Visual Evaluation of SQL Plan Cache Algorithms

Jan Kossmann, Markus Dreseler, Timo Gasda, Matthias Uflacker, and Hasso Plattner

Hasso Plattner Institute, Potsdam, Germany
jan.kossmann@hpi.de

Abstract. Caching optimized query plans reduces the time spent optimizing SQL queries at the cost of increased memory consumption. Different cache eviction strategies, such as LRU(-K) or GD(F)S exist that aim at increasing the cache hit ratio or reducing the overall cost of cache misses. A comprehensive study on how different workloads and tuning parameters influence these strategies does not yet publicly exist. We propose a tool that enables both such research as well as performance tuning for DBAs by visualizing the effects of changed parameters in real time.

Keywords: Query Plan Caching, Cache Eviction, Query Execution

1 Introduction

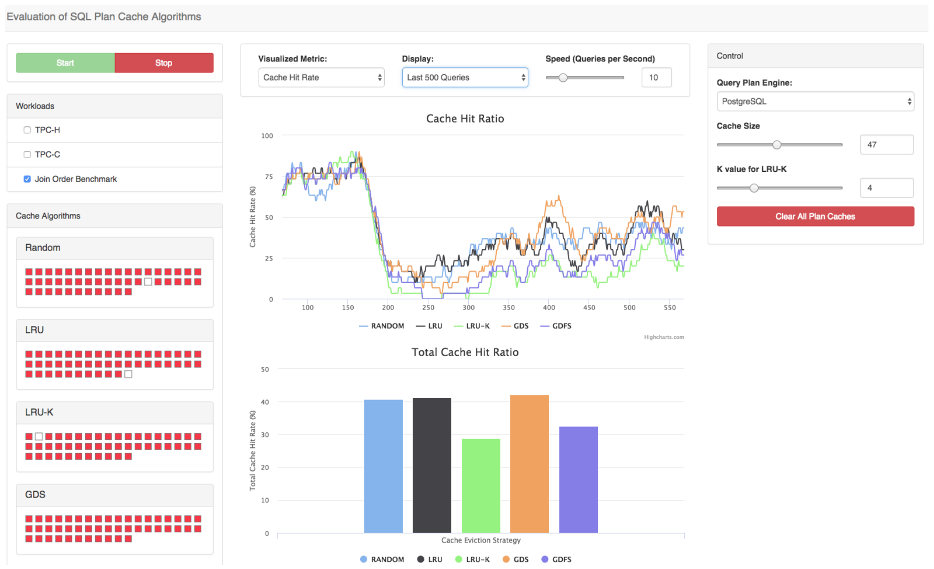
The importance of choosing a query plan that uses the available compute and memory resources efficiently can hardly be overstated. But while a good query plan can mean the difference between milliseconds and minutes of execution, finding it in the vast space of logically equivalent plans comes with its own costs. For workloads like the Join Order Benchmark [3], parsing and optimizing can take hundreds of milliseconds and in single cases even surpasses the execution time[1]. Luckily, this cost can be amortized if query plans are cached and reused [2].

Most commercial databases such as SAP HANA [1], Oracle 11g [5], and Microsoft SQL Server [4] use such a query cache. For SAP HANA, the cache consumes "a few GB out of a few TB of main memory to guarantee cache hit ratios above 99%" [1]. Others, such as Oracle BI 11g, use much smaller caches with only 1024 entries and warn users not to "raise this value without consulting Oracle Support Services" [5]. This serves to demonstrate why a careful selection of cache algorithms and their parameters is of relevance for the overall database performance. Surprisingly, only little academic research has been published that evaluates different cache implementations and eviction strategies.

We work on getting a better picture on how different cache implementations and eviction strategies influence the cache effectiveness and how this differs from workload to workload. To do so, we capture KPIs like the cache hit ratio and the overall time saved by the caches. When interpreting the results, we found it very helpful to have a visualization[2] that shows how changed parameters affect the KPIs. The traditional approach is to run a benchmark, modify the cache

[1] Query 28c takes roughly 600x longer to plan than to execute in PostgreSQL.
[2] https://www.dropbox.com/s/b4kdbmdfntatdse/

Accepted at Australasian Database Conference.
The final authenticated version is available online at https://dx.doi.org/10.1007/978-3-319-92013-9_31


Evaluation of SQL Plan Cache Algorithms

Workloads: TPC-H ☐, TPC-C ☐, Join Order Benchmark ☑

Visualized Metric: Cache Hit Rate

Display: Last 500 Queries

Speed (Queries per Second): 10

Query Plan Engine: PostgreSQL

Cache Size: 47

K value for LRU-K: 4

Fig. 1: Screenshot of the Cache Visualizer. The left side shows the different caches. Red blocks symbolize currently cached items. White blocks show which items are evicted. The main graph shows the performance of different compared cache algorithms. Parameters can be chosen on the right.

algorithm in the DBMS, and rerun the benchmark. This is not time-efficient and makes a live exploration of different influence factors impossible. Our presented system lets different cache algorithms compete against each other in real time. This allows the user, e.g., the DBA, to compare different algorithms, explore tuning factors and, thereby, tune the overall system performance.

A workload mix, chosen by the user, is assembled from a list of pre- or user-defined queries. It is then sent to our research DBMS Hyrise[3]. Currently, these queries include the TPC-C and -H as well as the Join Order[4] Benchmarks. Internally, we also work with a workload of a live ERP system. The system can be extended with further query sets. After assembling a workload mix, users can configure parameters that affect the caches. Most important is the cache size, which determines the number of queries that caches can hold. Algorithm-specific settings, such as K for LRU-K can also be chosen. Based on these, the KPIs of different caches are gathered and visualized in real-time as seen in Figure 1.

While a high *cache hit ratio* usually indicates a well-working cache, it ignores the different costs associated with generating the query plans. Algorithms can take this cost into account to make sure that an expensive query is not evicted by a query that is only slightly more frequent but significantly cheaper to optimize. Because of this, we found the *saved optimizer cost* to be a more accurate measure of a cache algorithm's performance and use it as the main KPI.

[3] `https://github.com/hyrise/hyrise`
[4] `https://github.com/gregrahn/join-order-benchmark`

Accepted at Australasian Database Conference.
The final authenticated version is available online at https://dx.doi.org/10.1007/978-3-319-92013-9_31

These costs depend on the implementation of the SQL parser, planner, and optimizer. What is a good configuration for one DBMS can be subpar for others. To account for this, our Cache Visualizer works with the optimizers in other DBMSs, currently MySQL and PostgreSQL. Queries are sent to these backends, the cost of optimizing is tracked and is then fed back into Hyrise's caches.

Since we are focusing on comparing cache algorithms and reducing the overall time spent in the optimizer, the plan execution time is out of scope. To increase the tool's throughput, plans are not executed. This is not a limitation of the architecture and could be enabled for tracking other components.

2 System Overview

Figure 2 shows the system's architecture. The selected queries and cache tuning parameters are sent to the Cache Visualizer's backend in Hyrise. Here, different caches are probed for an existing query plan. If a cache miss occurs, the query is planned using the selected DBMS's backend. Queries are only planned once, not repeatedly for each cache algorithm, because the time taken for planning is independent of the simultaneously evaluated cache algorithms. The result is then offered to the different caches. In the final step on the server side, the information on which cache had a miss and how long the required generation of the plan took is given back to the frontend. Here, it is aggregated and the metrics for the main graph are calculated.

Currently, our system provides five cache eviction strategies: LRU, LRU-K, GDS, GDFS, and a randomized strategy. Other strategies can be easily integrated by implementing an abstract cache interface.

In addition to the default metric *saved optimizer cost*, other metrics can be selected in the visualization options located above the graph. On the right, different DBMS backends and parameters for the caches can be selected.

The frontend interacts with Hyrise to provide additional information and options to the user. This includes the cache visualization seen on the left, where both the fill level and ongoing evictions are shown. In the future, more information such as the most recent use or the cost will be presented for each entry.

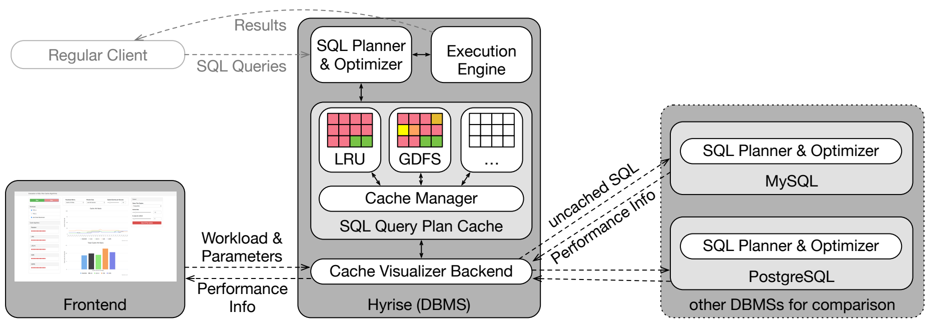


Fig. 2: Architecture of the Cache Visualizer.

Accepted at Australasian Database Conference.
The final authenticated version is available online at https://dx.doi.org/10.1007/978-3-319-92013-9_31

Hovering over an entry in the cache gives the stored query. Clicking on an entry will evict the entry manually.

3 Practical Use of the Tool and Demonstration

This tool makes it possible to compare the behavior of different plan cache algorithms depending on a number of parameters chosen by the user. Being able to see how they directly react to changes in the workload and the parameters allows researchers, database administrators, and developers to get a better understanding of how to tune a query plan cache. We are using this tool in ongoing research that looks at the cacheability of enterprise workloads. These differ from synthetic benchmarks in the number and complexity of tables and queries. In addition, the distribution of query executions rapidly changes over the workday which especially challenges cache algorithms.

The screenshot in Figure 1 allows a glance into the findings made possible. We can see how a change in the workload affects the different cache hit ratios. While GDS adapts quickly and takes only little time to reach the previous cache hit ratio, LRU-K takes longer because the wrong entries are chosen for eviction. Interestingly, the random eviction strategy performs quite well. This is a finding that can also be seen in the enterprise workload that we analyzed with this tool.

During the live demonstration, we will introduce the different configuration options available in the visualizer. After this, the audience will be able to interact with the tool, test different workload and parameter configurations, and see how the cache algorithms outperform each other. We expect that this leads to fruitful discussions on how to optimize databases' query plan caches and provoke additional questions that can be answered with the support of the tool.

References

[1] Norman May et al. "SAP HANA - The Evolution of an In-Memory DBMS from Pure OLAP Processing Towards Mixed Workloads". In: *Datenbanksysteme für Business, Technologie und Web (BTW)*. 2017.

[2] Gopi Krishna Attaluri and David Joseph Wisneski. *Method and system for transparently caching and reusing query execution plans efficiently.* US Patent 6,466,931. Oct. 2002.

[3] Viktor Leis et al. "How Good Are Query Optimizers, Really?" In: *Proceedings of the VLDB Endowment* (Nov. 2015).

[4] Microsoft. *Execution Plan Caching and Reuse.* URL: `https://technet.microsoft.com/en-us/library/ms181055(v=sql.105).aspx` (visited on 01/03/2018).

[5] Oracle. *Oracle Fusion Middleware System Administrator's Guide for Oracle Business Intelligence Enterprise Edition 11g.* URL: `https://docs.oracle.com/cd/E25178_01/bi.1111/e10541/configfileref.htm` (visited on 01/03/2018).