GADIS: A Genetic Algorithm for Database Index Selection

Priscilla Neuhaus, Julia Couto, Jonatas Wehrmann, Duncan D. Ruiz and Felipe Meneguzzi
School of Technology, PUCRS - Pontifícia Universidade Católica do Rio Grande do Sul - Porto Alegre, Brazil
Email: [priscilla.neuhaus, julia.couto, jonatas.wehrmann]@edu.pucrs.br, [duncan.ruiz, felipe.meneguzzi]@pucrs.br

***Abstract*—Creating an optimal amount of indexes, taking into account query performance and database size remains a challenge. In theory, one can speed up query response by creating indexes on the most used columns, although causing slower data insertion and deletion, and requiring a much larger amount of memory for storing the indexing data, but in practice, it is very important to balance such a trade-off. This is not a trivial task that often requires action from the Database Administrator. We address this problem by introducing GADIS, A Genetic Algorithm for Database Index Selection, designed to automatically select the best configuration of indexes adaptable for any database schema. This method aims to find the fittest individuals for optimizing both query response time, and disk required for the indexed data. We evaluate the effectiveness of GADISthrough several experiments we developed based on a standard database benchmark, compare it to three baseline indexing strategies, and show that our approach consistently leads to a better resulting index configuration.**

***Index Terms*—Database, Indexing, Artificial Intelligence, Genetic algorithms, Learning system.**

I. INTRODUCTION

Creating indexes is the main action to improve database query performance [7, Chap. 15], since the indexes are the most used technique to speed up queries response [9]. However, it is important to properly choose the columns to be indexed, given that it also affect time to insert and update data and increase disk consumption. The Database Management Systems (DBMS) optimizer is responsible for analyzing queries and choosing the most efficient way to access information. The goal of an optimizer is to find options for running a given query and evaluate the cost of each choice, so that the chosen one would provide the best performance for retrieving the data.

Achieving the best indexing configuration for a database is not a trivial task [6]. Ideally, all the frequently queried columns should be indexed for a faster data retrieval. However, it is quite complex to find a balanced trade-off between performance and storage required. It is quite often the case when the cost-based optimizer is not able to find a proper solution, requiring the DBA to make a final decision on the database architecture regarding indexing strategies.

In this paper we developed GADIS, an approach based on Genetic Algorithms (GA) to help finding an optimal index configuration. We use two fitness functions: 1) an optimization objective to maximize the database performance considering INSERT, DELETE and SELECT queries; and 2) designed to optimize the query response time by search for faster index configurations when compared to the initial one.

We perform a set of experiments using TPC-H, a well known database benchmark [11], and we compare our results with three baseline approaches: (i) the initial index configuration; and (ii) indexes derived from a random-search algorithm. Results show that GADIS outperforms the baselines, allowing for better index configuration on the optimized database. Finally, we observe that our approach is much more prone to find a proper solution that the competitive methods.

II. BACKGROUND

A. Genetic Algorithms

Holland [3] developed the concept of Genetic Algorithm inspired by the evolutionist theory. GA simulates the principles of biological evolution by repeatedly modifying a population of individual using rules modelled on reproduction and gene combinations. In this simulation of evolution, it leverages the best genes from the fittest individuals of a population to continue across the next generations. GA models individuals in terms of their genome, typically represented as strings of bits. Each generation replaces the previous population by its fittest *offspring*, where fitness is computed by a fitness function that assigns a score to each individual. Fitness typically measures how well an individual solves the problem at hand. The algorithm initializes with a random population and works through *selection*, *crossover*, and *mutation*. That process continues until the optimize criterion is satisfied or a certain number of generations is reached. Due to its random nature, GA improves the chances of finding a global solution.

B. TPC-H Benchmark

TPC [1] is a a non-profit corporation that produces database benchmarks. We chose TPC-H because it models a business database having realistic ad-hoc queries. TPC-H has a database schema, a workload and performance metric tests. The database size varies according to a constant named scale factor (SF). The workload is composed of 22 queries of varying complexity, and 2 refresh functions, that simulate data insertions and deletions in the two larger tables (`orders` and `lineitens`). In the performance test, the benchmark executes a power test and then a throughput test. The power

DOI reference number: 10.18293/SEKE2019-135

[1]TPC: http://www.tpc.org/

TABLE I
GA DEFINITIONS APPLIED IN THE CONTEXT OF THE DBMS

GA Definition	DBMS Application
Gene	0 if the column is not indexed; 1 otherwise
Individual	a database state represented by a vector that refers the columns of the schema
Population	collection of database states
Parents	two database states selected to be combined and next create a new database state
Mating pool	a collection of parents that are used to create the next population
Fitness	a function that tells us how good each database state is by running the benchmark
Mutation	a way to introduce variation in our population by randomly swapping the genes (0 - 1) of two individuals
Elitism	a way to carry the best individuals into the next generation

test (POWER@SIZE), calculates how fast the system computes answers to single queries. It executes a function to insert data, then it runs all queries in parallel, then it executes another function to delete data. The throughput test (THROUGHPUT@SIZE) measures how many queries were executed in the elapsed time for parallel query streams to simulate multiple users. It computes the ratio between the total number of queries and the total time spent to run the queries. TPC-H also presents the query-per-hour metric (QPHH), obtained from the geometric mean of power and throughput test. It captures the overall performance level of the system, both for single-user and multi-user mode.

III. METHOD

We already saw that indexes can speed up the data access. However, when we create or delete indexes we must verify which combination of indexes is the best one for queries selection. Specifically, for optimizing the TPC-H database, there are 2^{45} possible combinations for column indexing. Our approach is based on GAs trained directly on a running database, designed to evolve individuals that represent the whole index structure (i.e., all the columns on the database). We map GA definitions to the DBMS context in Table I.

A. Individual Representation

We use a straightforward individual representation that is based on binary vectors. In this strategy, each vector position denotes whether a column is indexed or not. Formally, we represent an individual as a binary vector $\mathbf{x}$, where $|\mathbf{x}| = C$ and C is the number of all columns in the whole database schema. Hence, $\mathbf{x}_i$ denotes whether the i^{th} column should be indexed by the DBMS. Naturally, both primary and foreign keys are always indexed, and therefore not affected by any crossover, mutation or additional random-based action on $\mathbf{x}$.

The initial population with n individuals is randomly created by sampling bits from an uniform distribution. Each bit corresponds to a specific gene having two possible actions: *create* or *drop* one index. Each individual is a concatenation of the binary representation of all columns from all tables. TPC-H contains 61 columns across 8 tables, with 16 primary and foreign keys indexed by default. From the remaining columns, only 24 are used on the benchmark queries. Hence, each individual is comprised by $C = 24$ mutable genes, which generates a search space containing $2^{24} = 16,777,216$ possible combination of indexes.

B. Fitness Function

During the evolutionary process, we use a fitness function to estimate the degree of adaptation to the environment for each individual in the population. We first use the QPHH as fitness function that aims to optimize the performance for running queries while being computationally cheaper for data insertion and removal. We also propose a simpler approach, which optimizes the speed-up time for running all the 22 select queries in the benchmark.

More specifically, we want to find an individual represented by a genome that is capable of achieving high QPHH, but using the very least memory as possible. Formally, let $\mathcal{Q}(\mathbf{x})$, $\mathcal{H}(\mathbf{x})$ and $\mathcal{P}(\mathbf{x})$ be the functions that estimate the QPHH, THROUGHPUT@SIZE and POWER@SIZE for a given $\mathbf{x}$. Note that the estimate of $\mathcal{P}(\mathbf{x})$ is calculated by running all the queries in the benchmark, including delete and insert ones that are quite slow in heavily indexed databases. Thus, a database in which most of the columns are indexed would most likely yield lower POWER@SIZEvalues. The QPHH-based fitness function would be hereby referred as $\mathcal{Q}(\mathbf{x})$, calculated by Eq. 1.

$$\mathcal{Q}(\mathbf{x}) = \sqrt{\mathcal{P}(\mathbf{x}) \times \mathcal{H}(\mathbf{x})} \tag{1}$$

Our second fitness function optimize the time performance for running the SELECT queries in the benchmark. In this case, we want to find individuals that are capable of retrieving data efficiently, without considering data insertion and removal. This is achieved by optimizing the speed-up of the current individual when compared to a baseline one, namely, the initial state of the database. In this case, we refer to the function that calculates the total query time for a given individual as $\mathcal{T}(\mathbf{x})$. Finally, the speed-up-based fitness function is given by

$$\mathcal{S}(\mathbf{x}) = \frac{\mathcal{T}(\mathbf{x}_I)}{\mathcal{T}(\mathbf{x})} \tag{2}$$

where $\mathbf{x}_I$ denotes the individual for the initial state of the database. Therefore, individuals with lower values of $\mathcal{T}(\mathbf{x})$ have higher fitness values than those that take more time to run the benchmark. In this case, we are necessarily optimizing the database storage requirements for index data. In summary, the proposed fitness functions are two-fold: (i) $\mathcal{Q}(\mathbf{x})$, which optimizes the performance for INSERT, DELETE and SELECT commands, and by transitivity, it also helps to lower memory requirements for indexes; and (ii) $\mathcal{T}(\mathbf{x})$ that directly optimizes the running time for all queries.

C. Selection

Our approach uses the popular and effective tournament method as selection technique, developed by Horn et al. [8]. It is a strategy for selecting the fittest candidates from the current generation in a GA. The process initiates with two candidate points selected randomly from the current population, that compete for survival in the next generation. The next step is to compile points randomly to compose a tournament set, where each member is compared with other members. We need to specify the size of the tournament set as a percentage of the total population. Hence, the tournament set size implies the degree of difficulty in surviving. If the tournament size is larger, weak candidates have a smaller chance of getting selected as it has to compete with a stronger candidate. We use the selection pressure to determine the rate of convergence of the GA. This is a probabilistic measure of a candidate likelihood of participation in a tournament. Here, the convergence rate is proportional to the selection pressure, and the GA is capable of identifying optimal or near-optimal solutions over a wide range of selection pressures. The tournament selection works either for positive or negative fitness values.

D. Crossover and Mutation

After obtaining the individual from selection method, we apply the crossover genetic operator on population. In this phase, two individuals exchange information to produce the offspring. Both crossover and mutation occur only with respect to some probability previously defined. The main goal of the use of crossover is to make it possible for the genes of two individuals to generate an improved individual. Higher crossover values lead to in-depth exploitation of the current population individuals, but constraining the exploration of the search space. We employ the two-point operator [10], that determines two random crossover points to mark at which points of the two parents will occur the split. Next, the tails of their two parents are swapped to get a new offspring, which would integrate the population. It is important to define the crossover probability (*cp*), which controls the frequency of the application of the crossover operator on the individuals. For instance, when using $cp = 1.00$, crossover is applied over the entire population. On the other side, when $cp = 0.00$, the entire new generation is made from exact copies of individuals from old population (which can suffer mutation as well).

The next step is the mutation operator, that introduces random changes into the characteristics of the individuals. Mutation plays a critical role in GA, as crossover leads the population to converge by making the individuals in the population alike. Mutation reintroduces genetic diversity back into the population. In this phase, we need to set the mutation probability and the mutation rate. The first parameter sets the chance of each individual to be mutated, whilst the second one refers to the number of genes that would be changed.

Frequently, the mutation rate is rather small and depends on the length of the individuals. Therefore, new individuals produced by mutation will not be very different from the original one. Since we set each individual as a bit vector, the mutation operator is responsible to change the bit value of 1 to 0 or vice-versa. We also provide some experiments using a linear decay for the mutation probability. Such a decay allows for larger exploration during the initial generations, increasing the genetic diversity, while allowing for in depth exploitation during the latter stages of the optimization procedure.

IV. Experimental Setup

We carried out all experiments in hardware with Intel(R) Xeon(R) Platinum 8175M CPU @ 2.50GHz, 30GiB RAM and Non-Volatile memory controller (SSD), running Ubuntu Linux 16.04. We used DEAP [2], an open-source Python package, to implement the genetic algorithm. To evaluate all approaches, we use the TPC-H benchmark with scale factor of 1GB. We run the performance test and calculate the Power@Size and Throughput@Size metrics for each execution of TPC-H. Then we calculate the QphH for each individual ($\mathcal{Q}(\mathbf{x})$).

A. Baselines

For evaluating our approaches we defined three main baseline strategies for database indexing. We compared our approach with the following strategies: (1) TPC-H initial state (only primary and foreign keys are indexed); (2) results achieved via a random-search algorithm that optimizes the proposed fitness functions. We run the random search using the same number of individuals as in our optimization algorithm.

B. Evaluation Measures

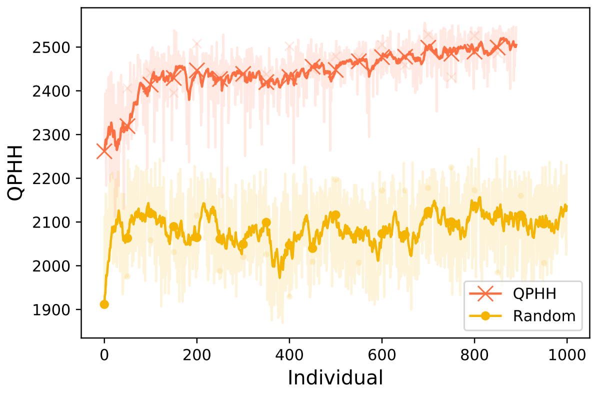
In our experiments, we run the final evaluation using a different set of query streams to the database, in order to make sure that our GA was not overfitting the training examples. First, we set the best indexing configurations found during the whole training phase, in order to run the complete benchmark. For quantitative evaluation, we used QphH and the storage required by the index data (in MB).

C. Parameters

The first three experiments were designed to optimize the fitness function $\mathcal{Q}(\mathbf{x})$, in order to maximize the QphH metric. The two latter ones focused on minimizing the second fitness function, namely the benchmark runtime given by $\mathcal{T}(\mathbf{x})$. For all experiments we used a initial population of 50 individuals, evolved during 50 generations, with mutation rate of 0.05 and elite size of 4. Parameters such as fitness function, mutation probability (mp), crossover probability (cp) and mutation decay (md) are defined as follows: Gadis-[$\mathcal{Q}$, mp=0.9, cp=0.8, md=0.05] and Gadis-[$\mathcal{T}$, mp=0.9, cp=0.6]. The complete training procedure for each of those experiments (including random search) took about a week.

V. Results

Figure 1 shows the performance of our approaches during the training phase. The chart shows values of QphH as fitness for all evolved individuals; and the second chart depicts values of time required to run the test benchmark (only with select queries), by using our second fitness function, namely $\mathcal{T}(\mathbf{x})$. Gadis have shown to be much better in terms of consistency, once it can find several good database configurations with ease when comparing to the baseline. It is clear that both Gadis-[$\mathcal{Q}$] Gadis-[$\mathcal{T}$] present similar performance during



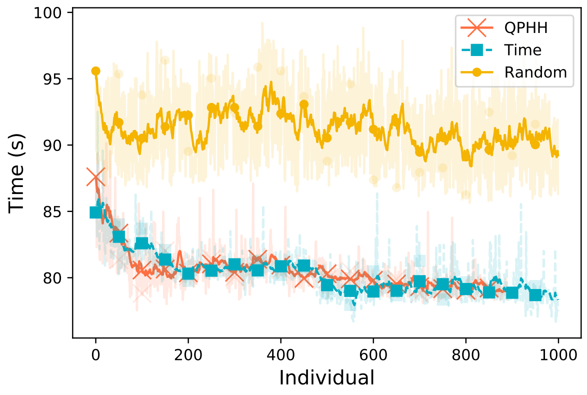


Fig. 1. QPHH and Time during training procedure.

TABLE II
RESULTS IN TERMS OF QPHH, TIME AND INDEX SIZE

Methods	QPHH	Time	Index Size
Initial State	1678	149.3s	**599 MB**
Random Search	1864	145.3s	1196 MB
GADIS-[$\mathcal{Q}$]	2631	71.2s	*1193* MB
GADIS-[$\mathcal{T}$]	**3077**	**60.6s**	1600 MB

the training phase, thought the latter presents itself as a much faster approach.

Table II shows quantitative results in terms of QPHH, Time and Index Size for each one of the approaches. Note that the best performing approach as measure in QPHH and Time is achieved by GADIS-[$\mathcal{T}$]. Additionally, both versions of GADIS were able to achieve top results in all used metrics. GADIS-[$\mathcal{Q}$] also requires roughly the same disk space when compared to the baseline, though with 40% better QPHH performance.

VI. RELATED WORK

In the past years, other researchers have used GA to improve database performance. Korytkowski et al. [4] present an automatic way to find the best set of indexes for a database, using GA. Different from our approach, their fitness function is based on time spent in insert operations and a single query, and they make experiments with just one table.

Pedrozo et al. [5] modelled an index tuning architecture applied to hybrid storage environments using GA to create indexes in the DBMS. They also use TPC-H benchmark to evaluate their approach, but unlike us, they did not apply the performance test provided by the benchmark.

Boronski et al. [1] propose a model to optimize the response time of a set of queries by creating indexes in a relational database. Unlike us, they use the response time of each group of queries to measure their performance and then compare it with the Oracle advisor.

VII. CONCLUSIONS

In this paper we developed a Genetic Algorithm-based approach for automatic index selection in databases called GADIS. This approach can find database configurations that outperform all the baselines in most of the scenarios, while saving storage requirements. We have observed that the training procedure of GADIS is very consistent, which allows us to find proper database configurations within only a few generations. In addition, GADIS is easily suited to be implemented on any database system. The main limitation of our work is that to find a good solution one has to run several distinct database configurations and evaluate them by using a benchmark. Such a procedure is somewhat costly, and required about a week of processing to optimize the indexes for the used database. For future work, we plan to improve GADIS so we can learn general rules for database agnostic indexing using metadata rather than performing a per-database optimization.

REFERENCES

[1] R. Boronski *et al.*, "Relational database index selection algorithm," in *CN*, A. Kwiecień *et al.*, Eds. Springer, 2014, pp. 338–347.
[2] F.-A. Fortin *et al.*, "DEAP: Evolutionary algorithms made easy," *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, jul 2012.
[3] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.
[4] M. Korytkowski *et al.*, "Genetic algorithm for database indexing," in *ICAISC*, L. Rutkowski *et al.*, Eds. Springer, 2004, pp. 1142–1147.
[5] W. G. Pedrozo *et al.*, "An adaptive approach for index tuning with learning classifier systems on hybrid storage environments," in *HAIS*. Springer, 2018, pp. 716–729.
[6] E. Petraki *et al.*, "Holistic indexing in main-memory column-stores," in *SIGMOD*. ACM, 2015, pp. 1153–1166.
[7] R. Ramakrishnan *et al.*, *Database Management Systems*, 3rd ed. McGraw-Hill, Inc., 2003.
[8] J. rey Horn *et al.*, "A niched pareto genetic algorithm for multiobjective optimization," in *CEC*, vol. 1, Citeseer. IEEE, 1994, pp. 82–87.
[9] P. Rob *et al.*, *Database Systems Design, Implementation and Management*, 5th ed. Course Technology Press, 2002.
[10] W. M. Spears *et al.*, "An analysis of multi-point crossover," in *Foundations of genetic algorithms*. Elsevier, 1991, vol. 1, pp. 301–315.
[11] A. Thanopoulou *et al.*, "Benchmarking with TPC-H on off-the-shelf hardware," in *14th International Conference on Enterprise Information Systems*. Springer, 2012, pp. 205–208.