Workload-Driven Data Placement for Tierless In-Memory Database Systems

Ben Hurdelhey[1], Marcel Weisgut[2], Martin Boissier[3]

Abstract: High main memory consumption is a significant cost factor for in-memory database systems. Tiering, i.e., placing parts of the data on memory or storage devices other than DRAM, reduces the main memory footprint. A controlled data placement can assign rarely accessed data to slow devices while frequently used data remains on fast devices, such as main memory, to maintain acceptable query latencies. We present an automatic data placement decision system for the in-memory database Hyrise. The system organizes the memory and storage devices in a tierless pool, with no fixed device class categorization or performance order. The system supports data placement use cases, such as minimizing end-to-end query latencies and making cost-optimal purchase recommendations in cloud environments. In this paper, we introduce an efficient calibration process to derive cost models for various storage devices. To determine data placements, we introduce a linear programming-based approach, which yields optimal configurations, and an efficient heuristic. With a set of main memory and SSD devices, we can reduce the main memory consumption for base table data of the TPC-DS benchmark by 74 percent when accepting a workload latency increase of 52 percent. In a comparison of data placement algorithms and cost models, we find that simplistic algorithms (e.g., greedy algorithms) can present viable alternatives to optimal linear programming algorithms, especially under cost prediction inaccuracies.

Keywords: Tiering; Data Placement; In-Memory Database Systems; Linear Programming; Cost Models

1 Data Placement for In-Memory Database Systems

In contrast to traditional disk-based database management system (DBMS), in-memory database management systems (IMDBMSs) store their data in dynamic random-access memory (DRAM) instead of comparatively slow hard disk drives (HDDs) or solid state drives (SSDs). Holding all data in main memory allows for faster query processing, which in turn facilitates business applications, for example, by flexibly computing aggregates on-the-fly [ÖTT17, Pl14]. However, IMDBMSs inherently come with high main memory consumption, which can result in increased operating costs for pure in-memory database systems [Lo19]. The continuously growing amounts of data aggravate this problem, increasing the need for larger-than-memory DBMS [Ma16]. Furthermore, DRAM capacity growth is slowing down and is expected to reach an upper limit [Ma02, Sh20].

Concluding, we argue that it can be desirable to reduce the main memory consumption of IMDBMS. Tiering, i.e., placing selected data on other memory/storage devices with a lower cost per dollar, is a viable strategy to reduce the DRAM consumption [Du16]. When

[1] Hasso-Plattner-Institut, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Deutschland ben.hurdelhey@student.hpi.de
[2] Hasso-Plattner-Institut, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Deutschland marcel.weisgut@hpi.de
[3] Hasso-Plattner-Institut, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Deutschland martin.boissier@hpi.de

moving data from DRAM to devices such as persistent memory (PMem) or SSD, we have to keep in mind that these memory/storage devices offer worse performance characteristics than DRAM, i.e., higher latencies and lower bandwidths. Moving the entire stored database to another device could therefore cause substantial data access cost increases and, thus, increased query latencies. However, data access in database queries is often highly skewed because some part of the data is frequently accessed while another part is only rarely or never queried. This skew is demonstrated by analyses of production database systems by Höppner et al. [HWR14, p. 68], Dreseler [Dr22, p. 12], and Boissier et al. [BSU18, p. 210]. With tiering, we can exploit data access skew by preferably storing the infrequently accessed data on the slower but less expensive memory/storage devices. The critical challenge is determining a *data placement* [Dr22, Vo20], i.e., an assignment of data to devices, to minimize the runtime performance impacts while reducing the DRAM usage. We refer to the series of instructions that determines a data placement as the *placement algorithm*.

We make the following contributions to advance automatic data placement for IMDBMS.

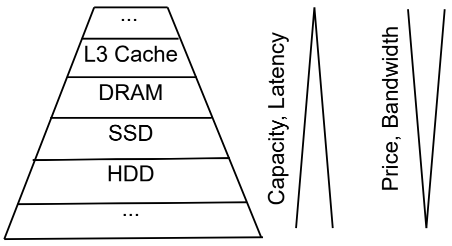
- We propose a placement system for columnar relational in-memory database systems with horizontally partitioned tables. The system is based on linear programming (LP) algorithms and supports multi-constraint multi-device data placement decisions before and after the hardware purchase (Sect. 2). We exemplarily implemented the placement selection system for the open-source in-memory research database Hyrise [Dr19].
- For efficient and accurate placement cost prediction, we propose and evaluate a calibrated cost model based on access tracking data, which allows for efficient cost prediction (Sect. 3).
- We propose and compare multiple placement algorithms and determine a Pareto-optimal trade-off between resource efficiency and result quality, demonstrating that simple algorithms (e.g., Greedy, Knapsack) can be viable alternatives to optimal linear programming algorithms (Sect. 4).

2 Automatic Placement Decisions

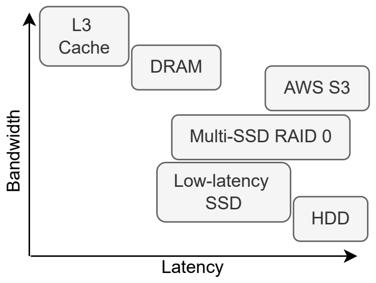
Our data placement approach has the following general characteristics.

- The data placement module works **autonomously**. Manual database administration is laborious and error-prone [Ma21]. Placement decisions can be complex, and it can be infeasible to manually determine the optimal data placement.
- While it is possible to place temporary data structures used during query processing on secondary devices (i.e., devices other than DRAM) [Da21], we focus on evicting append-only data structures of the database system's base tables. Eviction of temporary data structures is required for database operator execution when an operator requires more DRAM than available.
- Our goal is to move infrequently accessed data to slower devices to maintain acceptable query latencies. The *workload* of the database, i.e., the queries being run, determines the *access characteristics* of the stored data. We incorporate access tracking information. Thus, our approach is **workload-driven**.

- While some of the related research has considered only two devices [BSU18, Dr22, La22], we examine data placement techniques that support an **arbitrary number of devices**. In addition, we organize the devices in a **tierless pool**, with no fixed categorization or performance ordering of the devices, similar to [Vo20]. Contrarily to the traditional memory and storage hierarchy shown in Fig. 1a, we regard the devices as separate entities with different access characteristics, as illustrated in Fig. 1b. The term *tiering* is widely used and understood [BSU18, Dr22, Du16]. However, we use the term *devices* instead of *tiers* to not imply any fixed categorization or ordering of the devices by their price or access performance[4].



(a) Traditional memory and storage hierarchy.



(b) Bandwidth and latency of exemplary device classes.

Fig. 1: Tierless device pool concept. Device characteristics based on [Am21, Ap19, Dr22, Wu21] and `fio` [Ax22] measurements for the devices in Sect. 2.2 as specified at `https://github.com/benrobby/hyrise-data-placement#fio`.

2.1 Data Placement in Hyrise

We propose a data placement selection module for Hyrise. The module's main objective is to support placement algorithm and cost model experiments. The placement module consists of a plugin for Hyrise, which interacts with the database system, and a standalone module to determine the data placements[5].

Hyrise is a columnar in-memory database system. As described by Dreseler et al. [Dr19, p. 316], tables are horizontally partitioned into *chunks*, whereas chunks are mutable as long as data is added and become immutable and read-optimized once their capacity is reached. With the default chunk size used in this work, a single chunk stores 65 535 tuples. Each chunk is vertically partitioned into segments, whereas each segment corresponds to one fraction of a column of the table. Each segment can be encoded independently. Dictionary encoding is the default encoding in Hyrise, which we also use in this work. Hyrise uses

[4] The Encyclopædia Britannica defines a *tier* as "a row or layer of things that is above another row or layer". The definition can be found at `https://www.britannica.com/dictionary/tier`

[5] More information on the placement selection module for Hyrise can be found at `https://github.com/benrobby/hyrise-data-placement`.

multi-version concurrency control (MVCC) to isolate concurrent transactions. Instead of updating a row, a new row is written, and the old row is marked as invalid [Dr19, p. 320].

The *placement granularity* defines the data unit at which we make placement decisions and move data between tiers. We make placement decisions at a segment granularity to capture both unused columns and vertical skew with multi-dimensional access tracking and placement decisions [Dr22, pp. 97–100].

Access tracking allows observing the access frequencies of stored data. This information is crucial for deciding on which device a specific part of the data is to be stored on. Corresponding to the segment granularity of the placement decisions, we track data accesses for each individual segment using Hyrise's per-segment access counters [Dr22, pp. 102–104]. For each segment, Hyrise maintains access counters for sequential, monotonic, random, and point *access patterns* [Dr22, p. 92].

Hyrise uses `C++17's` polymorphic memory resources (PMRs) to encapsulate the allocation behavior on different devices and add eviction capabilities to arbitrary data structures [Dr22, p. 79-81]. We supply custom memory resources to allocate and deallocate memory on given devices. This approach is based on an existing implementation for Hyrise [We22]. The memory resources use jemalloc to manage the memory allocations and deallocations. In particular, the resource for block devices supplies hooks (i.e., function pointers) to jemalloc to control the underlying memory allocations of the memory allocator. These hooks allocate memory from memory-mapped UMap regions corresponding to one file on the given device (e.g., on an SSD). *UMap* [Pe19] is a user-space page fault handler that allows for configurations such as adjusting the page size or buffer size. Limiting the memory mapped buffer size gives us control over UMap's eviction behavior. For our experiments, restricting the memory mapped buffer size is crucial so a device does not degenerate to a buffer that can hold all stored data in DRAM. Therefore, we limit the UMap buffer size to 250 MB. Previous research found the optimum page size to be workload-dependent [We22, p. 1205]. However, we use a fixed page size of 128 KiB as this configuration is not our research focus. With the segment granularity, we migrate given segments between devices during the runtime of the DBMS. The database system might perform work during the migration and even access the exact segment currently being migrated. For this reason, we first copy the segment to the new device and then replace the old segment in the chunk with the new segment using an `std::atomic_store` [Dr22, p. 83].

2.2 Multi-Objective Data Placement with Linear Programming

In modern cloud environments, traditional on-premise assumptions regarding the device purchase no longer apply. For example, the device hardware is no longer purchased and used throughout the entire device lifespan. Instead, cloud environments allow users to easily migrate between different compute and storage hardware[6] with pay-as-you-go pricing models. This ability to flexibly reconfigure the used devices allows for new data placement

[6] For example, virtual machines can be rented and migrated on Amazon AWS (`https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-resize.html`).

applications throughout the entire DBMS life-cycle. Our placement selection module supports the following three use cases, which we refer to as *objectives*. For objective O1, we assume the devices are already purchased, and their respective byte capacities constrain the data placement. The goal is to achieve the best possible query runtime performance within the memory budget. Objective O2 occurs during the purchase phase of database deployment. This objective aims to determine purchase recommendations that satisfy a latency constraint for the workload runtime while minimizing the monetary costs for the memory and storage devices. Objective O3 aims to determine purchase recommendations that minimize the predicted query runtimes given a fixed monetary budget for memory/storage devices. We formulate these placement selection problems as linear programming models.

Objective O1: Byte Capacity Constrained Placement Configurations The first objective that our solution supports is minimizing the end-to-end runtime of the database system for a given set of devices, each with a fixed byte capacity. The task of the placement selection algorithm is to determine values for the decision variable $x_{t,a,p,d}$, which corresponds to assigning a segment in table t, attribute a, partition p, to a device d. T and D are the numbers of tables and devices. A and P are the maximum numbers of attributes and partitions over all tables. For these numeric parameters, we use the notation $t \in T$, which is equivalent to $t = 1, ..., T$, for improved readability. For example, for three devices, $d \in D \equiv d \in \{1, 2, 3\}$.

$$minimize_{x_{t,a,p,d} \in \{0,1\}^{T \times A \times P \times D}} \quad \sum_{\substack{t \in T, a \in A,\\ p \in P, d \in D}} x_{t,a,p,d} \cdot c_{t,a,p,d} \tag{1}$$

$$s.t. \quad \sum_{\substack{t \in T, a \in A,\\ p \in P}} x_{t,a,p,d} \cdot s_{t,a,p} \leq b_d \qquad \forall d \in D \tag{2}$$

$$(\sum_{d \in D} x_{t,a,p,d}) = i_{t,a,p} \qquad \forall t \in T, a \in A, p \in P \tag{3}$$

The objective (1) of the integer linear programming (ILP) model is to minimize the predicted costs $c_{t,a,p,d}$ of all segment assignments. Depending on the cost model's accuracy, the objective value can become an accurate runtime prediction for data placements. The LP objective is subject to two constraints. The constraint (2) ensures that the capacity of each device b_d is not exceeded by the accumulated segment size $s_{t,a,p}$ of the segments assigned to it. Furthermore, the constraint (3) requires the model to assign each segment to exactly one device: $i_{t,a,p} \in \{0, 1\}$ is a binary function guaranteeing that only existing segments (due to some tables having more attributes than other tables) will be assigned to a device. Contrarily, non-existing segments will not be assigned to any device.

Objective O2: Latency Constrained Buying Recommendations Objective O2 assumes that the user, i.e., the database administrator, wants to pose latency constraints on the database system and spend as little money as possible on devices to satisfy these constraints. In our case, we select the latency constraint that the cumulative runtime of all queries in a given workload must not exceed a given maximum latency. The algorithm minimizes the dollar costs for the devices required to comply with the given latency constraints. For this, the algorithm determines the hypothetical data placement that satisfies the given latency constraint. In the resulting data placement, the memory or storage usage per device will become the buying recommendation for the memory/storage devices.

$$minimize_{x_{t,a,p,d} \in \{0,1\}^{T \times A \times P \times D}} \quad \sum_{\substack{t \in T, a \in A, \\ p \in P, d \in D}} x_{t,a,p,d} \cdot s_{t,a,p} \cdot g_d \tag{4}$$

$$s.t. \quad \sum_{\substack{t \in T, a \in A, \\ p \in P, d \in D}} x_{t,a,p,d} \cdot c_{t,a,p,d} \leq o \tag{5}$$

$$(\sum_{d \in D} x_{t,a,p,d}) = i_{t,a,p} \qquad \forall t \in T, a \in A, p \in P \tag{6}$$

The ILP model is similar to the O1 model. We introduce the maximum runtime cost value o and the variable g_d for the cost of a device d in dollars per byte. The objective (4) of this ILP model is to minimize the total dollar cost of the data placement. The model calculates the dollar cost for a specific data placement using the segment sizes in bytes and the respective device prices in dollars per byte. The constraint (5) calculates the total predicted runtime cost value and poses an upper limit to this cost.

Our cost model's runtime predictions are inaccurate. Therefore, we infer the maximum runtime cost value o from a given maximum end-to-end latency using experimentally-determined linear interpolation. Calculating the parameters for this formula requires measurements of end-to-end database execution times, which can be slow. Furthermore, a recalculation is necessary for every database and hardware change. For these reasons, we consider improving the underlying cost model to output more accurate end-to-end runtime predictions as the critical challenge.

Objective O3: Dollar-Budget Constrained Buying Recommendations The third objective serves the use case that a user has a certain amount of money to spend on their infrastructure. While allocating more central processing unit (CPU) resources can be a strategy to reduce query runtimes, we focus on a given budget for memory and storage devices. With their monetary budget m, the user wants to buy the devices that allow for the best runtime performance. The parameter g_d stores the cost of a device d in dollars per byte.

$$minimize_{x_{t,a,p,d} \in \{0,1\}^{T \times A \times P \times D}} \quad \sum_{\substack{t \in T, a \in A, \\ p \in P, d \in D}} x_{t,a,p,d} \cdot c_{t,a,p,d} \tag{7}$$

$$s.t. \quad (\sum_{\substack{t \in T, a \in A, \\ p \in P, d \in D}} x_{t,a,p,d} \cdot s_{t,a,p} \cdot g_d) \leq m \tag{8}$$

$$(\sum_{d \in D} x_{t,a,p,d}) = i_{t,a,p} \qquad \forall t \in T, a \in A, p \in P \tag{9}$$

The objective (7) of this ILP model is to minimize the predicted runtime costs of the data placement. The objective is subject to two constraints. The constraint (8) asserts that the data assignment to the devices does not exceed the monetary budget. Furthermore, this ILP model also includes constraint (9) that forces segments to be assigned to exactly one device.

In the above model for objective O3, we considered the dollar costs of the devices as a continuous price in dollars per used byte. This assumption can hold for fine-granular storage rentals from cloud service providers. However, we are limited to a discrete set of available byte capacities when purchasing raw storage device hardware. To support discrete device capacities, we replace the constraint (8) with the constraint (11). For each device d, we assume a given list of length J that is sorted in ascending order. The list contains the discrete device capacities $e_{d,j}$ where $j = 1..J$ indexes the J distinct available capacities. Furthermore, equation (10) introduces a variable $s_{x,d}$ for the size in bytes of the segments assigned to a device d according to the values of the decision variables $x_{t,a,p,d}$.

$$s_{x,d} = \sum_{\substack{t \in T, a \in A, \\ p \in P}} x_{t,a,p,d} \cdot s_{t,a,p} \tag{10}$$

$$(\sum_{d \in D} g_d \cdot argmin_{e_{d,j}, s_{x,d} \leq e_{d,j}} e_{d,j}) \leq m \tag{11}$$

The formalization above contains a non-linearity in the *argmin* expression (11) as it is not a linear combination of its input variables. Therefore, we cannot solve this model in the presented form using linear solvers. However, the non-linearity can be substituted by an equivalent formulation in linear terms.

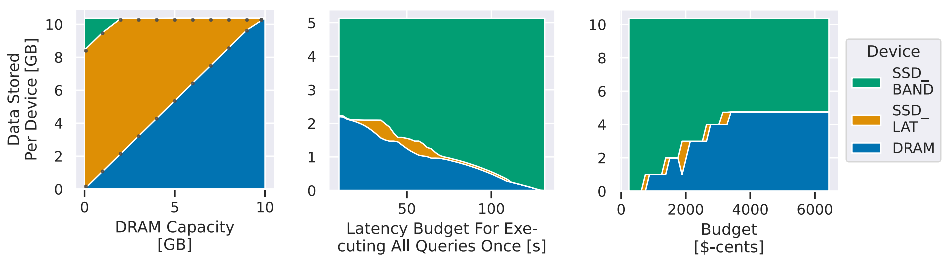
Evaluation Setup We conduct our measurements on a machine with two AMD EPYC 7F72 CPUs, each with 24 physical cores, 48 threads, a 192 MB shared L3 cache, and 256 GB of DDR4 memory with a theoretical per-socket memory bandwidth of 204.8 GB/s [Ad21]. Per CPU, the DRAM is distributed across eight Samsung M393A4G43AB3 dual in-line memory modules (DIMMs), each with a size of 32 GB. We pin processes and memory to a single node using `numactl` for our measurements on this multi-socket system. The machine runs Ubuntu 22.04 LTS with the Linux kernel 5.15.0-41-generic. We compile Hyrise with GCC 10.3 and use Python 3.10.4 to execute the placement selection module. We configure Hyrise to use all available cores on the given non-uniform memory access (NUMA) node.

For our measurements, we set the number of cores that Hyrise can use to 24, which is the number of available physical cores on one NUMA node of our test system. Furthermore, we set the number of clients to eight. Each client corresponds to one stream of queries we send to the database system concurrently. Moreover, we randomize the order of the queries for each client to utilize the database's resources evenly. In combination, our parameter values for the number of cores and clients allow us to measure the multi-threaded database performance. To evaluate the query latencies, we execute queries of the join order benchmark (JOB), TPC Benchmark DS (TPC-DS), and TPC Benchmark H (TPC-H) benchmarks. Unless specified otherwise, we set the scale factors of TPC-H and TPC-DS to ten to obtain a data size that exceeds the benchmark machine's cache sizes. The JOB does not have a scale factor.

In our experiments, we use the following three devices: DRAM, a redundant array of independent disks (RAID) of two Micron 7450 NVM Express (NVMe) SSDs (*SSD_BAND*), and a low-latency Intel Optane DC Series SSD (*SSD_LAT*). The SSD RAID is of type 0. SSD_LAT has a lower access latency, while SSD_BAND offers higher bandwidth. In measurements with the `fio` utility [Ax22] and the Intel Memory Latency Checker (MLC) [Vi21], the devices DRAM, SSD_BAND, and SSD_LAT had a read latency of 130, 70 000, and 12 000 nanoseconds, respectively. Furthermore, the sequential multi-core read bandwidth was 141.3, 12.5, and 2.4 GB per second. As a baseline, we also investigated a second set of devices consisting of DRAM, an SSD, and an HDD. We refer to this set as the *ordered device set*, opposed to the *tierless device set*. These devices have a clear ordering by their access performance.

We use the commercially available Gurobi solver [Gu21a] to solve the specified LP models. Previous research has found this solver to offer good runtime performance for similar applications [Bo22, p. 785] compared to other solvers, such as SCIP [Ga20] or Cbc [Lo03]. The optimality gap [Gu21b] is the maximum accepted gap between the objective value of the solution that the solver terminates with and the optimal objective value. We set this optimality gap to 1% as we experienced solver timeouts for smaller values. Furthermore, we limit the maximum execution time to 500 seconds. Finally, we set the number of threads to the number of cores of the test machine, i.e., 24 threads. We formulate the LP models using Pyomo [By21, HWW11], a Python-based open-source optimization modeling language that allows for interchangeable solvers.

Exemplary Placement Decisions Fig. 2 shows exemplary placement decisions of our system for the TPC-H benchmark using the three objectives O1, O2, and O3. The placement behavior for the JOB and TPC-DS benchmark shows similar patterns. For each given constraint (e.g., DRAM capacity, latency budget, dollar budget) that we vary along the x-axis, the plots display the percentage of data stored on the respective devices on the y-axis. Fig. 2a shows the objective O1 placements. For each DRAM budget, the algorithm uses all available DRAM as it allows for the lowest predicted workload runtime. We define the



(a) O1 device usages. The DRAM budget is varied between zero and ten GB in 11 steps.

(b) O2 purchase recommendations. The latency budget is varied between 9 and 132 seconds in 50 steps.

(c) O3 purchase recommendations for discrete device sizes. The dollar budget is varied between 241 and 6436 dollars in 50 steps.

Fig. 2: Exemplary Data Placements for objective O1, O2, and O3 for the TPC-H benchmark with scale factor ten. TPC-H scale factor five for the O2 purchase recommendations.

workload runtime as the runtime required to execute all queries of a workload (e.g., all TPC-H queries) once. For a DRAM budget of zero GB, most segments are assigned to SSD_LAT while SSD_BAND holds two GB of data. The latency-optimized SSD_LAT holds the segments with predominately random accesses, while SSD_BAND holds segments that are frequently accessed sequentially. In a comparison between the tierless device set and the alternative ordered device set, we found that our placement system can leverage the tierless property adequately. Fig. 2b shows the placements for objective O2. For the minimum latency budget, approximately 45 percent of the data is stored in DRAM. In comparison, the remaining 55 percent of data are unused segments that can be stored on SSD_BAND without affecting the workload latency. For the JOB and TPC-DS benchmark, 25 and 49 percent of the data is unused, respectively. In our algorithm, a post processing step assigns the unused segments to the least expensive device with available capacity. Directly implementing this functionality in the LP model by adding a device penalty to the unused segments' cost proved infeasible. The device penalty value had to be larger than the optimality gap but smaller than the minimum cost for used segments. With the minimum cost for used segments being smaller than the optimality gap in our experiments, this was not possible. Fig. 2c shows the objective O3 placements for discrete device sizes. With an increasing dollar budget, the algorithm can gradually afford more device space for faster devices, such as DRAM. As we artificially limited the available discrete device capacities to integer GB values (e.g., 1 GB, 2 GB, 3 GB), the algorithm affords the devices in steps. The less expensive SSD_LAT precedes the DRAM purchase. Interestingly, the DRAM usage is not monotonically increasing. For a dollar budget of 2000 cents, the model increases the SSD_LAT usage to two GB while it reduces the DRAM usage from two GB to one GB. Random access is the dominant access pattern of the segments assigned to SSD_LAT. As the bandwidth-optimized SSD_BAND has a higher read latency than the other devices, this decrease in DRAM usage thus allows the model to assign more random access-heavy segments to other devices than SSD_BAND and minimize the predicted runtime.

2.3 Dynamic Workloads

We define a workload as dynamic if it has a temporal skew in the queries being run during its duration. Our system supports these workloads by updating the placement regularly. We use windowing to collect access tracking information. The system bases the placement decisions on only the tracked segment accesses that occurred since the last placement update. In our experiments, we set the window size to two minutes. With these periodic updates, we successfully adapted the placement to the changing workload and reduced the query latencies accordingly. However, this reactive approach cannot predict future workloads. Furthermore, recurring workload changes might cause the placement to oscillate between multiple configurations, incurring high segment migration costs. Frequent and fast updates of the data placement are critical to quickly react to workload changes. The update frequency is limited by the runtime of the placement algorithm and the runtime required to apply a placement configuration. Thus, dynamic workload support is an application that requires low-latency placement algorithms, which we investigate in Sect. 4.2.

3 Data Placement Cost Models

We want to investigate how to efficiently estimate data placement effects on query runtime with sufficient accuracy using simple cost models. For this, we build cost models based on (i) Hyrise's segment access counters and (ii) device calibration data.

3.1 Assumptions

All cost models proposed in this work make the following similar underlying assumptions.

I/O-Dominated Workloads Our approach focuses on estimating data access costs. Thus, we do not directly estimate the costs of operators, such as joins or aggregates, executed during query processing. However, we indirectly include their costs as these operators might perform data accesses tracked with Hyrise's access counters. Similarly to Vogel et al. [Vo20, p. 2666], we argue that our focus on data access costs might decrease the absolute accuracy of our runtime predictions. However, the runtime predictions can still be correct in relation to each other. Nevertheless, even this relative measure is subject to the accuracy of the data access runtime cost predictions. As an example for input/output (I/O)-focused runtime predictions, let us consider a workload defined by one query that is executed. For this query, we assume that the CPU-heavy work in the query's operators (e.g., building a hash map in a join operator) is independent of the data access in the operators (e.g., materializing all values of a position list before building the hash map). The CPU-heavy work then adds a static overhead that is independent of the data placement of the segments. Therefore, we can compare the cost estimations of different data placements, and the differences between the estimates are correct, except for a constant overhead. In general, I/O is often a bottleneck for modern CPUs [HSY01]. Thus, I/O-dominated workloads are a relevant category of workloads. We argue that data access costs can be a good approximation for these workloads' overall query latency performance with the previous reasoning.

Independent Data Placement Decisions In this work, a placement decision for one segment can be made independently of a placement decision for another segment. While interactions between placement decisions could be possible, we exclude this case to limit the complexity of the placement algorithms. Similarly, our cost models make independent predictions per segment. This simplification allows us to limit the complexity of the cost models but can lead to inaccuracies. For example, an operator execution might read two columns in parallel, and its execution time might be determined by the maximum scan time of both columns. Furthermore, we consider the devices' access performance characteristics unaffected by our placement decisions. Similarly to the simplification above, this assumption allows us to limit the complexity of the data placement algorithms and cost models. To illustrate where this assumption can fail, let us assume that an operator execution reads two segments in parallel. If these two segments reside on the same device, the device's bandwidth can be impacted, and it could be beneficial to distribute the segments to different devices. On HDDs, multi-threaded reads can even decrease the read throughput due to increased seek time [Vo20, p. 2666].

3.2 Cost Model Definition

We propose the calibrated and workload-based cost model C3. Furthermore, we compare model C3 with previous development iterations C0, C1, and C2. The cost model C3 uses device calibration data and segment access information to estimate the runtime performance impact when a segment is assigned to a specific device.

$$c^3_{t,a,p,d} = \sum_{\gamma \in \Gamma} u_{d,\gamma,\xi_{t,a,p}} \cdot h_{t,a,p,\gamma} \cdot \frac{s_{t,a,p}}{n_{t,a,p}} \tag{12}$$

Model C3 predicts the runtime cost of assigning a segment in table t, attribute a, and partition p to device d using the formula in (12). The formula sums over all access patterns $\gamma \in \Gamma$ (sequential, monotonic, random, and point). Per access pattern, the model calculates the cost prediction as a product of the calibration value $u_{d,\gamma,\xi_{t,a,p}}$, the access counter $h_{t,a,p,\gamma}$ for the respective access pattern, and the byte size per value $\frac{s_{t,a,p}}{n_{t,a,p}}$, where $n_{t,a,p}$ is the number of values in a segment. The calibration value u takes the parameter $\xi_{t,a,p}$ that yields the segment's data type. The set of possible values $\xi_{t,a,p} \in \Xi = \{string_{\neg SSO}, float\}$ contains two data types: non-small string optimization (SSO) strings and floating-point numbers. Our specialized $string_{\neg SSO}$ calibration reads strings with an average length of 44 bytes, which exceeds the SSO threshold. The *float* calibration is used as the default for all other segment data types. SSO is a technique that compilers use to avoid dynamic memory allocations on the heap and improve data locality. As Hyrise stores the values of string segments as C++ `std::string` objects, this optimization applies in Hyrise. If a string's size is below the SSO threshold, the content of the string can be stored on the stack in the string object itself. For example, GCC has an SSO threshold of 15 bytes. Contrarily, the content of strings that exceed this threshold is stored on the heap, which means that the string object has to hold a heap pointer to the underlying string buffer. This additional pointer redirect when reading the string's content is equivalent to random access, as the memory layout of the underlying string buffers is undefined. For strings residing on devices with poor random access characteristics (i.e., a high read latency), access to string segments can incur high runtime costs. We experimentally determine the calibration values $u_{d,\gamma,\xi_{t,a,p}}$ by benchmarking a read workload for each combination of access pattern, device, and data type. For the data type, we consider strings with a length larger than 15 bytes and floating-point numbers with single precision. However, segments of integer data type showed the same calibration values in our experiments. The calibration workload reads all values of a column with an uncompressed total size of 720 MB. Google Benchmark [Go22a] repeats the measurements until a stable runtime is reached. Between benchmark iterations, random data is read to flush the UMap cache and the CPU caches, so we do not measure cache effects. The calibration finishes in less than 90 minutes, though it could be reduced to a fraction of that by reducing the number of values read without significantly sacrificing accuracy.

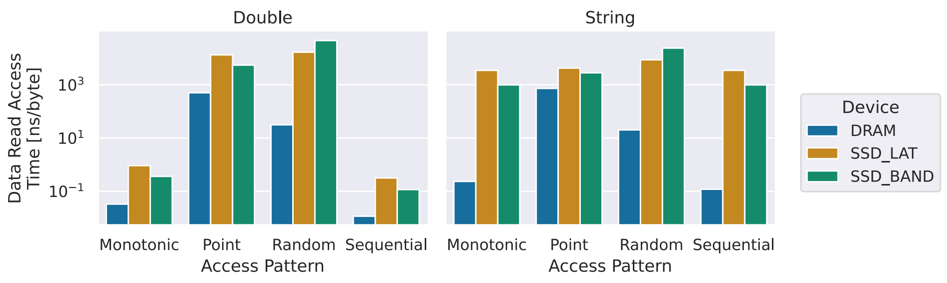


Fig. 3: Device calibration for different access patterns, data types, and devices. Multi-threaded reads (24 threads) measured on TPC-H data with scale factor ten.

Fig. 3 shows the calibrated runtimes for both data types. The calibrated values are the read times normalized as nanoseconds to read one byte with the respective access pattern. For example, the calibrated sequential byte access time for DRAM is 0.011 nanoseconds, equivalent to a read bandwidth of 90 GB per second. This calibrated DRAM bandwidth is smaller than the maximum bandwidth measured with Intel MLC of 141.3 GB per second, demonstrating that the calibration can improve the cost model's accuracy. We explain this difference between theoretical and utilized bandwidth with CPU overhead for decoding and processing the read data. In the calibration data, the low-latency SSD_LAT shows faster random access speed while SSD_BAND is faster for the remaining access patterns. For example, for the sequential accesses, the SSD_BAND achieves a bandwidth of 8.25 GB per second, which exceeds the theoretical maximum bandwidth of SSD_LAT by 3.4 times. The calibration not only allows us to distinguish the runtime performance of the devices but also supplies information about the importance of specific access patterns and data types. Fig. 3 shows that sequential access allows for the fastest read speed while the random access pattern has the highest runtime. Previous research showed that the ability to pre-fetch and cache data significantly influences the runtime performance of random data accesses [He21, p. 32]. Therefore, data accesses that follow the random and point access pattern incur the highest runtime. However, in an exemplary TPC-H benchmark run in

Hyrise, 10^6 as many random accesses as point accesses were recorded. These measured calibration values thus support the findings of Dreseler [Dr22, p. 120] that random access runtime performance greatly influences end-to-end database system runtime for Hyrise. The calibrated runtime for the string data type is significantly higher than for the *float* data type. The geometric mean of the string calibration values is 389.6 nanoseconds, which is 20 times higher than 19.9 nanoseconds, the geometric mean of the *float* calibration. We explain these differences with the second pointer indirection required to read non-SSO strings, which is equivalent to random memory access.

We compare the presented cost model with three iterations shown in (13)-(15). Model C0 does not use calibration data and corresponds to the model proposed by Dreseler [Dr22] using manually-determined weights w_γ. The superscript number v in the cost formula $c^v_{t,a,p,d}$ indicates the cost model version.

$$c^0_{t,a,p,d} = \sum_{\gamma \in \Gamma} w_\gamma \cdot h_{t,a,p,\gamma} \tag{13}$$

$$c^1_{t,a,p,d} = \sum_{\gamma \in \Gamma} u_{d,\gamma} \cdot h_{t,a,p,\gamma} \tag{14}$$

$$c^2_{t,a,p,d} = \sum_{\gamma \in \Gamma} u_{d,\gamma} \cdot h_{t,a,p,\gamma} \cdot \frac{s_{t,a,p}}{n_{t,a,p}} \tag{15}$$

Furthermore, we considered extending our cost model C3 with a cache miss rate prediction, as introduced by Lasch et al. [La22]. However, the proposed function did not model the CPU cache behavior accurately in our experiments. The maximum segment size in Hyrise was one order of magnitude smaller than the last level cache size, and thus the predicted cache miss rate was 0.05 for all segments. Therefore, the authors' assumption that the cache miss rate is independent for each single data structure was not met in our application.

3.3 Evaluation

The upper plots in Fig. 4 show the measured workload runtime for data placements determined with the objective O1 algorithm based on the respective cost model. The runtimes were measured for the JOB, TPC-DS, and TPC-H benchmark with scale factor ten and the DRAM capacity is varied between zero GB and full capacity. We consider only DRAM and SSD_BAND due to the limitations of cost model C0. The data placements for zero GB DRAM capacity and full capacity are thus the same across all cost models because all segments are assigned to the same device. Consequently, the measured query latencies are almost equal for these two cases. For all three workloads, the figures show that DRAM capacity thresholds exist where adding additional capacity does not decrease the end-to-end runtime. These thresholds correspond to the unused data per benchmark. Compared to the C3 cost model, the mean end-to-end measured runtimes for the C0, C1, and C2 models are 31, 18, and 3 percent higher, respectively. The lower plots in Fig. 4 show the predicted runtimes by the respective cost models. The cost models predict the

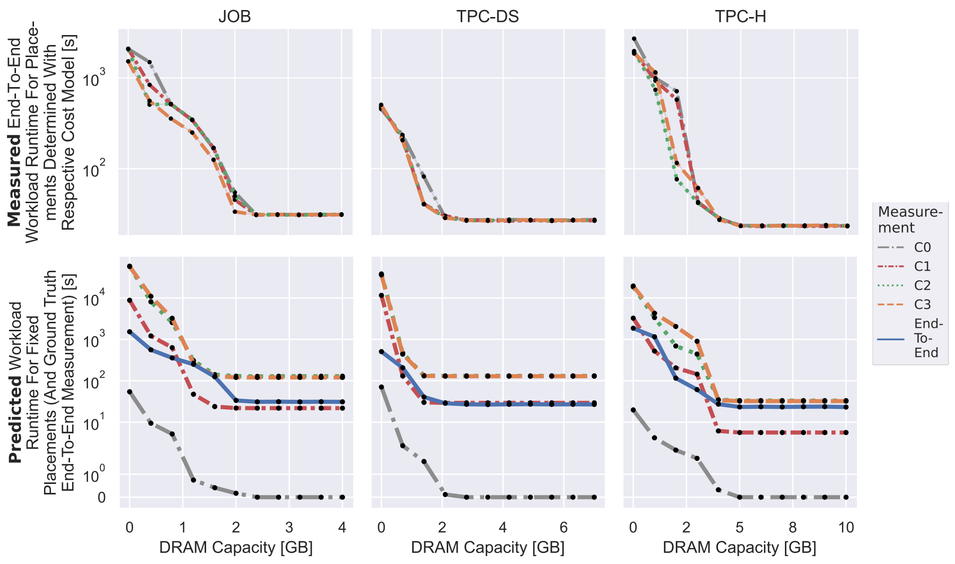


Fig. 4: Comparison of (i) measured end-to-end workload runtime and (ii) runtime prediction accuracy for data placements determined with the objective O1 ILP algorithm based on the respective cost model.

workload runtimes for data placements determined with the O1 algorithm and cost model C3. Model C0 underestimates the runtime by up to two orders of magnitude compared to the end-to-end measured runtimes. We argue that the costs predicted by C0 are on an arbitrary scale as it is not calibrated and cannot be seen as runtime predictions. Model C3 often overestimates the workload runtime. Potential reasons why the end-to-end measured runtimes are lower than expected are that model C3 overestimates data access costs because parallelization and caching speed up data accesses.

In conclusion, we showed that calibrated cost models offer significantly better cost predictions than models with manually-determined weights. To efficiently estimate data placement effects on runtime, we argue that our calibrated cost models are both simple and offer sufficient accuracy for our application. Assessing the accuracy of single cost predictions is challenging because no *gold standard* cost model exists to compare against. Experimentally determining the cost of each segment placement is infeasible due to the large size of the placement decision space. However, the demonstrated end-to-end placement decision and the runtime prediction accuracy of cost model C3 show solid results and allow us to recognize differences between the placement selection algorithms. Further adjustments to the device calibration (e.g., by better modeling the caching behavior) could improve the runtime prediction accuracy. Based on our development efforts, we argue that both (i) the usage of calibration data and (ii) the calibration data quality are significant factors for making calibrated cost models a reliable decision basis for placement selection algorithms.

4 Placement Selection Algorithms

Previous research [Bo22] has shown that simple heuristics can be tractable alternatives to optimal solutions because they often offer comparable results at lower algorithm runtime and memory usage. We investigate which trade-offs users have to make between different data placement algorithms. We compare algorithms for objective O1 as a proxy for all objectives. The algorithms include the LP algorithm introduced in Sect. 2, a second linear programming algorithm that makes placement decisions with column granularity, a greedy heuristic, and a multi-tier Knapsack algorithm.

4.1 Algorithm Descriptions

In the Greedy and Knapsack algorithm, we require computing an access performance metric v_d per device d and calculate it with the formula in (16). The existence of such an access performance metric implies an ordering of the devices by their access speeds (e.g., latency, bandwidth). This ordering contradicts our goal to organize the memory and storage mediums in a tierless device pool with no fixed ordering. However, this sorting is not used in our ILP solution.

$$v_d = \sum_{\gamma \in \Gamma} \Big(\sum_{\xi \in \Xi} u_{d,\gamma,\xi} \cdot \sum_{t \in T, a \in A, p \in P} h_{t,a,p,\gamma} \Big) \tag{16}$$

Multi-Device Greedy The Greedy algorithm orders the devices by their access performance and assigns the segments greedily to the fastest devices. This algorithm bases on the greedy heuristics proposed by Boissier et al. [BSU18, p. 214] and the *HOT* strategy by Vogel et al. [Vo20, pp. 2667–2668]. These two related works use cost models optimized for their respective database systems and intertwine the cost models with the placement selection algorithms. In our work, we focus on the Greedy algorithm itself and distinguish the placement selection algorithm from the cost model.

In the algorithm described in Algorithm 1, we first sort the devices by their access performance v_d in ascending order. To assign segments to devices, we regard the devices in a pairwise manner, starting with the fastest devices. We thus model the decision problem as a series of binary decisions between two devices. For each pair, we compute in Line 6 the segment scores and sort the segments according to these scores in descending order. A segment's score is calculated as the difference between the segment's cost values for the two corresponding devices. The variable $c_{w,d}$ holds the predicted costs for assigning a segment w to device d. We then greedily assign the segments to the current device as the device's byte capacity permits. Assuming S is the number of segments and D the number of devices, the asymptotic complexity of this Greedy algorithm is $O(S \cdot \log S \cdot D + D \log D)$. However, the number of devices D is constantly three in our experiments. Another difference between our Greedy algorithm and the previously-mentioned greedy algorithms from related work is our focus on supporting more than two devices. For example, the *HOT* algorithm at table granularity by Vogel et al. "places tables descending in order of their number of accesses on the fastest device with enough space for the whole table" [Vo20, p. 2668]. In comparison

Algorithm 1: Greedy Placement Selection Algorithm

```
Data: set of all segments $W$, segment sizes, segment costs $c_{w,d}$, device byte capacities, device
      calibration
Result: data placement
sorted devices = sort devices by $v_d$ ascending;
for ($d_i$, $d_{i+1}$) in sorted devices do
    if all segments assigned then
        return data placement;
    end
    sorted segments = sort segments $w \in W$ that are unassigned by $(c_{w,d_{i+1}} - c_{w,d_i})$ descending;
    for segment $w$ of sorted segments do
        if $w$ fits onto device $d_i$ then
            assign $w$ to $d_i$;
        end
    end
end
```

with these related work algorithms, we argue that our Greedy algorithm can produce data placements for multiple devices that induce lower end-to-end query latencies because we model the cost increase between the two devices instead of the absolute costs. With the proposed technique, our Greedy algorithm resembles the cost model usage of our ILP models.

Linear Programming with Column Granularity We compare the objective O1 LP algorithm and an adapted version that uses columns instead of segments as the placement decision unit. We refer to this segment-granular LP algorithm as *LP* and the algorithm that makes decisions at a column granularity as *Column-LP*. In related work by Vogel et al. [Vo20, p. 2674], the authors of the Mosaic storage engine find that their column-granular *HOT column* strategy outperforms the table-granular *HOT table* algorithm by a factor of 1.99. Similarly, we compare column and segment granularities. Our Column-LP is inspired by Mosaic's LOPT optimization model, which also uses column granularity. A complete re-implementation of the LOPT model was infeasible for our comparison because the LOPT model builds on a cost model that is intertwined with the authors' proposed LP model and specific to the database system's execution model.

Multi-Tier Knapsack The multi-tier Knapsack algorithm is an implementation of the multilevel generalized assignment problem (MGAP) simplification proposed by Dreseler [Dr22, p. 113]. The author models the placement selection problem as a series of independent binary decision problems between pairs of devices, similarly to our proposed Greedy algorithms in Sect. 4.1. Due to this simplification, the MGAP approach does not necessarily yield the optimal solution. The algorithm first sorts the devices by their access performance v_d in ascending order. The devices are regarded in a pairwise manner to assign the segments to them, starting with the fastest devices. Each binary decision problem is formulated as a Knapsack problem to determine which segments to place on the current device d_i. All segments not selected for the current device will be reconsidered for the next device pair. Thus, we do not assign segments to device d_{i+1}. The segments correspond to the Knapsack items. The computed segment costs $c_{w,d_{i+1}} - c_{w,d_i}$ are the items' values, and the segment sizes are the items' weights. The byte capacity of the current device defines the size of the Knapsack. We use the Google OR-Tools `KNAPSACK_MULTIDIMENSION_BRANCH_AND_BOUND_SOLVER` with a timeout of 500 seconds [Go22b].

4.2 Evaluation

The upper plots in Fig. 5 show the predicted workload runtime for data placements determined with the respective algorithm. The LP algorithm consistently determines the placements with the lowest runtime predicted by cost model C3. Relative to the LP algorithm's mean predicted runtime, the Column-LP algorithm's placements incur the highest mean predicted runtime with 203% of the LP placement's runtime. The Column-LP is followed by the Greedy algorithm at 136% and the Knapsack algorithm at 115% of the LP placement's runtime. The column-granular decisions of the Column-LP incur significantly higher runtimes because (i) they cannot capture vertical data access skew and (ii) they cannot exhaust the device capacities as efficiently as segment-granular placements. Furthermore, the Knapsack and Greedy algorithms are unable to determine the optimum data placement in some conditions due to their greedy strategy.

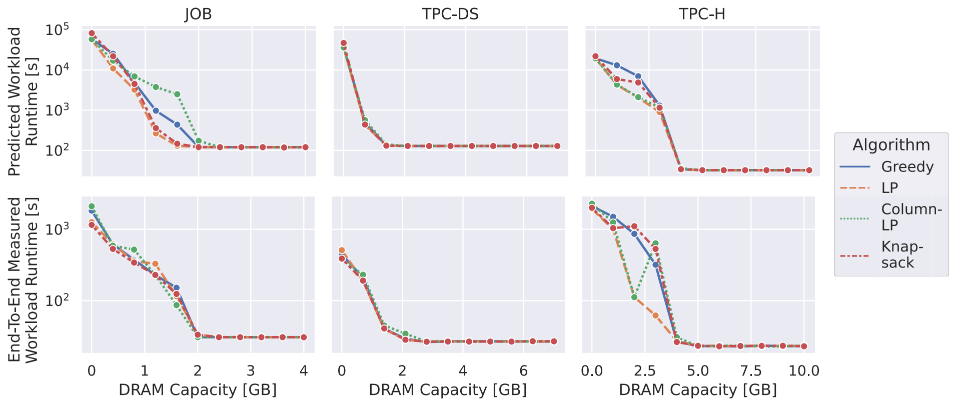


Fig. 5: Predicted and measured runtime of data placements determined with different placement algorithms based on cost model C3. Measurements for JOB, TPC-DS with scale factor ten, and TPC-H with scale factor ten.

The lower plots in Fig. 5 show the end-to-end measured workload runtime for data placements determined with the respective algorithm. Due to cost prediction inaccuracies, not all measured runtimes show the same patterns as the predicted runtimes. In our experiments, we also discovered cases where simpler algorithms (e.g., Greedy, Knapsack) produced placements that incurred lower end-to-end runtimes than the placement determined by the LP algorithm because of these inaccuracies. However, on average, the LP algorithm determines the placements with the lowest end-to-end workload runtime. The Knapsack algorithm's results have the highest mean predicted runtime at 147 percent of the LP algorithm's results. The Greedy algorithm follows at 135 percent and the Column-LP algorithm at 132 percent.

An evaluation of the trade-off between algorithm runtime and solution quality is shown in Fig. 6. The algorithm choice is subject to a pareto-optimal trade-off between both metrics. Compared to the LP algorithm, the Knapsack algorithm's resulting data placements are only 15 percent less optimal, while the algorithm has an 80 percent shorter runtime. Similarly, the Greedy algorithm trades a 36 percent optimality decrease for an 81 percent runtime decrease, making it a viable alternative to the optimal LP algorithm. For example, for the TPC-H benchmark with scale factor 1 000, 1 818 000 segments need to be assigned to the devices. In this context, the LP algorithm takes 60 minutes to determine a data placement, whereas the Pyomo setup time and the Gurobi solver runtime are responsible for one-third of the runtime, respectively. In contrast, the Greedy and Knapsack algorithms terminate within 20 minutes, and the Column-LP algorithm takes only 81 seconds. Memory consumption measurements show similar patterns.

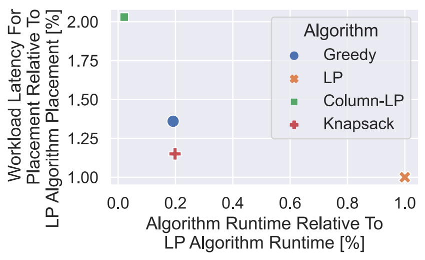


Fig. 6: Mean algorithm runtime and solution quality across JOB, TPC-DS, and TPC-H benchmarks. The solution quality corresponds to the cost model C3's predicted runtime to execute the workload queries once per data placement.

5 Related Work

Several related works on automated decision-making are shown in Tab. 1. Some of the stated research does not determine data placements but related configurations, such as encoding or index selection. Similar to our approach, the Mosaic storage system for the Umbra DBMS [Vo20] determines data placements with a linear optimization model and two greedy algorithms. However, Mosaic uses a cost model focused on sequential reads, matching Umbra's data access patterns. The authors compare column-granular data placements with table-granular placements and find a 1.99× relative speedup. Further related works include the Hybrid Data Layouts for Tiered HTAP Databases [BSU18] and the automatic tiering research by Dreseler [Dr22].

Boissier [Bo22] proposes an optimal linear programming encoding selection algorithm and a greedy heuristic that he often found on par with the optimal solution. The heuristic strategy weighs the candidates by their benefit-to-cost ratio [Va00]. To predict runtime costs of encoding configurations, Boissier uses multiple linear regression models comparable to the work of Ma et al. [Ma21]. Another cost model approach are zero-shot models [HB22].

Related Decision-Making Work	Granularity	Config Optimization				Placement Features			
		Data Placement	Encoding	Index	Storage Layout	Device Count	Objective O1	Objective O2	Objective O3
Mosaic [Vo20]	Columns	●	◐	○	○	n	●	○	●
Hybrid Layout Hyrise [BSU18]	Columns	●	○	○	○	2	●	○	○
Automatic Tiering [Dr22]	Segments	●	○	○	○	2	●	○	○
Encoding Configuration [Bo22]	Segments	○	●	○	○	-	-	-	-
Config Optimization [RSB22]	Segments	●	●	●	○	n	●	○	○
Cost Modeling HANA [La22]	Data Structures	●	○	○	○	2	●	○	○
Proteus and Tiresias [ALD22]	Varying	●	●	●	●	2	●	○	○
This Work	Segments	●	○	○	○	n	●	●	●

Tab. 1: Overview of Related DBMS Configuration Selection Research

However, compared to our calibrated cost models, such learned models have high complexity, long training times, and are hard to generalize.

Richly et al. [RSB22] optimize multiple configuration aspects jointly. The underlying cost model exhaustively calibrates scan operations for various configurations, which results in a long preparation phase to establish cost estimates.

Lasch et al. [La22] propose a data placement cost model for PMem, limiting the number of devices where a data unit can be placed to two (DRAM and PMem). Their model uses (i) device calibration data and (ii) workload information in the form of access counts, similar to our model C3. However, the authors also model the costs for additional data structures and regard cache miss ratios.

Finally, Abebe et al. [ALD22] propose Tiresias, a storage cost model that can predict future workloads to optimize multiple configuration aspects jointly. Such a predictive capability could be useful for our data placement system to anticipate dynamic workload changes.

6 Discussion

We proposed an automatic placement selection system for IMDBMS that supports multiple placement objectives, makes workload-driven placement decisions, and manages its devices in a tierless pool. In our exemplary implementation for the database system Hyrise, we compared several cost models and placement selection algorithms. The proposed cost models are applicable for database systems making similar assumptions as Hyrise (cf. Sect. 3.1), while the proposed placement algorithms have general applicability.

Based on our comparison of an optimal linear programming algorithm, a heuristic based on the Knapsack problem, and a greedy algorithm, we argue that the algorithm choice is subject to a pareto-optimal trade-off between algorithm resource usage and solution quality. The Knapsack and Greedy algorithms are viable alternatives to the resource-intensive LP algorithm, especially under inaccurate cost predictions. Placement granularity significantly influences the solution's optimality, as we found column-granular placements to incur, on average, 103 percent higher query latencies than segment-granular decisions. However, such column-granular decisions can be viable for low-latency applications (e.g., frequent placement updates for dynamic workloads) requiring fast re-computations of the data placement, as the column-granular linear programming algorithm to required up to two orders of magnitude less runtime compared to the segment-granular LP algorithm.

We found our cost model using (i) data access pattern tracking information and (ii) device calibration data to offer sufficient accuracy to distinguish the differences between placement selection algorithms and determine suitable placements. While other approaches, such as learned cost models, can offer higher accuracy, they can be complex, hard to generalize, and require expensive data collection. In comparison, our cost model is inexpensive to calibrate.

Merely moving the unused data from DRAM to secondary devices already allows for significant DRAM usage reductions. For the JOB, TPC-DS, and TPC-H benchmark, the share of data that could be removed from DRAM without increasing the workload latencies were 25, 49, and 55 percent, respectively. Even naive algorithms and cost models can determine such placement decisions, as it suffices to track data accesses.

Future Work To further improve our system's data placements and determine the optimal end-to-end placement, the choice of both the placement algorithm and the cost model is relevant. Placement selection algorithms determining placements close to the optimal solution are a necessary condition. In comparison, an optimal cost model accurately predicting all real-world database system behavior does not exist. Thus, placement cost modeling is a complex challenge that requires further research. Future improvements to our cost model include calibration for all available data types and predicting database system behavior such as caching. Additionally, operator-granular placement cost models could enable robustness guarantees for single query runtimes. Furthermore, an open question is whether the Greedy placement algorithm could be improved with alternative segment sorting metrics (e.g., by their benefit-to-cost ratio). In addition, we consider extending objective O3 to optimize purchases of the entire infrastructure, including CPU resources, a future work item. In this work, we focused on block-level devices as placement alternatives to the system's DRAM. Extending our system to place data on heterogeneous memory with different access qualities, e.g., CPU-local DRAM and Compute Express Link (CXL)-attached [CX22], disaggregated memory, is a potential future effort.

References

[Ad21] Advanced Micro Devices, Inc: AMD EPYC 7F72 - Technical Specification, https://www.amd.com/en/product/9656, 2021, visited on: 10/05/2022.

[ALD22] Abebe, M.; Lazu, H.; Daudjee, K.: Tiresias: Enabling Predictive Autonomous Storage and Indexing. Proc. VLDB Endow. 15/11, pp. 3126–3136, 2022.

[Am21] Amazon Web Services, Inc.: Maximum Transfer Speed between Amazon EC2 and Amazon S3, https://aws.amazon.com/premiumsupport/knowledge-center/s3-maximum-transfer-speed-ec2/, 2021, visited on: 10/05/2022.

[Ap19] Appuswamy, R.; Graefe, G.; Borovica-Gajic, R.; Ailamaki, A.: The Five-Minute Rule 30 Years Later and Its Impact on the Storage Hierarchy. Commun. ACM 62/11, pp. 114–120, 2019.

[Ax22] Axboe, J.: Flexible I/O Tester, https://github.com/axboe/fio, 2022, visited on: 10/04/2022.

[Bo22] Boissier, M.: Robust and Budget-Constrained Encoding Configurations for In-Memory Database Systems. Proc. VLDB Endow. 15/4, pp. 780–793, 2022.

[BSU18] Boissier, M.; Schlosser, R.; Uflacker, M.: Hybrid Data Layouts for Tiered HTAP Databases with Pareto-Optimal Data Placements. In: Proceedings of the IEEE International Conference on Data Engineering, ICDE. Pp. 209–220, 2018.

[By21] Bynum, M. L.; Hackebeil, G. A.; Hart, W. E.; Laird, C. D.; Nicholson, B. L.; Siirola, J. D.; Watson, J.-P.; Woodruff, D. L.: Pyomo–Optimization Modeling in Python. Springer Science & Business Media, 2021.

[CX22] CXL Consortium: Compute Express Link: The Breakthrough CPU-to-Device Interconnect, https://www.computeexpresslink.org, 2022, visited on: 10/09/2022.

[Da21] Daase, B.; Bollmeier, L. J.; Benson, L.; Rabl, T.: Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. Pp. 339–351, 2021.

[Dr19] Dreseler, M.; Kossmann, J.; Boissier, M.; Klauck, S.; Uflacker, M.; Plattner, H.: Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management. In: Proceedings of the International Conference on Extending Database Technology, EDBT. Pp. 313–324, 2019.

[Dr22] Dreseler, M.: Automatic Tiering for In-Memory Database Systems, DOI: 10.25932/publishup-55825, PhD thesis, Universität Potsdam, 2022, 143 pp.

[Du16] Dulloor, S.; Roy, A.; Zhao, Z.; Sundaram, N.; Satish, N.; Sankaran, R.; Jackson, J.; Schwan, K.: Data tiering in heterogeneous memory systems. In: Proceedings of the Eleventh European Conference on Computer Systems, EuroSys. 15:1–15:16, 2016.

[Ga20] Gamrath, G.; Anderson, D.; Bestuzheva, K.; Chen, W.-K.; Eifler, L.; Gasse, M.; Gemander, P.; Gleixner, A.; Gottwald, L.; trin Halbig, K.; Hendel, G.; Hojny, C.; Koch, T.; Bodic, P. L.; Maher, S. J.; Matter, F.; Miltenberger, M.; Mühmer, E.; jamin Müller, B.; Pfetsch, M. E.; Schlösser, F.; Serrano, F.; Shinano, Y.; Tawfik, C.; Vigerske, S.; Wegscheider, F.; Weninger, D.; Witzig, J.: The SCIP Optimization Suite 7.0, `http://www.optimization-online.org/DB_HTML/2020/03/7705.html`, 2020, visited on: 10/04/2022.

[Go22a] Google, LLC: Google Benchmark: A Microbenchmark Support Library, `https://github.com/google/benchmark`, 2022, visited on: 10/08/2022.

[Go22b] Google, LLC: Google OR-Tools, `https://developers.google.com/optimization`, 2022, visited on: 10/04/2022.

[Gu21a] Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual, `https://www.gurobi.com/documentation/9.5/refman/index.html`, 2021, visited on: 10/04/2022.

[Gu21b] Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual - Parameter Documentation: MIPGap, `https://www.gurobi.com/documentation/9.5/refman/mipgap2.html#parameter:MIPGap`, 2021, visited on: 10/05/2022.

[HB22] Hilprecht, B.; Binnig, C.: Zero-Shot Cost Models for Out-of-the-box Learned Cost Prediction. Proc. VLDB Endow. 15/11, pp. 2361–2374, 2022.

[He21] Heinzl, L.; Hurdelhey, B.; Boissier, M.; Perscheid, M.; Plattner, H.: Evaluating Lightweight Integer Compression Algorithms in Column-Oriented In-Memory DBMS. In: International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS. Pp. 26–36, 2021.

[HSY01] Hsu, W. W.; Smith, A. J.; Young, H. C.: I/O reference behavior of production database workloads and the TPC benchmarks - an analysis at the logical level. ACM Trans. Database Syst. 26/1, pp. 96–143, 2001.

[HWR14] Höppner, B.; Waizy, A.; Rauhe, H.: An Approach for Hybrid-Memory Scaling Columnar In-Memory Databases. In: International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS. Pp. 64–73, 2014.

[HWW11] Hart, W. E.; Watson, J.-P.; Woodruff, D. L.: Pyomo: modeling and solving mathematical programs in Python. Mathematical Programming Computation 3/3, pp. 219–260, 2011.

[La22] Lasch, R.; Legler, T.; May, N.; Scheirle, B.; Sattler, K.-U.: Cost Modelling for Optimal Data Placement in Heterogeneous Main Memory. Proc. VLDB Endow. 15/11, pp. 2867–2880, 2022.

[Lo03] Lougee-Heimer, R.: The Common Optimization Interface for Operations Research: Promoting open-source software in the operations research community. IBM J. Res. Dev. 47/1, pp. 57–66, 2003.

[Lo19] Lomet, D. B.: Cost/Performance in Modern Data Stores: How Data Caching Systems Succeed. In: Proceedings of the IEEE International Conference on Data Engineering, ICDE. P. 140, 2019.

[Ma02] Mandelman, J. A.; Dennard, R. H.; Bronner, G. B.; DeBrosse, J. K.; Divakaruni, R.; Li, Y.; Raden, C. J.: Challenges and future directions for the scaling of dynamic random-access memory (DRAM). IBM J. Res. Dev. 46/2-3, pp. 187–222, 2002.

[Ma16] Ma, L.; Arulraj, J.; Zhao, S.; Pavlo, A.; Dulloor, S. R.; Giardino, M. J.; Parkhurst, J.; Gardner, J. L.; Doshi, K. A.; Zdonik, S. B.: Larger-than-memory data management on modern storage hardware for in-memory OLTP database systems. In: Proceedings of the International Workshop on Data Management on New Hardware, DaMoN. 9:1–9:7, 2016.

[Ma21] Ma, L.; Zhang, W.; Jiao, J.; Wang, W.; Butrovich, M.; Lim, W. S.; Menon, P.; Pavlo, A.: MB2: Decomposed Behavior Modeling for Self-Driving Database Management Systems. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. Pp. 1248–1261, 2021.

[ÖTT17] Özcan, F.; Tian, Y.; Tözün, P.: Hybrid Transactional/Analytical Processing: A Survey. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. Pp. 1771–1775, 2017.

[Pe19] Peng, I. B.; McFadden, M.; Green, E. W.; Iwabuchi, K.; Wu, K.; Li, D.; Pearce, R.; Gokhale, M. B.: UMap: Enabling Application-driven Optimizations for Page Management. In: Workshop on Memory Centric High Performance Computing, MCHPC@SC. Pp. 71–78, 2019.

[Pl14] Plattner, H.: The Impact of Columnar In-Memory Databases on Enterprise Systems. Proc. VLDB Endow. 7/13, pp. 1722–1729, 2014.

[RSB22] Richly, K.; Schlosser, R.; Boissier, M.: Budget-Conscious Fine-Grained Configuration Optimization for Spatio-Temporal Applications. Proc. VLDB Endow. 15/13, pp. 4079–4092, 2022.

[Sh20] Shiratake, S.: Scaling and Performance Challenges of Future DRAM. IEEE International Memory Workshop, IMW/, pp. 1–3, 2020.

[Va00] Valentin, G.; Zuliani, M.; Zilio, D. C.; Lohman, G. M.; Skelley, A.: DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In: Proceedings of the IEEE International Conference on Data Engineering, ICDE. Pp. 101–110, 2000.

[Vi21] Viswanathan, V.; Kumar, K.; Willhalm, T.; Lu, P.; Filipiak, B.; Sakthivelu, S.: Intel Memory Latency Checker, `https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html`, 2021, visited on: 10/04/2022.

[Vo20] Vogel, L.; van Renen, A.; Imamura, S.; Leis, V.; Neumann, T.; Kemper, A.: Mosaic: A Budget-Conscious Storage Engine for Relational Database Systems. Proc. VLDB Endow. 13/11, pp. 2662–2675, 2020.

[We22] Weisgut, M.; Ritter, D.; Boissier, M.; Perscheid, M.: Separated Allocator Metadata in Disaggregated In-Memory Databases: Friend or Foe? In: IEEE International Parallel and Distributed Processing Symposium, IPDPS Workshops. Pp. 1202–1208, 2022.

[Wu21] Wu, K.; Guo, Z.; Hu, G.; Tu, K.; Alagappan, R.; Sen, R.; Park, K.; Arpaci-Dusseau, A. C.; Arpaci-Dusseau, R. H.: The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus. In: 19th USENIX Conference on File and Storage Technologies. Pp. 307–323, 2021.