SH2O: Efficient Data Access for Work-Sharing Databases

PANAGIOTIS SIOULAS*, Oracle, Switzerland
IOANNIS MYTILINIS*, Oracle, Switzerland
ANASTASIA AILAMAKI, EPFL& RAW Labs SA, Switzerland

Interactive applications require processing tens to hundreds of concurrent analytical queries within tight time constraints. In such setups, where high concurrency causes contention, work-sharing databases are critical for improving scalability and for bounding the increase in response time. However, as such databases share data access using full scans and expensive shared filters, they suffer from a data-access bottleneck that jeopardizes interactivity.

We present SH_2O: a novel data-access operator that addresses the data-access bottleneck of work-sharing databases. SH_2O is based on the idea that an access pattern based on judiciously selected multidimensional ranges can replace a set of shared filters. To exploit the idea in an efficient and scalable manner, SH_2O uses a three-tier approach: i) it uses spatial indices to efficiently access the ranges without overfetching, ii) it uses an optimizer to choose which filters to replace such that it maximizes cost-benefit for index accesses, and iii) it exploits partitioning schemes and independently accesses each data partition to reduce the number of filters in the access pattern. Furthermore, we propose a tuning strategy that chooses a partitioning and indexing scheme that minimizes SH_2O's cost for a target workload. Our evaluation shows a speedup of $1.8 - 22.2$ for batches of hundreds of data-access-bound queries.

CCS Concepts: • **Information systems → Database query processing**; **Data access methods**.

Additional Key Words and Phrases: databases, indexing, work sharing, analytical query processing

ACM Reference Format:
Panagiotis Sioulas, Ioannis Mytilinis, and Anastasia Ailamaki. 2018. SH2O: Efficient Data Access for Work-Sharing Databases. *J. ACM* 37, 4, Article 111 (August 2018), 26 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

When supporting interactive applications, analytical databases need to sustain highly concurrent workloads with stringent response time constraints. For example, dashboards and reporting in Meta [32] and Youtube [6] require processing hundreds of queries concurrently and need to maintain low response times (i.e., from sub-second to a few seconds). However, in query-at-a-time databases, high concurrency causes *load interaction*; as contention between queries is increased, their response time is also increased. Hence, load interaction jeopardizes response time constraints.

Work-sharing databases [3, 5, 10, 33] mitigate load interaction by using a shared execution model. A *global plan*, which consists of a network of *shared operators*, simultaneously accesses and processes data for multiple queries, eliminating overlapping work. During query processing, input

*Work done while at EPFL.

Authors' addresses: Panagiotis Sioulas, panagiotis.sioulas@oracle.com, Oracle, Zurich, Switzerland; Ioannis Mytilinis, ioannis.mytilinis@oracle.com, Oracle, Zurich, Switzerland; Anastasia Ailamaki, anastasia.ailamaki@epfl.ch, EPFL& RAW Labs SA, Lausanne, Switzerland.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Association for Computing Machinery.
0004-5411/2018/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

data flow from the data-access operators to the roots of the plan. In a query batch, individual query response time is bounded by the global plan's total processing time [10].

Nevertheless, work-sharing databases suffer from a data-access bottleneck. Global plans access data using shared scans followed by specialized shared filters that batch-process the predicates of all queries. This execution pattern incurs high processing time for two reasons: i) scans access the full data, including tuples that are redundant for all queries, and ii) while shared filters optimize predicate evaluation by reducing comparisons, they are implemented as index lookups [22, 36] and hence are more time-consuming than traditional filters. *Global plans that collectively process a small fraction of the data or require several shared filters spend most of their processing time in data access.* The cost is severe for access-heavy workloads, and also notable even for join-heavy workloads.

Traditionally, scan-oriented systems employ data skipping [34, 35, 37] to reduce the data access cost. Skipping operates over predetermined partitions that have been chosen based on the predicates of historical workloads. Then, by using lightweight metadata, it only scans the partitions that may contain parts of the requested data. However, in work-sharing environments, data skipping is inefficient and works well only in limited use cases. This is because it is not robust to workload shifts, and misalignments between partition boundaries and predicates that appear in runtime batches lead to overfetching: even if only a single tuple is required, we have to scan and filter the entire partition. Large query batches amplify this effect as they increase the probability of a misalignment.

We propose SH_2O: a shared data-access operator that, by exploiting data organization and the workload's access patterns, optimizes data access and filtering in work-sharing environments. It targets batches of tens to hundreds of queries with unpredictable filter values. Its only requirement for improving performance is that, in each batch, at least one of the filter attributes is predictable. Prior analysis on real-world workload from Sun et al. [34] supports the existence of predictable filters, hence also filter attributes: 30% of the queries contain all predicates used for 80% of the workload. Barring that assumption, which is necessary for most access methods, SH_2O achieves the same performance as shared scans and filters.

SH_2O is motivated by the following novel insight: *for every set of filters in a batch, there exists a partition of data into multidimensional regions where filtering decisions are the same for all contained tuples.*

Example. Consider a dataset of (X, Y) tuples in $[0, 100]^2$ and the following batch of queries:

```
Q1: SELECT COUNT(*) FROM T WHERE X < 50
Q2: SELECT COUNT(*) FROM T WHERE Y < 50
```

Therefore:

- All tuples of $[0, 50) \times [0, 50)$ are processed by both Q1, Q2.
- All tuples of $[0, 50) \times [50, 100]$ are processed only by Q1.
- All tuples of $[50, 100] \times [0, 50)$ are processed only by Q2.
- Region $[50, 100] \times [50, 100]$ is not processed by any query.

where $A \times B$ denotes the cross product of sets A and B.

SH_2O's key premise is that shared access to the required regions eliminates redundant accesses and the need to process the filters used for constructing the regions. Still, to put this idea to use, there are two challenges: i) efficiently accessing the regions and ii) handling dimensionality. First, access to the regions needs to be exact. Overfetching data requires post-filtering which incurs high processing cost in a work-sharing environment. Hence, SH_2O requires that data organization supports exact access to the regions that are likely to be requested. On the other hand, as the number of filtering predicates in the batch is increased, accessing the regions suffers from the curse of dimensionality. An increase in the number of attributes has a multiplicative effect on the number

of regions. In turn, regions become too sparse, and the number of contained tuples is insufficient to amortize the cost of accessing each region.

SH_2O addresses the two challenges with a three-tier approach:

On-demand multidimensional access. We observe that spatial indices enable efficient and exact range queries on their indexed attributes. *SH_2O* introduces a novel shared access strategy on top of existing spatial indices: it first identifies the regions to query based on a subset of the filter attributes that appear in a batch, and then, uses the index for shared, efficient, and *filter-free* access to each region. Hence, we can query exact ranges regardless of partition boundaries and avoid redundant costs that the expensive shared filters introduce.

Access-pattern optimizer. As dimensionality is increased, there is a cross point where adding one more attribute to the access strategy incurs more overhead than the shared filter that it strives to avoid. The cross point depends on the choice of attributes; hence, choosing the optimal set of attributes poses an optimization problem. *SH_2O* introduces a cost-based optimization strategy that uses an analytical cost model. As a result, *SH_2O* maximizes the benefit for the available index and given workload, and guarantees at least as fast access as shared scans.

Subspace specialization: In case the data is partitioned and each partition is indexed separately, *SH_2O* further reduces the number of regions. As only a subset of queries is interested in each partition, the number of filters and the number of distinct attributes per partition is decreased. By exploiting data-skipping information and by adapting the multidimensional access to each partition's queries and index, *SH_2O* further mitigates the curse of dimensionality.

The above properties show the importance of data organization for *SH_2O*'s performance. Thus, we propose a *partition/index selection strategy* that organizes data such that it minimizes *SH_2O*'s cost by tuning for recurring workload correlations and access patterns.

We implement *SH_2O* inside RouLette, a state-of-the-art work-sharing database. *SH_2O* drastically accelerates data-access-heavy workloads, achieving up to 22.2*x* speedup compared to shared scans. While *SH_2O* is most effective for data-access-heavy workloads, it is beneficial even for join-heavy benchmarks such as SSBM and TPC-H. Finally, in adversarial cases, it is at least as fast as shared scans and filters.

The contributions of this work are:

- Work-sharing databases suffer from a data-access bottleneck. To overcome this, we build *SH_2O* and propose an access strategy which is based on multidimensional regions instead of shared scans and filters.
- Overfetching and post-filtering is expensive in work-sharing environments. By using spatial indices, *SH_2O* achieves efficient and exact access to multidimensional regions.
- Choosing the access strategy's attributes for probing the spatial index introduces a trade-off between savings from shared filters and index-access overhead. We propose an optimization strategy that, by using a data- and workload-aware cost model, maximizes net benefit.
- Predicate correlations isolate predicates in specific data subspaces. We exploit this to propose a joint partition/index selection scheme that takes advantage of emerging workload patterns and further reduces the number of regions and *SH_2O*'s cost.

2 RELATED WORK

We first present an overview of the state-of-the-art in work-sharing databases and discuss common data-access strategies.

Work sharing. Work-sharing databases [3, 5, 10, 13, 33] address the need for high-throughput analytical processing. They exploit overlapping work across queries to reduce the total processing time, and hence increase throughput and mitigate load interaction. They express overlaps by using a *global (query) plan* for all running queries. The global plan is a directed acyclic graph (DAG) of

relational operators that process tuples for one or more queries and multi-cast their results to one or more parent operators. In each of its roots, it produces the results of queries that participate in the batch. Overlaps are represented by common paths in the DAG. Figure 1a shows a two-query example. Join $A \bowtie B$ is common across queries Q1 and Q2, and thus we can compute it only once and appropriately route the results. In this work, we are only interested in the data-access and filtering operators in global plans.

Work-sharing databases can share operators between partially overlapping queries (i.e., that use different predicates) using the *Data-Query model* [10]. The Data-Query model annotates each intermediate tuple with a *query-set* that indicates to which queries the tuple contributes. Operators in work-sharing databases process both the actual tuples and the corresponding query-sets and produce correctly annotated output tuples. Hence, the Data-Query model increases sharing opportunities and is used in recent work-sharing databases [3, 5, 10, 33]. SH_2O is applicable to such systems: it can accelerate their data access and directly produces Data-Query model tuples for each accessed table.

Shared scans. Shared scans amortize the cost of data access across multiple queries. They have been used for both disk-based [13] and in-memory databases [28]. Cooperative scans [40] further optimize shared scans for queries accessing different data ranges by scheduling I/O requests to maximize bandwidth sharing and minimize latency penalties at the same time.

Crescando [36] enhances in-memory shared scans with the Data-Query model to achieve fast and predictable performance for simple concurrent queries. Work-sharing databases either use Crescando [10, 23] or implement shared scans and filters similarly [3, 33]. SH_2O outperforms in-memory shared scans because it accesses data more selectively and amortizes downstream filtering costs.

Index access. Indices provide each individual query with efficient data access, without costly full scans and filters. However, as the number of concurrent queries is increased, index latency is also proportionally increased due to load interaction. As Kester at al. [18] demonstrate, there is a crosspoint where shared scans become more efficient. Furthermore, indices restrict work sharing for downstream computation across queries. Hence, shared scans are typically preferred in work-sharing databases. To our knowledge, the only work-sharing database that implements index probes is SharedDB, which batches index probes to improve instruction and data cache locality and to produce a shared set of tuples across participating queries [9].

Beyond work-sharing databases, OLTPShare [30] merges OLTP point queries into bulk index lookups. Also, in the context of information filtering, Fischer and Kossmann [8] propose merging identical probes into one index lookup and avoiding overlapping accesses between consecutive probes. While merging probes reduces data access, it requires post-filtering for the probe's results, which is time-consuming with shared filters. Also, it has not been studied for workloads that use predicates on multiple attributes, which complicates detecting overlaps between index traversals. SH_2O minimizes post-filtering and generalizes techniques that merge probes for multi-attribute setups.

Data skipping. Data skipping accelerates selective queries in scan-oriented databases. For each query, it prunes out data partitions that contain redundant data – it identifies such partitions using a compact set of aggregates [24] over them. State-of-the-art approaches, such as Qd-tree [37], Jigsaw [17], and the work of Sun et al. [34, 35], formulate optimization problems for partitioning the data such that data skipping minimizes access for a target workload. SH_2O both competes and takes advantage of data skipping. On the one hand, it outperforms data skipping for shared data access, as it avoids excessive access and filtering due to misaligned predicates (i.e., with filter constants not on partition boundaries). On the other hand, SH_2O uses data skipping information to specialize

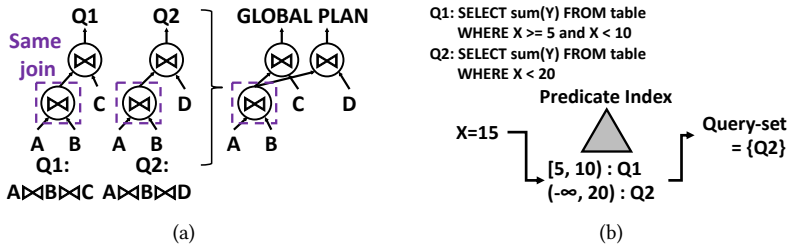


Fig. 1. (a) Global plan (b) From queries to predicate indices

data access for each partition and reduce dimensionality. Furthermore, this paper proposes a novel partition/index selection algorithm for minimizing SH_2O's cost.

Materialized views. While this work focuses on work sharing, a large body of work [15, 27, 31, 39] uses materialized views to reduce latency. Answering queries using views eliminates runtime computations, such as joins and aggregations, but at the cost of amplified memory consumption. In addition, views require that computation recurs across time so that the selected views are reused. By contrast, work sharing is transient, and thus does not require a long-term memory investment, and eliminates common computation only within each batch. As such, under high concurrency, both views and work sharing suffer from data access and filtering, which SH_2O optimizes. As the experiment of Figure 9b indicates, SH_2O is also beneficial for accelerating view access.

3 OVERCOMING THE ACCESS BOTTLENECK

SH_2O addresses the problem of efficient data access in work-sharing databases. In this section, we provide an overview of both the problem and the solution that SH_2O provides.

3.1 Scan-Filter in Shared Execution

In work-sharing databases data access is defined as the sequence of operators that access the data and compute the Data-Query model query-sets for each tuple. Then, the tuples can be processed by downstream operators such as joins and GROUP-BYs. Typically, shared data access is implemented using shared scans that are followed by shared filters, i.e., filters that evaluate the predicates for multiple queries at once and set the tuples' query-sets accordingly.

The performance of shared filters is critical because i) filters are the first operators after scans, hence they process a large fraction of the input, and ii) each filter may need to evaluate tens of predicates. A naive implementation would construct each tuple's query-set by going over all the query predicates in the batch and checking which are satisfied and which are not. However, this algorithm is linear to the batch size and introduces a significant overhead for batches of tens of queries. To make shared filters efficient, prior work [5, 10, 22, 33, 36] proposes using *predicate indices (PIs).*

Unlike conventional indices, which are pre-constructed, PIs are built at runtime and their lifetime is the duration of the batch's execution. Rather than index data, a PI indexes a set of predicates that belong to different queries – typically each PI indexes the predicates on a specific attribute [10, 22, 33, 36]. Indexing all predicates in the batch may require multiple PIs.

Shared filters evaluate predicates on different attributes by using the corresponding PIs. For each tuple, they probe PIs, and set the tuple's query-set accordingly. If the query-set becomes empty, shared filters discard the tuple. Figure 1b shows the process of probing a PI that serves two queries; probing with $X = 15$ identifies that $X \in (-\infty, 20)$, so the tuple belongs to Q2. After passing through shared filters, tuples have query-sets that represent the results of the predicates for all queries.

For implementing PIs, this paper relies on *grouped filters* which are used in TelegraphCQ [22], CJOIN [5], and RouLette [33]. Each grouped filter builds one index for all the predicates on a specific attribute. Grouped filters evaluate query-sets by intersecting the queries with satisfied predicates for each attribute. However, the same insights generalize for Crescando's implementation [36], also used in SharedDB [10], which inserts only one predicate per query to a grouped filter, and processes the rest of the predicates as post-filters only for the queries whose indexed predicate is satisfied.

3.2 Data-Access Overhead

Sharing data access amortizes the cost of both scans and predicate evaluation and thus mitigates the impact of concurrency. However, in absolute numbers, data access per batch is still expensive: each shared scan accesses the full table, which can be too large to read within milliseconds. Shared filters are also time-consuming: each PI probe is essentially a join between the probe's attribute and the indexed predicates, and the cost of the ensuing query-set operations is increased with the number of queries.

Expensive scan-filter-based data access introduces a performance bottleneck and adversely affects interactive applications with stringent time constraints. The impact is maximum for batches with limited downstream computation, such as queries over denormalized tables or selective queries. Both classes of workloads occur in highly-concurrent environments in industry, e.g., the former in decision-support queries in Amadeus [36] and the latter in dashboards that analyze a specific user's data [6, 32]. In such cases, data access dominates the total batch execution time. However, as our experiments show, there is a significant performance penalty even for join-heavy workloads such as SSBM and TPC-H.

3.3 SH2O: Efficient and Adaptive Access

We observe that much of the processing time spent in shared data access is unnecessary. First, PI probes incur repetitive work. Similar tuples, which have almost the same attribute values, make PI probes that produce the same access patterns, retrieve the same satisfied predicates and perform the same query-set updates. Hence, the process of constructing the same query-set multiple times is also redundant. Second, for selective queries, shared data access spends too much time for processing tuples that are eventually filtered out.

We propose SH_2O, a shared data-access operator that aims to reduce the redundant data access cost. To do so, it exploits the collective access pattern for the batch of queries to skip redundant tuples and amortizes the cost of shared filters across groups of similar tuples. Naturally, SH_2O has maximum benefit for workloads where data access makes up most of the processing time and performs at least as well as shared scans and filters for all workloads regardless of selectivity, predicate correlations, and filter attributes.

SH_2O works as a stand-in for data access operators, i.e., a shared scan followed by shared filters, in work-sharing databases. It assumes a work-sharing database that works under the Data-Query model, processes submitted queries one batch at a time and uses a set of predicate indices, one for each filter attribute, to facilitate filtering. First, in an offline phase, a dedicated tuner partitions the data, and for each partition p, it builds a spatial index $\mathcal{I}_p$ on a set of attributes $A(\mathcal{I}_p)$. The indexed attributes may differ from partition to partition. This scheme enables isolating query predicates that co-occur with other conditions to fewer partitions and specializing the selected index for the partition-local access patterns. Assuming a workload with recurring patterns, both the partitioning and the indices can be reused across batches.

Figure 2 shows how SH_2O performs data access at a high level. At execution time, SH_2O processes each partition independently for the queries interested in the partition; hence, only the filters for

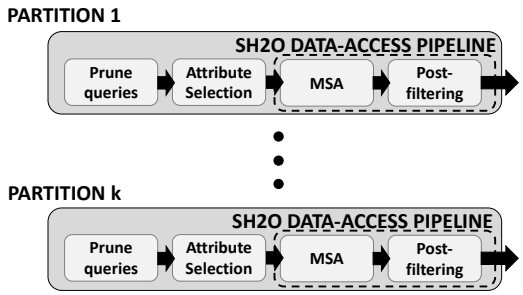


Fig. 2. SH_2O framework

the interested queries are considered. Instead of using the typical scan and filter pattern, partition-local data access is driven by *multidimensional shared access (MSA)*, our novel workload-driven data-access technique which exploits the spatial indices that were built during the tuning phase. In each partition, MSA replaces the shared filters on a subset of the indexed attributes $\mathbf{F}_p \subset A(\mathcal{I}_p)$. The *attribute selection* optimizer chooses $\mathbf{F}_p$ such that it hits a sweet spot between eliminated filter cost and spatial index overhead. Finally, SH_2O processes the tuples that MSA retrieves using the remaining filters that are not in $\mathbf{F}_p$.

Evidently, SH_2O's performance depends on the data partitioning scheme and the index used in each partition. For this reason, we also investigate the problem of organizing the data to minimize SH_2O's total processing cost for a target workload. This paper formulates the reorganization problem and proposes a unified partition/index selection algorithm, used by the offline tuner, based on Iterative Dynamic Programming [19].

We present the above mentioned components as follows. Section 4 presents the MSA-driven data-access pipeline for each partition. Section 5 shows the cost model and the enumeration strategy for choosing the $\mathbf{F}_p$ attributes. Section 6 generalizes SH_2O's data-access pipelines for partitioned data. Finally, Section 7 presents the formulation and solution for the partition/index selection problem.

4 MULTIDIMENSIONAL SHARED ACCESS

For each partition p, SH_2O processes a data-access pipeline. The pipeline revolves around MSA, which optimizes a scan-filter operator pattern for filters on attributes $\mathbf{F}_p$. In this section, we assume that attributes $\mathbf{F}_p$ are already given.

MSA is built on the following idea: there exists a set of multidimensional regions, which logically partition the data, where the batch's filters on $\mathbf{F}_p$ always make the same filtering decisions for all contained tuples. In these regions, filters on $\mathbf{F}_p$ produce the same query-set for all tuples. MSA identifies these regions and uses $\mathcal{I}_p$ to access all regions with a non-empty query-set. By doing so, it fetches only the corresponding tuples hence post-filtering is unnecessary. Thus, by accessing each required region's data once and augmenting it with the region's query-set, MSA amortizes expensive query-set operations and filtering costs across the entire region, and minimizes accessed data.

In this section, we put MSA in the context of SH_2O's per-partition pipeline. We present the mechanism for identifying MSA's regions, and, subsequently, present the end-to-end data access strategy.

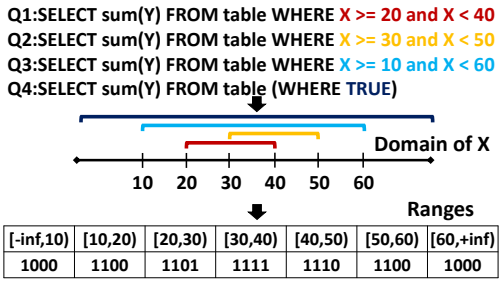


Fig. 3. From queries to predicate index ranges

4.1 Computing Single Query-set Regions

Identifying MSA's regions requires understanding under which conditions the filtering decisions for a set of tuples remain invariant. We provide an analysis of these conditions.

A PI on a specific attribute exhibits locality. It defines a function that maps each value in the attribute's domain to a set of queries for which the indexed predicates are satisfied. Figure 3 shows an example for the PI of four queries. Adjacent values are then likely to map to the same query-set, e.g., $X = 25$ and $X = 26$ both produce the same query-set $\{Q1, Q3, Q4\}$. Thus, we can represent each PI as a set of (r, Q) pairs, where $r = [\bullet, \bullet)$ is a range in the corresponding attribute and Q is a query-set that indicates which queries are satisfied in the specific range. The ranges define a partitioning on the attribute's domain. When building the PI, we compute the ranges and their corresponding query-sets by partitioning the domain across the boundaries of predicates values and statically computing the predicates in each partition. In the given example, analyzing the predicates gives 7 range-query-set pairs.

To generalize locality to a set of multiple attributes $\mathbf{F}_p$, we compose the ranges of one-dimensional PIs. Let $d = |\mathbf{F}_p|$ be the number of attributes used for the filters that MSA replaces, and $PI_1, PI_2, \ldots, PI_d$ the corresponding predicate indices. Each predicate index PI_i is represented as a set of (r, Q) pairs. The following theorem gives the computation of single query-set regions:

Theorem 4.1. *Let us assume d predicate indices $PI_1, PI_2, \ldots, PI_d$, and a random tuple (r_i, Q_i) from each index PI_i. Then, all data in $r_1 \times r_2 \times \cdots \times r_d$ will share the same query-set $Q* = Q_1 \cap \cdots \cap Q_d$.*

This theorem motivates a hyperrectangle-oriented access strategy for two reasons: First, as all tuples in the same hyperrectangle share the same query-set, we can perform the expensive query-set operations only once per region and then just use the result to annotate each tuple. Second, if $Q* = \emptyset$ for a hyperrectangle, we can skip its tuples altogether. By using hyperrectangles, we drastically reduce the data processed and the amortized cost per tuple.

MSA enumerates the hyperrectangles by computing the set of $r_1 \times \cdots \times r_d$. To avoid materializing the cross-product of PIs, we produce each hyperrectangle using an iterator that computes the next $(r_1 \times \cdots \times r_d, Q*)$ pair on-the-fly.

4.2 Index-based Access to Regions

For each partition, we build a spatial index $\mathcal{I}_p$ on a set of attributes $A(\mathcal{I}_p)$. Each time the iterator produces a hyperrectangle with a non-empty query-set $Q*$, we fetch the corresponding tuples by issuing a range query to $\mathcal{I}_p$. The query collects the tuples of interest and annotates them with the already computed $Q*$, so they can be processed in Data-Query model by subsequent shared operators.

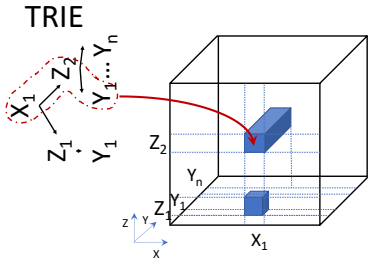


Fig. 4. Trie-based grid index

Our method is modular and does not stand for a specific type of spatial index. It is compatible with any technique that enables efficient multidimensional range queries, such as Ub-tree[25], k-d tree[4], R-tree[12], Hilbert- and Z-curve [20]. The performance characteristics of the employed technique are expected to affect the absolute cost of index access but not the overall trends.

In our implementation, we use a grid index, as the one depicted in Figure 4. We sort the tuples based on their projection on a selected permutation of $A(\mathcal{I}_p)$ and organize the resulting grid in a trie. Each level of the trie corresponds to an attribute, and each node corresponds to a distinct value in the domain of its level's attribute. Each level's nodes are stored in the same array, and nodes with the same parent are in contiguous positions and sorted among themselves; thus, a binary search finds children nodes that fall within the range query. At the leaves of the trie, we store contiguous sequences of tuples with the same projection. The only tuning knob for our trie implementation is the selected permutation of $A(\mathcal{I}_p)$, which is chosen when building the index.

In each range query, we traverse the trie such that the prefix satisfies the range query's predicates on the respective dimensions. In dimensions without predicates, we assume a $(-\infty, +\infty)$ range predicate. When the traversal reaches the leaves, it retrieves the corresponding individual tuples. Henceforth, when we mention an index probe during MSA, we refer to a range query over $\mathcal{I}_p$.

Optimizing for data correlations. In some cases, such as when data is correlated, range queries contain no results. For example, if columns A and B are correlated through the constraint $A - 10 \leq B \leq A + 10$, then ranges $(A, B, C) \in [50, 70) \times [0, 20) \times r_C$ are empty. To avoid redundant index probes, we eliminate empty hyperrectangles as follows: suppose that $r_1, \ldots, r_k$ is the shortest prefix of ranges that is not satisfied by any node in the k-th level. Because all suffixes $r_{k+1}, \ldots, r_d$ lead to empty hyperrectangles, the traversal abandons the current range query and searches in PI_k for the first non-empty range r_k^* after r_k. The r_k^* range has the property that $r_1, \ldots, r_k^*$ is satisfied by at least one node in the k-th level. If such a range exists, the iterator of hyperrectangles jumps to the first region with the non-empty prefix. Otherwise, the iterator jumps to the first hyperrectangle with prefix: $r_1, r_2, \ldots, r_{k-1}^*$, where r_{k-1}^* is the next range after r_{k-1}. This way, large numbers of empty hyperrectangles are pruned early, significantly reducing the index probing cost.

4.3 Hybrid Execution

Probing all index attributes (i.e., $\mathbf{F}_p = A(\mathcal{I}_p)$) may incur a high cost, as $|\mathbf{F}_p|$ has a multiplicative effect on the number of hyperrectangles. For this purpose, SH_2O uses a hybrid MSA/post-filtering approach: when processing a batch of queries with filters on a set of attributes U_F, it uses MSA to replace only filters on $\mathbf{F}_p \subset A(\mathcal{I}_p)$ and then uses $U_F - \mathbf{F}_p$ to post-filter the tuples that the spatial index retrieves. Post-filtering is necessary because the query-sets after probing the index correspond only to the filters on $\mathbf{F}_p$. As we will see in Section 5, the choice of the subset is based on the *attribute selection process* that maximizes the benefit of eliminating filters.

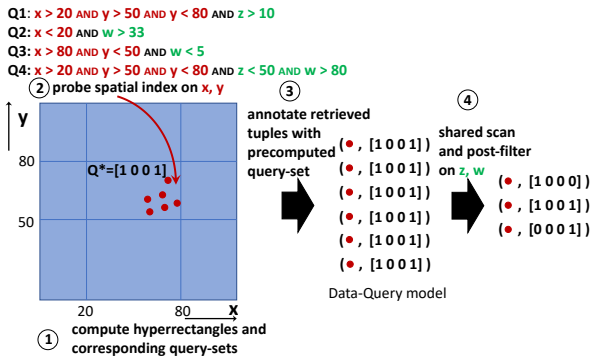


Fig. 5. SH_2O data-access pipeline

Post-filtering applies filters one after the other in a vectorized manner. The order in which filters are applied is determined by the query optimizer. For each filter, we probe the filter's predicate index and update the query-sets of retrieved tuples accordingly. We drop the tuples with empty query-sets and forward the remaining tuples to the next filter in the pipeline.

This way, SH_2O benefits from eliminating filters while keeping MSA overhead low. Figure 5 shows an example. First, we apply MSA on attributes x and y. In the first step, we identify the data regions that share the same query-set. Then, we fetch the corresponding tuples by probing the spatial index for each region. Finally, we process the retrieved tuples using shared filters on the remaining attributes z, w.

Processing the results of each hyperrectangle separately can reduce the benefit of vectorization. If each region contains very few tuples, interpretation overhead become comparable to the one in tuple-at-a-time execution. Post-filtering further aggravates the situation. To reduce the interpretation overhead, SH_2O consolidates its results. It packs small data vectors into larger vectors whose size exceeds a threshold. Hence, per-vector overhead is amortized across many more rows. In our experiments, we also use consolidation for non-SH_2O workloads to equalize downstream costs.

4.4 Supported Predicates

MSA models predicate indices as sets of (r, Q) pairs. This representation is compact and efficient for i) range predicates, ii) equality predicates, iii) disjunction on the same attribute/IN operator, and iv) conjunction. However, more complex predicates can be supported using techniques such as the feature vector from [34].

5 SELECTING PROBING ATTRIBUTES

MSA substitutes the shared full scan and part of the shared filters with a single query-set computation and index lookup for each hyperrectangular region. However, an increase in the number of either the probing attributes or of the ranges in the corresponding predicate indices has a multiplicative effect on the number of the resulting hyperrectangles. Hence, there are cases where it is beneficial to probe only a selected subset of the spatial index's attributes. Adding more attributes to this subset would increase overhead more than it would save cost. This way, we reduce: i) the number of hyperrectangles, ii) the cost of multidimensional index lookups, and iii) the hyperrectangle overheads per tuple. Nevertheless, identifying the optimal probing attributes is still a problem. Ideally, the chosen subset of attributes hits a sweet spot between the cost of probing the spatial index and the cost of the post-filtering.

The attribute selection mechanism is triggered each time a table partition access starts and consists of two parts: i) an analytical cost model that estimates the cost of MSA and shared filters for a given set of probe attributes for a specific partition and ii) an optimization algorithm that, given the cost model, finds the optimal subset.

5.1 Cost Model

The proposed cost model considers several factors, such as: what index is available, which attributes have filters in the workload, how many query-set ranges exist in each attribute's PI, what is the data distribution, and how many tuples are retrieved with each index access. Based on the queries at hand, our cost model partitions filter attributes in two sets: i) a subset to use for MSA and ii) a subset to use for post-filtering. Each of the subsets can be empty.

Our analytical model is easy to understand and tune using regression. It consists of two components: the spatial index access cost (*IC*) and the shared filter cost (*FC*). While we have motivated the general case, where a set of filter attributes is used for probing and the remaining for post-filtering, our model also inherently covers the trivial cases: i) use MSA for all available attributes, or ii) avoid using MSA and do a full-scan instead.

5.1.1 Index Cost. The Index Cost ($IC^{\mathcal{I}}$) represents the cost of MSA when the index $\mathcal{I}$ is used. Given i) a data partition of n tuples, ii) a set of filter attributes $\mathbf{F}$, which are used to probe the index, iii) a set of predicate indices on $\mathbf{F}$: $\mathbf{PI} = \{PI_1, \ldots, PI_{|\mathbf{F}|}\}$, and iv) a vector of data-partition statistics $\mathbf{D}$, $IC^{\mathcal{I}}$ can be expressed as:

$$IC^{\mathcal{I}}(n, \mathbf{F}, \mathbf{PI}, \mathbf{D}) = (\prod_{f \in \mathbf{F}} N(PI_f)) * Cost_H(n, \mathbf{F}, \mathbf{PI}, \mathbf{D})$$

With $\prod_{f \in \mathbf{F}} N(PI_f)$, we denote the number of non-empty hyperrectangles with non-empty query-sets that MSA creates. To estimate the number of hyperrectangles, we assume the worst case in which all hyperrectangles are non-empty and have non-empty query-sets hence MSA cannot exploit data correlations to skip hyperrectangles. $\prod_{f \in \mathbf{F}} N(PI_f)$ rapidly increases with more or larger predicate indices, and thus choosing strategically the set $\mathbf{F}$ is critical. Note also that the number of accessed ranges in a predicate index $N(PI_f)$ is directly affected by the workload access patterns. Intuitively, the more correlated the queries, the smaller $N(PI_f)$ would be. Thus, our cost model also captures latent workload correlations.

The second term, $Cost_H(n, \mathbf{F}, \mathbf{PI}, \mathbf{D})$, denotes the cost each hyperrectangle incurs. $Cost_H$ is proportional to three quantities: i) the cost QC of query-set operations for computing the query-set $Q*$, that all tuples of the hyperrectangle share, ii) the lookup cost SIC on the spatial index, and iii) the cost VC that the resulting tuples will incur on post-filtering. The latter is practically how many vectors we will need to post-filter. This can be expressed as:

$$Cost_H(n, \mathbf{F}, \mathbf{PI}, \mathbf{D}) = c_q * QC + c_s * SIC + c_v * VC$$

where c_q, c_s, c_v are constants. The value of these constants reflects latent variables such as the underlying hardware, the query-set implementation, the spatial index implementation, etc., and thus should be tuned to match the specific deployment at hand. We tune the parameters by running SH_2O for generated sets of data and queries and then fitting the cost model results to the measured response time using least-squares.

Now, we further discuss and expand the QC, SIC, and VC quantities. The query-cost QC depends on the implementation of query-set operations. Makreshanski et al. discuss the trade-offs of different implementations [23]. Our implementation uses bitsets, and in that case, $QC = |\mathcal{B}|$, where $|\mathcal{B}|$ is the size of the submitted query batch.

The lookup cost SIC is a function of the available predicate indices and data distribution ($SIC = SIC(\mathbf{PI}, \mathbf{D})$). The exact formula of SIC depends on the spatial index implementation. In our case, where a grid index is used, we observe that the cost depends on the number of visited nodes at each level of the trie:

$$SIC(\mathbf{PI}, \mathbf{D}) = \sum_{d=1}^{d_{max}} \prod_{i=1}^{d} sel(PI_{\mathcal{I}_i}) * \mathbf{D}(\mathcal{I}_i)$$

where d_{max} the dimensionality of the spatial index, $sel(PI_{\mathcal{I}_i})$ is the average selectivity across the accessed ranges of predicate index $\mathcal{I}_i$, and $\mathbf{D}(\mathcal{I}_i)$ denotes the distinct values in attribute $\mathcal{I}_i$. $\mathcal{I}_i$ maps each index dimension to the corresponding filter attribute. The formula assumes that the attributes follow independent distributions and that the number of nodes is lower than the number of tuples: it estimates that for each node of level $i-1$, index traversal accesses $sel(PI_{\mathcal{I}_i}) * \mathbf{D}(\mathcal{I}_i)$ nodes in level i. Thus, the number of visited nodes per level is increased multiplicatively. The assumption of independence is commonly used in query optimizers [21]. Estimating selectivity, especially for predicates across multiple attributes, is an orthogonal and active research area [7, 38].

Finally, the VC cost represents the number of vectors retrieved with each index lookup. We approximate this cost by assuming that all hyperrectangles contain the same number of tuples. The formula used is:

$$VC = \lceil \frac{n}{vs * \prod_{f \in \mathbf{F}} N(PI_f)} \rceil$$

where vs is the vector size used during processing. The ceiling shows that even when fewer than vs tuples are retrieved from a hyperrectangle, we have to "pay" for the whole vector.

Putting it all together:

$$IC^{\mathcal{I}}(n, \mathbf{F}, \mathbf{PI}, \mathbf{D}) = \prod_{f \in \mathbf{F}} N(PI_f) * (c_q * |\mathcal{B}| +$$
$$c_s * \sum_{d=1}^{d_{max}} \prod_{i=1}^{d} sel(PI_{\mathcal{I}_i}) * \mathbf{D}(\mathcal{I}_i) + c_v * \lceil \frac{n}{vs * \prod_{f \in F} N(PI_f)} \rceil)$$

5.1.2 Filter Cost. The filter cost models the cost of shared filters using the Data-Query model. For each tuple, if there are m predicate indices available, the tuple's query-set is produced as $probe(PI_1) \cap \cdots \cap probe(PI_m)$. Thus, the cost comprises the predicate index probe and the cost of query-set operations. Again, the cost of query-set operations is proportional to the query batch size $|\mathcal{B}|$.

Regarding the probing cost, our analysis indicates that it mostly depends on data locality. In our implementation, predicate indices take the form of binary trees. If consecutive tuples follow the same path in the predicate index's binary search, the branch prediction mechanism of modern processors significantly accelerates probing. Otherwise, if subsequent tuples follow different paths, performance degrades. To quantify this effect, for each filtering attribute f, we define a locality indicator L_f that shows the expected number of consecutive tuples that follow the same path when probing the corresponding predicate index. Then, using regression, we train a monotonically increasing decay function that maps the lack of locality to a factor of performance degradation: $loc(L_f) \rightarrow [1, \infty]$. In practice, we have observed that $loc(L_f) \in [1, 2.5]$.

In contrast to the spatial index cost, in shared filters, the number of ranges in the predicate index is less important. Finding the correct ranges, here, depends on a binary search and thus the cost is increasing logarithmically to the number of ranges. Summing up, the cost of shared filters is:

Algorithm 1: Enumerate Candidate Dimension-Sets

input: Tuple (n, U_F, PI, D, L)

1 $level = \{\emptyset\}$; $best\_set = \emptyset$; $best\_cost = SC(n, U_F, L)$;
2 **while** $level.nonEmpty()$ **do**
3 $next\_level = \emptyset$;
4 **for** $F \in level$ **do**
5 **for** $f \in U_F - F$ **do**
6 $IC_{old} = IC^I(n, F, PI, D); SC_{old} = SC(n, U_F - F, L)$ $IC_{new} = IC^I(n, F \cup \{f\}, PI, D)$;
7 $SC_{new} = SC(n, U_F - F - \{f\}, L)$;
8 **if** $IC_{new} - IC_{old} < SC_{old} - SC_{new}$ **then**
9 $next\_level = next\_level \cup \{F \cup \{f\}\}$;
10 **if** $TC_{new} < best\_cost$ **then**
11 $best\_cost = TC_{new}$; $best\_set = F \cup \{f\}$;
12 $level = next\_level$;
13 **return** $best\_set$;

$$FC(n, \mathbf{F}, \mathbf{L}) = n * c_f * \sum_{f \in \mathbf{F}} |\mathcal{B}| * loc(L_f)$$

where $\mathbf{L} = [L_1, \ldots, L_{|\mathbf{F}|}]$ is a vector with the locality indicators for each filter attribute and c_f is a constant.

5.1.3 *Total Cost.* Overall, the cost of SH_2O when probing the spatial index on attributes $\mathbf{F}$ is:

$$TC(n, \mathbf{F}, \mathbf{PI}, \mathbf{D}, \mathbf{L}, \mathcal{I}) = IC^{\mathcal{I}}(n, \mathbf{F}, \mathbf{PI}, \mathbf{D}) + FC(n, \mathbf{U_F} - \mathbf{F}, \mathbf{L})$$

where $\mathbf{U_F}$ is the set of all filter columns.

5.2 Enumerating Candidate Attribute Sets

The cost model can estimate whether a set of probed attributes is preferable to another. SH_2O takes advantage of these estimates to identify the attributes that minimize the total data access cost for each partition: It enumerates candidate attribute sets, and for each candidate, it computes the corresponding cost model's estimate. The set that yields the lowest cost is finally selected.

However, candidate selections are, in the worst case, exponential to the number of attributes. To explore all candidates, we traverse the lattice of attribute-sets in a bottom-up way, which takes $O(|F| \times 2^{|F|})$ in the worst-case (without pruning). To render enumeration efficient, we build Algorithm 1 which prunes parts of the space by using Observation 5.2 (line 8). The process terminates when no more candidates are available.

Observation. *If adding an extra attribute to a candidate-set increases IC more than it decreases FC, then the resulting candidate-set and its super-sets can be safely pruned.*

6 PARTITION SPECIALIZATION

Thus far, we have described how SH_2O processes data-access pipelines in a per-partition fashion. Here, we show how it exploits local patterns in the query predicates, and hence the need for partitioning.

In cases where there are correlations between the workload's predicates, directly applying SH_2O on the entire dataset processes an unnecessarily large number of hyperrectangles. Consider the

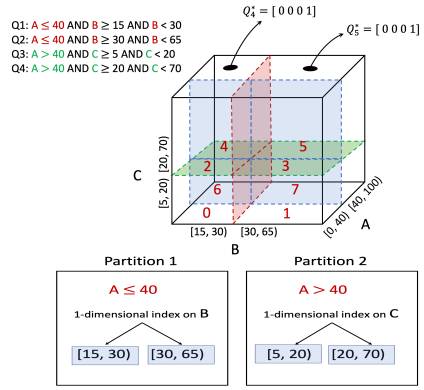


Fig. 6. Applying SH_2O at partition-granularity

example of Figure 6. The work-sharing database processes a batch of queries with filters on attributes A, B, C. Computing MSA's hyperrectangles for the entire dataset results in $|PI_A| * |PI_B| * |PI_C| = 8$ probes. However, consider the case where the data is partitioned on the predicate $A > 40$. In each partition, we need to consider only predicates from queries that intersect with the partition. Then, SH_2O probes $|PI_B| = 2$ hyperrectangles in the first partition ($A \leq 40$) and $|PI_C| = 2$ in the second partition ($A > 40$). Moreover, the partitioned case requires indices with fewer dimensions which translates to decreased probing cost.

To exploit local patterns within relevant partitions and reduce both the number of hyperrectangles and the dimensionality of indices, SH_2O adapts access to each partition. It identifies which predicates are relevant in each partition, using a variant of data skipping, and plans multidimensional shared accesses independently. First, SH_2O identifies the set of queries that process each partition. A query processes a partition if the partition overlaps with the query's predicates. To compute the query-set of each partition, we use data skipping based on zone maps [11]. SH_2O finds the queries that intersect with the zonemap's `min-max` ranges – by exploiting predicate indices – and computes their union. If the query-set is empty, SH_2O completely skips the partition. Next, SH_2O optimizes multidimensional shared access for each partition's queries. Filters that do not belong to the partition's query-set or statically evaluate to `TRUE` are excluded from the PI ranges and the attribute selection. Hence, SH_2O produces drastically fewer hyperrectangles.

7 SH2O-AWARE DATA ORGANIZATION

The effectiveness of SH_2O depends on the constructed partitions and indices. To reduce data-access and filtering costs, SH_2O requires i) spatial indices that can replace shared filters in a query batch arriving at runtime, thus enabling efficient access, and ii) partitions that exploit predicate correlations. In this section, we formulate and address the problem of partitioning/indexing the data such that we minimize SH_2O's cost for a set of target batches.

We partition/index the data in an offline manner. As long as predicate patterns recur (i.e., predicates on the same attributes and correlations between predicates) the partition/index-building cost is an investment that is amortized with time. In this work, we do not elaborate on the predicate monitoring process or on the details of how often partitions should be updated, but we theoretically formulate and solve the joint partition-index selection problem for SH_2O. Continuously tuning the physical design is a well-known problem which is orthogonal to ours [2]. Here, we assume that the set of (one or more) batches to optimize for is known in advance, and we build partitions and indices once in the beginning.

In the literature, there are several data-layout optimization techniques that partition a multi-dimensional dataset in a way that captures query correlations and favors data skipping (e.g., [37]). However, existing partitioning approaches: i) do not account for shared access across a batch of queries and ii) are index-oblivious. Here, we address both deficiencies at the same time. More specifically, we take advantage of emerging access patterns in the workload and partition the data space into a set of hyperrectangles. For each of these hyperrectangles, partition-index selection uses the cost model of Section 5.1 to choose an index that best fits the specific subspace. The goal is to *minimize the aggregate data access time across all partitions.* At runtime, SH_2O processes each resulting partition independently. Formally, we define the *Index-Aware Partitioning for Shared Access (IPSA)* problem:

Problem (IPSA)**.** — *Given w query batches with predicate index sets* $PI^1, PI^2, \ldots, PI^w$*, find a set of partition-index pairs*

$$\boldsymbol{P} = \{(S_1, I_1), \ldots, (S_m, I_m)\}$$

such that

$$\min \sum_{(S_i,I_i)\in \boldsymbol{P}} \sum_{j=1}^{w} \min_{\boldsymbol{F}} TC(|S_i|, \boldsymbol{F}, \boldsymbol{PI^j}|S_i, S_i, \boldsymbol{L}, I_i)$$

with the constraint that $S_1, \ldots, S_m$ *are hyperrectangles* [1].

We observe that any partitioning can be generated by applying a series of recursive cuts, where each cut corresponds to a predicate (e.g., Figure 6). The recursion starts by taking as input the entire dataset. Then, at each step, the current partition is either kept intact, the optimal index is selected and the partition-index pair is returned as part of the solution, or is bi-partitioned by a new cut. In the latter case, the two resulting partitions are further processed recursively. This structure enables a Dynamic Programming (DP) formulation. Let us denote the optimal data access cost for partition S as

$$OC(S) = \min_{\mathbf{I}} \sum_{j=1}^{w} \min_{\mathbf{F}} TC(|S|, \mathbf{F}, \mathbf{PI^j}|S, S, \mathbf{L}, I)$$

Moreover, given a cut c, we use $V_c(S)$ to denote the subspace of S that satisfies c. Then DP is expressed as:

$$\begin{cases} P(S) = min\{OC(S), min_c\{P(V_c(S)) + P(S - V_c(S))\}\} \\ P(S) = OC(S) \quad \text{, if no other cut can be applied} \end{cases}$$

7.1 Iterative Partitioning

Solving IPSA using DP is prohibitive as it requires tabulating and estimating access costs for all the possible subspaces that the filter attributes of the batches define. This number is expected to be very high and much larger than the number of hyperrectangles that the predicate indices define.

To efficiently approximate the optimal solution, we use an iterative algorithm that is inspired by Iterative Dynamic Programming [19]. The algorithm works in iterations and starts from a single partition that contains all the data. At each iteration, it chooses a partition and finds the optimal sequence of k recursive cuts (Algorithm 2, line 13) that minimize the total cost across all new subpartitions. At the same time, it also selects the optimal index for all the partitions that these cuts produce (line 15). If the cost reduction, after making the k cuts, is more than a relative threshold $t\%$

[1]The notation $\mathbf{PI}|S_i$ signifies the subset of each predicate index that is within the boundaries that define S_i

Algorithm 2: Index-Aware Partitioning for Shared Access

```
1  Function MAKE_PARTITIONS(n, U_F, PI, D, k, t):
2      output = ∅; partitions = newQueue() ;
3      partitions.push(D) ;
4      while partitions.nonEmpty() do
5          target = partitions.pop();
6          kcuts = BEST_KCUTS() ;
7          if kcuts.bestCost >= t * target.cost then
8              output.add(target) ;
9          else
10             for p ∈ kcuts.results do
11                 partitions.push(p) ;
12 return output ;
13 Function BEST_KCUTS(n, U_F, PI, D, k):
14     if k == 0 then
15         ret.bestCost = CHOOSE_INDEX(n, U_F, PI, D) ;
16         ret.results = {D}; return ret ;
17     cuts_in_partition = {f ∈ U_F | f cuts D} ;
18     for cut ∈ cuts_in_partition do
19         (lhs, rhs) = estimatePartition(n, D) ;
20         for i = 0 to k do
21             j = k − i − 1 ;
22             ret1 = BEST_KCUTS(lhs.n, U_F, PI, lhs.D, i) ;
23             ret2 = BEST_KCUTS(rhs.n, U_F, PI, rhs.D, j) ;
24             total_cost = ret1.bestCost + ret2.bestCost ;
25             if ret.bestCost > total_cost then
26                 ret.bestCost = total_cost ;
27                 ret.results = ret1.results ∪ ret2.results ;
28 return ret ;
```

(line 7), the cuts are actuated and the new partitions become candidates for the next iteration (line 11). The iterations stop when k cuts cannot significantly reduce the cost of any of the partitions. The algorithm's behavior is tunable based on the value of k. For small values of k, the algorithm is fast but can only discover simple correlations. For larger values of k, the algorithm is more expensive but can discover more complex correlations.

The cuts we select are always filter constants that appear in at least one batch of the target workload. The idea is that if a cut does not correspond to a filter, then aligning it to an adjacent filter would further reduce the cost. Thus, cuts derived from the workload's filters reduce the search space without jeopardizing the quality of the solution.

7.2 Index Selection

The last component is to select the optimal index for processing the target workload. Index selection i) estimates the optimal data access cost $OC(S)$ of accessing a partition as is, without further subpartitioning, and thus is critical for IPSA and ii) finds which index minimizes the sum of SH_2O costs for a sequence of batches. For each batch, the incurred cost is the cost of the access strategy for the optimal attribute selection. The optimal set differs for each candidate index.

We use an algorithm that enumerates and evaluates all possible indices for a set of candidate index attributes. The candidate index attributes are the filter attributes in the given batches. For the case of the trie, the possible indices are the permutations of the subsets of candidate attributes. For each candidate index, the index selection algorithm runs attribute selection for each target batch in a what-if manner to estimate the sum of costs. In the end, it chooses to build the index that yields the lowest cost estimate. We omit the algorithm for brevity.

In the general case, the number of possible indices is exponential to the number of attributes. However, in practice, we see that SH_2O maximizes cost savings with a relatively small number of indexed attributes (up to 10). In that case, this approach is sufficient. However, scaling to a higher number of attributes, or adding more types of indices, may require heuristics.

8 IMPLEMENTATION

Our implementation is based on RouLette [33], an in-memory state-of-the-art work-sharing database. Our implementation for SH_2O modifies the *ingestion* and *shared filter* components: When scheduling a scan, the *attribute selection* process of Section 5 estimates the best subset of columns on which to use MSA. If the selected subset is empty, the original code path is used. Otherwise, we modify RouLette to use MSA for the best subset and exclude the corresponding columns from shared filters. After filtering, RouLette consolidates results.

MSA can retrieve more than a vector's worth of elements with each lookup. Then, we affinitize all the rows contained in the specific hyperrectangle to the worker performing lookup. The worker caches the lookup results, and in subsequent calls to ingestion, it extracts a vector of tuples directly from the cache. When all cached results are returned, the worker moves to the next hyperrectangle. By doing so, each worker processes its own exclusively-owned hyperrectangles, and synchronization overheads are reduced.

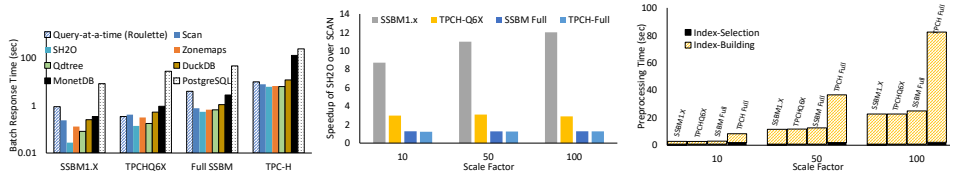
By default, Roulette uses single-partition tables. To enable the partitioning of Section 7, we modify ingestion. To minimize synchronization, initially, each worker is assigned a different partition. Once a worker finishes processing a partition, it requests another one. If there are no more unassigned partitions, the worker is attached to the same partition that has been assigned to another worker. Workers processing the same partition pull hyperrectangles from the same iterator.

RouLette, and by extension SH_2O, processes memory-resident data. Thus, the current implementation is optimized for in-memory processing. However, the approach is also beneficial for disk-based systems: First, selective access can significantly reduce I/O. Second, with modern SSDs achieving several GB/s in read bandwidth, probing a sequence of predicate indices is still too expensive to be masked by I/O. Still, a disk-based implementation requires extra optimizations: i) to avoid spreading range queries across several disk pages, data needs to be organized on disk using space-filling curves, and ii) similar to cooperative scans [40], to maximize bandwidth, we need to implement I/O scheduling. An extension for disk-based systems is outside the scope of this work.

9 EXPERIMENTAL EVALUATION

Our experiments evaluate SH_2O across three axes:
i) We first analyze the cost of shared scans and filters and show how multidimensional shared access can eliminate it.
ii) We discuss the introduced overhead by the number of hyperrectangles and demonstrate the merits of attribute selection.
iii) We show that multidimensional partitioning significantly reduces the data access cost for different workload families.

Access Methods. We evaluate the following methods: i) **Scan**: Shared full scan and filtering using the Data-Query model, ii) **MSA**: Access data by exclusively using MSA, iii) **SH2O**: This is our



(a) Batch response time for sf10 (b) Speedup over Scan (c) Preprocessing Time

Fig. 7. SSBM and TPC-H benchmarks

hybrid approach where we first use MSA and then apply shared post-filtering, iv) **Qd-tree [37]:** This is a state-of-the-art data skipping approach that partitions the data based on the workload, in a way that maximizes partition pruning, v) **Zonemaps**: This is standard data skipping over horizontal partitioning. When we assess this technique, the dataset is always sorted on the filtering attribute. We implement all aforementioned methods on Roulette.

We also compare against well-known databases: **MonetDB** [14], **PostgreSQL**, and **DuckDB** [29] that we use as baselines for query-at-a-time execution over indices. DuckDB and MonetDB are optimized for efficient columnar data access, and support the ART and ORDERED index, respectively. Finally, PostgreSQL has a mature B+-tree design and supports multidimensional indexing with GiST. We configure all databases to keep data and execution in-memory. Note that, as our workload is analytical (i.e., queries process hundreds of thousands tuples), the bottleneck is data access and processing, not the index traversal itself. Thus, we do not expect alternative indexing techniques to affect the showcased trends and conclusions.

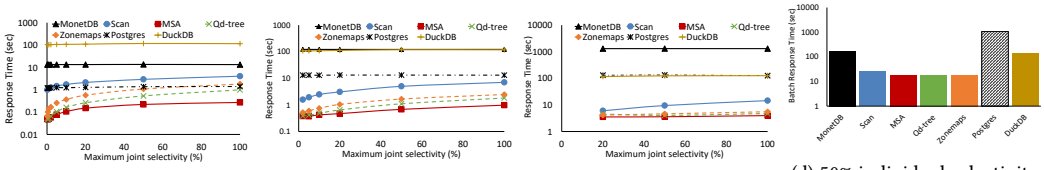
Data & Workload. We run both macro- and micro-benchmarks. First, we show how the proposed technique can accelerate analytical applications such as the widely used SSBM [26] and TPC-H benchmarks with scale factor 10. The order of the tuples is randomized. Then, we perform an extensive sensitivity analysis. More specifically, we investigate the effect of: i) concurrency (query batch size), ii) selectivity, iii) the number of filtering attributes, iv) data correlations, v) workload shift, vi) existing pre-computed aggregates, vii) the size of the resulting predicate indices, and viii) the predicate correlations. To allow controlling the experiment variables, for the purpose of the micro-benchmarks, we generate synthetic data. We use a single table of $256M$ rows and 4-byte integers. The number of columns and their distribution is presented in each experiment.

Hardware. We run the evaluation on a two-socket machine with Intel Xeon Gold 5118 CPU with 12 cores per socket, running at 2.30 GHz, and 378 GB of main-memory. We isolate execution on one NUMA-node and run all our experiments with 12 threads. The threads and memory are affinitized to the used NUMA node. We run all the experiments after the data has been loaded in-memory in columnar format, and report the average of 3 runs.

9.1 End-to-end Performance for Analytics

We evaluate the effect of SH2O on analytical queries using the Star Schema Benchmark (SSBM) [26] and TPC-H. Both SSBM and TPC-H measure the performance of databases for data warehousing applications. SSBM defines four select-project-join-aggregate query templates over a star schema. For each template, there are multiple variants with different selectivity. In total, SSBM has thirteen queries. TPC-H defines twenty-two query templates that cover more advanced SQL queries.

We compare SH2O against all other considered methods. We compare total response time for batches that contain the *full SSBM* and the *full TPC-H* as well as batches that consist of SSBM `Q1.1`, `Q1.2`, and `Q1.3` (*SSBM 1.x*), which are the only SSBM queries that compute filters directly on the fact table, and eight instances of TPC-H Q6 with different predicates (*TPCHQ6X*). TPC-H represents an adversarial example with limited sharing, and *TPCHQ6X* simulates dashboard queries with the



(a) 0.1% individual selectivity (b) 1% individual selectivity (c) 10% individual selectivity (d) 50% individual selectivity and 100% joint selectivity

Fig. 8. Effect of joint selectivity on data-access.

same template and different parameters. SH2O uses only one partition and chooses indices that use all the filter attributes on each fact table. The trie for TPC-H takes 8.7 sec to build and requires 216MB, whereas the trie for SSBM takes 2.2 sec to build and requires 53kB. The databases do not use indices in this experiment, Qd-tree uses the greedy algorithm from [37], and Zonemap splits the data, which uses the same ordering as the trie, into 1000 horizontal partitions.

Figure 7a shows the results of the end-to-end comparison in logarithmic scale. Scan performs better than query-at-a-time execution (both in Roulette and in databases), and SH2O always performs better than all competing methods. For SSBM 1.x, which aggressively filters the fact table, eliminating the shared filters makes a substantial difference and SH2O results in 12.9× over Scan. For the full SSBM, which accesses all rows in the fact table, SH2O still achieves a moderate 1.6×. Similarly, for the full TPC-H, the achieved speedup is 1.18×, while for *TPCHQ6X* it is 3×. To factor out join costs, for the full TPC-H case, we also compare the time that Scan and SH2O spend in data access and we find SH2O being 2.69× faster.

Scalability. We also compare Scan and SH2O for scale factors 50 and 100 and measure the response time, and the indexing-optimization times that are required by SH2O. Figure 7b shows results that are qualitatively similar to SF 10. For the data-intensive benchmarks the gains range from 3× to 11×. Figure 7c shows that the index-selection time is small and invariant to the scale factor, whereas indexing is proportional to the scale factor. Finally, as it only depends on the distinct values, which are the same, the index size is also the same across scale factors.

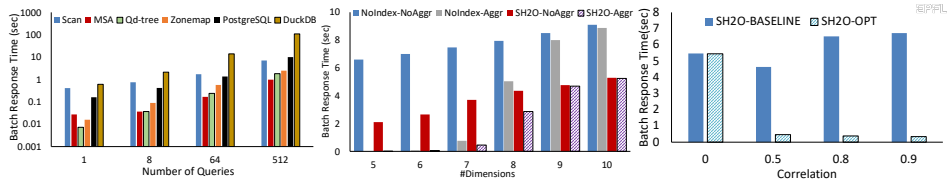
Takeaway: SH2O reduces response times even for complex queries. The benefit of SH2O is maximum for queries that filter large tables. However, by eliminating filter overhead, it still reduces response times even when almost all data is accessed.

Discussion: SH2O accelerates the *scan & filtering phase*. In join-heavy workloads, the join costs mask SH2O's benefit in data access. At worst, SH2O's benefit is marginal. Also, SH2O assumes use of work sharing and improves over shared scans. However, there exist workloads where work sharing and/or shared scans are suboptimal, such as point queries, and non-concurrent or non-overlapping queries. These are adversarial cases for SH2O as well.

9.2 Efficient Multidimensional Data Access

In this section, we show the costs from which existing techniques suffer when we vary i) the *joint selectivity*, ii) the *size* and iii) the *filtering attributes* of a query batch, and how MSA eliminates these costs. Moreover, we show how MSA handles the high number of hyperrectangles that data correlations produce.

Joint selectivity. We compare Scan, MSA, Qd-tree, Zonemaps, and the baseline databases (MonetDB, PostgreSQL, DuckDB), to analyze the behavior of shared scans and index-based methods under varying workload parameters. In this experiment, SH2O only uses MSA. We use a table with two uniformly distributed columns. We index the first column, whose domain is $[0, 100k)$. Building the trie takes 7.98 *sec* and requires 781kB. MonetDB uses the ordered index, PostgreSQL uses B-trees, and DuckDB always uses full scans even though we have built an ART index. All queries



(a) concurrency (b) precomputed aggregates (c) data correlations

Fig. 9. Effect of concurrency, precomputed aggregates and data correlations on response time.

filter the first column. For all the experiments of the section, we use batches of 512 filter-aggregate queries. The filter is a range on the first column.

Figure 8 shows the impact of the amount of common accesses for varying query selectivities (0.1%, 1%, 10%, and 50%). We control the *maximum joint selectivity*, that is the fraction of the table's rows that can be accessed by one or more queries. When maximum joint selectivity is 1%, all queries of the batch are generated such that their predicates request a subset of the selected fraction. When maximum joint selectivity is 100%, each query can access any possible range. To demonstrate the robustness of MSA to workload shifts compared to partition-based data-skipping approaches, the predicates of the query workload are shifted by 0.1% compared to the tuning workload of the Qd-tree. In the absense of workload shift, Qd-tree has identical performance with MSA.

The response time of all databases depends exclusively on the performance of individual queries and is unaffected from changes to the maximum joint selectivity. However, for an individual selectivity of 0.1% and a joint selectivity 100%, PostgreSQL is competitive to Qd-tree and Zonemaps.

Scan always processes the entire dataset. Thus, redundant filtering overheads are introduced and latency is high even when all queries access the same 0.2% of the data. Decreasing the maximum joint selectivity only changes the cost of aggregation at the end, and moderately affects the result.

By contrast, MSA significantly benefits from low maximum joint selectivity, and gradually pays the cost of spreading accesses across the table. When the workload contains queries with 0.1% selectivity and accesses less than 2% of the table, MSA gives a speedup of more than an order of magnitude compared to Scan. Even in the worst case, where accesses are completely uncorrelated, it achieves a lower response time by 45% as i) it still accesses fewer data, ii) incurs lower filtering overheads, and iii) achieves higher locality in the router before aggregations.

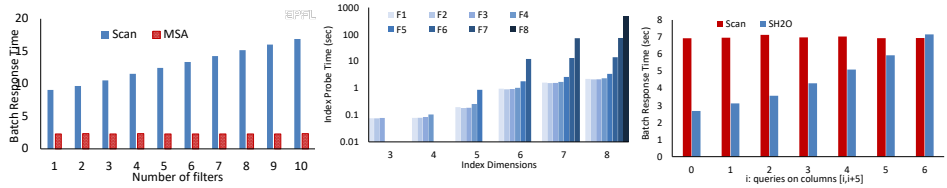
Data-skipping approaches also benefit from low joint-selectivity. However, as data accesses spread with increasing joint selectivity, they suffer from overfetching and filtering costs. Thus, MSA achieves up to 3.68× over Qd-tree and 6.66× over Zonemaps.

We also observe that MSA outperforms all databases regardless of individual selectivity. For 0.1% selectivity, it is 10× faster on average than PostreSQL, which is the fastest DB alternative (as it uses B-trees), and for 50% selectivity it is 7.7× faster than DuckDB.

Takeaway: Techniques that combine selective and shared access drastically improve interactivity in workloads that span a small portion of the table. Nevertheless, MSA is superior to data-skipping approaches, which are prone to overfetching and filtering costs.

Batch Size. Figure 9a shows the impact of concurrency on Scan, MSA, the data-skipping techniques, PostgreSQL, and DuckDB by varying the query batch size. We use the same data, indices, and workload as the scenario with 1% selectivity and 10% joint selectivity in Figure 8.

MSA exploits the fact that smaller batches effectively access a very small portion of the table. As the batch size is increased, response time is also increased but does so sublinearly until 64 queries. Scan starts with almost two orders of magnitude higher response time. Cost is increased along with the number of queries due to heavier query-set operations, larger predicate indices, and more aggregations. When the query batch size exceeds the 64 queries, query-set operations become



(a) filter dimensions (b) probing cost for various index dimensions (c) workload shift

Fig. 10. Effect of filter dimensions and workload shift on response time.

particularly heavy and cause a sharp latency increase for both methods. However, even for 512 queries, MSA is 7.2× faster than Scan and 1.8× faster than Qd-tree.

Takeaway: MSA gradually spreads data access across the table. It outperforms both indices and shared scans across the whole concurrency spectrum.

SH2O over pre-aggregations. Figure 9b shows that SH2O is both complementary and essential for minimizing response time when answering queries using pre-aggregated results. We use a table with $100M$ rows and a variable number of columns (from 5 to 10). Each column has 10 distinct values. We use 512 queries, and each filters on one of the columns. We compare four configurations: shared scans on the original table (NoIndex-NoAggr), shared scans on a pre-aggregation on the columns (NoIndex-Aggr), SH2O on the original data (SH2O-NoAggr), and SH2O on the pre-aggregated results (SH2O-Aggr). SH2O indexes 5 columns. SH2O-NoAggr eliminates 5 shared filters, outperforming NoIndex-Aggr. For both approaches, the response time is increased linearly with the number of columns. NoIndex-Aggr is efficient when aggregating on few columns, however, the number of resulting groups is exponential to the number of columns. Hence, for high-dimensional workloads, the number of groups is high enough that shared filters are time-consuming. Combining the two optimizations (SH2O-Aggr) both reduces the number of rows and reduces filtering overhead and thus outperforms all other approaches.

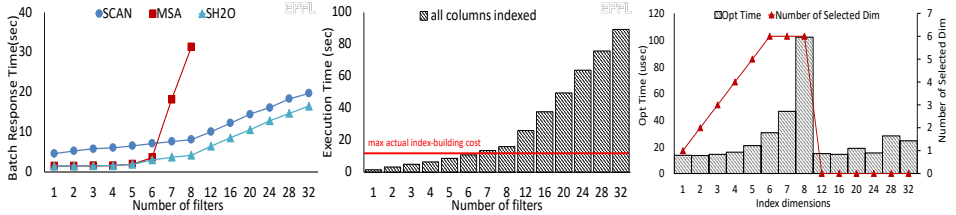
Takeaway:. While preaggregation reduces data size and thus latency, it still suffers from the data-access bottleneck. Combining it with SH_2O alleviates the overhead.

Data correlations. Figure 9c shows the effect of the early elimination optimization during hyperrectangle iteration. We use a table with three columns, where the first, C_1, is uniformly distributed in $[0, 1k)$. We make the 2^{nd} and 3^{rd} correlated to C_1 by adding uniform random variables. The range of the variables determines the correlation. Therefore, we use $[-500, 500]$ to achieve a correlation of 0.5, $[-250, 250]$ for 0.8 etc. Correlation 0 means that all columns are generated independently. Each query has a filter on one of the three columns and retrieves approximately 1% of the rows. We build an index on the three columns, which takes 21.2 − 22.5sec.

Without the optimization of Section 4.2, MSA suffers from the high number of hyperrectangles. The observed variance is due to differences in the trie's structure depending on the correlation. Enabling the optimization decreases the number of hyperrectangles and increases the benefit with correlation. When correlation becomes 1, the optimization reduces response time by 21.2×.

Takeaway: The response time of MSA depends on the actual number of non-empty hyperrectangles. Eliminating them is critical when the data contains correlations.

Filter attributes.Figure 10a assesses the impact of the number of filtering attributes on a table with 10 columns. The workload accesses the full table. To scale the number of filters without exploding the number of resulting hyperrectangles, which we test in the next Section, we assume that 9 columns of the table contain boolean $\{0, 1\}$ values. Each query contains two filters and has 5% selectivity: i) the first filter is on the same attribute across all queries and has 10% selectivity, ii) the second filter is on one of the boolean columns and has 50% selectivity. We vary the number of



(a) Scalability for wide tables (b) Cost of building the spatial index (c) Optimization time

Fig. 11. Impact of dimension selection during probing

boolean columns used for filters and, for each run, we use an index, which covers all used columns (first the common filter, and then the rest).

Scan shows a linear increase with the number of filters: more filters directly translate to increased query-set operations per tuple. MSA demonstrates near-constant performance; a slight decrease in execution time is due to better load balancing between threads. The efficiency of MSA stems from the fact that filtering costs are not spread to the whole dataset. Instead of the expensive query-set operations, MSA only probes and computes the corresponding query-sets for up to 10240 hyperrectangles. By amortizing the filtering overhead among all tuples of a hyperrectangle, it is 4.69× faster than Scan, despite that they both access the whole table. Finally, the indexing time is linear to the number of columns and varies from $4.5 - 45.8$sec.

Takeaway: The benefit of amortizing query-set overhead across a hyperrectangle is proportional to the number of filter attributes.

Index Access Cost. We measure the impact of dimensionality on the efficiency of the index. We vary the number of indexed attributes and the number of attributes we use for probing; *Fi* signifies that we probe i dimensions. The dataset contains 30 distinct values. Measurements that use the same probes attributes result in the same hyperrectangles and are thus comparable. Increasing the number of index dimensions increases the cost exponentially. By contrast, when only a small number of dimensions is used, index probe is in the sub-second range.

Thus, SH2O shines when the workload accesses few dimensions with relatively few distinct values. If the data characteristics are invariant to data scale, this holds even for large data sizes as demonstrated in Figure 7b. On the contrary, in cases where scaling the data also increases the data-value domains and combinations, and the accessed columns, SH2O is less effective: our cost model captures this trend and mitigates dimensionality by indexing and probing only the subset of the filter attributes in the workload that maximizes the net gain over Scan.

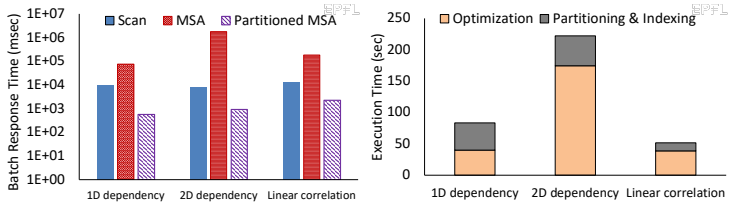
Takeaway:. Probing high-dimensional indices is expensive. The cost model is critical as it averts choosing an index or probe attributes that degrade performance.

Workload Shift. In all experiments thus far, queries target a subset of the columns that were used to build the index. Figure 10c shows the effect of workload shifts. We build indices on columns $[0, 5]$ and each bar of the plot represents a batch with queries on columns $[i, i + 5]$. Hence, in each step of the x-axis, we include one more non-indexed column. The takeaway here, is that there is no cliff and performance gracefully degrades until it converges to Scan.

Takeaway:. SH2O is robust to workload shift. The performance benefit decreases proportionally to the filter-attribute shift between the target and the runtime workload.

9.3 Scaling using Attribute Selection

Next, we demonstrate the effect of scaling the hyperrectangles and the importance of attribute selection. As this is a latent parameter that depends on the number of filter columns and the size



(a) Batch response time (b) Preprocessing cost

Fig. 12. Impact of partitioning for various workloads

of the corresponding predicate indices, we design two experiments: in each of these, we fix one parameter and vary the other.

First, we fix the size of each predicate index to 10 ranges and vary the number of filtering columns. To scale the experiment to wider tables, we use 32 uniformly distributed columns in $[0, 10)$. Due to the large number of attributes, we downsize the table to 100M rows. At each run, we build the spatial index on all the columns that are used for SH_2O. Each query has a single filter column and different query groups access non-overlapping columns (uncorrelated access pattern). We compare MSA, Scan, and SH_2O. We also compare against the GiST index of PostgreSQL, which is a generalized search tree. It provides support for multidimensional point of type cube. However, SH_2O achieves 14.1-73.4x faster response time hence we exclude GiST from the plots.

Figure 11a shows how attribute selection improves scalability. For MSA, both the number of hyperrectangles and the response time grow exponentially with the number of filter columns. After a crossing point, MSA becomes significantly slower than Scan, whose response time grows linearly.

Attribute selection chooses to use a subset of the filter columns such that expanding it with one more column would make the overhead higher than the savings from skipping the scan. Our cost model detects such cases and we observe that in this experiment, it never probes more than 6 columns (the remaining attributes are post-filtered). SH2O is more efficient than both MSA and Scan, as it trades filters for some of the attributes for a small overhead. The gains for the eliminated filters persist, and SH2O achieves 1.19× over Scan, even when the total number of filters is thirty-two.

Figure 11b shows the index building cost. If we index all available columns, the cost is minimal at first, but it sharply increases as the dimensionality is increased. The space overhead is also increased, at first exponentially until 7 attributes (85MB) and then linearly as the trie's leaves degenerate to single tuples. However, as our cost model is never going to probe more than 6 attributes, we leverage this information and only index the attributes that SH_2O would use based on the current batch. Thus, we guarantee that preprocessing cost is bounded and that it pays off.

Figure 11c shows how the optimization time varies with dimensionality. To vary dimensionality, we index all used filter columns. Optimization time grows exponentially to the number of columns. However, after 8 dimensions, our cost model detects that traversing the index deteriorates performance, prunes high-dimensional candidates, and falls back to shared scans. This way optimization is bounded to 100us and is applicable to real-time execution.

Takeaway: For MSA, the data-access time is determined by the number of hyperrectangles. For a large number of hyperrectangles, SH2O outperforms both pure scan- and index-based techniques.

9.4 Decoupling Dimensions using Partitioning

We evaluate the impact of partitioning on decorrelating filter dimensions. We use three common correlations in data analysis:

(1) *1D dependency*: a query in the batch has a predicate in column C_i iff it also has a predicate p_i in column A. $i \in \{1, 2, 3, 4\}$ and each column C_i has 100 distinct predicates. Before partitioning,

we index columns $A, C_1, \ldots, C_4$, in this order. After partitioning, we index in each partition only the filtered C_i.

(2) *2D dependency*: if a query in the batch has predicate p_i in column A and predicate q_j in column B, it also has a predicate in column $C_{(i+j)\%n+1}$. $i, j \in \{1, 2, 3, 4\}$ and each column C_k has 100 distinct predicates. Before partitioning, we index columns $A, B, C_1, \ldots, C_4$, in this order. After partitioning, we index in each partition only the filtered C_i.

(3) *Linear correlations*: if a query in the batch has a predicate A *between* X *and* Y, then it also has predicates $.C_i$ *between* $X + E$ *and* $Y + E$ where $E \in [-10, 10]$, $i \in \{1, \ldots, 5\}$. There are 100 distinct predicates in column A. We index columns $A, B_1, \ldots, B_5$, in this order, both before and after partitioning.

We run the iterative algorithm to partition the data based on the filters of the batch, actuate the partition, and finally run the batch itself. We report the response time in Figure 12a and the tuning time in Figure 12b. Note that response time is in msec and in log-scale.

For all three workloads, the number of hyperrectangles is high. As such, in all cases, single-partition MSA is orders of magnitude more expensive than Scan; 212× slower in the worst case. The partitions chosen by the iterative algorithm reduce MSA's response time by three orders of magnitude hence it outperforms Scan.

Takeaway: When filters occur in uncorrelated columns, data access time for MSA is prohibitive. In high-dimensional workloads, partitioning is necessary for making MSA the best option.

10 DISCUSSION

Although the current work focuses on reducing data-access cost, with sufficient optimizer support, SH_2O can also accelerate joins. Invisible joins [1], and data-induced predicates [16] propagate filters across joining tables that SH_2O can use.

As presented, SH_2O naturally comes with trade-offs and limitations. First, it requires space for storing the index, and processing time both for building and maintaining the index. Second, for workloads that filter on a large number of columns, and which have no correlations that partitioning can exploit, it effectively indexes and probes a limited number of columns and achieves low speedup. Third, it currently uses one clustered index for the whole workload and misses opportunities for using multiple unclustered indices. Finally, SH_2O is contingent on using work sharing, whereas in some use cases other alternatives are preferable, e.g., unclustered indices for point queries and materialized views in low-concurrency, real-time applications with static workload.

11 CONCLUSIONS

To provide interactivity for highly concurrent workloads, we propose SH_2O, a novel data-access method that combines efficient selective access with minimal filtering cost, and scalability. SH_2O amortizes filtering cost by exploiting multidimensional regions where filtering decisions are invariant across all tuples. However, multidimensional data access suffers from the curse of dimensionality. To avoid dimensionality pitfalls, SH_2O employs two complementary mechanisms, attribute selection and partitioning. By probing only a select subset of dimensions and taking advantage of query and data correlations, SH_2O outperforms shared scan and filters from 1.8× to 22.2×.

ACKNOWLEDGMENTS

We would like to thank our anonymous reviewers for their thoughtful comments and constructive criticism. Furthermore, we would like to thank our friends and colleagues for their feedback and support. Panagiotis Sioulas was supported by the 2020 Facebook Fellowship Program.

REFERENCES

[1] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. 2008. Column-Stores vs. Row-Stores: How Different Are They Really?. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 967–980. https://doi.org/10.1145/1376616.1376712

[2] Sanjay Agrawal, Nicolas Bruno, Surajit Chaudhuri, and Vivek R Narasayya. 2006. AutoAdmin: Self-Tuning Database SystemsTechnology. *IEEE Data Eng. Bull.* 29, 3 (2006), 7–15.

[3] Subi Arumugam, Alin Dobra, Christopher M Jermaine, Niketan Pansare, and Luis Perez. 2010. The DataPath system: a data-centric analytic processing engine for large data warehouses. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 519–530.

[4] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (sep 1975), 509–517. https://doi.org/10.1145/361002.361007

[5] George Candea, Neoklis Polyzotis, and Radek Vingralek. 2009. A scalable, predictable join operator for highly concurrent data warehouses. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB)*.

[6] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew McCormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, Roee Aharon Ebenstein, Nikita Mikhaylin, Hung ching Lee, Xiaoyan Zhao, Guanzhong Xu, Luis Antonio Perez, Farhad Shahmohammadi, Tran Bui, Neil McKay, Vera Lychagina, and Brett Elliott. 2019. Procella: Unifying serving and analytical data at YouTube. *PVLDB* 12(12) (2019), 2022–2034. https://dl.acm.org/citation.cfm?id=3360438

[7] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity Estimation for Range Predicates Using Lightweight Models. *Proc. VLDB Endow.* 12, 9 (May 2019), 1044–1057. https://doi.org/10.14778/3329772.3329780

[8] Peter M. Fischer and Donald Kossmann. 2005. Batched Processing for Information Filters. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*. IEEE Computer Society, USA, 902–913. https://doi.org/10.1109/ICDE.2005.25

[9] Georgios Giannikis. 2014. *Work Sharing Data Processing Systems.* Ph. D. Dissertation. ETH Zurich, Zürich, Switzerland. https://doi.org/10.3929/ethz-a-010265242

[10] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2012. SharedDB: killing one thousand queries with one stone. *arXiv preprint arXiv:1203.0056* (2012).

[11] Goetz Graefe. 2009. Fast loads and fast queries. In *International Conference on Data Warehousing and Knowledge Discovery*. Springer, 111–124.

[12] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '84)*. Association for Computing Machinery, New York, NY, USA, 47–57. https://doi.org/10.1145/602259.602266

[13] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. 2005. Qpipe: A simultaneously pipelined relational query engine. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 383–394.

[14] Stratos Idreos, F. Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.* 35 (01 2012).

[15] Panos Kalnis, Nikos Mamoulis, and Dimitris Papadias. 2002. View selection using randomized search. *Data & Knowledge Engineering* 42, 1 (2002), 89–111.

[16] Srikanth Kandula, Laurel Orr, and Surajit Chaudhuri. 2019. Pushing Data-Induced Predicates through Joins in Big-Data Clusters. *Proc. VLDB Endow.* 13, 3 (nov 2019), 252–265. https://doi.org/10.14778/3368289.3368292

[17] Donghe Kang, Ruochen Jiang, and Spyros Blanas. 2021. Jigsaw: A data storage and query processing engine for irregular table partitioning. In *Proceedings of the 2021 International Conference on Management of Data*. 898–911.

[18] Michael S Kester, Manos Athanassoulis, and Stratos Idreos. 2017. Access path selection in main-memory optimized data systems: Should I scan or should I probe?. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 715–730.

[19] Donald Kossmann and Konrad Stocker. 2000. Iterative Dynamic Programming: A New Class of Query Optimization Algorithms. *ACM Trans. Database Syst.* 25, 1 (mar 2000), 43–82. https://doi.org/10.1145/352958.352982

[20] Jonathan K. Lawder and Peter J. H. King. 2000. Using Space-Filling Curves for Multi-Dimensional Indexing. In *Proceedings of the 17th British National Conferenc on Databases: Advances in Databases (BNCOD 17)*. Springer-Verlag, Berlin, Heidelberg, 20–35.

[21] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[22] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. 2002. Continuously Adaptive Continuous Queries over Streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (Madison,

Wisconsin) *(SIGMOD '02)*. ACM, New York, NY, USA, 49–60. https://doi.org/10.1145/564691.564698

[23] Darko Makreshanski, Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2016. MQJoin: Efficient Shared Execution of Main-memory Joins. *Proc. VLDB Endow.* 9, 6 (Jan. 2016), 480–491. https://doi.org/10.14778/2904121.2904124

[24] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 476–487.

[25] Der Technischen Universität München and Volker Markl. 1999. MISTRAL: Processing Relational Queries using a Multidimensional Access Technique.

[26] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *TPCTC*. 237–252.

[27] Apache Pinot. 2023. https://pinot.apache.org/.

[28] Lin Qiao, Vijayshankar Raman, Frederick Reiss, Peter J. Haas, and Guy M. Lohman. 2008. Main-Memory Scan Sharing for Multi-Core CPUs. *Proc. VLDB Endow.* 1, 1 (aug 2008), 610–621. https://doi.org/10.14778/1453856.1453924

[29] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.

[30] Robin Rehrmann, Carsten Binnig, Alexander Böhm, Kihong Kim, Wolfgang Lehner, and Amr Rizk. 2018. OLTPshare: The Case for Sharing in OLTP Workloads. *Proc. VLDB Endow.* 11, 12 (aug 2018), 1769–1780. https://doi.org/10.14778/3229863.3229866

[31] Nicholas Roussopoulos. 1982. View indexing in relational databases. *ACM Transactions on Database Systems (TODS)* 7, 2 (1982), 258–290.

[32] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1802–1813. https://doi.org/10.1109/ICDE.2019.00196

[33] Panagiotis Sioulas and Anastasia Ailamaki. 2021. Scalable Multi-Query Execution using Reinforcement Learning. In *Proceedings of the 2021 International Conference on Management of Data*. 1651–1663.

[34] Liwen Sun, Michael J Franklin, Sanjay Krishnan, and Reynold S Xin. 2014. Fine-grained partitioning for aggressive data skipping. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1115–1126.

[35] Liwen Sun, Michael J Franklin, Jiannan Wang, and Eugene Wu. 2016. Skipping-oriented partitioning for columnar layouts. *Proceedings of the VLDB Endowment* 10, 4 (2016), 421–432.

[36] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. 2009. Predictable Performance for Unpredictable Workloads. *Proc. VLDB Endow.* 2, 1 (aug 2009), 706–717. https://doi.org/10.14778/1687627.1687707

[37] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning data layouts for big data analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 193–208.

[38] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep Unsupervised Cardinality Estimation. *Proc. VLDB Endow.* 13, 3 (nov 2019), 279–292. https://doi.org/10.14778/3368289.3368294

[39] Jingren Zhou, Per-Ake Larson, Jonathan Goldstein, and Luping Ding. 2007. Dynamic materialized views. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 526–535.

[40] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. 2007. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *Proceedings of the 33rd International Conference on Very Large Data Bases* (Vienna, Austria) *(VLDB '07)*. VLDB Endowment, 723–734.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009