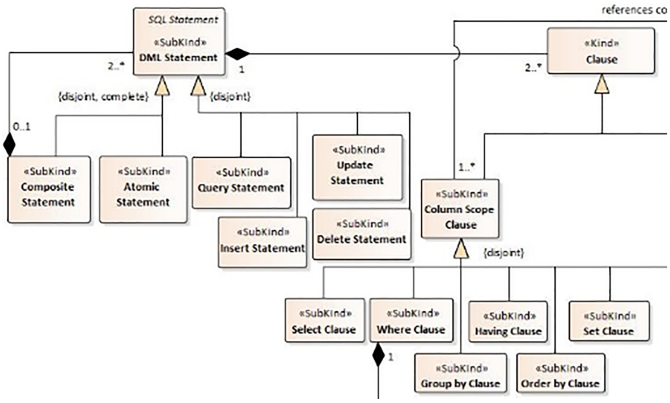


Figure 2. Fragment of OnDBTuning

Figure 3 shows a state diagram representing the states and events of the instantiation process in OnDBTuning. In the initial state, the individuals in the ontology are only those representing the classes and properties of the tuning domain. In this article, we used the prefix *tuning* to identify classes, object properties, data properties, and instances of the OnDBTuning ontology.

When OnDBTuning receives the workload (receive workload) composed of DML statements, schema metadata, and optimizer statistics, new triples are created (asserted)

[6]https://www.ime.uerj.br/ondbtuning/

as individuals of ontology's classes with their respective properties. After that, the rule engine executes the ontology rules, and new triples are created (inferred). Some inferred instances are the result of the SQL parse rules and others are the result of the tuning rules, as described afterwards. The last ones contain suggestions for tuning actions. The instantiation process repeats for every new workload received, accumulating the acquired knowledge (asserted and inferred) from previous workloads for the same database instance.

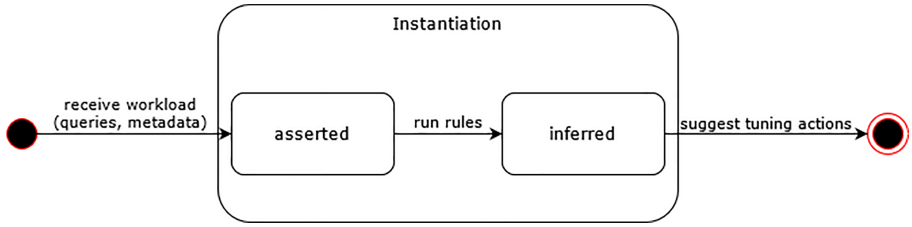


Figure 3. Ontology Instantiation State Diagram

Initially, OnDBTuning's inference rules were designed in SWRL [Almeida et al. 2019]. However, we decided to use SPIN to implement the tuning rules due to SWRL technological and expressiveness restrictions. First, a technological limitation was found using Protégé[7] and the SWRLTab[8]: we cannot use the *OR* operator with SWRL, and there was also a limitation regarding the rule sizes. Therefore, we tried to use several rules, one for each type of clauses combination [de Almeida 2013] in a DML query, which makes rule creation and maintenance very complex. Moreover, SPARQL could be more expressive than SWRL since it offers operators such as FILTER, OPTIONAL, UNION, and the possibility of defining new functions and templates in SPIN, besides spin built-in functions. Another feature is related to the performance of the rule engines: with SPIN, the rules are checked when new instances or changes occur, while in SWRL we need to check all the rules at each execution [Bassiliades 2020].

SQL Parsing Rules

Abstract syntax trees (AST) is a structured (tree) representation from the code of any language with grammar. The SPIN vocabulary enables SPARQL statements' ASTs representation in RDF. SQL queries can also be represented by their ASTs. Thus, a database workload can be represented by an RDF graph, where nodes are resources typed with language's abstract classes, whose syntactic structures can be annotated with additional metadata.

The DML statements from input workloads are submitted to a parsing process since most tuning heuristics are highly dependent on the queries' structure (preconditions). This process used to split SQL into indivisible nodes was supported by 06 (six) spin:rules. We have considered the CONSTRUCT command to infer new triples, categorize their nodes and create relationships based on OnDBTuning classes and properties.

Figure 4 shows the result of a parse rule for the *tuning:WhereClause*: one triple with the data property correspondent to its content (*tuning:hasClauseDescription*), five

[7] https://protege.stanford.edu/
[8] https://github.com/protegeproject/swrltab

triples linking to the referenced columns (*tuning:clauseReferencesColumn*) and the triple that links with the input workload DML (*tuning:componentOfDMLSt*).

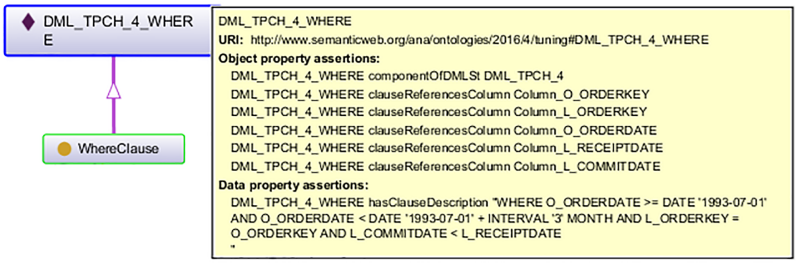
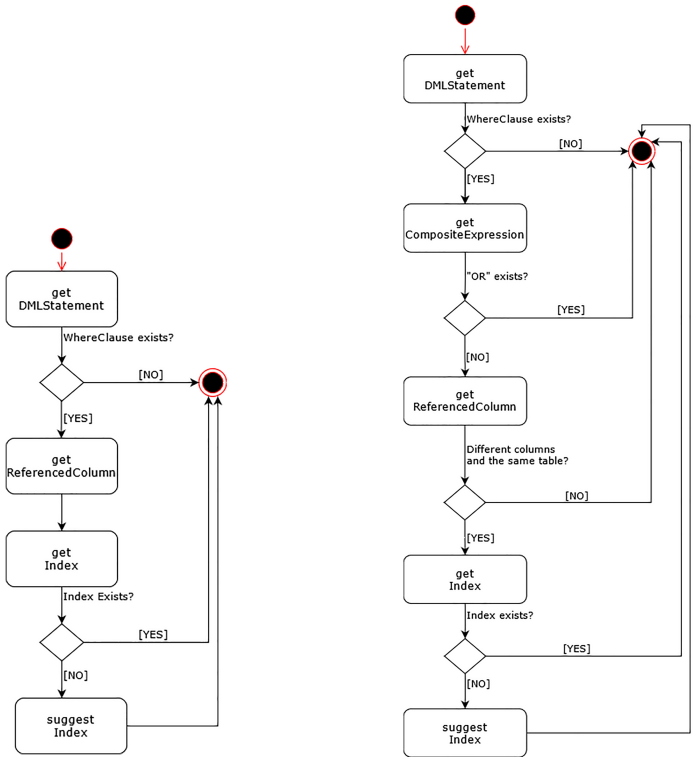


Figure 4. Parsing of the WhereClause

Instantiating ontology concepts through rules attached to their classes rather than other approach makes any reasoning explicit and consistent. It also makes the ontology self-contained and with high cohesion, reducing external dependency and increasing reuse. In the presence of new DB tuning heuristics that rely on a concept not previously defined, it is quite straightforward to extend the ontology with the classes, properties, and their respective parse rules.

Modeling Tuning Rules

Activity Diagrams of figures 5a and 5b demonstrate the rationale behind the rules used to suggest (a) simple and (b) composite indexes. Each one is more detailed as follows.



(a) Simple **(b)** Composite

Figure 5. Activity Diagrams

For the simple (non composite) index rule (Figure 5a): (i) **get DMLStatement**: for each query instance, check if there is a "WHERE" clause; (ii) **get ReferencedColumn**: select each column referenced in its "WHERE" clause; (iii) **get Index**: check if this column has an (physical) index; and (iv) **suggest Index**: suggest an index based on this column.

For composite indexes (Figure 5b): (i) **get DMLStatement**: for each query instance, check if there is a "WHERE" clause; (ii) **get CompositeExpression**: for each Composite expression of the "WHERE" clause it checks if it has the logical connector "OR"; (iii) **get ReferencedColumn**: select a pair of columns referenced in its "WHERE" clause and checks if they are different columns and belong to the same table; (iv) **get Index**: check if this pair of columns has an (physical) index; and (v) **suggest Index**: suggest a index based on this pair of columns.

4. Evaluation

SPIN rules were added to OnDBTuning using the TopBraid Composer Free Edition[9] tool. Topbraid's front end enables users to define rules using a GUI interface rather than writing their SPARQL rules using RDF syntax. This is one of the most used tools for creating Ontologies, besides Protégé, OWLGrED[10] and SWOOP[11] [Suganya and Porkodi 2018]. To infer new individuals, we used the TopSPIN SPIN rule engine.

Tuning rules named *tuning:RuleHypCompositeIndex* and *tuning:RuleHypSimpleIndex* are instances of *tuning:HeuristicDefinitionRule* and were implemented as spin:rules attached to the *tuning:DMLStatement* concept. Using the property *tuning:isGeneratedBy* of the triple generated by each rule, the user can identify and even filter specific rule suggestions.

For each suggested hypothetical index, we calculate its bonus based on equation 1:

$$bonus = \frac{count(hypIndex \rightarrow queries)}{count(hypIndex \rightarrow column \leftarrow table \leftarrow queries)} \quad (1)$$

A *bonus* refers to the total number of queries that originate the suggested hypothetical index divided by the sum of queries that references the corresponding table. The underlying idea is that the closer the bonus value is to 1, the better. It indicates that it can be helpful in most queries present in the accumulated workload. Above a user-defined threshold, this bonus can show hypothetical index candidates to become actual indexes.

Indeed, for each real index, its usefulness is calculated, dividing the total *tuning:WhereClause*, for the involved columns, by the total *tuning:DMLStatment* that references its table. The closer the result is to 1, the better, as it indicates that the real index can be useful for most DMLs from the accumulated workload. Values close to 0 suggest index candidates to be dropped, as they require maintenance and are not being used.

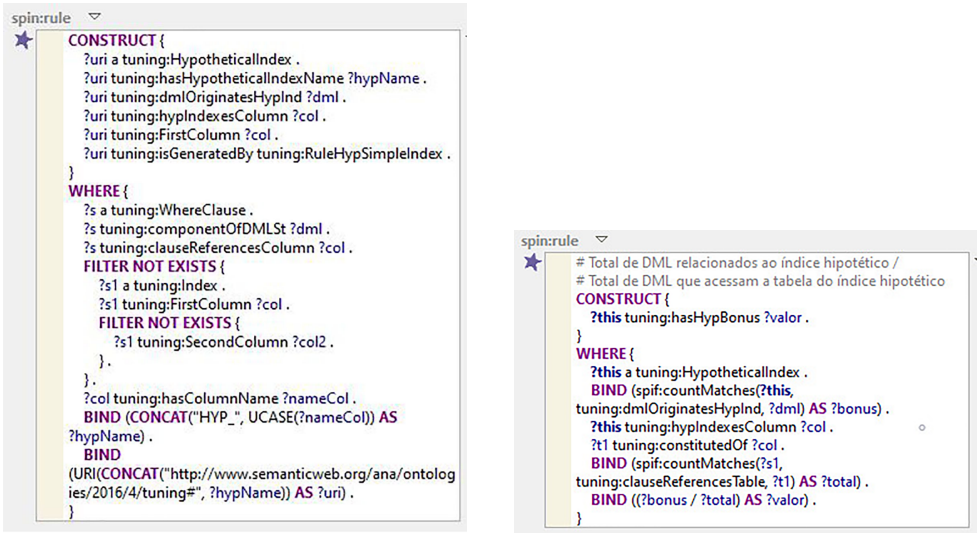
We consider SPIN built-in functions in conjunction with SPARQL in the implementation of parsing and tuning rules. In the parser process, the *spif:split* function allows the division of some strings into substrings. In the tuning rule called *tuning:RuleHypSimpleIndex* (Figure 6a), for each referenced column in the where clause

[9] https://www.topquadrant.com/

[10] http://owlgred.lumii.lv/

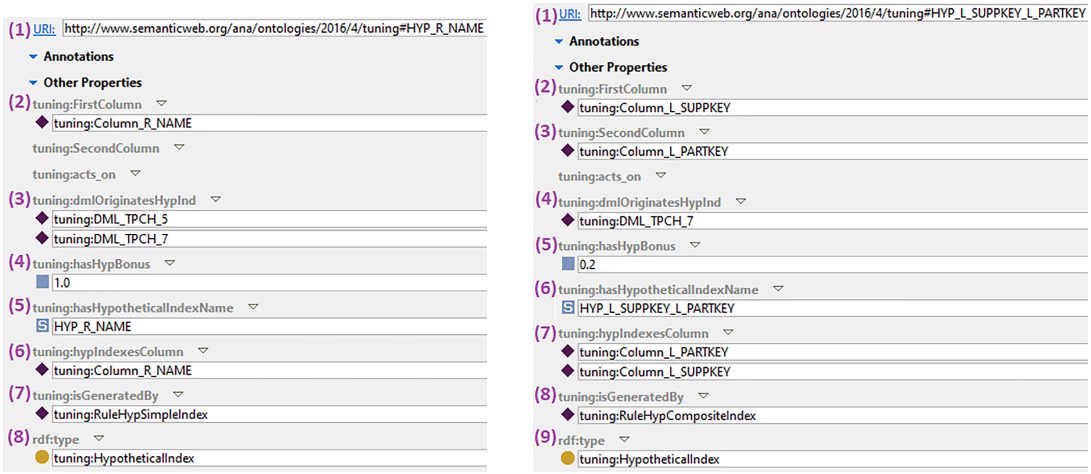
[11] https://www.w3.org/2001/sw/wiki/SWOOP

(*tuning:WhereClause*) of a SQL statement, it was checked if there are indexes for that column (FILTER NOT EXISTS). For bonus calculation, the function *spif:countMatches* is called, which sums the occurrences of a triple, according to the rule (see Figure 6b).



(a) RuleHypSimpleIndex **(b)** RuleBonus

Figure 6. Tuning Rules



(a) Inference RuleHypSimpleIndex **(b)** Inference RuleHypCompositeIndex

Figure 7. Hypothetical Index suggestion

For this experiment, we used the well-known TPC-H benchmark[12], which targets analytical workload (OLAP). We used a subset of its schema, consisting of six tables and 28 columns. Real indexes initially exist for 6 primary keys and 6 foreign keys.

First, the schema metadata and 5 queries (Q01, Q03, Q04, Q05, and Q07) composed the workload and instantiated OnDBTuning. For this workload, 38 hypothetical indexes are suggested: 9 by the *tuning:RuleHypSimpleIndex* rule and 29 two-column

[12] `http://www.tpc.org/tpch/`

indexes by the *tuning:RuleHypCompositeIndex* rule. All bonus values are calculated (*tuning:hasHypBonus*) for each hypothetical index, as well as real indexes utility (*tuning:hasIndexUsefulness*). See Figure 7 for one example of each type.

Figure 7a shows an inferred hypothetical index identified by its URI (line 1) and described by its properties. Line 3 shows *tuning:DML_TPCH_7* and *tuning:DML_TPCH_5* as the names of the benefited queries (*tuning:dmlOriginatesHypInd*) which are instances of *tuning:QueryStatement*. Line 4 indicates its bonus value as 1.0 (*tuning:hasHypBonus*), line 5 *HYP_R_NAME* as hypothetical index name (*tuning:hasHypotheticalIndexName*), line 6 *tuning:Column_R_NAME* that relates to the referenced column (*tuning:HypIndexesColumn*) named *R_NAME* (*tuning:hasColumnName*), and at line 7 *tuning:RuleHypSimpleIndex* indicates the tuning rule (*tuning:isGeneratedBy*) that suggested this hypothetical index (*rdf:type tuning:HypotheticalIndex*).

In Figure 7b, identified by its URI (line 1), lines 2-3 describe the properties content about the first (*tuning:FirstColumn*) and the second (*tuning:SecondColumn*) column analyzed to generate the index: *tuning:Column_L_SUPPKEY* and *tuning:Column_L_PARTKEY*, respectively. Line 4 presents the input query that generates the hypothetical index, named *tuning:DML_TPCH_7* (*tuning:dmlOriginatesHypInd*). Line 5 has the bonus value (*tuning:hasHypBonus*), its name is shown at line 6: *HYP_L_SUPPKEY_L_PARTKEY* (*tuning:hasHypotheticalIndexName*) and indexed as in line 7: *tuning:Column_L_SUPPKEY* and *tuning:Column_L_PARTKEY*(*tuning:HypIndexesColumn*). This composite hypothetical index was generated by *tuning:RuleHypCompositeIndex* (*tuning:isGeneratedBy*) rule - see line 8. TopSPIN only inferred one composite hypothetical index for these pair of columns since there already exist a real index for the same pair of columns in reverse order.

After the first workload, the resulting triples are persisted at OnDBTuning, and a second workload, with the TPC-H Q02 query, was added. The rule engine runs once more, and both bonus (*tuning:hasHypBonus*) and utility (*tuning:hasIndexUsefulness*) are calculated, considering the accumulated workload. A new triple with the respective properties for hypothetical and real indexes, were added where necessary.

In Figure 8 we can observe that bonus values of some hypothetical indexes have decreased, increased, or neither. For instance, *tuning:HYP_C_NATIONKEY*, previously 0.66666667, dropped to 0.5. This result showed that there was a decrease in the number of queries benefiting from this hypothetical index.

The *tuning:LINEITEM_L_PARTKEY_FKEY* index has utility value of 0.2, Figure 9, and the second workload value decreased to 0.1666667, evidencing that it has low potential to increase the performance of the workload queries. It is essential to mention that, in the second workload, new triples were created only for those indexes that had usefulness value variation due to the new query.

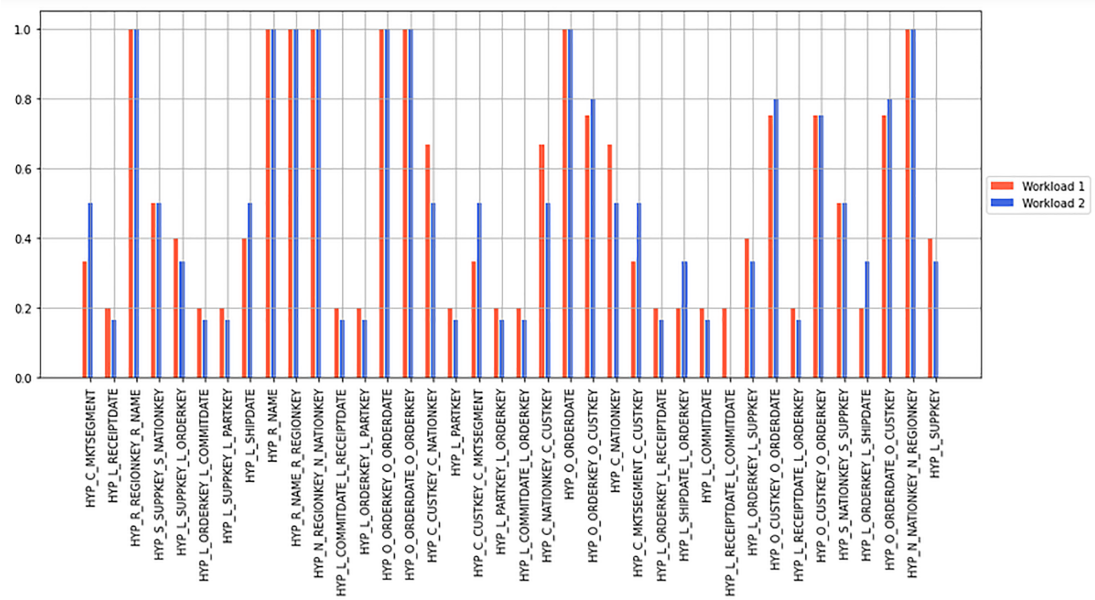


Figure 8. Hypothetical Index Bonus

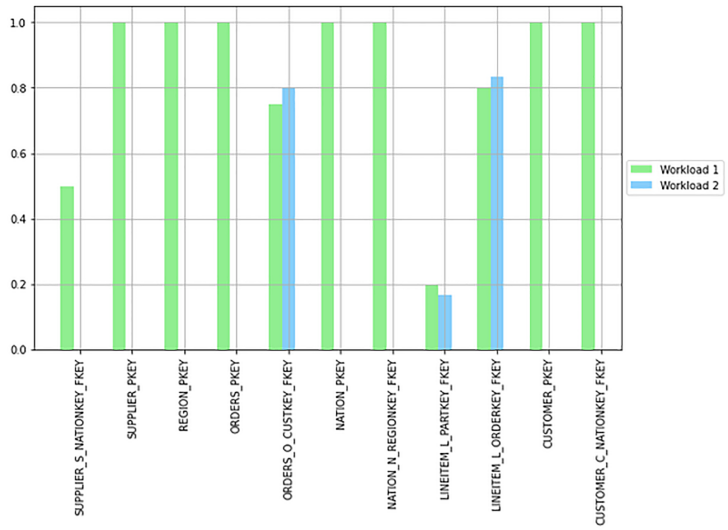


Figure 9. Real Index Usefulness

5. Conclusions

In this research work we designed the inference of tuning suggestions based on index heuristics. Two rules, *tuning:RuleHypSimpleIndex* and *tuning:RuleHypCompositeIndex*, were implemented in the OnDBTuning ontology, and index hypothetical suggestions were evaluated using TPC-H workload. The inferred suggestions supports a DBA in his tuning actions with rationale explanations that black-boxes tools do not provide.

As future work we plan to extend OnDBTuning by refining index heuristics with optimizer statistics, modelling and implementing others heuristics about partial indexes, materialized views and partitions strategies as well as new database tuning concepts, properties and rules concerning NoSQL databases. In addition, OnDBTuning may also be

extended to capture concepts defined in $Tun - O_{CM}$, a conceptual model that supports database tuning decision making, CM-OPL methodology and its underlying language, incorporating heuristics versioning [Almeida et al. 2021]. We will also integrate the ontology and a rule engine with a tool that captures RDBMS workload and executes the inferred tuning suggestions for a complete evaluation of the database performance.

References

Aken, D. V., Yang, D., Brillard, S., Fiorino, A., Zhang, B., Billian, C., and Pavlo, A. (2021). An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. *PVLDB*, pages 1241–1253.

Almeida, A. C., Baião, F. A., Lifschitz, S., Schwabe, D., and Campos, M. L. M. (2021). Tun-o$_{\text{cm}}$: A model-driven approach to support database tuning decision making. *Decision Support Systems*, page 113538.

Almeida, A. C., Campos, M. L. M., Baião, F. A., Lifschitz, S., de Oliveira, R. P., and Schwabe, D. (2019). An ontological perspective for database tuning heuristics. In *Int. Conf. on Conceptual Modeling (ER)*, pages 240–254. Springer.

Bassiliades, N. (2020). A tool for transforming semantic web rule language to SPARQL infererecing notation. *Intl. Journal Semantic Web Information Systems*, pages 87–115.

de Almeida, A. C. B. (2013). *Framework para apoiar a sintonia fina de banco de dados (in portuguese)*. PhD thesis, PUC-Rio.

Doulaverakis, C., Koutkias, V., Antoniou, G., and Kompatsiaris, I. (2016). Applying sparql-based inference and ontologies for modelling and execution of clinical practice guidelines: a case study on hypertension management. In *Knowledge Representation for Health Care*, pages 90–107. Springer.

Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowledge Acquisition*, pages 199–220.

Morelli, E. T., Almeida, A. C., Lifschitz, S., Monteiro, J. M., and Machado, J. C. (2012). Autonomous re-indexing. In *Symp. on Applied Computing*, pages 893–897. ACM.

Promkot, A.-n., Arch-int, S., and Arch-int, N. (2019). The personalized traditional medicine recommendation system using ontology and rule inference approach. In *4th Intl. Conf. on Computer and Communication Systems*, pages 96–104. IEEE.

Shasha, D. E. and Bonnet, P. (2002). *Database Tuning - Principles, Experiments, and Troubleshooting Techniques*. Elsevier.

Staab, S. and Studer, R. (2010). *Handbook on ontologies*. Springer Sci & Bus. Media.

Suganya, G. and Porkodi, R. (2018). Ontology based information extraction-a review. In *Intl. Conf. on Current Trends towards Converging Technologies*, pages 1–7. IEEE.

Valentin, G., Zuliani, M., Zilio, D. C., Lohman, G., and Skelley, A. (2000). Db2 advisor: an optimizer smart enough to recommend its own indexes. In *16th Intl. Conf. on Data Engineering*, pages 101–110. IEEE Computer Society.

Zhang, J., Zhou, K., Li, G., Liu, Y., Xie, M., Cheng, B., and Xing, J. (2021). Cdbtune+: An efficient deep reinforcement learning-based automatic cloud database tuning system. *VLDB*, pages 1–29.