Deployment of Query Plans on Multicores

Jana Giceva[§] Gustavo Alonso[§] Timothy Roscoe[§] Tim Harris[†]

[§]Systems Group
Department of Computer Science, ETH Zürich
{gicevaj, alonso, troscoe}@inf.ethz.ch

[†]Oracle Labs
Cambridge, UK
timothy.l.harris@oracle.com

ABSTRACT

Efficient resource scheduling of multithreaded software on multicore hardware is difficult given the many parameters involved and the hardware heterogeneity of existing systems. In this paper we explore the efficient deployment of query plans over a multicore machine. We focus on shared query systems, and implement the proposed ideas using SharedDB.

The goal of the paper is to explore how to deliver maximum performance and predictability, while minimizing resource utilization when deploying query plans on multicore machines. We propose to use *resource activity vectors* to characterize the behavior of individual database operators. We then present a novel deployment algorithm which uses these vectors together with dataflow information from the query plan to optimally assign relational operators to physical cores. Experiments demonstrate that this approach significantly reduces resource requirements while preserving performance and is robust across different server architectures.

1. INTRODUCTION

Increasing data volumes, complex analytical workloads, and advances in multicores pose various challenges to database systems:

First, most database systems experience performance degradation with increasing data volume and workload concurrency. The loss in performance and stability is due to contention and load interaction among concurrently executing queries [23,38]. These effects will become worse with more complex workloads.

Second, multicores, due to their parallelism and heterogeneity, are a difficult target for databases since poor deployments and/or scheduling lead to performance penalties [8,19]. The internal properties of the architecture dominate performance and require tailoring of the operators to the architecture [5,9,29,39].

Finally, a generous allocation of resources to guarantee performance and stability leads to overprovisioning. Overprovisioning results in lower efficiency and prevents the system from leveraging the full potential of the underlying architecture. This problem is being addressed in the context of virtualization and multitenancy but it also exists on multicore machines [14,33].

Proceedings of the VLDB Endowment, Vol. 8, No. 3

In this paper we address the question of how to efficiently deploy query plans on multicore machines. In particular, we focus on two problems: (i) consolidation of relational operators running on multicore according to temporal (when operators are active) and spatial requirements (memory bandwidth, CPU demand), and (ii) resource allocation to a given query plan, i.e., decide which operators should be placed on which cores, rather than treating the cores as a homogeneous array of processors. The goal is to find a deployment that minimizes the amount of resources used while maintaining performance and stability guarantees.

To provide a concrete context and a platform for experimentation, we work on global query plans such as those used in shared works systems and evaluate the ideas proposed on top of SharedDB [17]. Shared work systems are good examples of increasing complexity in query plans [4, 17, 20]. Even though most of these systems are motivated by multicores, to our knowledge no work has been done in mapping their complex plans onto modern architectures. Thus, with the results in the paper we provide both a concrete solution for a class of emerging systems, as well as a number of ideas valuable for conventional engines running on multicore.

Based on this, the paper makes the following contributions:

- We show that scheduling query plans on multicores is a non-trivial problem. We identify its performance implications and characterize the elements needed to solve it.
- We introduce the concept of *resource activity vectors* (RAVs) as a means to characterize and quantify the computational and memory bandwidth requirements of relational operators.
- We present a novel deployment algorithm for temporal and spatial scheduling that performs two important tasks: (i) computes the minimum computational requirements of a query plan, and (ii) suggests a deployment considering the NUMA-properties of the underlying architecture. Our algorithm operates on input consisting of the machine characteristics, operators' RAVs, and the dataflow properties of a query plan.

We evaluate our algorithm against existing deployment approaches for a complex query plan (TPC-W workload) on two multicore machines with different architectures. We show that the proposed algorithm decreases resource utilization (number of cores) by a factor of 7.4 without sacrificing either system's performance or its predictability. Results indicate that our approach based on RAVs accurately characterizes resource utilization properties and requirements of relational operators. Furthermore, the characterization is valid across different hardware architectures and over different data volumes, thereby establishing a solid basis for future query optimizers targeting multisocket multicore machines.

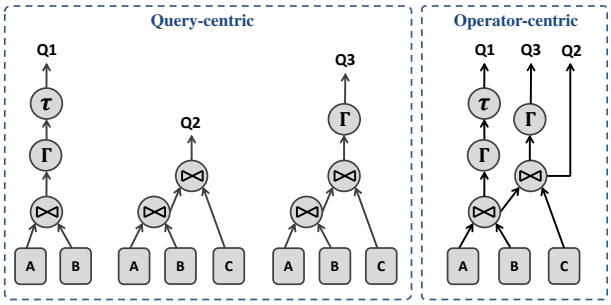


Figure 1: Examples for query-centric and operator-centric execution models. Three different query types (Q1, Q2, and Q3) need to be executed. The query-centric model generates three different query plans, while the operator-centric uses data and work sharing among operators and handles the three queries in a single plan.

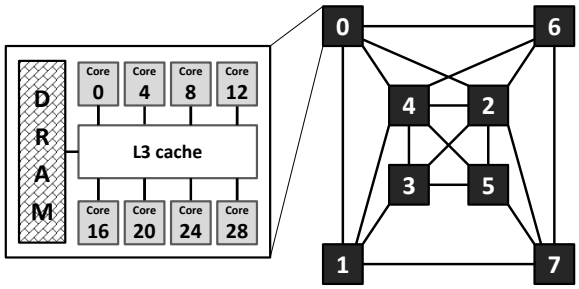


Figure 2: Layout of a typical four-socket AMD Bulldozer system. One the right is the interconnect topology of the eight NUMA nodes. On the left is the layout of cores on NUMA node 0. Note that the used NUMA and core IDs are taken from the output of the `numactl --hardware` command.

2. BACKGROUND

2.1 Complex Query Plans

Several recent systems suggest sharing of computation and data as a means to overcome resource contention and provide good and predictable performance. Such shared-work (SW) systems were first introduced at the storage engine level in the form of shared (cooperative) scans implemented in systems such as Blink [37], Crescando [40] and in MonetDB/X100 [45]. Psaroudakis et al. [36] describe the main approaches to work and data sharing: (a) simultaneous pipelining (SP) – originally introduced in QPipe [20], and (b) global query plans (GQP) – introduced for joins in CJOIN [10] and then extended to support more complex relational operators in DataPath and SharedDB [4, 17]. All these systems abandon the traditional query-at-a-time execution model (*query-centric*) and implement an *operator-centric* query execution model (see Figure 1 for an illustration also motivated from the classification presented by Psaroudakis et al. [36]).

Operator-centric systems try to maximize the sharing of both computation and data among concurrent queries using shared operators. By executing more queries in one go they achieve higher throughput, and in some cases, also more stable performance. We distinguish three common properties of these systems:

1. Operators are deployed as a pipeline (QPipe) or dataflow graph (DataPath, SharedDB). This is particularly important because the information about dataflow relationship can help in achieving better data-locality when deploying an operator.
2. Plans are composed of shared and 'always-on' relational operators (CJOIN, Blink, SharedDB, QPipe): the operators are active throughout the whole execution of the workload and are shared among the concurrently executing queries. Different systems leverage different techniques in order to maximize sharing of computation and data (batching, detecting common sub-plans, or sub-expressions, etc).
3. Operators can be characterized as either blocking or non-blocking, depending on whether one requires the full input before starting to work or can start processing as data arrives.

We have implemented and evaluated our ideas on SharedDB. It compiles the entire workload into a single global query plan that serves hundreds of concurrent queries and updates, and can be reused for a long period of time.

Sharing data and work can be easily exploited in a scalable and generic way as a result of SharedDB's batching of incoming queries and updates. SharedDB's query optimizer can automatically generate the global query plan, and assigns each operator to one core [18].

2.2 Scheduling of Shared Systems

Scheduling parallel query plans on multisocket multicore machines is a challenging, multifaceted problem. Not only do modern machines add many more factors to consider (shared caches, shared processing units, NUMA regions, processor interconnects, etc.), but are also far more heterogeneous even among processors of the same vendor. Although motivated by and designed for multicores, operator-centric database engines have not yet addressed the problem of efficient resource utilization and deployment on multicore machines. In this subsection we provide a short overview of the current approaches.

Existing shared work systems have different approaches for assigning processing threads to their relational operators/operator's work units: a micro-engine's thread pool [20], per-operator threads [17], fixed worker threads executing specific work-units [4], etc. All of these approaches, however, fix the number of threads assigned to the operators. Each thread is pinned to a particular core and there is only one thread assigned to a core. Such an implementation provides (i) predictable performance because there is no thread migration [17], and (ii) progress is always guaranteed [20].

The cost for this performant deployment is system-wide resource overprovisioning. This problem is further aggravated by the rigidity of assigning the same amount of resources for all operators, which does not account for the individual resource footprints and requirements. Moreover, none of the approaches currently employed in operator-centric systems takes into consideration the architecture and properties of the multicore machine or the data-dependency of the shared query plan.

2.3 Problem Statement

Our goal is to determine how to deploy a query plan minimizing the amount of resources used while maintaining required performance and predictability guarantees. There are two different aspects of the problem: (i) temporal and (ii) spatial scheduling.

Temporal scheduling aims at deciding which operators are suitable candidates to time-share a CPU, i.e., can be deployed to run on the same core. The challenging part is to avoid co-locating operators which will interfere with each other, so that we maintain the required system stability and performance. Example of a suitable pair of operators that could time-share a core are pipelined blocking operators where it is certain that the downstream operator is only active after its predecessor has finished processing the result-set. Additionally, they may also benefit from data locality.

Spatial scheduling, on the other hand, aims at determining which cores should be used for the deployment of the operators. This has

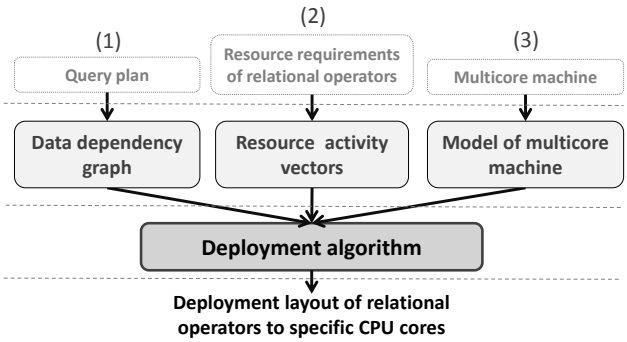


Figure 3: Sketch of the solution – overview of the information flow in the deployment algorithm.

two versions: collocating operators to run concurrently on the same core if one is CPU and the other is memory bound; and placing operators communicating with each other in ways that optimize the data traffic on the processor interconnect network. This is a difficult problem which is architecture-dependent and must be addressed accordingly. In order to illustrate the complexity of the problem and the necessity of appropriate solutions, we present a sample multicore architecture in Figure 2. By looking at the inter-NUMA links one can easily grasp the difference in communication cost and redundant data traffic on the interconnect network when one chooses to place two communicating operators on cores 0 and 4 which share a last level cache (LLC) and a local NUMA node, as opposed to using core 0 on socket 0 and core 1 which is on a remote socket (one or two NUMA-hops away).

2.4 Sketch of the Solution

Optimized resource management and scheduling must take into consideration (Figure 3): (1) the data-flow dependency graph in the query plan which is used to determine temporal dependencies; (2) the resource footprint and characteristics of the individual operators, which are used to determine spatial placement; and (3) the intrinsic properties of the underlying architecture, which determine the available resources and constraints on the operator placement and inter-operator communication.

The next two sections describe in detail the two main contributions of the paper, the *resource activity vectors* as a means to characterize the resource requirements of the database operators (Section 3) (addressing points 2 and 3), and a novel algorithm (Section 4) mapping operators to cores (under the constraints of point 1) while maintaining the system's performance and stability.

3. RESOURCE ACTIVITY VECTORS

Good resource management and relational operator deployment requires awareness of the thread's resource requirements. In the last decade, we have witnessed significant progress in tuning DBMS performance to the underlying hardware [2, 3, 5, 9, 25, 29, 30, 39, 42, 45]. As a result databases and their performance have become more sensitive to the resources they have at hand, and poor scheduling can lead to performance degradation [19, 27].

In order to capture the relevant characteristics, in this paper we introduce *resource activity vectors* (RAVs). These vectors concentrate on the most important resources, CPU and memory bandwidth utilization. This approach of characterization is inspired by the notion of activity vectors, initially introduced for energy-efficient scheduling on multicores [32].

3.1 RAV Definition

We use RAVs to characterize the resource footprints of relational operators. They summarize the amount of resources needed such that each thread delivers its best performance. In the current implementation we consider two dimensions:

1. *CPU utilization* represents how efficiently the operator thread uses the CPU throughout the workload execution. This is highly relevant for the deployment algorithm as it identifies the threads that are either rarely active, or when active make poor use of the CPU time. These types of threads are usually good candidates for sharing the core with another task. At the other extreme, operator threads with high CPU utilization are both active for a very long time and efficiently use the CPU and should be thus left to run in isolation. This way their performance will not be hurt by time-sharing the CPU. We elaborate more in Section 3.3.
2. *Memory bandwidth utilization* identifies both interconnect and DRAM bandwidth consumption of the operator threads throughout the workload execution. This information is relevant for deployment because, for instance, if several bandwidth thirsty operators are placed on the same NUMA node, one can easily hit the bandwidth limit and affect performance. Similar to the arguments used for the CPU utilization, it is not enough to look only at the memory bandwidth utilization as an average over the active time but over the total duration of the workload execution. Only then will we be able to avoid over-provisioning of the machine resources by reducing the significance of short running tasks with heavy memory access requirements. We elaborate more in Section 3.4.

3.2 RAV Implementation

One approach to obtain the RAV values is to model the relational operators and use the model to deduce their behavior on a set of resources. The alternative is to treat the operator threads as 'black boxes' and use available hardware instrumentation tools to calculate their resource requirements. We chose the second option because we believe it offers better scalability and could also handle future hardware extensions.

The operator threads were instrumented on two different architectures (introduced in Section 5.1). The Performance Measuring Units (PMUs) on both architectures differ in their structure, organization and supported events [1, 16, 21, 22]. For simplicity of argumentation in this paper we are only going to use generic names, as summarized in Table 1.

The instrumentation is performed while running a representative sample of the workload in the following way: in the initial system deployment all operator threads are pinned to different cores with nothing else scheduled at the time. Every PMU has a number of registers that can be used for gathering performance event counts.

Table 1: Instrumentation events, terminology description

Term	Description
`cycles`	CPU clocks unhalted, i.e., # of clocks, when the CPU was not idle
`ret_inst`	# of useful instructions the CPU executes on behalf of the program
`DRAM_accesses`	# of DRAM accesses by the program as measured by the memory controller(s)
`sys_read`	# of system reads by coherency state
`sys_write`	# of octwords written to system
`LLC_misses`	# of last level cache misses

Because there is a limited number of such registers, we execute a number of runs to gather all data required for the statistics. Upon completion of data gathering, we postprocess the measurements to derive the final values of the two RAV components.

When the number of cores on the machine is smaller than the total number of operators, one can use the 'separate-thread' option in the profiling tool. This way the post-processing step can distinguish between events that occurred as a result of another thread's activity, running on the same core.

3.3 Capturing CPU Utilization

The CPU utilization needs to represent both efficient utilization of the CPU cycles when in possession of the core, as well as the active time of the operator thread as a fraction of the total duration of the workload-sample execution.

In order to derive the efficient utilization of the CPU, we calculate the IPC (instructions per cycle) value for each operator. The IPC can be calculated using the `ret_inst` and `cycles` events.

Although we work with 'always-on' operators, they are not active at the same time, for instance, because of a data-dependency. Furthermore, we expect to see discrepancies among operators when comparing their active runtime. As mentioned earlier, the deployment algorithm should consider the substantial difference in long- and short-running threads and handle each class accordingly. The active runtime of a thread is derived as the ratio of total number of unhalted CPU cycles measured on that core to the total duration of the workload-sample execution.

The formula used to calculate CPU utilization for the RAVs normalizes the thread's IPC value with its active runtime as presented in Equation (1). Please note that after simplifying the formula, the CPU utilization is only dependent on the measured number of `ret_inst` events and the `total_cycles` (duration of the workload sample). Also note that the latter one is constant for all operators in the same query plan.

$$\begin{aligned} CPUutil &= \text{IPC} \times \text{active time} \\ &= \frac{\texttt{ret\_inst}}{\texttt{cycles}} \times \frac{\texttt{cycles}}{\texttt{total\_cycles}} \\ &= \frac{\texttt{ret\_inst}}{\texttt{total\_cycles}} \end{aligned} \tag{1}$$

3.4 Capturing Memory Utilization

The values for memory bandwidth utilization can be calculated in a similar fashion. The first factor is the measured data bandwidth transfer (in `bytes`). We gather information for both the interconnect and the DRAM bandwidth utilization. The events used for data gathering and exact formulas used for deriving these values are highly architecture-dependent. While on AMD the core PMUs can gather specific events that capture both DRAM and system accesses [16], on Intel one is limited to using approximation formulas based on the last level cache (LLC) [21].

The second factor, just as in the CPU utilization case, is the active runtime of the operator threads. The total duration of the workload sample is used to normalize the memory bandwidth requirements of the threads of interest. Equation (2) shows the derivation for memory bandwidth utilization:

$$\begin{aligned} MEMutil &= \text{bandwidth} \times \text{active time} \\ &= \frac{\texttt{bytes}}{\texttt{cycles}} \times \frac{\texttt{cycles}}{\texttt{total\_cycles}} \\ &= \frac{\texttt{bytes}}{\texttt{total\_cycles}} \end{aligned} \tag{2}$$

The bandwidth utilization of an operator is only dependent on the amount of data transfered over the duration of the workload sample (expressed as `total_cycles`). For simplicity the total `bytes` transferred is the sum of interconnect and DRAM bandwidth consumption (events `DRAM_accesses`, `sys_read`, `sys_write`).

Note that with this particular profiling setup ('operator-per-core' deployment, and distributing all operators across the cores), the memory and in particular the interconnect bandwidth utilization for the operators is overestimated. If the operators are placed on the same NUMA region, it is likely that most of the data-transfer would occur via the shared last-level cache.

3.5 Parallel Operators

Until now, we have discussed how to capture the resource footprints and requirements of single-threaded operators. We assume that the degree of parallelism assigned to an operator is decided by the optimizer. Multithreaded operators are then supported in a similar manner – we just consider the individual operator threads as separate entities to be scheduled and hence RAV-annotated. In fact, in the context of SharedDB, we already do this for the scans. The performance of a scan operator, like Crescando (used in SharedDB), can be improved by increasing the number of scan threads working on horizontal partitions. In our experiments we RAV-annotate and schedule each scan thread separately.

4. DEPLOYMENT ALGORITHM

The deployment algorithm we propose aims at delivering a deployment plan of the operators which (i) minimizes the computational and bandwidth requirements of the query plan; (ii) provides NUMA-aware deployment of the relational operators; and (iii) enhances data-locality.

As presented in Figure 4, the algorithm consists of four phases: (1) operator graph collapsing, (2) bin-packing of relational operators to clusters based on the CPU utilization dimension of the RAVs, (3) bin-packing of operator-clusters (output of (2)) to number of NUMA nodes based on the RAV's memory bandwidth utilization dimension as well as the capacity of the NUMA nodes and (4) deployment mapping of the computed number of NUMA nodes onto a given multicore machine. The first two phases compute the required number of cores which corresponds to the temporal scheduling subproblem, the third phase approximates the minimum number of required NUMA nodes, and the fourth computes the final placement of the cores on the machine such that it minimizes bandwidth usage – spatial scheduling subproblem.

4.1 Operator Graph Collapsing

The first phase of the algorithm takes as input an abstract representation of the complex query plan – dataflow graph of database operators. It iterates over the operators in the dataflow graph and compacts each *operator-pipeline* into one so-called *compound operator*. An operator-pipeline is characterized by non-branching dataflow between the operators belonging to the pipeline. An example of such pipeline is presented in Figure 5.

This phase of the algorithm targets operator-pipelines with blocking operators. Here, by design, there is a guarantee that the involved operators will never overlap each other's execution and as a result can be temporally ordered. Since there is a temporal ordering between the blocking operators belonging to the same operator-pipeline, one can easily think of them as an atomic scheduling unit (i.e., they can be grouped into one compound operator). This way the scheduler can safely place all such operators to run on the same set of resources, one after another. Additionally, the new compound operator is expected to have better data locality. Scheduling the

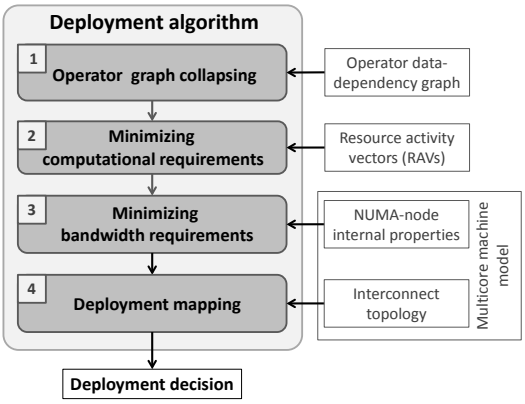


Figure 4: Overview of the deployment algorithm

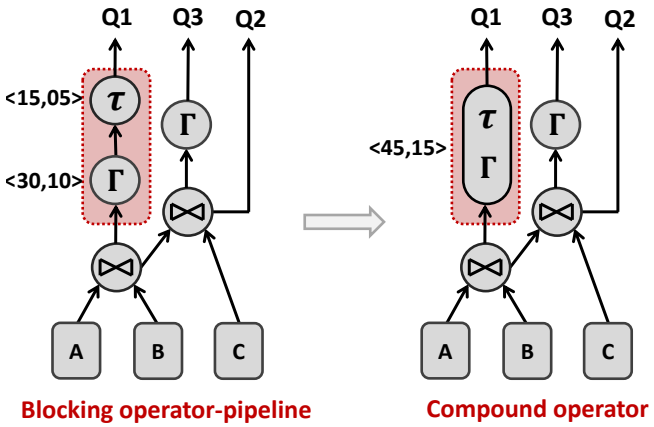


Figure 5: The highlighted rectangle illustrates: (left) an example of an operator pipeline that can be collapsed into one compound operator, (right) the resulting compound operator. For readability, only the operator-pipeline and the corresponding compound operator are RAV-annotated.

component operators sequentially on the same core will leverage the warm data caches and reduce unnecessary memory movement.

The new composed compound operator inherits the RAV characteristics of its components as presented in equations (3) and (4).

$$C.\text{cpu\_util} = \sum i.\text{cpu\_util}, \forall i \in P \tag{3}$$

$$C.\text{mem\_util} = \sum i.\text{mem\_util}, \forall i \in P \tag{4}$$

where C denotes the compound operator, and P denotes the set of all operators belonging to the operator-pipeline.

Both dimensions of the compound operator's RAV are computed as the sum of the values of the corresponding dimensions of its components. The formulas are a direct consequence from the definitions of CPU and memory bandwidth utilization in equations (1) and (2). Intuitively, the compound operator now has to execute the cumulative number of instructions of its components in the same amount of time (`total_cycles`). The same reasoning applies for the total number of bytes transferred.

4.2 Minimizing Computational Requirements

Once the original set of operators has been compressed by collapsing the operator-pipelines into compound operators, the subsequent phases of the deployment algorithm operate on a smaller set of operators. Please note that due to the 'always-on' nature of the operators of the shared-work systems, the operators of the new set can no longer be temporally ordered, i.e., we have to assume that they could run concurrently.

The second phase iterates over this new set of operators in search of suitable clusters of operators that can be safely placed on the same CPU core. This clustering is based on the values of the CPU utilization component of the operators' RAVs. The goal is to determine the minimum number of CPU cores needed to accommodate all relational operators. Note that the second dimension of the RAVs does not play a role. This is because the operator clustering solely determines which operators can be placed safely on the same CPU core. The operators belonging to the same cluster technically only share CPU time. Hence, they will never be executed concurrently so memory bandwidth is not going to be shared.

This phase of the algorithm is an instance of the bin packing problem, defined as follows: *Items of different sizes must be packed into a finite number of bins, each of capacity X, in a way that minimizes the number of bins used.* Even though it is an NP-hard problem, there are many approximation algorithms proposed that give nearly optimal solution in polynomial time [13]. Since we know the items (operators) and their properties in advance, we can use offline bin packing solutions. Therefore, in our algorithm, this phase is implemented using an adapted version of one of the simplest heuristics – the first-fit decreasing (FFD) algorithm. Algorithm 1 provides a pseudo-code implementation of the bin-packing algorithm. The change we added to the standard FFD is related to the support for parallel operators, i.e., checks whether another thread of the same operator (aka sibling item) was previously placed in the same bin and avoids the bin if so (line 5). The standard FFD algorithm has been proven to have a tight approximation bound of $11/9 \cdot OPT + 6/9$, where OPT is the optimal number of bins [15].

Eventually, it returns an approximate of the minimum number of cores that can fit all operator threads such that the computational constraints are met. Furthermore, it also outputs the contents of the bins, i.e., how the relational operators are clustered.

Data: List of items
Result: Number of bins, contents of bins

```
1  SortDecreasing (items);

2  BinPacking (items)
3  for items i=1,2,...n do
4      for bins j=1,... do
5          if item i fits in bin j and i's sibling is not in bin j then
6              pack item i in bin j
7              break the loop, and pack the next item
8          end
9      end
10     if item i was not placed in any bin then
11         create new bin k
12         pack item i in bin k
13         add bin k to bins
14     end
15 end
16 return number of bins, and bins' contents
```

Algorithm 1: FFD bin-packing algorithm

4.3 Minimizing Bandwidth Requirements

The third phase of the algorithm operates on the following input:

- Model of the internal NUMA-node properties: (1) number of cores, and (2) local bandwidth capacity.
- Memory bandwidth requirements of the (bins) operator clusters, which were computed during the previous phase of the algorithm. One can compute the resource requirements of these operator clusters in a similar manner as with the compound operators (equations (3) and (4)).

The goal is to compute the minimum number of NUMA nodes required to accommodate all operator-clusters and their bandwidth requirements given the node's capacity constraints.

This can be formalized as another instantiation of the bin packing problem: the items to be packed are the operator-clusters and the bins are the NUMA nodes. The capacity of the bins is determined by the maximum attainable local DRAM bandwidth. Furthermore, the bins are also constrained on the cardinality of items they can accommodate (the number of CPU cores on the corresponding NUMA node). Hence, it is an instantiation of *cardinality-constrained* offline bin packing problem. We can, thus, use the same FFD algorithm to compute an approximate solution. The only modifications needed are when evaluating whether an item can fit in a certain bin, and when updating the corresponding data structures (lines 5 and 6 of Algorithm 1).

4.4 Deployment Mapping

The last phase of the algorithm computes the final deployment mapping of the operator-clusters onto actual CPU cores of the multicore machine. It uses the output of the previous phase, which computed the minimum number of NUMA nodes required to accommodate the operator-clusters. If one NUMA node is sufficient, the output is trivial – any subset of the cores belonging to the same NUMA node will do, provided it is of cardinality k, where k denotes the required number of CPU cores (as computed in phase (2)). The rest of this subsection explains the steps needed should the number of NUMA nodes be larger than one.

In order to determine the optimal mapping, this phase first models the multicore's NUMA interconnect topology as a graph. The graph is defined as $G(V, E)$, where the set of vertices V corresponds to the set of all NUMA nodes and the set of edges E is composed of all direct links between the NUMA nodes. Furthermore, G is an undirected graph, with maximum one link (edge) allowed between a pair of NUMA nodes.

As initially presented in Section 7 (Figure 2) two communicating operators should ideally be placed close to each other so that we reduce data-access latency and interconnect bandwidth usage. Thus, if it is not possible to accommodate them on a single NUMA node, then a priority should be given to the neighboring nodes.

As an example, we present the following problem: it has been determined that the deployment of a certain query plan on a given machine needs four NUMA nodes. To demonstrate the generality of the approach, let us assume that the interconnect topology of the machine is not symmetric and looks like shown in Figure 7. We have denoted with $D1$ and $D2$ two of the many possible deployments (subgraphs) encapsulating four NUMA nodes. Ideally, the deployment algorithm should return deployment $D1$ as a preference because that way all operators will be at the shortest possible distance of each other (1-hop) as opposed to the other alternative where the average distance between the NUMA nodes is 1.33 hops.

Therefore, the algorithm needs to also quantify how close the nodes of a certain subgraph are. In order to do that it leverages the concept of *graph density*. Graph density (d_G) of a graph $G(V_G, E_G)$ is defined as the number of edges divided by the number of vertices, or more formally:

$$d_G = \frac{|E_G|}{|V_G|}$$

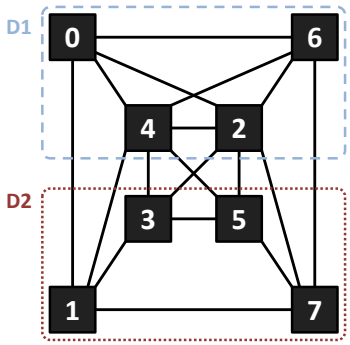


Figure 7: Example of two possible deployments (D1,D2) of 4 NUMA nodes within an 8-node AMD Bulldozer machine. The asymmetric topology means deployment D1 is preferable.

Using this metric we can formalize the problem that needs to be solved in this phase as an instantiation of *finding the densest k-item subgraph* problem. Khuller and Shaba [24] proved that the solution is NP-hard, but given the small size of our graph this is still within acceptable boundaries. Given a multicore machine a prephase of our naïve implementation iterates over all subgraphs of size k, and computes the density of each. The desired output is a query for the subgraph with highest density. The final operator-to-core deployment mapping is chosen accordingly.

4.5 Discussion and Possible Extensions

With the deployment algorithm presented in this section we leverage the dataflow information from the database query plan, the RAV properties of the operators, and the NUMA model of the multicore machine. Using this input the algorithm is able to significantly reduce the total number of resources required by a query plan and at the same time avoid contention for the most critical resources: CPU cycles and memory bandwidth.

One immediately applicable enhancement is the following: The chosen implementation of the bin-packing algorithm (sorted first fit) can provide a good approximation on the minimum number of cores/NUMA nodes required but does not guarantee that all the bins will be equally balanced. Consequently, one can extend it with an additional epilogue phase that will perform load-balancing of the content in the bins.

Moreover, it can be extended with an OS system-wide knowledge of the current resource utilization among all running tasks, similar to the approach suggested in COD [19].

5. EVALUATION

In this section we show that the performance, stability and predictability of the query plan remains unaffected despite the heavy reduction in the allocated resources by the deployment algorithm. We evaluate the performance of the deployment of a TPC-W query plan on different dataset sizes on two different multicore architectures. We demonstrate the accuracy of the RAV characterization, and conclude the section with an analysis on the different phases of the deployment algorithm.

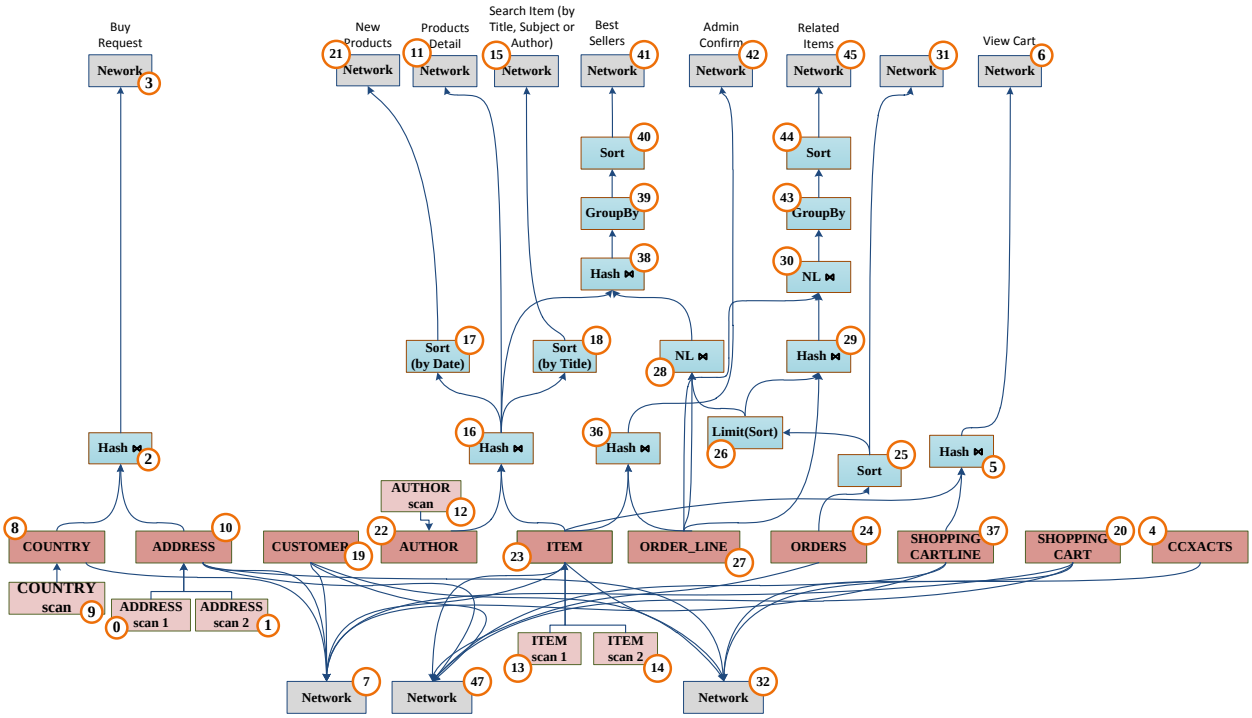


Figure 6: TPC-W shared query plan – as generated for SharedDB

5.1 Experiment Setup

Infrastructure: For our experiments we used two multicore machines from different vendors in order to compare the influence of their architectures both on the operators' RAVs and the outcome of the deployment algorithm. The machines used are:

1. **AMD MagnyCours**: Dell 06JC9T board with four 2.2GHz AMD Opteron 6174 processors. Each processor has two 6-core dies. Each die has its own 5MB LLC (L3 cache) and a NUMA node of size 16GB. The operating system used was Ubuntu 12.04 amd64 with kernel 3.2.0-23-generic.
2. **Intel Nehalem-EX**: Supermicro X8Q86 board with four 8-core 1.87GHz Intel Xeon L7555 processors with hyperthreads. The hyperthreads were not used for the experiments. Each processor has its own 20MB LLC (L3 cache) and a NUMA node of size 32GB. The operating system used was Debian with Linux kernel-3.9.4 x86_64.

The clients were running on four machines with two 2.26GHz Intel Xeon L5520 quadcore processors, and a total of 24GB RAM.

Workload: In order to evaluate the deployment algorithm for a query plan we used SharedDB's global query plan for the TPC-W benchmark. The TPC-W workload consists of 11 web-interactions, each consisting of several prepared statements, which are issued based on the frequencies defined by the TPC-W browsing mix. The query parameters were also generated as defined in the TPC-W specification. SharedDB's TPC-W query plan has 44 operators, as presented in Figure 6. In the figure we marked all 44 operators with an ID which corresponds to the OS core-ID assigned in the initial deployment. Please note that the range of the IDs is between 0-47 but some core IDs (4 to be exact) are skipped[1]. The storage engine operators work with data local to their threads, and the internal logic operators fetch their input from their predecessors and generate data on their local NUMA node. Our analysis focuses on two dataset sizes: 5GB (1.2k emulated browsers(EBs), 100k items), and 20GB (5k EBs, 100k items).

Setup: Every experiment was run for 10 minutes, plus two minutes dedicated for warm-up and cool-down phases. The record logs obtained in the warm-up and cool-down phases were not taken into account in the final results presented here. The profiling on both machines was done with Oprofile (operf).

Metrics: Throughput is reported in Web Interactions Per Second (WIPS), and all the reported numbers on latency are in seconds (s). Maximum attainable bandwidth is not taken from the machine specifications but measured with the STREAM benchmark [31]. In the figures it is expressed in Gigabytes per second (GB/s).

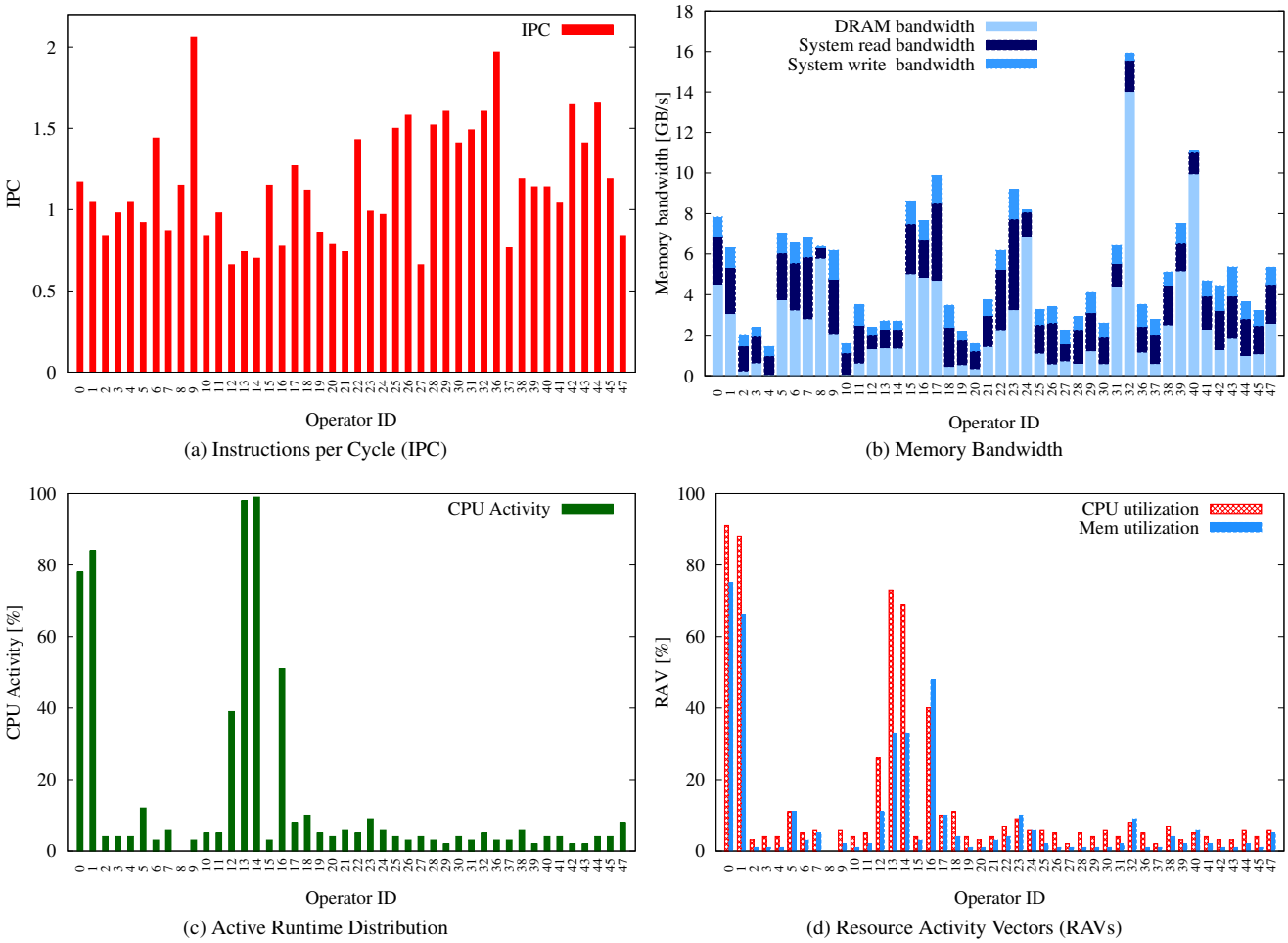
5.2 Resource Activity Vectors (RAVs)

As explained in Section 3, the RAVs are derived from statistics gathered by profiling the operator threads. Here we present a breakdown of the main factors that constitute the values for CPU and memory bandwidth utilization dimensions. The results are from an experiment run on the AMD MagnyCours on the 20GB dataset.

We measured `cycles` and `ret_instr` to derive the IPC values. Moreover, in order to calculate the memory bandwidth consumed we used the formulas as presented in [16] and collected the following events: `sys_read`, `sys_write` and `DRAM_accesses`. The last one included measurements of two different DRAM channels (DCT0 and DCT1). The per-core PMUs on AMD MagnyCours can collect up to four events per run, so in total we needed two runs to derive the RAV properties for the operators.

The results presented in Figure 8 illustrate the measured values of the operator threads in terms of IPC (8a) and memory bandwidth (8b) (consisting of DRAM, and system- read and write bandwidth).

[1]Due to difference in the core-ID mapping between the application and the OS



(a) Instructions per Cycle (IPC)

(b) Memory Bandwidth

(c) Active Runtime Distribution

(d) Resource Activity Vectors (RAVs)

Figure 8: Understanding the derivation of RAVs, AMD MagnyCours, 20GB dataset

From these two graphs, one can notice variety in the distribution of resource consumption among the different operators.

As discussed on Section 3, looking at the raw performance metrics presented in Figures 8a,8b is not enough to make a sound decision on the threads' resource requirements. One must also consider the total *active* time for each operator thread, and normalize the derived values for both RAV dimensions accordingly. Figure 8c presents the active runtime of the operators with respect to the total duration of the experiment. It shows that only a few operator-threads are actively using the CPU time. This is an important observation, as it emphasizes the large number of idle threads and the opportunity for resource consolidation.

The final values for CPU and memory bandwidth are presented in Figure 8d. Please note that the memory bandwidth utilization no longer contains the bandwidth breakdown but rather considers their sum. Figure 8d shows that both the CPU and memory bandwidth utilization values look significantly different than on the raw performance/resource metrics in Figures 8a,8b. This confirms that the resources of the machine are overprovisioned and that there is room for improvement.

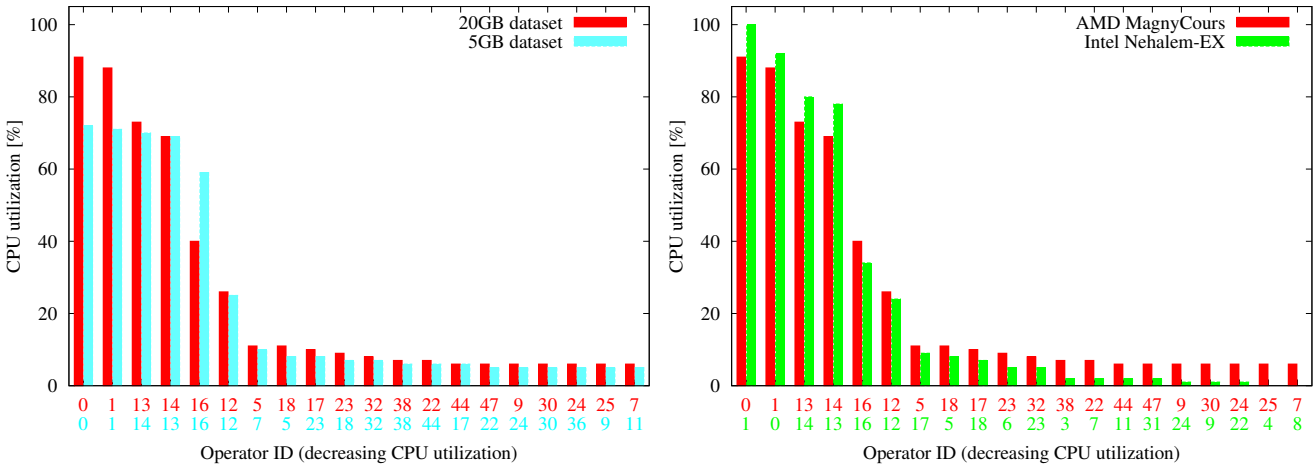
In the rest of the evaluation section we focus on the CPU utilization dimension of the RAVs. The same observations also hold for memory bandwidth utilization.

Impact of Dataset Size on RAVs

This subsection analyzes the effect that dataset size has on operator's RAVs. The experiments were executed on the AMD MagnyCours machine and the two dataset sizes used are 5GB and 20GB.

A summary of the output of the experiment is presented in Figure 9a. It displays the derived CPU utilization values for the two experimental configurations. For readability, in the figure we only present the values (in a decreasing order) for the operators with CPU utilization higher than 5 percent (in this case the top 19). Each row in the x-axis denotes the operator-IDs for the corresponding experiment run. As shown in the figure, the distribution of the CPU utilization varies with the changes in the dataset size. Intuitively, the larger dataset puts more strain on some of the scan operators (operator IDs 0 and 1), while the CPU-heaviest join (ID 16) is busier in the smaller dataset. The difference in CPU-utilization of the other operators in both datasets is almost negligible.

The difference in both distribution and absolute values of the CPU utilization influence the output of the deployment algorithm. In this case their effects canceled each other and consequently the second phase of our algorithm derived the same number of bins (cores) for both configurations – six.



(a) CPU utilization: Varying dataset size

(b) CPU utilization: Influence of architecture (20GB dataset)

Figure 9: Analyzing the impact of dataset size and multicore architecture on the RAVs

Influence of Architecture on RAVs: Intel vs AMD

Another important factor that influences the values of the RAVs is the underlying multicore architecture. In order to show its impact on the RAVs we executed the same workload with dataset size of 20GB on the two different machines introduced in Section 5.1 (AMD MagnyCours and Intel Nehalem-EX).

In Figure 9b we present the results of the CPU utilization of the operators in decreasing order. We focus on operators with CPU utilization higher than 5 percent (i.e. the top 19). Once again, there are two x-axis denoting the operator-ids on the corresponding architectures. On Intel we observe that the heavy scan operators (IDs 0,1,13,14) have higher utilization of the CPU, but also that fewer operators have CPU utilization higher than 5 percent (only the first 8). The observed difference in both distribution and absolute values confirms the benefits of using the RAVs as a means to detect architecture-specific sensitivity. For this setup the deployment algorithm allocated 5 cores for the run on Intel Nehalem-EX and 6 for AMD MagnyCours.

5.3 Performance Comparison: Baseline vs Compressed deployment

The following experiment compares the performance of the query plan when deployed on a *compressed*-set of resources (based on the output of our algorithm) versus the approach using operator-per-core deployment. As baseline we use the performance when the deployment of operators is handled by the OS scheduler.

The results show that both performance and stability of the query plan are unaffected by the heavy reduction of resources allocated by the deployment algorithm. This can be observed for a range of workload configurations and architectures (Table 2). The table summarizes the performance expressed both in throughput (WIPS), and latency (s). In order to capture the stability of the system, alongside with the aggregated average, we also present the standard deviation in parenthesis. Additionally, for the latency measurements we present the 50th, 90th and 99th percentile of the requests' response time.

The presented values for throughput and latency confirm that the resulting system performance was not compromised by the significant reduction in allocated resources. Furthermore, in all experiments the stability and predictability of the system remain intact, which is important for databases and their SLAs.

The performance of the query plan when the OS scheduler was in charge of deployment (represented in row 3 in Table 2) is poorer both in terms of absolute values and stability than the other two approaches, including the operator-per-core deployment (row 2). The latter suggests that performance can be affected as a result of thread migration.

Performance/Resource Savings Ratio

In order to observe the significance of the gain in deployment efficiency we introduce a new metric: performance/resource efficiency savings factor. It is calculated using the measured throughput (`Tput`) on the allocated set of resources (`Res`) using the following formula:

$$\text{savings factor} = \frac{\texttt{Tput}_c}{\texttt{Res}_c} \times \frac{\texttt{Res}_b}{\texttt{Tput}_b} \quad (5)$$

where the subscript c denotes the compressed, and b the operator-per-core deployment.

Table 3 illustrates the efficiency boost obtained by the compressed deployment for the different workload setups. The gain in performance/resource (calculated from the throughput values in Table 2) is usually in the range of 6-7x compared with the operator-per-core deployment. In other words, with our deployment algorithm the query plan can achieve the same performance by using only 14% of the resources, or even less when compared to the baseline OS operator scheduling.

5.4 Analysis of the Deployment Algorithm

The deployment algorithm delivers an approximation to the minimum number of cores needed by the query plan, and a mapping on how to optimally choose cores on the given multicore architecture, in order to reduce bandwidth consumption. As presented in Figure 4, it consists of four phases and each phase of the algorithm contributes to the final result from different aspects:

Phase (1) reduces the total number of operators by collapsing the operator-pipelines to compound operators. In the case of the TPC-W global query plan, this phase decreases the number of operators from the original 44 down to 32. A second contribution is the constructive cache sharing between the operators in the original operator-pipelines. These are now scheduled on the same core and benefit from data-locality.

Table 2: Performance on default vs. compressed deployment

row #	Architecture	#Cores	Throughput [WIPS]	Response Time[s]			
	(exp. config)		Average (stdev)	Average (stdev)	50th	90th	99th
	AMD						
1	(20GB)	6	428.07 (+/- 32.80)	14.62 (+/- 0.76)	15.36	23.73	36.13
2		44	425.86 (+/- 54.34)	14.69 (+/- 0.85)	14.59	22.93	36.08
3	(OS baseline)	48	317.30 (+/- 31.11)	20.81 (+/- 2.55)	8.22	72.43	82.03
4	(5GB)	6	645.71 (+/- 38.24)	8.41 (+/- 0.46)	7.00	16.44	19.69
5		44	703.51 (+/- 55.66)	7.38 (+/- 0.55)	5.65	14.81	17.87
	Intel						
6	(20GB)	5	362.62 (+/- 62.16)	18.05 (+/- 2.77)	18.35	31.73	43.94
7		32	386.97 (+/- 59.34)	16.70 (+/- 2.47)	16.95	28.03	41.93

Table 3: Performance/Resources efficiency savings

Setup	savings factor
AMD 5GB	x6.73
AMD 20GB	x7.37
Intel 20GB	x5.99

Phase (2) is responsible for a more aggressive reduction in the allocation of computational resources. It further decreases the total number of required cores down from 32 to 6 or 5 cores, depending on the architecture. The optimality of the algorithm was briefly addressed in Section 4. In order to evaluate the accuracy of the output of this phase, we did another experiment on the AMD MagnyCours (20GB) where we took the content of the sixth bin and evenly distributed it across all other bins (i.e, test a deployment on a smaller number of cores). The results are presented in a row dedicated to Phase (2) in Table 4. The first row shows the system's performance based on the output of the algorithm, and we use it as a baseline for comparison. Comparing the first two rows confirms that both the absolute performance and the stability of the system are decreased when reducing the number of cores allocated from six down to five.

Table 4: Evaluating the design choices of algorithm phases

Phase	layout	#Cores	Throughput [WIPS]
	same NUMA	6	428.07 (+/- 32.80)
(2)	same NUMA	5	366.08 (+/- 51.01)
(3,4)	dist NUMA	6	401.84 (+/- 33.21)

Phases (3) and (4) compute the final placement of operators to cores. In order to evaluate the heuristic and importance of the actual core placement on the machine, we compare the output of our algorithm (the heuristic being, if no contention for memory bandwidth, place the bins as close as possible) to the other extreme where every bin is placed on a different NUMA node. The results are presented in row dedicated to Phase (3,4) in Table 4. The difference in performance favors the heuristic approach, although the results are still within the error bars. We expect that it will have higher impact when dealing with bandwidth-heavier workloads.

5.5 Discussion

The results confirmed that RAVs successfully characterize the properties of relational operators, and can accurately capture the changes in resource requirements for different dataset sizes. Furthermore, the evaluation on the two machines (Intel Nehalem-EX and AMD MagnyCours) points out that both the operator's resource requirements and consequently the optimal resource allocation and query plan deployment are architecture-dependent.

Most importantly, the performance of SharedDB on the compressed set of resources (compressed-deployment) is as good as the performance on the operator-per-core deployment both in terms of throughput and latency, and in their stability, and better when compared to the OS operator scheduling baseline.

Eventually, we emphasized the significance in efficient resource utilization when delivering performance with a performance/resource savings factor of 6-7x when compared to the baseline.

6. GENERALIZING THE APPROACH

6.1 Parallel Operators

In SharedDB and its original operator-per-core deployment policy, because of the 'always-on' nature of the operators, even though it was conceptually possible, in practice one was not able to parallelize all operators at the same time, simply because there were not enough contexts available on current multicores. Therefore, one had to choose to parallelize only a few heavy operators (in our case the scans) that could improve the performance.

One of the benefits of this work is that it allows the system to improve its performance by adding more resources to the existing operators, as a result of the reduction of the amount of resources allocated to the original query plan. We have shown how multi-threaded operators are supported with our deployment algorithm and RAV-annotations. Here, in order to demonstrate the immediate benefits of the smart deployment, without having to parallelize the internal 'logic' operators, we increased the number of threads of all Crescando operators and replicated the internal 'logic' operators (the KV store operators were not parallelized/replicated). Since the original plan deployment fitted on one NUMA, we deployed every new replica on a new NUMA node.

Table 5: Plan replication

Dataset size	# of replicas (Throughput [WIPS])			
	1	2	4	8
5GB	452.87	1073.51	1923.41	2610.30
20GB	495.64	1031.64	1969.14	2834.98

Table 5 summarizes the performance of the system in terms of throughput. The experiments were executed on the AMD MagnyCours machine. The results indicate that with the simple technique of plan-replication the system scales almost linearly up until four replicas, and achieves a scale-up of almost six when using eight replicas. We would like to point out that this boost in performance was achieved without any system fine-tuning.

6.2 Dynamic Workload

We distinguish three different types of dynamism in a workload: (i) known-in-advance queries vs. query-types, where in the latter case the known part is implemented in the form of JDBC-like prepared statements, (ii) changes in the workload distribution, which refers to the percentage of query types being present in the workload mix, and (iii) ad-hoc queries and their arrival rates and distribution.

In this work we present a solution handling the first type of workload dynamism – which follows by design and implementation of SharedDB. Furthermore, given the reduction of the amount of resources allocated to the global query plan, it is immediately possi-

ble to handle incoming ad-hoc queries by running them on the side, using some of the extra, now available, resources.

The only dynamism that is not supported by the current implementation of the global query plan and its static deployment of operators is the second one – with changes in the distribution of query types in the workload mix which will affect the hot-paths/spots in the query plan. We believe that adding support for continuous monitoring of the operators' activity and computing the RAVs can assist the DBMS and its optimizer by providing additional information about potential bottleneck operators and heated paths in the system. This way the optimizer can adapt the global query plan and use the same deployment algorithm to re-deploy its operators.

6.3 Non-shared (Traditional) Systems

To our knowledge, there is very little work focusing on deployment and scheduling of query plans on multicore machines. While in this paper we work on shared work systems, the approach of using RAVs and basing the deployment on temporal and spatial considerations can also be used in conventional systems. RAVs can be obtained by instrumenting and observing ongoing execution of queries (much as, e.g., selectivity estimates, result caching, hints for indexing, and data statistics are collected today). Using RAVs the query optimizer can be extended to consider the CPU and memory requirements of the operators and use the algorithm proposed in this paper to identify how many cores to allocate to a plan and how to deploy the operators of the plan among the cores that have been allocated. Obviously, the more complex the plan in terms of overall costs, data movement, and number of operators, the more gains are to be expected from using an approach such as the one outlined in this paper.

Although our prototype has been evaluated using SharedDB, we believe that our techniques generalize beyond this kind of shared work systems. In fact most of the placement decisions apply to conventional query plans. For instance, blocking operators within the same query plan can be placed on the same bin (core). Operators streaming to each other can be placed on adjacent cores. Operators that will not be active at the same time can be placed on the same core. Operators active at the same time but complementary in terms of resource usage (CPU vs. memory bound) can be placed on the same core, etc. The spatial scheduling in these systems would additionally have to take into account the physical data placement and data access patterns of the operators (similar to [34]), in order to decide the most suitable cores/NUMA regions (spatial scheduling subproblem) but, overall, the same concepts as those used here would apply.

7. RELATED WORK

Scheduling on multicores has inspired a lot of research targeting different systems, which we cover here as part of related work.

7.1 General Scheduling

There is a considerable amount of research aimed at contention-aware scheduling on multicore machines (Zhuravlev et al. [44] provide a comprehensive survey). These efforts (e.g. [7,26,43]) achieve higher resource efficiency by classifying the application's behaviour and assigning cores so that contention, typically in the memory subsystem, is reduced.

Additionally, there are several methods for characterizing and modeling thread performance on multicores [32,41], and their possible interference when sharing resources [6, 11]. Some of these models are used to aid comprehension and optimization of code performance, while others are used to minimize overall resource contention and performance degradation. Although these methods provide valuable insight in the performance bottlenecks of multicore systems as well as techniques to identify and alleviate contention, they are application-agnostic and consequently unable to provide any performance guarantees to the executing application. Furthermore, none of these optimizes the overall resource allocation so as to maximize their utilization efficiency.

7.2 Scheduling for Databases

Databases, traditionally, have dealt pessimistically with the challenges imposed by modern hardware by exclusively allocating (or assuming exclusive access to) hardware resources such as cores and memory. This practice leads to hardware resources being overprovisioned and underutilized.

However, scheduling becomes an increasing topic of interest in databases. For example, cache-aware scheduling (e.g. [12,27]) concentrates on minimizing the cache-conflicts and benefiting of constructive cache sharing via cache-aware scheduling on multicore machines. Existing work by Porobic et al. [35] also argues that the topology of modern hardware favors coupling communicating-threads and deploys them onto cores on the same 'hardware island' (NUMA node, or CPU-socket) in order to minimize cross-node communication. Follow up work enhances system's data-locality and reduces redundant bandwidth traffic by providing means for suitable data placement and adaptive re-partitioning techniques as the workload changes [34]. Leis et al. [28] take it a step further and propose a novel morsel-driven query execution model which integrates both NUMA-awareness and fine grained task-based parallelism. This allows for maximizing the utilization of the CPU resources and provides elasticity with respect to load balancing the resource allocation to dynamic query workloads. Furthermore, they also advocate the deployment of operator pipelines and assign hard CPU affinities to threads in order to maintain locality and stability.

These examples provide highly valuable techniques, mechanisms and execution models but none uses the knowledge at hand to solve the problem we address, which is how to use operator characteristics and inter-thread relationships to minimize the total number of resources allocated to a query plan without affecting performance and predictability. Nevertheless, we corroborate previous works' observations that leveraging the knowledge of the underlying hardware for operator deployment is essential for minimizing bandwidth traffic and maximizing data locality [28, 35].

8. CONCLUSION

In this paper we address the problem of minimizing resource utilization for complex query plans on modern multicore architectures without affecting performance or sacrificing desired performance properties such as stability and predictability.

We base our solution on two contributions: (i) *resource activity vectors* (RAVs), an abstraction for the performance profile of each relational operator that can be derived from offline measurements, and (ii) a deployment algorithm that computes the minimum amount of resources needed and which proposes an optimal assignment of the operator threads to processor cores.

Our evaluation confirms that RAVs can accurately characterize database operators and that our deployment algorithm significantly reduces the computational resource requirements, while leaving performance unaffected.

As a continuation to this work, we plan to evaluate and possibly augment our model to accommodate alternative scenarios to the ones we presented here. We consider an on-line monitoring of the resource utilization of the operator threads, and operator-to-core mapping which will additionally exploit runtime information about the machine topology and overall system-state.

Acknowledgements

The authors would like to thank the anonymous reviewers for useful feedback to improve this paper. Part of this work has been funded through a grant from Oracle Labs. Jana Giceva is supported by a Google PhD Fellowship.

9. REFERENCES

[1] Advanced Micro Devices, Inc. (AMD). *BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors*, 2013.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *VLDB '99*, pages 266–277.

[3] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB '12*, 5(10):1064–1075.

[4] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. Perez. The DataPath system: a data-centric analytic processing engine for large data warehouses. In *SIGMOD '10*, pages 519–530.

[5] C. Balkesen, J. Teubner, G. Alonso, and M. T. Ozsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. *ICDE '13*, 0:362–373.

[6] M. Banikazemi, D. Poff, and B. Abali. PAM: a novel performance/power aware meta-scheduler for multi-core systems. In *SC '08*, pages 39:1–39:12.

[7] M. Bhadauria and S. A. McKee. An approach to resource-aware co-scheduling for CMPs. In *ICS '10*, pages 189–199.

[8] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali. A Case for NUMA-aware Contention Management on Multicore Systems. In *PACT '10*, pages 557–558.

[9] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR '05*, volume 5, pages 225–237.

[10] G. Candea, N. Polyzotis, and R. Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *PVLDB '09*, 2(1):277–288.

[11] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *HPCA '05*, pages 340–351.

[12] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *SPAA'07*, pages 105–115.

[13] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for NP-hard problems. In D. S. Hochbaum, editor, *Approximation algorithms for bin packing: a survey*, pages 46–93. 1997.

[14] S. Das, V. R. Narasayya, F. Li, and M. Syamala. CPU Sharing Techniques for Performance Isolation in Multitenant Relational Database-as-a-Service. *PVLDB '13*, 7(1):37–48.

[15] G. Dósa. The tight bound of first fit decreasing bin-packing algorithm is $FFD(I) \leq 11/9OPT(I) + 6/9$. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pages 1–11. Springer, 2007.

[16] P. J. Drongowski and B. D. Center. Basic Performance Measurements for AMD Athlon 64, AMD Opteron and AMD Phenon Processors. *AMD whitepaper*, 25, 2008.

[17] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: killing one thousand queries with one stone. *PVLDB '12*, 5(6):526–537.

[18] G. Giannikis, D. Makreshanski, G. Alonso, and D. Kossmann. Shared Workload Optimization. *PVLDB '14*, 7(6).

[19] J. Giceva, T.-I. Salomie, A. Schüpbach, G. Alonso, and T. Roscoe. COD: Database/Operating System Co-Design. In *CIDR '13*.

[20] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: a simultaneously pipelined relational query engine. In *SIGMOD '05*, pages 383–394.

[21] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2008.

[22] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 3A, 3B, and 3C: System Programming Guide*, 2013.

[23] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *EDBT '09*, pages 24–35.

[24] S. Khuller and B. Saha. On finding dense subgraphs. In *Automata, Languages and Programming*, volume 5555 of *Lecture Notes in Computer Science*, pages 597–608. Springer Berlin Heidelberg, 2009.

[25] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs. *PVLDB '09*, 2(2):1378–1389.

[26] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems. *Micro '08*, 28(3):54–66.

[27] R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang. MCC-DB: minimizing cache conflicts in multi-core processors for databases. *PVLDB '09*, 2(1):373–384.

[28] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age. In *SIGMOD '14*, pages 743–754.

[29] Y. Li, I. Pandis, R. Müller, V. Raman, and G. M. Lohman. NUMA-aware algorithms: the case of data shuffling. In *CIDR '13*.

[30] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *PVLDB '00*, 9(3):231–246.

[31] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *TCCA '95*, pages 19–25.

[32] A. Merkel, J. Stoess, and F. Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *EuroSys '10*, pages 153–166.

[33] B. Mozafari, C. Curino, and S. Madden. DBSeer: Resource and Performance Prediction for Building a Next Generation Database Cloud. In *CIDR '13*.

[34] D. Porobic, E. Liarou, P. Tozun, and A. Ailamaki. ATraPos: Adaptive transaction processing on hardware Islands. In *ICDE '14*, pages 688–699.

[35] D. Porobic, I. Pandis, M. Branco, P. Tözün, and A. Ailamaki. OLTP on Hardware Islands. *PVLDB '12*, 5(11):1447–1458.

[36] I. Psaroudakis, M. Athanassoulis, and A. Ailamaki. Sharing Data and Work Across Concurrent Analytical Queries. *PVLDB '13*, 6(9):637–648.

[37] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-Time Query Processing. In *ICDE '08*, pages 60–69.

[38] T.-I. Salomie, I. E. Subasu, J. Giceva, and G. Alonso. Database engines on multicores, why parallelize when you can distribute? In *EuroSys '11*, pages 17–30.

[39] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *SIGMOD '10*, pages 351–362.

[40] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *PVLDB '09*, 2(1):706–717.

[41] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM '09*, 52(4):65–76.

[42] J. Zhou and K. A. Ross. Buffering database operations for enhanced instruction cache performance. In *SIGMOD '04*, pages 191–202.

[43] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS XV '10*, pages 129–142.

[44] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv. '12*, 45(1):4:1–4:28.

[45] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: dynamic bandwidth sharing in a DBMS. In *VLDB '07*, pages 723–734.