Using Autotuning Techniques to Improve Database Performance

Nitya Subramanian, Jason Ansel, Saman Amarasinghe
Massachusetts Institute of Technology
Cambridge, MA
{nityas, jansel, saman}@mit.edu

I. Introduction

The focus of this project will to improve the performance of databases using program autotuning. The project will build a database that can adapt to optimize itself for specific systems and workloads. The key advantage this project will have is the ability to specialize a database configuration for a specific workload, rather than solving a more general problem of optimizing for all use cases. The projected impact of this project is an increase in database performance for a given environment and to build databases that can adapt to work best in any environment.

II. Problem Statement

Database performance is a critical component of any data-backed application. As such, current distributions of the database applications under consideration for this project have already been heavily tuned, both by hand and various other means, to optimize performance.

Tuning is a process for program optimization that traditionally involves tweaking application configuration parameters by hand to obtain performance improvements. Most programmers will tune a few environment variables with the goal of improving performance, but in many cases, the optimally performing configuration for a program is one that results from tuning hundreds of parameters.

Despite extensive prior hand tuning, databases are used under environments and circumstances that vary widely, and applications are generally tuned to optimize for average-case use. This project aims to leverage these inherent variations to enable autotuning of a database instance to a particular system's requirements to gain performance improvements over even heavily hand tuned application instances. This observation is the main motivator for the project. By exploiting the ability of autotuning to specialize a database configuration for only a specific task, we enable databases to be specialized and therefore configured for only tasks that are relevant to the specific database by discarding options that improve performance in many other cases, but not the specific performance case being tuned for.

III. Related Work

The primary resource this project will use to accomplish autotuning is OpenTuner, an ongoing project within the Commit Research Group at CSAIL.[4]OpenTuner is an open source framework for program autotuning that features an extensible configuration representation and uses ensembles of techniques to determine the optimal configuration for a program.

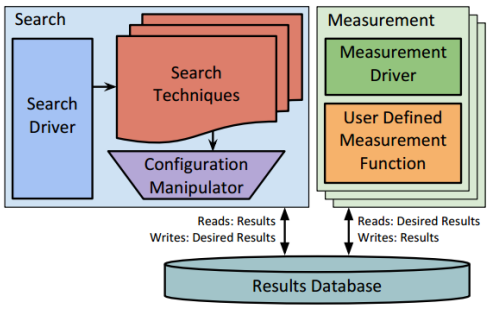

Fig. 1. Architecture of the OpenTuner Framework. OpenTuner achieves performance gains by using heuristics to navigate the large parameter search space of applications and find the optimal parameter configuration maximizing performance for a given benchmark. [4]

OpenTuner has demonstrated success in yielding performance improvements in other types of applications, including the GCC compiler as shown in Figure 2, though no such tests on databases have been completed.

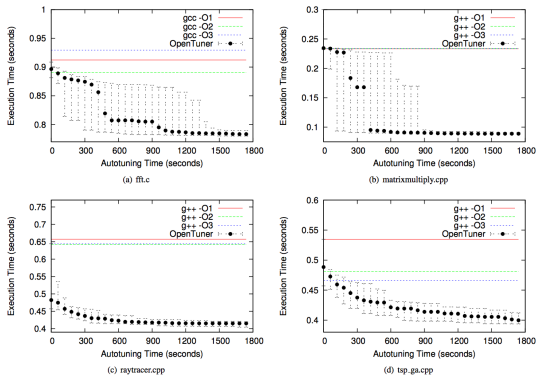

Fig. 2. Opentuner Performance on the GCC Compiler over four different programs.Speedups of up to 2.8xhave been observed using OpenTuner with the GCC compiler. [4]

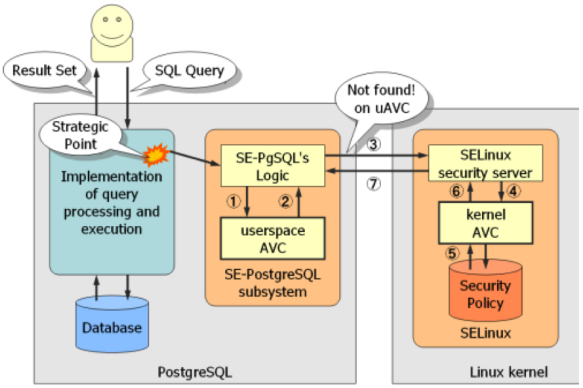

Fig. 3. The internal architecture of Postgres. All tuning for this project will occur through manipulation of query processing and execution to achieve performance gains on established benchmarks. Source: https://wiki.postgresql.org/wiki/SEPostgreSQL_Architecture

While a framework such as OpenTuner has never been extended to optimize databases, database performance is a field with numerous solutions that this project hopes to expand upon. One such solution is Pgtune, an open-source project built specifically for the PostgreSQL (or Postgres) database application. Pgtune automatically tunes variables referenced in the Postgres configuration file based on the amount of RAM available on the hardware used to run the application as well as the suggested workload on the instance.[1]It is expected that this project will yield performance improvements over Pgtune with added tuning on the Postgres query planner as well as manipulation of configuration variables not found in the primary configuration file modified by Pgtune.

IV. Methodology

The primary deliverable of this project is an end-to-end pipeline that searches the Postgres parameter space, running a benchmark on it until the optimal configuration has been found. The approach we used was to create an initial, functioning pipeline, then iterate upon receipt of preliminary results.

A. Performance Analysis

Because this project aims to measure and hopefully improve the performance of a system, the primary means of evaluation will be the speed at which a given instance of Postgres executes a set of database commands. There are existing sets of such queries, known as benchmarks, available specifically for Postgres. These benchmark speeds will be compared for instances running standard Postgres, Pgtuner, and our tuner in a given environment, as well as across disparate environments.

1) Benchmarking: The range of tested environments is key to showing general applicability of this project. As such, it is important that a range of environments is selected that is representative of the varied environments in which Postgres is typically run, particularly with regards to resource and processing capability of machines. Additionally, the benchmarks chosen must be representative of the range of production-relevant queries typically executed using Postgres. Candidates for relevant benchmarks include the Postgres-provided benchmarks, known as pgbench, as well as the Transaction Processing Council (TPC) Database Benchmark. While the TPC Suite has a greater level of recognition as a standard benchmark, it is highly likely that hand-tuned versions of many database applications are optimized for performance on this particular benchmark. Exclusive use of this benchmark could therefore result in findings that favor a specific use case over even average case performance. Over the course of evaluation of the project, both benchmarks as well as others available for use will be considered further.

Performance of database systems created as a result of autotuning will be evaluated on two variants of TPC-B, read-only and read-write. This will ensure the adaptability of the autotuning process to varied benchmarks, an important requirement for the results obtained to be generalizable.

2) Pgbench Testing Suite: The specific testing application that will be used to collect benchmark performance scores will be pgbench. This suite was selected on the basis of flexibility and ease of integration. Because Pgbench is the default testing suite for Postgres, it is a reasonable assumption that out of the box Postgres is hand tuned to perform well on this testing suite. Additionally, this testing framework is highly extensible, with a variety of options available to customize benchmarks run including duration of test, type of benchmark test, number of concurrent threads, and number of clients connected with access to the database.

From the ubuntu manpages [2] "pgbench is a simple program for running benchmark tests on PostgreSQL. It runs the same sequence of SQL commands over and over, possibly in multiple concurrent database sessions, and then calculates the average transaction rate (transactions per second). By default, pgbench tests a scenario that is loosely based on TPC-B, involving five SELECT, UPDATE, and INSERT commands per transaction."

Initially, a test involving 100000 transactions was performed. This series of transaction, under most configurations, took under a second to complete. One well-documented limitation of pgbench and TPC-B in general is a nondeterministic component to execution that creates variability in test scores obtained. To combat this, it is recommended that tests be run for at least ten minutes to obtain performance scores that are reproducible.

Each time pgbench is run, it creates four new tables, destroying existing tables of the same name that may exist prior to doing this. These four tables each contain between 1 and 10000 rows. After creating these four tables, pgbench runs a set of 100000 transactions that involve transfers between fields in these four tables. These transfers can either be default read-only or read-write TPC-B style benchmarks or user-specified benchmarks. In this series of experiments we elected to use the default TPC-B style benchmarks to increase the applicability of our results by having a wider range of previous results to compare our results against.

Because of the time cost associated with running multiple iterations of such ten minute tests on a very large search space, our initial results only take into account results found under the small 100000 transaction benchmark test. This method could yield numbers that are not fully representative of the actual performance of the database on this benchmark, but the results do still carry meaning as an indicator of potential performance on benchmarks run in a fully distributed dedicated testing environment.

B. Technical Approach

An initial challenge faced was in selecting the correct parameters to tune over. Postgres has hundreds of possible parameters that can be tuned, and representing all of them in OpenTuner proved exhausting. Initially, about 20 parameters related to Postgres query tuning were selected, with the goal of adding more parameters once obtaining preliminary results. This increased to about 50 parameters upon successful experimentation with the initial 20 parameters, and the search space used in results detailed here was of the size $10^80.5$. Selecting parameters proved challenging because many numerical parameters did not have clearly labeled numerical ranges and were thus difficult to represent. By beginning with only a few well-documented categories of parameters, we were able to focus on creating an end-to-end pipeline first and assess preliminary results before further optimizing.

Selection of a benchmark was approached in a similar manner to parameter selection. As noted above, we were concerned that Postgres would already be hand-tuned to perform optimally on the TPC-B Benchmark. However, we decided to run our initial experiments on TPC-B and entertain the possibility of switching benchmarks if needed. The primary motivation for this was that good performance on TPC-B was a priority of the project, so if promising results were achieved on this benchmark, no other benchmarking techniques needed to be considered. As detailed in the Benchmarking section above, the initial benchmark used involved 100000 operations and ran to completion within a few seconds in the worst case. However, variability in our experiments necessitated switching to a variant of the same benchmark that instead took ten minutes to complete under each configuration. The additional time requirements of this new benchmark created a need to parallelization, which will be detailed further in the AWS Computing Environment Section.

All initial experimentation was done on a laptop. A locally running instance of Postgres was initialized under various configurations, and the TPC-B

Benchmark was run on each configuration ten times. The metrics used to assess performance were the mean and median benchmark completion times, to protect against any undue outlier effects.

After the experiment was able to run in a local laptop environment, it needed to be moved to a more powerful computing environment. The main reason for this is that experiments run on laptops are prone to interference from other running programs and processes on the laptop, and numbers collected in this manner are therefore prone to noise from inconsistent noise. And ideal collection environment for benchmark scores is a dedicated server instance or node on which no other processes are ongoing and on which the database server can be repeatedly initialized and shut down.

1) Lanka Computing Environment: Once laptop results had been achieved, we moved the experiment to Lanka, a highly powerful cluster. Lanka is 24-node Intel Xeon E5-2695 v2 @ 2.40GHz Infiniband cluster with two sockets per node, each with 12 cores, for a total of 576 cores and a theoretical peak computational rate of 11059 GFlop/s.[3] Each node has 128GB of memory, of which 120GB is safely usable. A key advantage of this machine is that it is able to perform powerful computations consistently, making it a highly valuable tool for measuring performance.

Moving the experiment over to Lanka posed several difficulties. In addition to necessitating the use of a different variant of Postgres, use of Lanka required that all experiments be run under different conditions from that on a laptop. Specifically, all experiments needed to be run without root-level access, and all database queries needed to access one specific node. After several attempts to port the experiment over to Lanka from a laptop computing environment, we found that the permissions restrictions on database access to Postgres made the type of experimentation required an impossibility on the Lanka computing environment. We decided to circumvent this challenge by instead porting the experiment to an AWS dedicated instance.

2) AWS Computing Environment: Running the experiment in a cloud computing environment provided several marked advantages over previous approaches. It enabled quick startup and spinout of new machines to test on, a feature that became very important at later stages. Perhaps most importantly, it also enabled the computing environment in which tests were run to be fully customizable and free of environment-specific sources of noise and inconsistencies. In all experiments conducted thus far, t1.micro instances running Ubuntu 14.04 were employed due mainly to price and computation constraints and requirements.

Once the experiment was up and running on AWS, we noticed some anomalies in the data. Namely, the benchmark being employed, which consisted of 100000 operations on a database created specifically for the test, yielded results with a very low signal to noise ratio of about 0.5, indicating that there was a high variability present in the benchmark we were taking to be consistent. To mitigate this noise variability, we needed to run the benchmark for 10 minutes on each distinct configuration. Given that the experiment was run on a parameter search space of $10^8 0.5$, this scaled the time requirements of the experiment to an infeasible duration. To speed up the experiment, parallelization of the autotuning process was required. This stage has not yet been completed, and results detailed below are only indicative of the results garnered from the laptop experiments.

V. Results

Though results are still preliminary, they indicate a very positive potential of autotuning to improve the performance of databases by optimizing for a particular use case. Orders of magnitude speedup were observed in both tested experiments on read-only and read-write benchmarks.

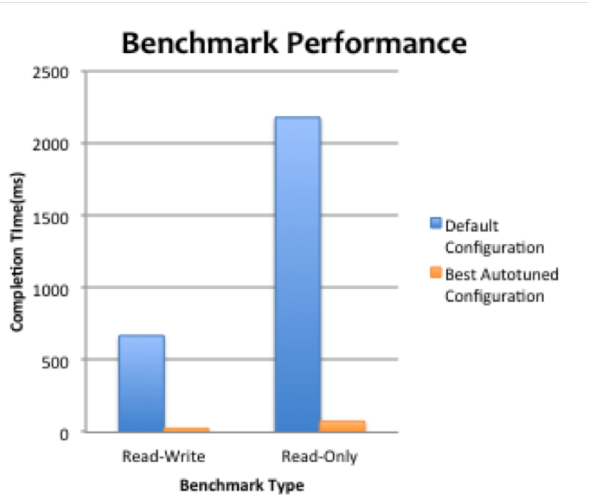

Fig. 4. Comparison of pgbench benchmark test results using both read-only and read-write benchmarks on out of the box vs autotuned Postgres.

In Figure 4, it is evident that the use of autotuning was able to produce orders-of-magnitude speedup in the performance of postgres on the pgbench benchmark, both for read-only and read-write queries. Read-only query benchmarks yielded larger improvements than read-write query benchmarks. This is an indicator that the default configuration of Postgres is optimized for the more common read-write case.

These tests are promising to a point that they could indicate errors in testing methodology. Verification of these results will involve examining other benchmarks and finding cases in which the default configuration of postgres performs optimally and comparing this specific case to the performance of the tuned configuration.

VI. Limitations

As mentioned above, the results detailed in this section were all obtained through experimentation in a laptop environment. As we were unable to obtain results from parallelization experiments on AWS, the results shared here serve as preliminary possibilities as to the potential of this project when using tuning in a noise environment.

The results here were obtained via the shorter benchmark of 100000 operations as opposed to the ten minute benchmark recommended by pgbench documentation. This was done due to time and computing constraints in a non-distributed environment. This methodology to data collection implies that the results described contained a high level of variability and are therefore not guaranteed to be significant. However, the results, though preliminary, do indicate a strong possibility that autotuning can yield benefits in database performance.

VII. Future Work

The work detailed above has yet to be run in a distributed AWS computing environment. Results obtained using a ten minute benchmark on such a distributed environment will likely provide further insight into the true potential of autotuning when applied to databases, as the nature of these experiments is such that benchmark variability and external process interference are minimized.

The quality of the results detailed above also needs to be verified against other benchmarks. In order to be fully confident in performance gains achieved via this series of experiments, it is essential that any specific use case for which postgres is tuned be found to perform similarly to the tuned configuration for that same case. If this is the case, we can safely conclude that performance gains are as a result of specialization because such a result would show that fewer gains relatively were made on a function for which postgres has been hand tuned.

This project can be further extended through the generation of different benchmarks, each testing a different specific capacity of databases. With such a set of benchmarks, it becomes possible to create instances of databases optimized for specific functionalities.

Another interesting application of this work is the capability to create self-tuning databases. The process described in this paper finds optimally performant configurations for a given benchmark, or task. The benchmark can be customized to the workload of a particular Postgres instance, and can be similarly altered to create databases that run with configurations specialized to the type of task they most commonly perform. This enables the capability of specialized database instances, each optimally tuned to do a specific task.

VIII. Impact

These results are highly extensible and can likely be improved upon with the addition of more parameters and consideration of additional benchmarks. The results of this experiment can also likely be applied to the many other widely used SQL based database frameworks.

This series of experiments has demonstrated that high potential for performance gains exist within the postgresql framework. By representing possible parameter configurations as a multidimensional search space, we show that it is possible to make significant improvement in the speed with which Postgres executes series of queries. We also demonstrate the potential of autotuning to garner improvements even in systems that have already been extensively hand-tuned to optimize performance.

References

[1] Pgtune: Postgresql configuration wizard. https://github.com/gregs1104/pgtune. Accessed: April, 2015.
[2] Ubuntu manuals: Pgbench. http://manpages.ubuntu.com/manpages/trusty/man1/pgbench.1.html. Accessed: April, 2015.
[3] Using the lanka cluster. http://groups.csail.mit.edu/commit/lanka. Accessed: April, 2015.

[4] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. Opentuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, Canada, August 2014.