Poster: Autotuning *PostgreSQL*

A Blueprint for Successful Autotuning of Real-World Applications

Thomas Karcher
Karlsruhe Institute of Technology
Karlsruhe
karcher@ira.uka.de

Mathias Landhäußer
thingsTHINKING GmbH
Karlsruhe
mathias@thingsTHINKING.net

ABSTRACT

Autotuning is a technique for optimizing the performance of sequential and parallel applications. We explore the problem of successfully applying on-line autotuning to real-world applications. We tune *PostgreSQL*, an open-source database server software, by optimizing tuning parameters that affect table scans. We evaluate the effects on the performance using the *TPC-H* benchmark and achieve speedups up to 3.9. A video subsuming the process is available at https://dx.doi.org/10.5445/DIVA/2018-192.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**; **Software performance**;

KEYWORDS

On-line autotuning; database; *PostgreSQL*; heuristic, search-based, run-time optimization; parameter tuning

ACM Reference Format:
Thomas Karcher and Mathias Landhäußer. 2018. Poster: Autotuning *PostgreSQL:* A Blueprint for Successful Autotuning of Real-World Applications. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3183440.3195045

1 INTRODUCTION

Imagine your task is to optimize the performance of the *PostgreSQL* database server: Fairly complex source code structure, the workload profile heavily depends on the tables and queries, and it comes with 261 tunable parameters.

On-line Autotuning (OAT) can tackle that task: It is an automated optimization technique that improves the performance of programs [7]. OAT executes a feedback loop that empirically determines the optimal performance of a program by systematically varying tuning parameters and measuring their effect on performance. *On-line* means the autotuner continually adapts the program's configuration to dataset and environment characteristics during production runs. The search stops when improvements become marginal.

Autotuning numerical applications has been around for over 20 years [1]. Active Harmony [8] and MATE [4] are designed for tuning programs in heterogeneous, distributed systems by manipulating tuning parameters. MATE targets MPI applications for dynamic tuning, focusing on MPI-specific tuning parameters. Other approaches consider so called algorithmic choices, i. e. different algorithms performing the same task [6].

We apply the on-line autotuner *AtuneRT* [3] to the database system *PostgreSQL* as follows: (1) We lay the proper groundwork to apply autotuning in general. (2) We setup the benchmark environment with TPC-H [9]. (3) We use SARD [2] to find the most promising tuning parameters. (4) We dissect *PostgreSQL*'s source code to locate suitable spots to introduce the autotuner's API calls. (5) We use TPC-H to evaluate performance changes.

2 ON-LINE AUTOTUNING EXPLAINED

On-line autotuning raises the challenge of amortization: The tuning overhead must not outweigh the performance gain. Amortization is easily missed by exhaustively testing all configurations or spending run-time on bad configurations.

The autotuner needs at least one tuning parameter p with a discrete value set V_p and a standard value $\text{def}(p) \in V_p$. The value of p influences the performance of a program. Run-time measurements are used to compare tuning parameter values.

A measurement point is embedded in a *tuning section*, defining the beginning and the end of a measurement. The autotuner's objective is to find a set of parameter values that is good enough by smartly probing the intervals of the tuning parameters.

New parameter configurations come from a search-based optimization algorithm. We currently use the *Nelder-Mead* simplex algorithm [5]. We modified the algorithm slightly to work on a discrete search space and to stay within the value ranges of the tuning parameters. Additionally, we randomly select a limited number of parameter configurations at the beginning to probe for bad regions within the intervals. The best randomly selected configuration serves as starting point for Nelder-Mead.

AtuneRT provides an API that is used (1) for exposing the program's tuning parameters and sections to the tuner and (2) for reconfiguring the program at run-time.

3 AUTOTUNING *POSTGRESQL*

The source code of the *PostgreSQL* 9.6 database server consists of approx. 600 KLOC. We need to select suitable tuning parameters out of 261 and find a location to put the tuning section into.

We use *TPC-H* as database benchmark [9]. It contains 22 SQL queries that answer business-oriented questions. We used default parameter values and warm disk caches to establish a baseline for the delivery state of *PostgreSQL*.

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3195045

Table 1: *PostgreSQL*'s top four tuning parameters (according to *SARD*) that affect query execution (not planning). We choose min and max to limit the search space.

Parameter	Default	Min	Max
work_mem	5 MB	1 MB	128 MB
effective_io_concurrency	1	1	4
temp_buffers	8 MB	8 MB	1 GB
maintenance_work_mem	64 MB	4 MB	512 MB

Exposing Tuning Parameters. According to *PostgreSQL*'s documentation, there are 37 parameters that influence the run-time of queries; 25 of them steer query planning and preparation, twelve steer query execution. Factoring in their values ranges, the search space is too big for an exhaustive search. The interval of acceptable values often manifests as either a plateau where it does not matter which particular value to choose or as a small window compared to the set of possible parameter values.

We use *SARD* [2] to pre-select tuning parameters. We then focus on tuning parameters affecting query execution and choose the top four from the result of *SARD*. We set reasonable value limits that eliminate unreasonable search space regions (see Table 1). From the *SARD* runs, we learn that tuning parameters affect the queries very differently: Some appear to be unaffected, others are susceptible to only a subset of the four parameters.

We use instruction-based profiling (*GNU gprof*) to locate the tuning parameters within *PostgreSQL*'s source code. We insert the autotuner's API calls to expose the tuning parameters after the basic session startup in `src/backend/tcop/postgres.c`.

Identifying a Tuning Loop. PostgreSQL's on-line documentation, source code comments and sample-based profiling with *perf* help us to identify the function `ExecScan` that reads a single tuple. Reading 200,000 tuples serves as tuning section.

4 RESULTS AND DISCUSSION

Table 2 gives a detailed overview the performance results. Of 22 queries, six did not produce useful results due to not reading at least 200,000 rows or containing sub-queries that break run-time comparability for Nelder-Mead. Some queries experience a speedup of less than 1.02. In these cases, the autotuner amortizes the tuning overhead but fails to exceed *PostgreSQL*'s default performance.

Some queries experience a slowdown; some tuning runs converge to a parameter configuration that is worse than the default. Others fail to run long enough for amortization. The performance gain achieved by autotuning depends on both the tuning parameters and the query.

Caveat. We did not inspect causality between the tuning parameters and the tuning section. There might be other parameters that influence the tuning section and the chosen parameters might be ineffective regarding the tuning section.

The query plan influences the execution; we did not investigate which tuning parameters affecting the planning phase implicitly tune the execution phase.

Table 2: Run-times of the tuning section (ms). "#It": Number of iterations. "NoOpt": Confidence interval ($\alpha = 0.05$) with parameter default values. "NM": Run-time w/ Nelder-Mead.

Q#	#It	NoOpt	NM	$\frac{NM}{NoOpt}$
1	145	504.15 ± 0.27	505.57	0.997
3	15	415.82 ± 4.78	106.49	3.905
5	5	751.71 ± 6.13	742.84	1.012
7	35	180.82 ± 0.62	181.70	0.995
8	150	50.16 ± 0.08	51.8	0.968
9	205	60.09 ± 0.27	61.49	0.977
10	35	215.10 ± 2.19	238.48	0.902
11	40	43.45 ± 0.23	47.86	0.908
13	35	153.67 ± 2.41	241.70	0.636
14	5	79.78 ± 0.92	75.46	1.057
15	10	1505.34 ± 4.55	1487.20	1.012
17	300	68.20 ± 1.67	62.80	1.086
18	150	68.50 ± 0.72	62.58	1.095
20	20	435.27 ± 4.08	427.49	1.018
21	90	160.47 ± 0.23	159.8	1.004
22	35	59.27 ± 0.36	63.4	0.935

5 CONCLUSION AND OUTLOOK

We successfully applied on-line autotuning to *PostgreSQL* with only a few lines of code. The effort lies in locating suitable spots in the source code for the autotuning API calls and in selecting fruitful tuning parameters. We demonstrated that autotuning either helps or maintains query run-time.

REFERENCES

[1] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. 1997. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. In *Proceedings of the 11th International Conference on Supercomputing (ICS '97)*. ACM, New York, NY, USA, 340–347. https://doi.org/10.1145/263580.263662

[2] B. K. Debnath, D. J. Lilja, and M. F. Mokbel. 2008. SARD: A Statistical Approach for Ranking Database Tuning Parameters. In *2008 IEEE 24th International Conference on Data Engineering Workshop*. 11–18. https://doi.org/10.1109/ICDEW.2008.4498279

[3] Thomas Karcher and Victor Pankratius. 2011. Run-Time Automatic Performance Tuning for Multicore Applications. In *Euro-Par 2011 Parallel Processing*, Emmanuel Jeannot, Raymond Namyst, and Jean Roman (Eds.). Number 6852 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 3–14.

[4] A. Morajko, P. Caymes-Scutari, T. Margalef, and E. Luque. 2007. MATE: Monitoring, Analysis and Tuning Environment for Parallel/Distributed Applications. *Concurrency and Computation: Practice and Experience* 19, 11 (Aug. 2007), 1517–1531. https://doi.org/10.1002/cpe.1126

[5] J. A. Nelder and R. Mead. 1965. A Simplex Method for Function Minimization. *Comput. J.* 7, 4 (1965), 308–313. https://doi.org/10.1093/comjnl/7.4.308

[6] Philip Pfaffe, Martin Tillmann, Sigmar Walter, and Walter F. Tichy. 2017. Online-Autotuning in the Presence of Algorithmic Choice. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1379–1388. https://doi.org/10.1109/IPDPSW.2017.28

[7] Patrick Siarry and Zbigniew Michalewicz. 2008. Advances in Metaheuristics for Hard Optimization. (2008). http://bvbr.bib-bvb.de:8991/F?func=service&doc_library=BVB01&doc_number=016716007&line_number=0001&func_code=DB_RECORDS&service_type=MEDIAhttp://dx.doi.org/10.1007/978-3-540-72960-0 In: Springer-Online.

[8] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. 2002. Active Harmony: Towards Automated Performance Tuning. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC '02)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1–11. http://dl.acm.org/citation.cfm?id=762761.762771

[9] Transaction Processing Performance Council. 2017. TPC-H. (2017). http://www.tpc.org/tpch/