Accelerating Concurrent Workloads with CPU Cache Partitioning

Stefan Noll [#+*2], Jens Teubner [#+1], Norman May [*2], Alexander Böhm [*2]

[#] *Databases and Information Systems Group, TU Dortmund University, Germany*
[+] *Informatik Centrum Dortmund e.V., Germany*
[1] jens.teubner@cs.tu-dortmund.de

[*] *SAP SE, Germany*
[2] {stefan.noll,norman.may,alexander.boehm}@sap.com

***Abstract*—Modern microprocessors include a sophisticated hierarchy of caches to hide the latency of memory access and thereby speed up data processing. However, multiple cores within a processor usually share the same last-level cache. This can hurt performance, especially in concurrent workloads whenever a query suffers from cache pollution caused by another query running on the same socket.**

In this work, we confirm that this particularly holds true for the different operators of an in-memory DBMS: The throughput of cache-sensitive operators degrades by more than 50 %. To remedy this issue, we devise a cache allocation scheme from an empirical analysis of different operators and integrate a cache partitioning mechanism into the execution engine of a commercial DBMS. Finally, we demonstrate that our approach improves the overall system performance by up to 38 %.

I. Introduction

In recent years, hardware advances gave rise to modern main-memory *database management systems* (DBMS) [1]. By keeping data in main memory, these systems are no longer constrained by the traditional bottlenecks, i.e., disk I/O. Instead, memory bandwidth and access latency emerge as the new performance bottlenecks that systems need to address.

With all data in memory, this class of DBMS is also referred to as Operational Analytics Data Management Systems [2], allowing applications and users to concurrently execute transactional and analytical workloads on the same data set. As a result, concurrent queries usually have different resource requirements depending on the workload, the number of records accessed, as well as the data structures (i.e., indices) and algorithms being used. In particular, some workloads are highly sensitive to the available amount of CPU cache (e.g., random accesses to a small hash table), contrary to cache-insensitive operations such as a sequential scan of a large memory area.

Consider the example of the mixed workload illustrated in Figure 1. We executed an OLTP query either isolated, concurrently to an OLAP query, or concurrently to an OLAP query with cache partitioning applied. Our measurements show that the throughput of the OLTP query *degrades* significantly when *executed concurrently* to the OLAP one, because they compete for shared resources such as the processor's *last-level cache* (LLC). The OLAP query *pollutes* the cache by frequently accessing data from DRAM, thus evicting data from the cache needed by the OLTP query.

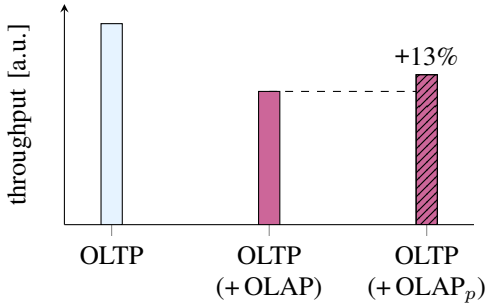


Fig. 1. Throughput of an OLTP query running either isolated, concurrently to an OLAP query, or concurrently to an OLAP query with cache partitioning (p) applied. Restricting the LLC for the OLAP query by partitioning the cache avoids cache pollution and improves performance of the OLTP query.

For system designers, the good news is that hardware manufacturers allow fine-grained control of cache allocation by offering mechanisms such as Intel's *Cache Allocation Technology* (CAT) [3] to avoid cache pollution. However, besides some isolated analysis of individual algorithms and data structures, it is still unclear how easy it is to apply these mechanisms in the context of real-world systems, and whether the corresponding integration effort pays off.

To fill this gap, we study in this work the impact of the CPU cache size on various query workloads and analyze the effect of careful cache partitioning on the overall system performance. To this end,

- *(i)* we empirically analyze the cache requirements of key DBMS operators varying available cache sizes;
- *(ii)* we derive cache partitioning schemes to reserve cache capacity for cache-sensitive operators and queries;
- *(iii)* we discuss how cache partitioning support can be retrofitted into an existing DBMS with low expenditure using SAP HANA as an example; and
- *(iv)* we evaluate the practical benefits of the proposed techniques using both standard benchmarks and queries from a modern HTAP business application (S/4 HANA).

We propose to integrate cache partitioning into a DBMS by restricting the LLC for scan-intensive operators causing cache pollution. We show that our approach can improve performance significantly without introducing regressions.

The rest of the paper is structured as follows. Section II introduces key data structures used by SAP HANA during query processing. In Section III, we describe our experimental set-up for the analysis, while we analyze the cache usage of key database operators in Section IV. In Section V, we discuss our approach of integrating cache allocation control into an existing system and derive cache partitioning schemes from our analysis. Subsequently, we evaluate our cache partitioning approach in Section VI. Finally, we discuss related work in Section VII and conclude in Section VIII.

II. Query Execution in SAP HANA

As a poster child for our analysis of cache usage, we use the in-memory database SAP HANA [4]. To better interpret the experiments described later in the paper, we give a brief overview of the most relevant implementation details of SAP HANA's query execution engine. In this section, we first present key data structures commonly used by different algorithms. In the subsequent section, we describe the database algorithms which we analyze in the evaluation regarding their cache usage.

The execution engine of SAP HANA uses tailor-made, cache-optimized data structures. For the scope of our current work, three data structures are most relevant: *(i) dictionaries*, which help to compress columnar data in SAP HANA, but also speed up value comparisons; *(ii) hash tables*, which are relevant for our current work regarding *aggregation with grouping*; and *(iii) bit vectors*, which accelerate the processing of *foreign key joins*. These data structures are used throughout SAP HANA's query processing engine (not just for the types of queries we study in this work).

Dictionaries play a significant role in the compression-optimized execution engine of SAP HANA. The ordered dictionary maps the domain values to a dense set of consecutive numbers. Instead of storing the actual value in the columns of a table, the storage engine of SAP HANA stores the typically much smaller number referencing an entry in the dictionary. In addition, each column can be further compressed using different compression methods. If data needs to be decompressed during query processing, *e.g.*, for projection or intermediate result construction, the dictionary is accessed frequently to look up the actual value.

Hash tables are a prominent example in the context of cache-sensitive data structures and operations. By nature, they are typically accessed in a random-access fashion, which can be very expensive when the hash table does not fit into the CPU caches. In SAP HANA, individual algorithms such as *grouped aggregation* use hash tables, *e.g.*, to store temporary results for different groups. They are used both locally per worker thread and globally to merge thread-local results. Characteristic is their very frequent access during query processing.

```
-- Query 1: Column Scan
SELECT COUNT(*) FROM A WHERE A.X > ?;

-- Query 2: Aggregation with Grouping
SELECT MAX(B.V), B.G FROM B GROUP BY B.G;

-- Query 3: Foreign Key Join
SELECT COUNT(*) FROM R, S WHERE R.P = S.F;
```

Fig. 2. The three SQL queries executed in the experimental analysis. Each query focuses on a specific database operator.

Bit vectors accelerate, *e.g.*, the evaluation of *foreign key joins* in the OLAP-optimized join algorithms of the execution engine of SAP HANA. The bit vectors map the primary key range to a highly compact representation, which can be kept in CPU caches even for a large key range. Using bit vectors is known to reduce memory loads and CPU cost since the CPU can perform the same operation on multiple elements of a bit vector at once [5], [6].

III. Micro-Benchmarks: Set-Up

To motivate why cache partitioning can improve the performance of concurrent workloads, we first study micro-benchmarks in order to analyze the cache usage of individual database algorithms. Our goal is to determine how much last-level cache a database operator needs to reach best performance and to determine how data distribution impacts the cache usage.

Furthermore, experimentally studying the cache characteristics of individual operators allows us to derive cache partitioning schemes for the concurrent execution of these operators. First, we present the experimental set-up of the analysis.

A. Queries

Since we want to see the effects of CPU caches on end-to-end performance, we express our benchmark queries on the SQL level and measure full query execution times. The three queries used in our experiments are listed in Figure 2. We keep the queries deliberately simple so that each query is dominated by a specific database operator: *(1) column scan*, *(2) aggregation with grouping*, and *(3) foreign key join*.

Column Scan: We execute Query 1 to analyze the performance of the *column scan* operator. The parameter "`?`" is used to vary the selectivity of the predicate. The operator sequentially reads an entire column of a table while evaluating a range predicate. The operator works on compressed data and uses SIMD instructions to process multiple encoded values at once, which significantly improves performance [7], [8].

Note that the *column scan* operator reads data from DRAM only once. It exploits data locality by processing each byte of a cache line. Thus, it profits from the hardware prefetcher, i.e., the CPU can load cache lines into the cache before they are requested. *Column scan* does not depend on any of the auxiliary data structures mentioned in the previous section.

```
-- Schema for Query 1
CREATE COLUMN TABLE A( X INT );

-- Schema for Query 2
CREATE COLUMN TABLE B( V INT, G INT );

-- Schema for Query 3
CREATE COLUMN TABLE R( P INT, PRIMARY KEY(P));
CREATE COLUMN TABLE S( F INT );
```

Fig. 3. The different SQL table schemata used in the experimental analysis.

Aggregation with Grouping: We use Query 2 to analyze the *aggregation with grouping* operator. The operator proceeds as follows. First, it distributes its input among a set of worker threads. Then, each worker thread collects aggregates locally for its partition. After all threads have finished aggregating, the algorithm merges the local results to build the global result for the next operator of the query plan.

The *aggregation with grouping* operator decompresses the input data to compute the aggregate [9]. As a result, the operator performs many random accesses to the dictionary. Furthermore, the algorithm uses hash tables to store intermediate, pre-aggregated results for every group as well as to store the merged results globally—similar to [10]. Thus, accessing the hash tables results in additional random memory accesses.

Foreign Key Join: By executing Query 3, we trigger the execution of the *foreign key join* operator of SAP HANA's execution engine. The join operator is optimized for OLAP workloads and exploits the fact that a foreign key maps to exactly one primary key.

In a first step, the join algorithm creates a very compact representation of the primary keys by mapping the keys to a bit vector. If the primary keys range from 1 to N, the algorithm creates a bit vector of length N and sets the i-th bit if the query's predicate evaluates to true for the row of primary key i. The resulting bit vector usually fits in the CPU caches even for a large number of keys. During the next step, the algorithm performs a look-up in the bit vector for each foreign key to check if it matches a primary key. In addition, it aggregates the matches.

B. Data Sets

Figure 3 illustrates the SQL schemata of the column tables used in the experiments. We fill the table with generated data (no null values) and vary the distribution of the data to study its impact on the cache usage of the operators.

Query 1 (Column Scan): The input data of Query 1 is a table consisting of one column with 10^9 integers. We randomly generate numbers between 1 and 10^6 with a uniform distribution. While the integers initially have a size of 32 bits, SAP HANA applies compression to store each integer using $\lceil \log_2(10^6) \rceil = 20$ bits. To vary the selectivity of the predicate, we set the parameter "?" to a random integer between 1 and 10^6 after every execution of the query.

Query 2 (Aggregation with Grouping): The input data of Query 2 is a table consisting of two columns with 10^9 integers. The first column `V` is used for aggregating while the second column `G` is used for grouping. We vary the number of distinct values by randomly picking integers from 1 to N. For column `V` we vary N between 10^6 and 10^8, which changes the size of the dictionary, and for column `G` we vary N between 10^2 and 10^6, which changes the number of groups and thus impacts the size of the hash tables used by the algorithm.

Query 3 (Foreign Key Join): The input data of Query 3 are two tables consisting of one column each. Column `P` of the first table contains distinct integers that form a primary key ranging from 1 to N. We vary N between 10^6 and 10^9, which impacts the number of matches and the size of the bit vector used by the join algorithm. Column `F` of the second table contains 10^9 integers referencing the primary key of the first table. We generate the foreign keys by randomly picking numbers from column `P`.

C. Hardware Platform

We perform the experiments with a prototype of SAP HANA running on a single socket system with $128\,\mathrm{GiB}$ of main memory. We use SUSE Linux Enterprise Server 12.1 as the operating system, but update the Linux kernel to version 4.10. The system features an Intel Xeon E5-2699 v4 processor with 22 cores. With simultaneous multithreading enabled, the processor can execute 44 threads in parallel.

Using the Intel Memory Latency Checker [11], we determine that DRAM has a memory read bandwidth of $64\,\mathrm{GB/s}$ and an access latency of $80\,\mathrm{ns}$. The shared L3 cache (LLC) has a size of $55\,\mathrm{MiB}$. It stores both data and instructions and is inclusive: This means the LLC contains all the information stored in the other caches of the hierarchy.

D. Measurement Method

To analyze the cache usage and to determine the performance impact of a smaller cache, we limit the size of the available LLC [3]. Thus, the entire instance of SAP HANA can only allocate data into a limited size of the LLC. We state the available size of the cache in MiB.

We execute SQL queries with SAP HANA and measure end-to-end response time, i.e., the total execution time including parsing, optimizing, query execution and result transfer. Note that we set the concurrency limit of a SQL statement to the number of physical cores of the system. Consequently, a query is potentially executed on all available cores of the processor.

As a result of the reduction in cache size in the experiments, we normalize a query's throughput to its maximum throughput using the entire cache. In addition, we measure the LLC hit ratio and the LLC misses per instruction using Intel's Processor Counter Monitor [12].

IV. Micro-Benchmarks: Results

In this section, we present the experimental analysis of the cache usage of isolated database algorithms. By varying the size of the LLC, we study the impact of the cache size on performance. Note that the results conceptually apply not only to SAP HANA but to any modern in-memory DBMS.

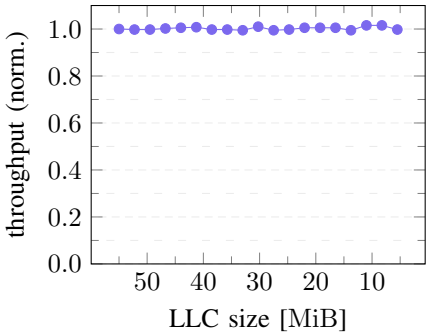


Fig. 4. Normalized throughput of Query 1 (*column scan* operator) at varying LLC sizes. The operator is hardly sensitive to the size of the cache.

First, we analyze the cache usage of the *column scan* operator. Afterwards, we focus on *aggregation with grouping*. Finally, we present the results of the *foreign key join* operator.

A. Query 1 (Column Scan)

Figure 4 shows the impact of the cache size on the throughput of the *column scan* operator. We observe that the throughput of Query 1 remains unaffected by the cache size. In addition, we measure that the LLC hit ratio is below 0.08, while the LLC misses per instruction amount to $1.9 \cdot 10^{-2}$, independent of the cache size. Thus, we conclude that the scan is not sensitive to the cache size.

The results do not come as a surprise because the *column scan* reads data from DRAM only once without reusing it. Moreover, the sequential memory access pattern of the scan operator features strong data locality. Therefore, it profits from the hardware prefetcher of the CPU.

Furthermore, the *column scan* operator takes advantage of the fact that it can process its input data without decompressing it first. This is possible because SAP HANA's dictionary encoding is *order preserving*. That is, it is sufficient to map the query parameter "`?`" to its dictionary code, then execute the query entirely on compressed data [7], [8]. During the *column scan*, the dictionary remains untouched and does not have to be cached.

B. Query 2 (Aggregation with Grouping)

The evaluation of the *aggregation with grouping* operator is split into three different experiments: We vary the number of distinct values in the column `B.V` to change the size of the dictionary to 4 MiB, 40 MiB and 400 MiB. Then, in each experiment we alter the number of groups in addition to changing the size of the LLC.

Dictionary Size of 4 MiB: Figure 5a shows the experimental results of *aggregation with grouping* using a data set with 10^6 distinct values in the column `B.V`. This results in a dictionary size of approximately 4 MiB. Thus, the dictionary fits completely in the LLC (55 MiB) but exceeds a single L2 cache (256 KiB).

The results show that for a group size of 10^2, 10^3 and 10^4 the throughput degrades as soon as the aggregation query is forced to use less than 20 MiB of the cache. In addition, we notice that the throughput degrades by more than 46 % if we limit the size of the cache to approximately 5 MiB. We observe the strongest throughput degradation with 10^5 groups: The curve breaks at a cache size of less than 40 MiB, resulting in a throughput degradation of 67 %.

If we increase the number of groups to 10^6, the throughput degrades less strongly. The throughput decreases by 28 % if we limit the cache size to 25 MiB. If we reduce the cache size even further, the throughput degrades by 46 %.

We explain the different performance impacts by the size of the hash table, which is decided by the number of groups. In case of 10^5 different groups, the hash table occupies all of the LLC. Thus, if we change the size of the available cache, we observe the most significant impact on performance.

If the number of groups is smaller, the hash table is so small that even a small portion of the cache is enough to store it entirely. If the number of groups is bigger, the size of the hash table exceeds the size of the LLC. As a result, the hash table does not completely fit in the cache and the algorithm suffers from cache misses even if the entire cache is used. Reducing the cache size results in an increasing number of cache misses and further degrades performance.

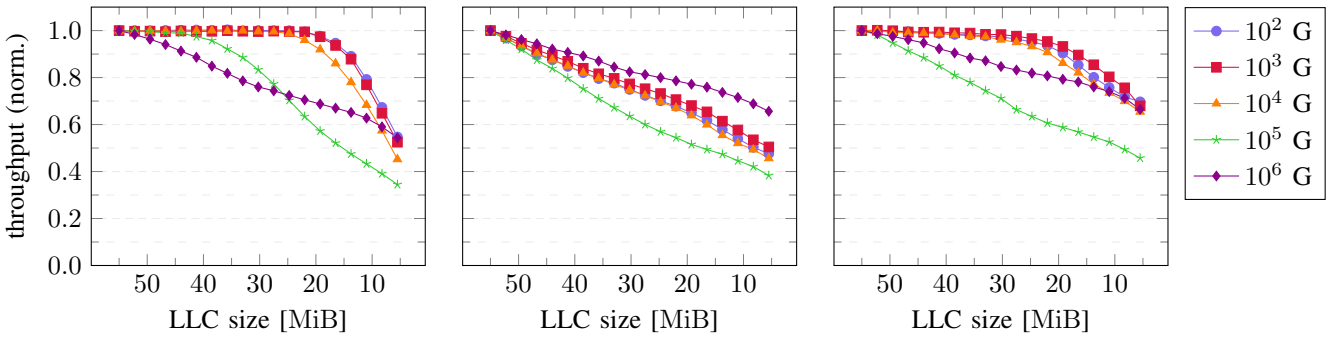
Accordingly, we measure that the LLC hit ratio as well as the LLC misses per instruction decline significantly between group sizes 10^2 to 10^5 and group size 10^6 when the size of the hash table exceeds the size of the LLC: the LLC hit ratio drops from more than 0.9 to less than 0.6, while the LLC misses per instruction increase by an order of magnitude.

Dictionary Size of 40 MiB: Figure 5b illustrates the second experimental results of executing Query 2 with varying cache sizes. In this experiment, we set the number of distinct values in the column `B.V` to 10^7. This results in a dictionary size of approximately 40 MiB. Thus, the dictionary can occupy a large portion of the LLC (55 MiB).

We observe that for group sizes 10^2 to 10^5 the throughput drops significantly by up to 62 % as we lower the size of the available LLC. In contrast, we observe that for the largest group size the impact on the performance is less significant. The results show that the throughput degrades by up to 34 %. Moreover, if we compare the results of this experiment to the previous one, we notice that the throughput degrades steadily for all group sizes—even for larger cache sizes.

We explain these results by the increased size of the dictionary. In contrast to the first experiment, the dictionary has a size of more than half of the LLC. During the aggregation, the algorithm performs lots of random memory accesses to the dictionary to decompress the encoded values of the column before aggregating them. Therefore, the execution time of the operator is dominated by accesses to DRAM, as soon as the dictionary size exceeds the LLC size. The results show that the dictionary cannot be held in the LLC either if the number of groups is large (10^6) or if we limit the size of available cache to less than 45 MiB.

Dictionary Size of 400 MiB: Figure 5c displays the third set of experimental results, where we set the number of distinct values in the column `B.V` to 10^8. The configuration results



(a) 4 MiB Dictionary: The operator is slightly sensitive to the size of the cache for smaller groups and highly sensitive for larger groups.

(b) 40 MiB Dictionary: The operator is highly sensitive to the size of the cache for all group sizes.

(c) 400 MiB Dictionary: The operator is slightly sensitive to the size of the cache for smaller groups and increasingly sensitive for larger groups.

Fig. 5. Normalized throughput of Query 2 (*aggregation with grouping* operator) at varying LLC sizes. We set the dictionary size of the column that is aggregated to 4 MiB (a), 40 MiB (b) and 400 MiB (c) and vary the number of groups (G).

in a dictionary size of 400 MiB. The size of the dictionary exceeds the size of the LLC (55 MiB) by a factor of more than 7.

The experimental results reveal that the cache size impacts the throughput less compared to the second experiment. We observe that, as we limit the size of the cache, the throughput degrades by more than 31 %. For group sizes 10^2 to 10^4, the curve breaks at a cache size of less than 30 MiB, while for a group size of 10^6 the curve breaks earlier at a cache size of less than 50 MiB. If the algorithm aggregates over 10^5 groups, the throughput degrades by up to 54 %.

The experimental results are similar to the results of the first experiment (Figure 5a) if we compare where the curves break. The reason for this behavior is again the increasing size of the hash table. This time, however, the dictionary exceeds by far the size of the LLC. Consequently, the algorithm suffers from lots of cache misses. We notice that compared to the first experiment, the cache hit ratio drops by at least 10 % to 20 %. Consequently, the overall performance degradation is less in comparison to the first experiment (Figure 5a). However, the size of the hash tables still impacts the cache sensitivity of the algorithm significantly.

C. Query 3 (Foreign Key Join)

Figure 6 illustrates the throughput of Query 3 with varying cache sizes. The query triggers the execution of the OLAP-optimized *foreign key join* operator, which uses a bit vector to represent the primary keys. The results show that the throughput of the join algorithm worsens by only 5–14 % for 10^6, 10^7 and 10^9 primary keys. For 10^8 primary keys, however, the throughput degrades by up to 33 %. We attribute the cache sensitivity of the algorithm to the size of the bit vector used to store the primary keys. To store 10^8 distinct keys ranging from 1 to 10^8, the algorithm uses a bit vector with a size of 10^8 bit = 11.92 MiB. Thus, the bit vector easily fits in the LLC. In all other cases the bit vector either exceeds the LLC or (almost) fits in the L2 cache.

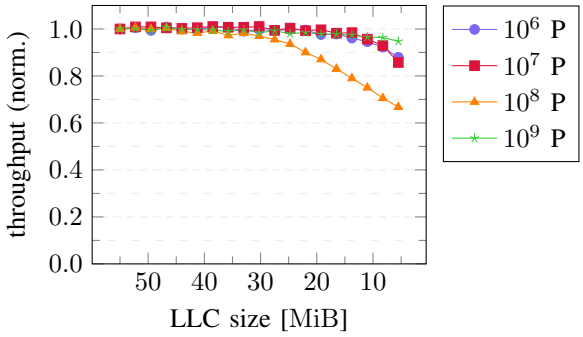


Fig. 6. Normalized throughput of Query 3 (*foreign key join* operator) at varying LLC sizes. We vary the number of primary keys (P). The operator is sensitive to the size of the cache only for 10^8 primary keys when the size of the bit vector is comparable to the size of the LLC.

D. Discussion

Our measurements show that the *column scan* operator is hardly sensitive to the size of the cache. *Column scans* do not benefit from a large portion of the LLC and run well with a small cache configuration (e.g., 10 %). This observation does not come as a surprise because, by nature, scans read data exactly once from DRAM without any data re-use.

Aggregations, by contrast, can be highly sensitive to the size of the cache. The *aggregation with grouping* operator that we consider is based on hashing, and is most cache-sensitive whenever the size of the hash tables is comparable to the (configured) LLC size. If the hash table is either very small or very large, cache sensitivity becomes less significant. In this regard, our observations are consistent with the findings of Lee *et al.* [13] who contrasted cache demand and cache usage within PostgreSQL.

In addition, we ran experiments for a *join* query with an OLAP-optimized join algorithm of SAP HANA. The *foreign key join* algorithm's cache sensitivity depends on the cardinality of the primary keys: If the size of the bit vector is comparable to the size of the LLC size, the operator becomes cache-sensitive. Otherwise the operator does not use the LLC.

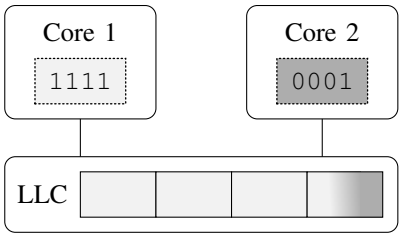


Fig. 7. Simplified example of using Intel Cache Allocation Technology to partition the LLC. A bitmask controls the cache allocation of each core. The first core can evict cache lines from the entire LLC, while the second core can only evict 25 % of the LLC and shares its portion with the first core.

Our results suggest that scan-intensive operators, such as the *column scan* operator or the *foreign key join* operator (for a small bit vector), cause cache pollution for cache-sensitive operators, such as the *aggregation with grouping* operator, if they are executed concurrently. The awareness of these different cache usage characteristics allows us to classify database operators based on their cache usage and use *cache partitioning* to manage the shared LLC more efficiently for concurrent workloads, which we study in the following.

In fact, we validate whether the simple approach of restricting scan-intensive operators to a minimum portion of the cache—while allowing a cache-sensitive operator to use the entire cache—can avoid *cache pollution* for concurrent workloads and improve performance.

V. Cache Partitioning in SAP HANA

In this section, we present the cache partitioning feature of current Intel processors and we describe how we implement cache partitioning in the execution engine of a prototype version of SAP HANA. By integrating cache partitioning into a database system, we can avoid cache pollution and thereby improve the performance of concurrent query execution.

A. Cache Partitioning with Cache Allocation Technology

Traditionally, the user has little control over the cache, as it is entirely managed by hardware. Techniques such as *page coloring* [13] offer the possibility of partitioning the cache by allocating memory in specific memory pages, known to map to a specific portion of the cache. However, their use in commercial systems is limited. Page coloring requires significant changes to the Linux kernel and to the application, resulting in poor usability and maintainability. In addition, page coloring is less flexible because re-partitioning the cache dynamically at runtime requires copying the allocated data [14], [15].

With the Haswell microarchitecture Intel introduced the possibility to partition the *last-level cache* of a processor (not L2 or L1), thereby giving the user more control over the CPU cache. Intel refers to this hardware feature as *Cache Allocation Technology* (CAT) [3]. It allows the user or the operating system to dynamically control from which portion of the last-level cache an individual (logical) core can evict a cache line in order to replace it with a new one. Figure 7 illustrates the feature using a simplified example.

The user partitions the cache by writing a bitmask of N bits in a specific processor register of a core, where N depends on the processor model. Setting the bit at the i-th position of the bitmask means that the core can evict cache lines from the i-th portion of the last-level cache, while un-setting the bit at the i-th position means that the core never evicts cache lines from the i-th portion of the cache. By choosing distinct bitmasks, the user allows cores to evict portions of the cache exclusively. Bitmasks can be dynamically changed at run time.

To illustrate, the Intel Xeon E5-2699 v4 processor used in our experiments has a 20-way associative LLC with a size of 55 MiB. The bitmask for controlling the cache partitioning feature has a size of 20 bits. As a result, one portion of the cache equals 55 MiB / 20 = 2.75 MiB. This means that setting, e.g., two bits in the bitmask, corresponds to a portion with a size of 5.5 MiB. Note that the processor allows up to 16 different bitmasks to be active at the same time.

The Linux kernel supports CAT since version 4.10 [16]. The extension allows the user to specify each core's bitmask used for cache partitioning by reading and writing to the pseudo file system *sysfs*. Furthermore, instead of specifying a bitmask for a core, the user has the option to specify a bitmask for a *process id* (PID) or a *thread id* (TID). This allows mapping a portion of the cache to an individual process or thread. During a context switch, the scheduler of the kernel is responsible for updating the bitmask of the core on which the process or thread is currently running.

B. Cache Partitioning Scheme

To avoid cache pollution and to improve the performance of concurrent workloads by partitioning the cache, we need to decide *how much* cache we need to allocate to an operator or a query, respectively. In fact, we can derive a cache partitioning scheme from the results of Section IV.

Query 1: Figure 4 showed that the performance of Query 1 does not depend on the size of the cache because the *column scan* operator does not re-use data and does not need to access the dictionary. However, the scan evicts lots of cache lines by continuously loading data from DRAM. Thus, we conclude that *column scans* will cause cache pollution for co-running queries. To avoid cache pollution, we give the *column scan* operator the smallest amount of cache (without reducing performance): 10 % using the bitmask "`0x3`".

Query 2: Figure 5 illustrated how the *aggregation with grouping* operator can be highly sensitive to the size of the cache, because it accesses the dictionary and the hash table frequently. Thus, we do not restrict the access to the LLC for the *aggregation with grouping* operator using the bitmask "`0xfffff`".

Query 3: Figure 6 demonstrated that the *foreign key join* operator is sensitive to the size of the cache depending on the cardinality of the primary keys: It causes cache pollution if the size of the bit vector is not comparable to the size of the cache, otherwise it becomes cache-sensitive.

Consequently, we restrict the *foreign key join* operator to 10 % of the LLC using the bitmask "`0x3`" in the first case

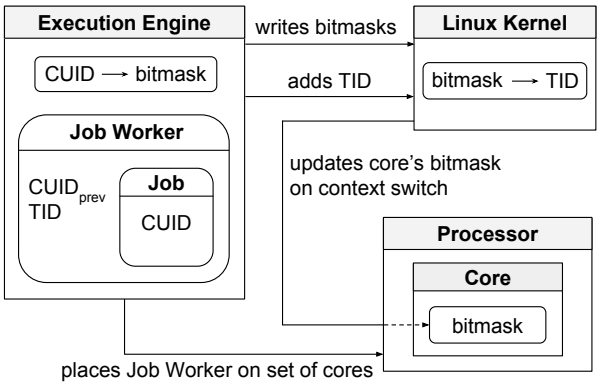


Fig. 8. Schematic overview of the interaction between the execution engine of SAP HANA, the Linux kernel and the processor. The execution engine maps *jobs* to cache partitions by associating the *cache usage identifier* (CUID) of a *job* with a *bitmask*. Then, it passes the *thread id* (TID) of the *job worker* and the *bitmask* to the kernel, which interacts with the processor to partition the cache.

and to $60\,\%$ (throughput degrades below $35\,\mathrm{MiB}$) using the bitmask "`0xfff`" in the other case. As a simple heuristic, we decide based on the size of the bit vector whether the operator is cache-sensitive or not.

Note that we also evaluated using the bitmask "`0x1`" to restrict a scan-intensive operator. We observed, however, that this configuration degrades performance severely (not shown in Figure 4 to 6)—even for Query 1. We explain this behavior with the current implementation of CAT (restricting access to certain ways for a N-way set associative cache) which may result in contention.

C. Integration into SAP HANA

We decided to use the Linux kernel interface of CAT to integrate cache partitioning into the execution engine of a prototype version of SAP HANA. Directly using the hardware interface would require to either pin threads to cores with specific bitmasks or to track which thread is running on which core and to manually update a core's bitmask upon thread migration. This would limit flexibility especially for changing workloads. A schematic overview of how we integrate cache partitioning is illustrated in Figure 8.

The execution engine of SAP HANA uses a thread pool of worker threads called *job workers* to execute *jobs* [17]. A job encapsulates a single operator or—together in a group of jobs—a parallelized operator. Thus, a job represents one operator at the maximum. Therefore, we implement cache partitioning for jobs to enable cache optimizations *per operator*—similar to existing NUMA optimizations [17].

In fact, we annotate a job with information of its *cache usage* by associating it with an *identifier* (CUID). We currently distinguish between three categories: *(i)* jobs which are not cache-sensitive and pollute the cache such as the *column scan*; *(ii)* jobs which are cache-sensitive and profit from the entire cache such as the *aggregation with grouping* operator for most cases; and *(iii)* jobs such as the *foreign key join* operator which can be both cache-polluting and cache-sensitive depending on the query or data. By default, a job belongs to *(ii)* to avoid regressions.

The execution engine maps the CUID to a bitmask, following the heuristics described in Section V-B ("`0x3`" for *(i)*; "`0xfffff`" for *(ii)*; and "`0x3`" or "`0xfff`" for *(iii)*), then passes the bitmask to the Linux kernel.

Interacting with the Linux kernel to associate a thread with a new CAT bitmask might incur an execution time overhead. Therefore, our implementation always compares old and new bitmasks and only associates a TID with a new bitmask if really necessary. In practice, however, the overhead is negligible at least for OLAP scenarios. We benchmarked our test system and measured an overhead of less than $100\mu\mathrm{s}$.

If at all, only short-running OLTP queries might see a small performance penalty due to the interaction with the kernel. However, SAP HANA handles such queries in a dedicated thread pool anyway. That thread pool always has access to the entire cache.

VI. Experimental Evaluation

To evaluate the integration of cache partitioning into the database system, we use the same hardware platform that was introduced in Section III-C with a prototype based on the SAP HANA code base. We implemented the cache partitioning feature in the execution engine as described in Section V-C.

We perform the following experiments: We start by executing Query 1 and Query 2, and Query 2 and Query 3 (cf. Figure 2) concurrently using the same data sets as described in Section III-B. We compare the performance with and without using cache partitioning. Afterwards, we evaluate the cache partitioning feature using the TPC-H benchmark and a mixed workload with a query extracted from a real-world SAP S/4HANA application.

A. Setup

TPC-H: We run each TPC-H query concurrently with Query 1 (column scan) using a generated data set of scale factor 100. Our goal is to study how a scan-intensive OLAP query (column scan), which causes cache pollution, impacts the performance of the individual TPC-H queries and how cache partitioning can improve performance.

S/4HANA Workload: S/4HANA is an enterprise resource planning application commercialized by SAP. The "Universal Journal Entry Line Items" table ACDOCA is one of the central data stores for processing core financial aspects, and is heavily used in both OLTP and OLAP query processing. Reflecting complex business logic, ACDOCA is a wide table with 336 attributes of type `NVARCHAR` (285) or `DECIMAL` (51). The instance of ACDOCA used in our experiments has 151 million rows and was extracted from a real customer system together with the most frequent OLTP query—executed more than 10 million times a week.

In each experiment, we execute all queries repeatedly for 90 seconds. This assures that each query is affected by another query for the same time. For each query, we report the

throughput of the query, when running concurrently to another query, normalized to the throughput of the query when running in isolation. If not stated otherwise, we restrict the query causing cache pollution such as the *column scan* to 10 % of the cache, while the other query can access the entire cache. Note that we tune the system for best throughput: The memory-intensive workloads are limited by memory bandwidth.

B. Query 1 (Column Scan) & Query 2 (Aggregation)

Figure 9 illustrates how the throughput of Query 1 and Query 2 degrades when executed concurrently. We set the dictionary size of the data set for Query 2 to 4 MiB, 40 MiB, and 400 MiB. Then, in each experiment we vary the number of groups. In addition, we evaluate if partitioning the LLC improves throughput: we allocate 10 % of the cache to Query 1 and 100 % to Query 2.

Dictionary Size of 4 MiB: We observe that the throughput degrades with increasing group sizes. If we increase the group size from 10^4 to 10^5, the throughput of the *aggregation* query drops from 80 % to 66 %. If we increase the group size from 10^5 to 10^6, the throughput of the scan query drops from 89 % to 69 %. Note that, as the number of groups increases, the *aggregation* algorithm uses larger hash tables to store temporary results. This heavily impacts the cache usage of the *aggregation*. Up to a group size of 10^4, the hash table has the size of only a fraction of the LLC. It mostly fits in the L2 cache. Thus, the *aggregation* is not sensitive to the capacity of the LLC and cache pollution is not a problem.

If we increase the number of groups to 10^5, the size of the hash table is comparable to the size of the LLC. The *column scan* evicts an increased number of cache lines used by the *aggregation*, thereby causing cache pollution. When aggregating over 10^6 groups, the situation changes, as the size of the hash table exceeds the size of the LLC. The *aggregation* query performs more DRAM accesses and uses more memory bandwidth. As a result, both queries increasingly compete for memory bandwidth, which explains why the throughput of the scan query degrades more strongly.

The results show that enabling the cache partitioning feature of the execution engine significantly improves performance of the *aggregation* query when the *aggregation* algorithm becomes sensitive to the capacity of the LLC (for group size of 10^5). By giving the *aggregation* the entire cache and the *column scan* only a small portion of the cache, we improve throughput by 20 %. At the same time, the throughput of the *column scan* improves by 3 %.

Note that the performance improvement correlates with other metrics, which we collected by sampling hardware performance counters for the entire system: The cache hit ratio increases from 0.78 to 0.82, while the LLC misses per instruction improve from $2.86 \cdot 10^{-3}$ to $2.32 \cdot 10^{-3}$. Thus, partitioning the cache avoids cache pollution and improves the overall cache efficiency of the workload.

Dictionary Size of 40 MiB: The results show that the throughput of the *aggregation* query drops below 60 % for up to 10^5 groups. At the same time, the throughput of the *column scan* query drops to 84 %. If we increase the number of groups from 10^5 to 10^6, the throughput of the *aggregation* degrades less, but the throughput of the *column scan* degrades more. By utilizing cache partitioning, we can improve the throughput of the *aggregation* query by up to 21 %. At the same time, the throughput of the *column scan* is improved by up to 6 %. Reserving 90 % of the cache exclusively for the *aggregation* query allows the entire dictionary to be kept in the cache, as long as the hash table does not exceed the size of the LLC (i.e., up to 10^5 groups). Otherwise, the dictionary and the hash table compete for cache capacity.

We determine that the overall cache hit ratio increases and that the LLC misses per instruction decrease because the *aggregation* has to perform fewer accesses to main memory. In addition, the *column scan* gets more memory bandwidth. By partitioning the cache, the database system uses hardware resources more efficiently and executes both queries faster.

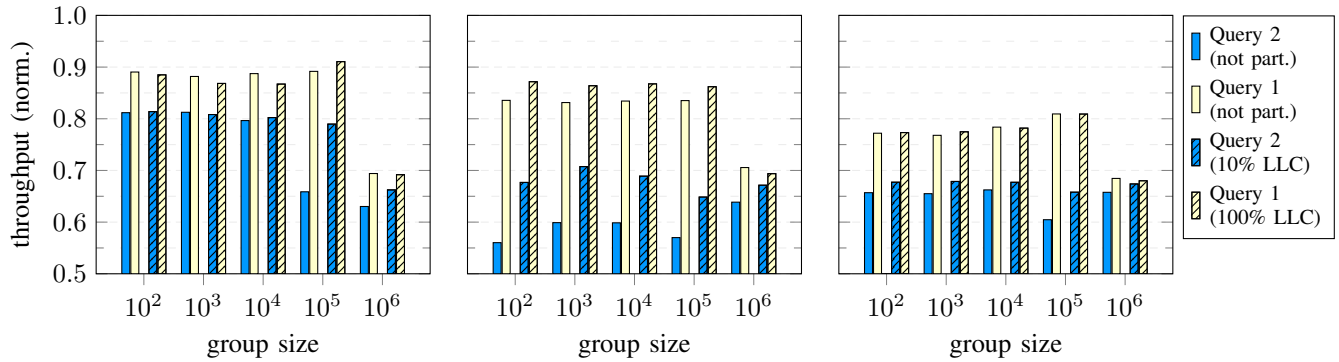
Dictionary Size of 400 MiB: We observe that, when the dictionary is several times larger than the cache, the throughput of the *aggregation* query decreases to 60–66 %. At the same time, the throughput of the *column scan* query decreases to 68–81 %, which is more significant than in the previous two experiments. This illustrates that both queries compete less for the LLC but more for memory bandwidth. The dictionary and the hash table cannot be kept in the cache at the same time. Thus, the *aggregation* algorithm performs more memory accesses to DRAM—independent of the group size. It consumes more memory bandwidth and impacts the *column scan* query more strongly. At the same time, the *aggregation* is less sensitive to the cache size, which explains why cache partitioning improves the throughput of the *aggregation* query only by 3–9 %.

C. Query 2 (Aggregation) & Query 3 (Join)

Figure 10 illustrates how the throughput of Query 2 and Query 3 degrades if they are executed concurrently. We change the number of primary keys in the data set for Query 3 to 10^6 and 10^8 to change the size of the bit vector used by the *foreign key join* algorithm. Then, in each experiment, we vary the number of groups for Query 2. In addition, we evaluate two cache partitioning schemes: Fist, we restrict Query 1 to 10 % of the LLC. Second, we restrict Query 1 to 60 % of the LLC. We allocate 100 % of the LLC to Query 2.

10^6 Primary Keys: We observe that for group sizes below 10^6 the throughput of the *aggregation* query drops to 41 %, while the throughput of the *join* query drops to 63 %. If we increase the group size to 10^6, the throughput of the *aggregation* query decreases to 51 %. The results match previous results presented in Section VI-B, albeit in this workload the performance of both queries suffers more.

The results show that enabling cache partitioning improves the throughput of the *aggregation* query by up to 38 %. At the same time, the throughput of the *join* query improves by up to 7 %. Note that the cache hit ratio increases from 0.55 to 0.67, while the LLC misses per instruction improve from $2.26 \cdot 10^{-3}$ to $1.93 \cdot 10^{-3}$ for, e.g., a group size of 10^3. Thus,

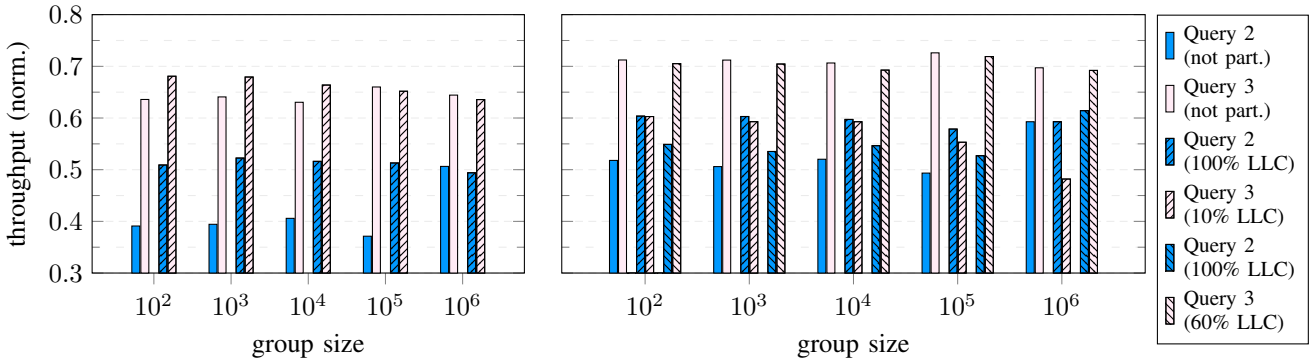


(a) 4 MiB Dictionary: Throughput degrades with increasing group sizes. For 10^5 groups cache partitioning significantly improves throughput.

(b) 40 MiB Dictionary: Throughput of Query 2 degrades significantly. Cache partitioning improves throughput by up to 21 %.

(c) 400 MiB Dictionary: Throughput of both queries degrades significantly. Cache partitioning improves throughput by up to 9 %.

Fig. 9. Normalized throughput of Query 1 (*column scan* operator) and Query 2 (*aggregation with grouping* operator) when executed concurrently. We set the dictionary size of the column that is aggregated to 4 MiB (a), 40 MiB (b) and 400 MiB (c), vary the number of groups (G), and disable or enable cache partitioning. We restrict Query 1 to 10 % of the LLC and allow Query 2 to access 100 %.



(a) 10^6 Primary Keys: Restricting Query 3 to 10 % of the LLC improves throughput by up to 38 %.

(b) 10^8 Primary Keys: Restricting Query 3 to 10 % of the LLC does *not* improve performance, while restricting Query 3 to 60 % improves total throughput by up to 8 %.

Fig. 10. Normalized throughput of Query 2 (*aggregation with grouping* operator) and Query 3 (*foreign key join* operator) when executed concurrently. We set the dictionary size of the column that is aggregated to 40 MiB, the number of primary keys to 10^6 (a) or 10^8 (b), vary the number of groups (G), and disable or enable cache partitioning. We restrict Query 3 to either 10 % or 60 % of the LLC and allow Query 2 to access 100 %.

we conclude that partitioning the cache improves the overall cache efficiency of the workload. If the group size is 10^6, both operators are only limited by the memory bandwidth and not by LLC contention. Consequently, partitioning the cache does not improve performance.

10^8 Primary Keys: If we choose the data set with 10^8 primary keys, we observe that the throughput of the *aggregation* query drops to 49 %, while the throughput of the *join* query drops to 70 %. By partitioning the cache with the configuration that gives Query 2 the entire cache and Query 3 only 10 % of the cache, we improve the throughput of the *aggregation* query by up to 19 %. However, the throughput of the *join* query worsens by 15–31 %. Based on the combined throughput, we lose more performance than we gain from applying this cache partitioning scheme.

This observation is consistent with the results from Section IV-C, where we learned that the throughput of the *join* query decreases if the cache size falls below 35 MiB. Thus, we need to partition the cache differently. We allow the *aggregation* query to evict cache lines from the entire cache, while we restrict the *join* query to 60 % of the cache. This means that we allocate 40 % exclusively to the *aggregation* query, while 60 % of the cache is shared between both queries. We prioritize Query 2 over Query 3 because it needs more cache and suffers more from cache conflicts.

The results show that this configuration improves the throughput of the *aggregation* query by up to 9 %. At the same time, the throughput of the *join* query varies around 2 %. When we consider the combined throughput, the second cache configuration improves the overall performance of the workload.

D. Query 1 (Column Scan) & TPC-H

Figure 11 illustrates how cache pollution caused by *column scans* (Query 1) impacts the throughput of each TPC-H query. In addition, we study whether our policy of limiting the *column scan* operator (to 10 % of the cache) improves the performance of the workload.

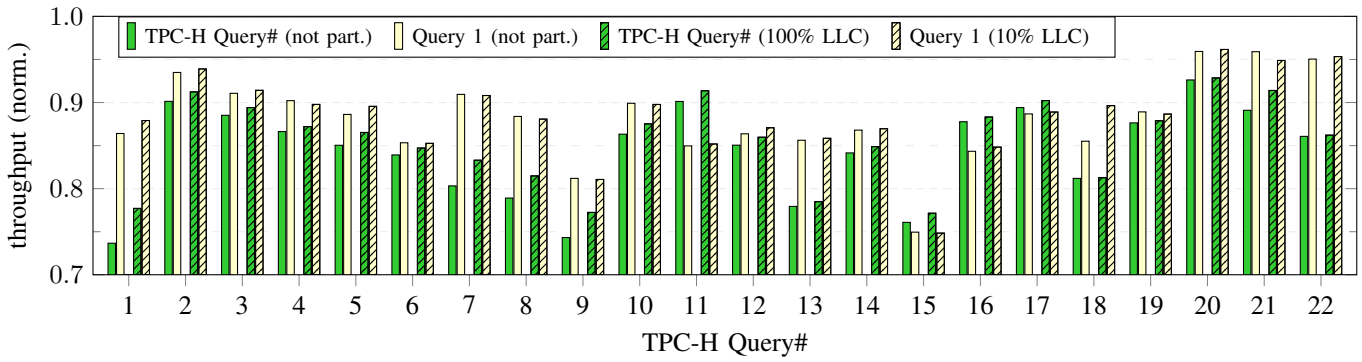


Fig. 11. Normalized throughput of Query 1 (*column scan* operator) and each TPC-H query when executed concurrently. We disable or enable cache partitioning. We restrict Query 1 to 10 % of the LLC and allow each TPC-H query to access 100 %. Cache pollution caused by Query 1 impacts especially TPC-H queries 1, 7, 8 and 9, while TPC-H queries 10 to 22 suffer less from cache pollution. Partitioning the cache improves throughput by up to 5 %.

We observe that the impact of a co-running query varies significantly with the TPC-H queries. The throughput of the TPC-H queries degrades to 74–93 %, while the throughput of the *column scan* query degrades to 65–96 %. If we enable cache partitioning, we improve the throughput of the TPC-H queries by up to 5 %. The results show that for TPC-H Queries 1, 7, 8 and 9 the cache partitioning approach improves the overall performance of the workload. For other queries the improvements are less noticeable.

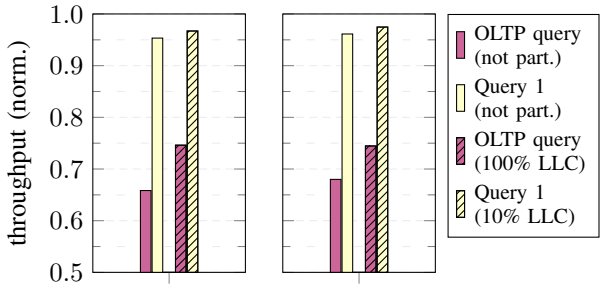
This shows that the performance of only some queries depends on the LLC. That is because the columns of the TPC-H data, which are aggregated, feature comparatively small dictionaries. Furthermore, grouping usually uses only a relatively small number of groups. Thus, most of the frequently accessed data structures are small enough to fit in L2 caches or in a small portion of the LLC.

One exception is the column "*L_EXTENDEDPRICE*" with a dictionary size of approximately 29 MiB, which is frequently accessed during the execution of, *e.g.*, TPC-H Query 1. The query aggregates the column causing lots of accesses to the dictionary, which explains why reducing cache pollution through cache partitioning improves its performance.

Interestingly, the avoidance of cache pollution sometimes reflects back on the *column scan* operator. Faced with fewer cache misses, the co-running TPC-H query takes away less bandwidth from the bandwidth-sensitive scan, resulting in a throughput increase of up to 5 % for the *column scan* query (e.g., with TPC-H Query 18 co-running).

E. OLAP & OLTP

Figure 12 presents the throughput of Query 1 and an OLTP query from the S/4HANA workload when executed concurrently. We use a modified version of the OLTP query, which contains a projection to 13 columns featuring the biggest dictionaries of the table (Figure 12a), and the unmodified query, which contains a projection to 6 different columns featuring smaller dictionaries (Figure 12b). The results show that the throughput of the OLTP query drops to 66 % and 68 %, respectively, while the throughput of Query 1 decreases to only 95 % and 96 %, respectively. Restricting Query 1 to 10 % of the cache improves the throughput of the OLTP query by 13 % and 9 %, respectively.



(a) Projection of 13 columns (b) Projection of 6 columns

Fig. 12. Normalized throughput of Query 1 (*column scan* operator) and an OLTP query from the S/4HANA workload running concurrently. We set the number of projected columns to 13 (a) and 6 (b). We restrict Query 1 to 10 % of the LLC and allow the OLTP query to access 100 %. Cache partitioning improves the throughput of the OLTP query by 13 % and 9 %, respectively.

During the processing of the OLTP query, the engine accesses the inverted index of five columns that are part of a primary key. Afterwards, it projects the selected rows to 13 (or 6) columns causing accesses to the dictionaries of the columns. We argue that, for a larger number of projected columns, cache partitioning improves the performance more significantly because more dictionaries need to be kept in the cache to avoid cache misses.

An additional experiment (not shown here) further demonstrates that the size of the working set, i.e. the size of the dictionaries and indices, affects the cache-sensitivity of the OLTP query: We varied the number of projected columns from 2 to 13 (featuring the biggest dictionaries) and observed that the throughput degrades with an increasing number of projected columns. At the same time, restricting Query 1 to 10 % of the cache improves the throughput of the OLTP query by 8 % to 13 %.

F. Discussion

Our evaluation confirms that *aggregations* are sensitive to *cache pollution* either caused by *column scans* or *joins*. Ag-

gregations are most sensitive to cache pollution whenever the size of their performance-critical data structures is comparable to the size of the LLC. While the *column scan* operator always pollutes the cache because it continuously evicts cache lines and does not re-use data, the *join* operator only causes cache pollution whenever its frequently accessed data structures fit in the L2 cache. In these cases, we can eliminate cache pollution and significantly improve performance by restricting the *column scan* or *join* to a small portion ($10\,\%$) of the LLC. In addition, the *column scan* operator profits from the fact that aggregation consumes less memory bandwidth: The throughput of the *column scan* increases, too.

If the size of the data structures used by a join is comparable to the size of the LLC, the *aggregation with grouping* and the *join* operators compete for cache capacity. Thus, we evaluate a different cache partitioning scheme: We restrict the *join* to $60\,\%$ of the cache, but observe that performance improves only slightly. Generally, the search for the "best" partitioning in any given situation will depend on accurate *result size estimates*.

Furthermore, we performed experiments using the *TPC-H benchmark* to evaluate whether limiting *scans* improves the performance of OLAP workloads. Our measurements show that the performance of TPC-H Queries 1, 7, 8 and 9 improves from partitioning the cache because these queries frequently access a column with a large dictionary. The performance of the other queries, however, does not improve noticeably. This demonstrates that not all queries are sensitive to cache pollution: The size of the working set, affecting, e.g, dictionary and hash table sizes, determines if the performance of an operator depends on the size of the available LLC.

Finally, we ran experiments with the *column scan* (OLAP) and an OLTP query from a real-world application. The results demonstrate that the performance of the OLTP query degrades significantly in the base configuration. This is because OLTP queries tend to use dictionaries aggressively, which the OLAP query evicts from the cache. By using cache partitioning to restrict the OLAP query to a small portion ($10\,\%$) of the cache, we avoid the eviction of dictionaries and improve the performance of the OLTP query.

The results illustrate that our simple approach for avoiding *cache pollution* is effective in improving overall system throughput. Thus, we propose to restrict scan-intensive operators which do not profit from using the LLC, such as the *column scan*, to a minimum portion of the cache without reducing performance. This approach has the advantage that it can improve the performance of *any* concurrent workload containing a scan-intensive operator. In addition, it does not depend on further knowledge of the workload.

VII. Related Work

A plethora of research efforts from the database community focuses on developing and optimizing *cache-aware* database algorithms, which exploit the cache hierarchy of modern computer systems [18], [19], [20]. They are finely tuned to current processors and usually depend on setting the correct hardware parameters. Hence, others propose *cache-oblivious* algorithms which achieve cache efficiency independent of specific hardware parameters [21], [22], [23], [24]. However, by design, both groups of algorithms are sensitive to the cache and thus sensitive to cache pollution. Thus, we expect our approach to benefit both groups of algorithms.

Plenty of work from the systems community focuses on cache partitioning based on *page coloring* [25], [26], [15]. The goal is to statically or dynamically partition the cache among competing threads to improve resource utilization or guarantee quality of service. Among others, Soares *et al.* [27] specifically aim to avoid *cache pollution*. They propose a dynamic mechanism, which first characterizes an application's cache behavior using hardware performance counters. Then, it maps the memory pages of applications with high cache miss rates to dedicated cache sets to avoid polluting memory pages of an application with low cache miss rates.

Lee *et al.* [13] build on the results from the systems community and present a method for minimizing last-level cache conflicts for PostgreSQL. They demonstrate a cache allocation mechanism based on *page coloring* to avoid cache capacity conflicts and classify queries based on their data locality and cache sensitivity. However, they mainly focus on a hash join and an index nested loop join. In addition, they evaluate a disk-based DBMS, which uses a memory buffer pool to keep a portion of the data in main memory. For such a system, allocating and copying memory (necessary to (re-)partition the cache via *page coloring*) is potentially less performance-critical compared to an in-memory system.

In contrast to software-based cache partitioning, Chiou *et al.* [28] or Qureshi *et al.* [29] propose hardware-based cache partitioning by restricting cache line replacement to a certain way for an n-way associative cache. Zhuravlev *et al.* [30] propose to mitigate contention for shared resources on multicore processors via thread scheduling. More recently, Herdrich *et al.* [31] introduced two technologies addressing quality of service on Intel's multi-core server platforms: Cache Monitoring Technology (CMT) and Cache Allocation Technology (CAT). They highlight the benefit of partitioning the LLC using the SPEC CPU2006 benchmark, network communications, and the STREAM benchmark.

While we derived the cache partitioning scheme from an experimental analysis, the application of existing characterization methods for *describing* the cache usage pattern of a database operator could be investigated. For instance, Chou and DeWitt [32] propose the query locality set model based on the knowledge of the various patterns of queries to allocate buffer pool memory efficiently. Others propose the cache miss ratio as an online model for characterizing workloads or operators [26], [33], [34].

VIII. Conclusion

In modern microprocessors, multiple processor cores share the same last-level cache. Conflicts over the shared cache can significantly degrade performance of concurrent workloads, whenever query execution suffers from *cache pollution* caused by the execution of another query.

In this work, we confirm that key in-memory database operators exhibit different performance characteristics depending on the available cache size. Based on micro-benchmarks, we derived a cache partitioning scheme that we deliberately kept simple: *Restrict memory-intensive operators that do not re-use data to a small portion of the cache.*

Furthermore, we demonstrate how to integrate a cache partitioning mechanism into the execution engine of an existing DBMS with low expenditure. Our evaluation shows that our approach avoids cache pollution and significantly reduces cache misses. We demonstrate that, by partitioning the cache, we can improve the overall system performance and showcase improvements for custom queries targeting column scans, aggregations, and joins, as well as for the TPC-H benchmark and a modern HTAP business application. Ultimately, our results show that integrating cache partitioning into a DBMS engine is worth the effort: it may improve but never degrades performance for arbitrary workloads containing scan-intensive, cache-polluting operators.

Looking ahead, cache allocation might give incentive to revisit *scheduling strategies* in a database context. For instance, it might be advisable to co-run operators with high cache pollution characteristics (cache usage identifiers *(i)* and *(iii)*, according to our taxonomy), but let cache-sensitive queries (identifiers *(ii)* and *(iii)*) rather run alone. Lee *et al.* [13] have found similar strategies to be successful in connection with page coloring and PostgreSQL.

Acknowledgments

We thank Roman Dementiev and Thomas Willhalm from Intel, Thomas Bach, Lucas Lersch, Ismail Oukid, Georgios Psaropoulos, Robin Rehrmann, Frank Tetzel, and our colleagues from SAP for their support and feedback. The work has received funding from the Deutsche Forschungsgemeinschaft (DFG), Collaborative Research Center SFB 876, project A2 (http://sfb876.tu-dortmund.de/).

References

[1] P.-A. Larson and J. Levandoski, "Modern Main-memory Database Systems," *Proc. VLDB*, vol. 9, no. 13, pp. 1609–1610, Sep. 2016.

[2] A. Böhm, J. Dittrich, N. Mukherjee, I. Pandis, and R. Sen, "Operational Analytics Data Management Systems," *Proc. VLDB*, vol. 9, no. 13, pp. 1601–1604, Sep. 2016.

[3] Intel Corporation, "Improving Real-Time Performance by Utilizing Cache Allocation Technology," *White Paper*, Apr. 2015.

[4] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees, "The SAP HANA Database – An Architecture Overview," *Data Eng. Bull.*, vol. 35, no. 1, pp. 28–33, 2012.

[5] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-stores vs. Row-stores: How Different Are They Really?" in *Proc. SIGMOD*. ACM, 2008, pp. 967–980.

[6] O. Polychroniou and K. A. Ross, "Vectorized Bloom Filters for Advanced SIMD Processors," in *Proc. DaMoN*. ACM, 2014, pp. 6:1–6:6.

[7] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner, "SIMD-Scan: Ultra Fast in-memory Table Scan using on-Chip Vector Processing Units," *Proc. VLDB*, vol. 2, no. 1, pp. 385–394, Aug. 2009.

[8] T. Willhalm, I. Oukid, I. Müller, and F. Faerber, "Vectorizing Database Column Scans with Complex Predicates," in *ADMS*, 2013, pp. 1–12.

[9] F. Transier, C. Mathis, N. Bohnsack, and K. Stammerjohann, "Aggregation in Parallel Computation Environments with Shared Memory," 2012, US Patent App. 12/978,194.

[10] Y. Ye, K. A. Ross, and N. Vesdapunt, "Scalable Aggregation on Multicore Processors," in *Proc. DaMoN*. ACM, 2011, pp. 1–9.

[11] V. Viswanathan, K. Kumar, T. Willhalm, P. Lu, and B. Filipiak, "Intel Memory Latency Checker v3.3," 2017. [Online]. Available: https://software.intel.com/en-us/articles/intelr-memory-latency-checker

[12] T. Willhalm, R. Dementiev, and P. Fay, "Intel Performance Counter Monitor," 2017. [Online]. Available: www.intel.com/software/pcm

[13] R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang, "MCC-DB: Minimizing Cache Conflicts in Multi-core Processors for Databases," *Proc. VLDB*, vol. 2, no. 1, pp. 373–384, Aug. 2009.

[14] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining Insights into Multicore Cache Partitioning: Bridging the Gap Between Simulation and Real Systems," in *Proc. HPCA*. IEEE Computer Society, Feb 2008, pp. 367–378.

[15] X. Zhang, S. Dwarkadas, and K. Shen, "Towards Practical Page Coloring-based Multicore Cache Management," in *Proc. EuroSys*. ACM, 2009, pp. 89–102.

[16] Intel Corporation, "User Interface for Resource Allocation in Intel Resource Director Technology," *Documentation of the Linux Kernel*, 2017. [Online]. Available: https://www.kernel.org/doc/Documentation/x86/intel_rdt_ui.txt

[17] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki, "Scaling Up Concurrent Main-memory Column-store Scans: Towards Adaptive NUMA-aware Data and Task Placement," *Proc. VLDB*, vol. 8, no. 12, pp. 1442–1453, Aug. 2015.

[18] S. Manegold, P. Boncz, and M. Kersten, "Optimizing Main-Memory Join on Modern Hardware," *Trans. Know. and Data Eng.*, vol. 14, no. 4, pp. 709–730, Jul 2002.

[19] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu, "Multi-core, Main-memory Joins: Sort vs. Hash Revisited," *Proc. VLDB*, vol. 7, no. 1, pp. 85–96, Sep. 2013.

[20] I. Müller, P. Sanders, A. Lacurie, W. Lehner, and F. Färber, "Cache-Efficient Aggregation: Hashing Is Sorting," in *Proc. SIGMOD*. ACM, 2015, pp. 1123–1136.

[21] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-Oblivious Algorithms," in *Proc. FOCS*. IEEE Computer Society, 1999, pp. 285–.

[22] R. Cole and V. Ramachandran, "Resource Oblivious Sorting on Multicores," *ACM Trans. Parallel Comput.*, vol. 3, no. 4, pp. 23:1–23:31, Mar. 2017.

[23] M. A. Bender, E. D. Demaine, and M. Farach-Colton, "Cache-Oblivious B-Trees," *SIAM J. Comput.*, vol. 35, no. 2, pp. 341–358, 2005.

[24] B. He and Q. Luo, "Cache-Oblivious Query Processing," in *CIDR*, 2007, pp. 44–55.

[25] S. Cho and L. Jin, "Managing Distributed, Shared L2 Caches Through OS-Level Page Allocation," in *Proc. MICRO*. IEEE Computer Society, 2006, pp. 455–468.

[26] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm, "RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations," in *Proc. ASPLOS*. ACM, 2009, pp. 121–132.

[27] L. Soares, D. Tam, and M. Stumm, "Reducing the Harmful Effects of Last-Level Cache Polluters with an OS-Level, Software-only Pollute Buffer," in *Proc. MICRO*, Nov 2008, pp. 258–269.

[28] D. Chiou, P. Jain, L. Rudolph, and S. Devadas, "Application-specific Memory Management for Embedded Systems Using Software-controlled Caches," in *Proc. DAC*. ACM, 2000, pp. 416–419.

[29] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *Proc. MICRO*. IEEE Computer Society, 2006, pp. 423–432.

[30] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing Shared Resource Contention in Multicore Processors via Scheduling," in *Proc. ASPLOS*. ACM, 2010, pp. 129–142.

[31] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache QoS: From Concept to Reality in the Intel Xeon Processor E5-2600 v3 Product Family," in *Proc. HPCA*. IEEE Computer Society, March 2016, pp. 657–668.

[32] H.-T. Chou and D. J. DeWitt, "An Evaluation of Buffer Management Strategies for Relational Database Systems," in *Proc. VLDB*. VLDB Endowment, 1985, pp. 127–141.

[33] S. Manegold, P. Boncz, and M. L. Kersten, "Generic Database Cost Models for Hierarchical Memory Systems," in *Proc. VLDB*. VLDB Endowment, 2002, pp. 191–202.

[34] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic Tracking of Page Miss Ratio Curve for Memory Management," in *Proc. ASPLOS*. ACM, 2004, pp. 177–188.