Challenges of Index Recommendation for Databases

[With Specific Evaluation on a NoSQL Database]

Parinaz Ameri
Karlsruhe Institute of Technology (KIT)
Hermann-von-Helmholtz-Platz 1, Bldg. 449
76344 Eggenstein-Leopoldshafen, Germany
parinaz.ameri@kit.edu

ABSTRACT

One important aspect of physical database design is the selection of a proper set of indexes for a workload. Creation of indexes in a database system is subject to storage constraints. It is also affected by the ratio of update operations in the workload. Therefore, the cost and benefit of each set of indexes should be evaluated by a proper optimization method. The large number of index sets that must be assessed and the iterative nature of such optimization methods impose an additional load on the database system. Therefore, an efficient algorithm is needed to develop a practical framework for automating index recommendation. Furthermore, due to the fundamental differences between data models and query languages of NoSQL databases to each other and the relational databases, evaluation of such index recommendation system for NoSQL databases faces many challenges. This paper reviews the challenges and my proposed solutions for developing a self-tuning index recommendation system, especially for a document-based NoSQL database instance.

General Terms

Index Recommendation

Keywords

Indexing, Query Optimizer, NoSQL Databases, Optimization, Representative Sampling, Synthetic Workload, Generation, Benchmark

1. INTRODUCTION

The performance of a database is dependent on its physical design. A crucial part of the physical design is the selection of the proper set of indexes concerning a particular workload. Given the large size of conventional in-production databases and the heavy workload of queries on them, automating the process of choosing proper indexes for them is necessary.

28th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 24.05.2016 - 27.05.2016, Nörten-Hardenberg, Germany.
Copyright is held by the author/owner(s).

The index recommendation problem is defined as the course of determining a subset of indexes so that the benefit of creating them for a particular workload is maximized concerning the storage capacity.

Each database normally has a query optimizer that executes the cost of running a query with different scenarios based on available indexes in the database. Each query optimizer itself has a cost function to evaluate different scenarios. The recent trend in developing index recommendation solution is to benefit from this cost function. This approach eliminates the risk of developing an entirely separated cost function for the index recommendation system that would recommend indexes which might not even be considered by the optimizer. The estimated cost by the optimizer can directly be used in benefit function of the optimization method.

On the other hand, utilizing the estimated cost of the query optimizer requires provoking it by the index recommendation system for all of the candidate indexes. A large number of calls to the optimizer can affect the performance of the database itself in response to its applications. Therefore, a good framework design is needed to avoid putting lots of overhead on the in-production database and having better performance for the index recommendation system. To prevent this problem, we introduce a virtual environment consisting of a sample of the targeted database. The candidate indexes can be estimated in this environment rather than in the in-production system itself.

The number of candidate indexes can grow drastically in proportion to the size of the database. Consecutively, a method is needed to eliminate candidate indexes considered by the recommendation system without removing the most relevant indexes. Our proposed solution is discussed more in Section 3.

Furthermore, the performance of the index recommendation system needs to be tested by a series of workloads. This evaluation on NoSQL databases runs to a lack of well-defined benchmarks. The reason is the various data model of NoSQL databases and the vast variety of their query languages. These differences result in the ineffectiveness of well-known benchmarking models of traditional relational database models for NoSQL databases. This issue and our developed solution for this challenge is discussed more in Section 5.

The rest of this paper is organized as following: some related work are surveyed in Section 2. In Section 3, some of the practical challenges in developing a framework for a self-tuning index recommendation system are explained. Addi-

tionally, some of our solutions for these difficulties and the proposed architecture for such a framework are presented. The Section 4 provides a mathematical definition of index recommendation problem followed by a discussion about possible optimization methods to solve the problem. The Section 5 provides a brief discussion on difficulties of evaluating such database related systems for NoSQL databases and a solution that we developed for this problem. At the end, Section 6 concludes the work.

2. RELATED WORK

There are several optimization methods to solve this issue. Some recent approaches [7] neglect storage limitations and instead calculate a lower bound for the cost of a workload based on each query's individual optimal index. Despite being advantageous, the derived bounds are not pragmatic in a context of the real storage limitations.

In the literature, many different optimization methods are used to find the optimal solution for this problem such as Knapsack problem usage [16], genetic algorithm [12] or even linear programming optimization techniques [8] and branch-and-bound [18]. However, this problem is often solved by using a greedy algorithm [6, 9, 2]. In order to estimate better how close we get to the optimal solution, other solutions such as Integer Linear Program (ILP) can be used [14].

Instead, we propose using ILP, because it not only enables us to explore more cases than for example the mostly used greedy algorithm, but it also allows us to evaluate the quality of the optimal solution. Also, by applying Linear Programming relaxation, we can have useful information about approximate solutions that had optimal performance, but due to lack of storage space are not chosen.

3. FRAMEWORK DESIGN AND ITS CHALLENGES

The practical challenges of developing a framework for recommending indexes and our solutions are presented in this section.

We refer to the set of indexes chosen for cost evaluation as *candidate indexes*. The first challenge in designing a good framework for recommending indexes is limiting the search space for candidate indexes.

On modern databases, there are not only traditional ascending and descending index types, but also many new index types, e.g. spatial, text indexes, etc. Consider there are s types of indexes in a database with n attributes in a collection. The following equation gives number of possible single and multi-attribute indexes on that collection:

$$\sum_{k=1}^{n} \frac{(s^k)n!}{(n-k)!} \tag{1}$$

where k is number of fields and $k \leq n$ [5]. On a collection of only five attributes and the possibility to create four types of indexes, the number of possible indexes exceeds 150,000.

Therefore, it is important to limit the search space for candidate indexes to the most relevant ones. As presented in [5], we consider the most relevant indexes as the ones derived from attributes of the most frequent queries in the workload. Accordingly, the search space reduces from all the possible combinations of the whole attributes in the data set to the single and repeatedly present combinations of the frequent queries in the workload. The *Frequent Itemset* [1] as a data mining algorithm is used to build single and compound indexes related to the most frequent queries of the workload.

However, as described in [3], the frequency of queries should not be the only determining factor for defining candidate index proposal. For example, consider a query that user wishes to issue frequently, but the response time of it on a large database takes very long time. Due to the long response time, its frequency might fall below the configured threshold of the frequent itemset. To avoid missing such important queries for candidate index evaluation, a proper combinatorial algorithm is needed to combine the frequency and length of a query as two determining factors for generating candidate indexes.

Extracting attributes from the most frequent queries and setting order for them in a compound index should be done automatically. This is the responsibility of *Syntax Analyzer* in Figure 1. This component is crucial for analyzing queries and also for scalability of the system to work with large workloads. The functionality of the syntax analyzer component is directly affected by different query languages and data models of NoSQL and relational databases.

The mostly denormalized structure of NoSQL databases allows each attribute to contain non-scalar values, e.g. arrays and nested documents in document-based databases. Such data models eliminate usage of join operations while querying the data. Consecutively, most NoSQL databases developed their query language (normally an object-oriented one) instead of using SQL queries.

The query for different data models influences the access path to the data. Successively, the query optimizer plans are affected. Therefore, while developing the syntax analyzer to gather the attributes and order them after each other, the order that the query optimizer of each database arrange its data should be taken into account. We considered developing the syntax analyzer for the particular case of MongoDB based on the rules explained in [5].

The input of this component is the workload of the database that is logged in the *Profiles*. Its output is an ordered set of attributes and their corresponding index type that is passed to the *Miner*. Then, the Miner component generates a set of most frequent single and compound attributes and sends it to the *Config Evaluator*.

The new tendency in developing index recommendation systems is to use the *query optimizer* of the database itself to estimate the costs of running a query with a configuration of indexes. This approach eliminates the effort to develop an additional cost function for index recommendation. Also, utilizing the query optimizer of the database itself as opposed to an external cost function prevents recommending indexes that in reality are not considered by the database to execute a query.

In our proposed and examined approach in [5], all of the candidate indexes are created on a sample set of the targeted data set. Then the chosen indexes by the query optimizer are returned as the recommended set of indexes. However, this approach relies entirely on the cost function of the database. The estimation of the distance of the recommended indexes to the optimal set of indexes is not possible.

Therefore, developing an ILP optimization method (discussed in Section 4) enables us to compare the performance of these techniques with each other and also evaluate the

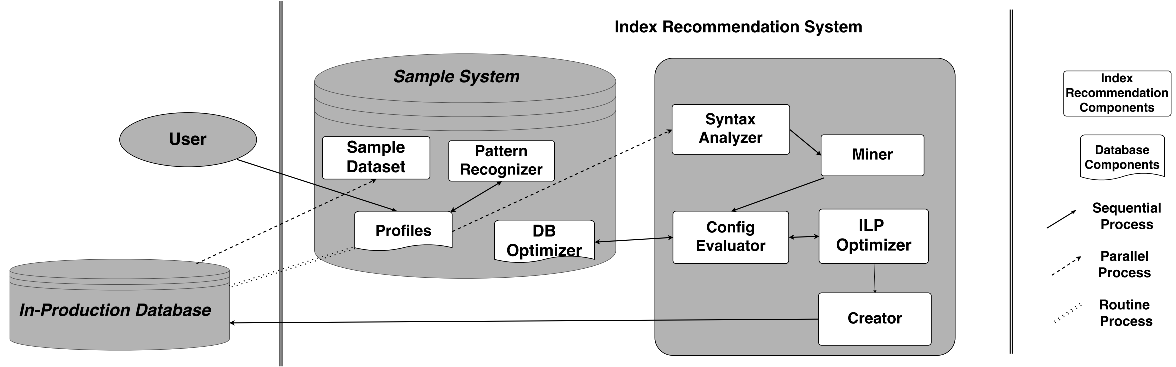

Figure 1: The Architecture of the Index Recommendation System.

recommended set of indexes. To avoid being separated from the database query optimizer and its internal evaluations while using our defined cost function, the cost of running queries in formula (2) are taken from the query optimizer.

For this purpose, the cost of running the query with or without each of candidate indexes should be inquired from the optimizer. Accordingly, the *Config Evaluator* in our design in Figure 1 creates all of the candidate indexes on a sample set of targeted collection and inquiries the cost of running each query in the workload with and without indexes. The obtained information from the optimizer along with the required storage space estimation for each index are passed to the *ILP optimizer*. This component - which can be replaced with any other optimization method - is responsible for recommending the optimal set of indexes to the *Creator* component to create them on the *In-Production Database*.

This approach brings us to the second major challenge in developing index recommendation framework: number of calls to the query optimizer. Each call to the optimizer for estimating the cost of a query applies an overhead on the system. For a large number of the candidate indexes, the process can interfere with the work of the in-production system. Moreover, creating all of the candidate indexes on a large in-production system takes lots of resources and effects the performance of the system negatively.

Thus, our index recommendation system contains a *Sample System* that all of its characteristics are the same as the original database system only smaller. Sand-boxing the data set in the sample system enables us with obtaining the query costs from the optimizer for a larger number of candidate indexes. This cost estimation and also estimating the required storage space can be done without disturbing the performance of the in-production system.

However, utilizing a sample of the data set imposes some challenges itself that require careful research. One problem is that the ratio of presence of each attribute in the original data set to its appearance in the sample changes with each write operation (i.e. insert, update and delete) to the system. The challenge is to keep the sample representative of the current state of the database over time.

Therefore, an appropriate interval for updating the sample set should be chosen. Since determining the sample set requires reading the entire data set and it implies an overhead on the database, the interval should not be too short. The optimization of selecting the appropriate interval for sampling should be done with consideration of the ratio of read to write operations to the database, the size of the data set and the overhead on the in-production system.

Evidently an evaluation of the performance of such index recommendation system is required. The assessment of this system for a NoSQL database in comparison to the similar systems for traditional relational databases is subject to many difficulties that are discussed in Section 5.

4. FORMULATION OF THE INDEX RECOMMENDATION PROBLEM

In this section, a description of the mathematical form of ISP and discussion on optimization methods to solve it are given.

The objective of index recommendation is to recommend the optimal set of indexes for a given database and a workload. The *workload* is a set of m queries as $Q = \{Q_1, Q_2, ..., Q_m\}$. Let $I = \{I_1, I_2, ..., I_n\}$ be the set of all possible indexes. Indexes can be single- or multi-attribute. Each index has a corresponding size of s_1 to s_n.

The execution cost of each query Q_i is different, depending on the indexes that are used by it. Not only a single index but a set of indexes can be used to run one query.

A *configuration* is defined as a subset of indexes that are all used for executing some query, $C_k = \{I_{k1}, I_{k2}, ...\}$. This is known as *atomic* configuration for a workload [9]. There is a set P of all the possible configurations that can be derived from indexes in I and potentially be used by some query, as $P = \{C_1, C_2, ...C_p\}$. Accordingly, each configuration $C_k \in P$ is associated with some subset of $I_\tau \subset I$. A configuration is known as *active* if all of its indexes are built.

Our objective is to find the optimal configuration which its indexes have the maximum benefit for a specific workload under the storage constraints of the system. Each query Q_i has a corresponding cost of running with usage of configuration C_k which we call as $cost(i, C_k)$. Likewise, the cost of running query i without any index is $cost(i, \varnothing)$. Therefore, the benefit of running each query with a configuration is defined as the following:

$$b_{ik} = cost(i, \varnothing) - cost(i, C_k) \tag{2}$$

It should be considered that having indexes for update queries enforces a maintenance cost on the system. Each update operation consists of two parts: finding the proper data unit and modifying it. The finding part is yet another query whose benefit can be calculated from formula (2). The

modification part can be considered as an *insert* (or a *delete*) that does not benefit from having indexes due to the lack of finding statement. Moreover, each update operation enforces maintenance of indexes that are associated with that update operation. In general, a negative benefit $-f_j$ can be associated with each index I_j that is related to the total overhead of that index in association with m' update operations.

Therefore the objective function can be defined as

$$\max(\sum_{i=1}^{M}\sum_{k=1}^{p} b_{ik} \cdot x_{ik} - \sum_{j=1}^{n} f_j \cdot y_j) \tag{3}$$

where $M = m + m'$. This objective function is subject to the following constraints:

$$\forall i \leq M: \qquad \sum_{k=1}^{p} x_{ik} \leq 1 \tag{4}$$

$$\forall k : I_j \in C_k, \forall i \leq M, j \leq n: \qquad x_{ik} \leq y_j \tag{5}$$

$$\forall i \leq M, j \leq n, k \leq p: \qquad x_{ik}, y_j \in \{0, 1\} \tag{6}$$

$$\sum_{j=1}^{n} s_j \cdot y_j \leq S \tag{7}$$

Constraint (4) ensures that at most one configuration is used for any query. Constraint (5) represents the fact that a configuration can not be used unless all of its indexes are built. Constraint (7) enforce the limitation of available storage S on the number of indexes that can be built.

Constraint 6 defines the binary nature of the two introduced decision variables x_{ik} and y_j. For each pair of query Q_i and configuration C_k, x_{ik} is defined as:

$$x_{ik} = \begin{cases} 1 & query\ Q_i\ uses\ configuration\ C_k \\ 0 & otherwise \end{cases} \tag{8}$$

In the same way, the variable y_j is associated with built indexes and can be defined as:

$$y_j = \begin{cases} 1 & index\ I_j\ is\ built \\ 0 & otherwise \end{cases} \tag{9}$$

We consider using ILP methods so that we can estimate the quality of the optimal solution in a tight bound, similar to the approach take by [14]. Also, usage of methods such as branch-and-bound enables us to improve the quality of an approximate solution that would have optimal performance, but cannot be built and used due to storage limitation. In general, usage of ILP can provide the same performance as the other optimization methods while it examines much more alternative solutions.

5. EVALUATION CHALLENGES ON NOSQL DATABASES

In the previous section, the theory to design the fitting objective function and our choice for the proper optimization method were discussed. This section explains the difficulty of evaluating such system, in particular on a document-based NoSQL database.

To compare the performance of databases and database-related systems, different sets of workloads are required to represent various applications. Some examples of such applications are Online Analytical Processing (OLAP) applications with their long and aggregated read-mostly set of queries and Online Transaction Processing (OLTP) applications with their short update-intensive set of queries.

Evaluation of traditional relational databases is mostly done by using variances of the well-known Transaction Processing Performance Council (TPC) [15] benchmarks. However, utilizing TPC benchmarks for evaluating NoSQL databases runs into at least two major challenges: first mapping of the tabular data format of the relational model to a specific NoSQL data format (e.g. documents, key-values, graphs, etc.) and second translating SQL queries to the commonly object-oriented query language of a NoSQL database.

The first challenge arises from the fact that the relational data models are highly normalized. Mapping such normalized model to the often denormalized models of NoSQL databases requires careful study. For example, in document-based databases, there is the danger that the mapped model either overdo the usage of nested documents or not using this capability at all. Both of these cases directly affect the query performance, because it is highly dependent on the data access path.

In the case of overdoing usage of nested documents, the access path to the deepest documents is long. Hence, the database performance is negatively affected. Not using the capability of nested documents at all enforces usage of references in documents to the related documents. In this case, due to the lack of join operation in most NoSQL database, more queries should be issued to fetch the required data. It would also affect the performance of the database in comparison to the relational model. A proper mapping between of these databases is hard and currently missing.

There are however other solutions targeting the problem of generating different workloads for various databases. One of them is the Yahoo Cloud Serving Benchmark (YCSB) [17]. Usage of this solution for evaluating our index recommendation system which is developed in combination with MongoDb [13], a document-base database had a major difficulty: the data set and all queries are handled by the MongoDB specific primary key, *_id* field, which is by default indexed in MongoDB.

Another available solution for this problem is the Apache JMeter [10]. Although JMeter provides a good environment for benchmarking, it requires the user to enter manually the set of queries for benchmarking. This setup does not provide an easy possibility of producing a large and diverse set of queries that are distributed over a time interval.

Therefore, in order to overcome this problem, we provided a generic workload generator named Not only Workload Generator (NoWog) which is available in [4]. NoWog provides many features to fulfill its main objective: generating synthetic workloads similar to realistic workload in an integrated layer. This layer should make NoWog independent from the data model of the database.

Some of these features are the possibility to generate large workloads by defining simple rules that configure the distribution of different query types in various time intervals, usage of arbitrary keys for querying.

Development of such flexible and generic workload generator was absolutely necessary for enabling the evaluation

process of our index recommendation system.

6. CONCLUSION AND FUTURE WORK

In this paper, first, we reviewed many challenges in developing a framework for index recommendation system. Some of the major challenges are providing a solution for reducing the search space for candidate indexes and also reducing the number of calls to the query optimizer for obtaining the costs of running a query with different indexes. Our solution for the former is to only consider the evaluation of the most relevant query attributes (most frequent and longest queries). We solved the latter with introducing a virtual environment containing a sample of the targeted data set.

Then, a mathematical definition of index recommendation is presented and some of the possible optimization methods to solve the problem are discussed. At the end, the challenges of evaluating such system for NoSQL database are briefly discussed. The NoWog is introduced as a solution for generating synthetic workloads with different distributions over time.

Investigating similarity of new queries to the previous one in order to estimate the cost of recommending and creating new indexes in contrast to using the already existing ones is part of future plans to expand this work.

7. ACKNOWLEDGMENTS

The author like to thank Large-Scale Data Management and Analysis project [11] by the German Helmholtz Association for funding this research.

8. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. `http://dl.acm.org/citation.cfm?id=645920.672836`.

[2] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 496–505, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[3] P. Ameri. On a self-tuning index recommendation approach for databases. Manuscript is accepted to be published by the IEEE International Conference on Data Engineering (ICDE) PhD Symposium, 2016.

[4] P. Ameri and H. Guan. Nowog. `https://github.com/ParinazAmeri/NoWog` accessed 22-April-2016.

[5] P. Ameri, J. Meyer, and A. Streit. On a new approach to the index selection problem using mining algorithms. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 2801–2810, Oct 2015.

[6] K. Aouiche and J. Darmont. Data mining-based materialized view and index selection in data warehouses. *CoRR*, abs/0707.1548, 2007.

[7] N. Bruno and S. Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 227–238, New York, NY, USA, 2005. ACM.

[8] A. Caprara, M. Fischetti, and D. Maio. Exact and approximate algorithms for the index selection problem in physical database design. *IEEE Trans. on Knowl. and Data Eng.*, 7(6):955–967, Dec. 1995. `http://dx.doi.org/10.1109/69.476501`.

[9] S. Chaudhuri and V. Narasayya. An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*. Very Large Data Bases Endowment Inc., August 1997.

[10] JMeter. Apache jmeter. `http://jmeter.apache.org/` accessed 22-April-2016.

[11] C. Jung, M. Gasthuber, A. Giesler, M. Hardt, J. Meyer, F. Rigoll, K. Schwarz, R. Stotzka, and A. Streit. Optimization of data life cycles. *Journal of Physics: Conference Series*, 513(3):032047, 2014. `http://stacks.iop.org/1742-6596/513/i=3/a=032047`.

[12] J. Kratica, I. Ljubic, and D. Tosic. A genetic algorithm for the index selection problem. Technical report, In Applications of Evolutionary Computing, 2003.

[13] MongoDB. Mongodb for giant ideas. `https://www.mongodb.com/` accessed 22-April-2016.

[14] S. Papadomanolakis and A. Ailamaki. An integer linear programming approach to database design. In *In ICDE Workshop on Self-Managing Databases*, 2007.

[15] tpc, 2016. `http://www.tpc.org/information/benchmarks.asp` accessed 22-April-2016.

[16] G. Valentin, M. Zuliani, D. C. Zilio, G. M. Lohman, and A. Skelley. Db2 advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of the 16th International Conference on Data Engineering*, ICDE '00, pages 101–, Washington, DC, USA, 2000. IEEE Computer Society. `http://dl.acm.org/citation.cfm?id=846219.847390`.

[17] YCSB. Home brianfrankcooper/ycsb wiki github. `https://github.com/brianfrankcooper/YCSB/wiki` accessed 22-April-2016.

[18] D. C. Zilio. Physical database design decision algorithms and concurrent reorganization for parallel database systems. Technical report, 1997.