Oligolithic Cross-task Optimizations across Isolated Workloads[*]

Eleni Zapridou
eleni.zapridou@epfl.ch
EPFL
Switzerland

Panagiotis Sioulas[**]
panagiotis.sioulas@epfl.ch
Oracle
Switzerland

Anastasia Ailamaki[**]
anastasia.ailamaki@epfl.ch
EPFL, Google
Switzerland, USA

ABSTRACT

Enterprises collect data in large volumes and leverage them to drive numerous concurrent decisions and business processes. Their teams deploy multiple applications that often operate concurrently on the same data and infrastructure but have widely different performance requirements. To meet these requirements, enterprises enforce resource boundaries between applications, isolating them from one another. However, boundaries necessitate separate resources per application, making processing increasingly resource-hungry and expensive as concurrency increases. While cross-task optimizations, such as data and work sharing, are known to curb the increase in total resource requirements, resource boundaries render them inapplicable.

We propose the principle of functional isolation: cross-task optimizations can and should be combined with performance isolation. Systems should permit cross-optimization as long as participating tasks achieve indistinguishable or improved performance compared to isolated execution. The outcome is faster, more cost-effective, and more sustainable data processing. We make an initial step toward our vision by addressing functional isolation for work sharing and propose GroupShare, a strategy that reduces both total CPU consumption and the latency of all queries.

1 INTRODUCTION

Data is increasingly important for producing value. Organizations use data-sharing platforms (i.e., common infrastructure giving multiple systems access to shared data) more and more to facilitate collaboration on data-driven projects both internally and with other organizations and extract value, fueling various of their applications. In this data economy, the ultimate goal is to maximize the extracted value by enabling more and more users to submit a growing number of increasingly complex data-intensive tasks.

Various teams and data practitioners run tasks that process the same data but have different requirements and characteristics. The characteristics stem from each task's nature and include its type (e.g., real-time, analytical, ML-based) and resource needs (e.g., CPU, memory, bandwidth). The requirements are specified explicitly by

[*]Cross-task optimization techniques are "monolithic", as they execute all tasks with a single execution schedule, while resource isolation techniques that use a different schedule per task are "polylithic". We propose that systems should follow an "oligolithic" strategy – they should utilize cross-task optimizations only when they do not compromise performance isolation.

[**]Work done entirely at EPFL.

14th Annual Conference on Innovative Data Systems Research (CIDR'24), January 14–17, 2024, Chaminade, USA
2024.

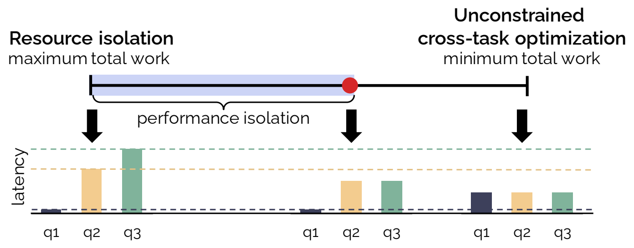


Figure 1: The red dot identifies the execution strategy that minimizes total work while preserving performance isolation. The dashed lines represent the latency requirements for each query.

the users to satisfy their performance needs. They vary from resource reservations per user, group of users, or group of applications to complex requirements such as desired accuracy and latency.

For example, consider the use case of analyzing e-commerce client logs that are stored in a database shared across applications. A real-time dashboard processes the logs to produce visualizations based on actions from analysts. It issues short analytical queries selecting only data for a specific time window, e.g., the last hour, but demands sub-second latency to enable analysts to be productive. Meanwhile, the service's recommendation system uses a periodic background job to analyze weekly trends for groups of similar users. While this job has loose latency requirements (in the range of hours), it submits queries that process all historical data and could occupy the entire infrastructure for several minutes.

To satisfy the requirements of concurrent tasks, data-sharing platforms adopt a resource-centric design. The system assigns a specific subset of resources to each task based on its requirements. The goal is to isolate tasks by enforcing resource boundaries between them so that one cannot affect the resources and, thus, the performance of another. For the above example, boundaries ensure that dashboard users experience short latency regardless of whether the recommendation system's queries are running.

Provisioning separate resources per task requires a large, expensive, and energy-hungry infrastructure that must grow with the level of concurrency. Exceeding the infrastructure's capacity has an adverse impact on tasks (e.g., the dashboard's response time can increase to the extent that it disrupts the analyst's productivity). Cross-task optimizations, such as data and work sharing, reduce the total resource requirements but are prohibitive because they can violate isolation and introduce unpredictable results, e.g., the dashboard's queries take minutes when running concurrently with the recommendation query and finish in seconds otherwise.

Resource isolation is a means to an end. What matters to users is achieving *functional isolation* — achieving performance metrics that are at least equal or better than those of isolated execution, rather than having exclusive resources, as long as security is guaranteed. Between resource isolation and unconstrained cross-task optimization lies a full spectrum of execution strategies (illustrated in Figure 1) that relax resource isolation and leverage some of the cross-task optimization opportunities. On one end, the resource isolation strategy respects the queries' latency requirements but incurs high total work. On the other end, the schedule that takes advantage of all cross-optimization opportunities minimizes total work but penalizes the fastest query. Crucially, many of the strategies between the two extremes achieve performance isolation. We propose that systems should aim to find the strategy that minimizes total work using cross-task optimizations given performance isolation constraints (red point in Figure 1). This strategy reduces resource consumption and improves performance when the infrastructure is overcommitted. As task concurrency increases, systems can then identify more optimization opportunities, becoming increasingly more sustainable and cost-effective than isolated execution.

We test our vision for the use case of sharing work across analytical queries. We propose GroupShare, an execution strategy that achieves performance isolation and, at the same time, exploits sharing opportunities, reducing the total processing time by up to 6×. To do so, GroupShare finds, by using runtime monitoring, *sharing groups*, that is, groups of queries that can meet the isolation guarantees of each participating query when pooling their resources to process a shared query plan.

Concretely, we make the following contributions:

- We propose the principle of functional isolation; systems should use cross-query optimizations to decrease the overall processing time while respecting performance isolation.
- We show that functional isolation for work sharing can be achieved by partitioning queries into sharing groups.
- Building on the idea of sharing groups, we propose GroupShare, an algorithm that exploits sharing while preserving fair allocation of CPU time.

2 THE CASE FOR FUNCTIONAL ISOLATION

In the early 1960s, researchers had already acknowledged the need for concurrent program execution and its performance implications and proposed scheduling mechanisms to limit interference between concurrent programs [15, 16]. The proposed concepts have been formative in various domains, including operating systems, cloud computing, databases, and "big data" stacks.

Databases and big data stacks achieve performance isolation through resource allocation. For each task, these systems choose a sequence of execution steps, then assign a portion of the overall resources to each step and execute it using these specific resources. Thus, the performance of each task depends solely on the execution steps and its allocated resources and remains unaffected by other concurrent tasks. For example, PostgreSQL [9], IBM DB2 [2], and SAP HANA [18] spawn a number of OS threads or processes for each request and rely on the OS's fair scheduling mechanisms. Other databases such as Umbra [40] and Microsoft SQL Server [7] directly manage the machine's CPU cores and allocate a time-share to each query using stride scheduling. Big data systems such as Hadoop and Spark rely on a resource negotiator such as YARN [39], Kubernetes [12], or Mercury [25] to fairly allocate specified requested resources. Finally, cloud providers such as Google [6], Oracle [8], and Amazon AWS [4] structure their pricing models based on resource reservations for at least one system resource, including CPU, memory, and/or storage resources.

Allocating resources per task comes with a price; as concurrency increases, the infrastructure must continuously grow to cover the cumulative demand. Two decades ago, when concurrency was low and single-machine databases were sufficient in most cases, this was a reasonable price to pay for the separation of concerns. However, this is no longer the case; Today's organizations are deploying an increasing number of data-driven applications with tens, hundreds, or even thousands of concurrent tasks [1, 3, 5, 14], each tasked with handling ever-expanding volumes of data. At the same time, the rate at which the price-to-performance ratio of hardware resources improves has slowed down [10, 22]. Consequently, coping with the increasing demand doesn't come for free with the hardware evolution. The infrastructure must grow proportionally to concurrency; otherwise, the quality of service will suffer, hindering mission-critical operations.

2.1 Beyond (Pure) Resource Isolation

To contain the growing demand for resources, we need to rethink siloing allocated resources. Cross-optimizing tasks saves resources and offsets increasing concurrency. For example, in Microsoft's data centers, 45% of the jobs have overlapping computations, which, if exploited, can save up to 40% of all machine hours [23]. Importantly, as concurrency increases, the cross-optimization opportunities increase as well. Cross-optimization techniques have been studied at least as far as the 1980s [33] with recent techniques ranging from machine learning-optimized [35] to heuristic-based strategies [11, 13, 28]. However, these techniques risk interference, e.g., for two partially overlapping jobs, one short and one long-running, computing the union of their results and filtering each answer separately penalizes the faster job. Thus, cross-optimization needs to be constrained so that it respects performance isolation.

We propose that, instead of resource isolation, systems should adopt functional isolation; they should cross-optimize tasks as long as each task achieves better or equal performance compared to isolated execution. In a sense, functional isolation encodes an incentive for individual tasks to give up resource isolation. Systems should provide the mechanisms for choosing which tasks should pool their resources, cross-optimize, cross-execute, and hence profit.

Functional isolation is achieved by co-optimizing planning and resource allocation across tasks and guarantees the isolation and performance goals achieved through schedulers and resource managers. This marks a clear departure not only from the existing model that decouples planning and scheduling but also from recent ideas that propose jointly determining each task's query and resource plan to maximize performance [24].

2.2 Use Case: Sharing across Isolated Workloads

In this section, we define functional isolation for the case of data and work sharing. This is the first definition of functional isolation,

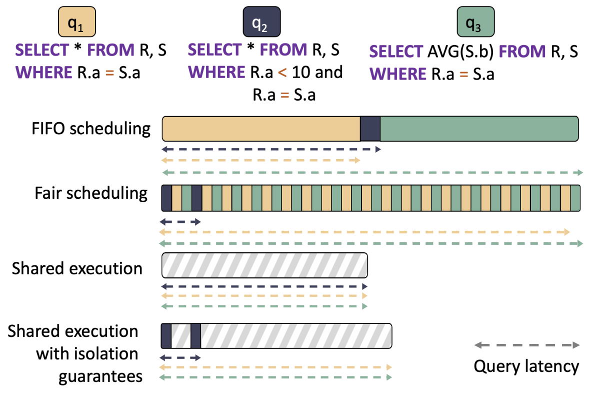


Figure 2: Per-query latency and total execution time comparison with different scheduling algorithms

which we believe will drive impactful research towards optimizing the next-generation data-sharing and cloud platforms.

Let $q_1, \ldots, q_n$ be a set of queries to execute. Each query q_i takes processing time $T(q_i)$ when using all the resources in the infrastructure. For simplicity, let us assume that all queries are submitted at the same time. Each query's latency $L(q_i)$ is defined as its end-to-end time. Furthermore, let us assume that execution takes place in a single machine and is in memory, and resource reservations concern only the CPU. The above assumptions serve the ease of presentation and can be lifted with a more complex formulation.

Performance isolation via scheduling: Concurrent queries can vary significantly in performance, e.g., $max_i T(q_i)$ may be orders of magnitude larger than $min_i T(q_i)$. Fairness ensures graceful degradation of query latencies proportional to the concurrency level, providing predictability and a form of performance isolation.

Databases achieve fairness with respect to CPU allocation through scheduling. Figure 2 shows, through a three-query example, the impact of scheduling on performance isolation. Colored bars signify which query occupies the CPU at a certain moment, while dashed arrows show the query's latency. Intuitive approaches such as First-In-First-Out (FIFO) scheduling, which executes queries in the order of arrival, can arbitrarily penalize short-running queries. Suppose the queries arrive in the order q_1, q_2, q_3. A FIFO scheduler would execute query q_2 only after q_1 has finished. The user of query q_2 would experience orders of magnitude higher latency. Reordering the queued queries to prioritize short-running queries can address the problem but requires estimates derived during query optimization, which are known to be error-prone.

An approach that does address fairness is to time-share the CPU across queries to achieve what is called fair scheduling. Queries take turns running on the CPU for a limited period of time, called a quantum. By the end of the quantum, if the query has not been completed, the query yields, and the CPU is given to the next query in the ready queue. As shown in Figure 2, with fair scheduling, all queries get 1/3 or higher percentage of the CPU. Fair scheduling can be further extended to proportional share scheduling, which allocates a specific fraction of the CPU to each query. Proportional share scheduling algorithms include lottery scheduling [41], stride scheduling [40, 42], and EEVDF [37]. Databases that use fair or proportional share scheduling achieve performance isolation as query latency is expected to be at most $L(q_i) \leq \frac{T(q_i)}{share_i}$, where $share_i$ is the time-share of the CPU requested by q_i.

Shared execution Shared-execution techniques reduce total processing time and, thus, resource consumption by detecting and exploiting overlaps between concurrent queries. They are classified in data and work-sharing approaches[21].

Data-sharing techniques exploit common data accesses across queries. Disk-based databases like Microsoft SQL Server and Teradata [21], which are I/O-bound, use scan sharing to amortize the cost of disk accesses. Cooperative scans [43] improve shared disk access by scheduling I/O requests to maximize the effective bandwidth and minimize latency penalties. BatchSharing aims to mitigate the memory bandwidth bottleneck [31]. It is noteworthy that both cooperative scans and BatchSharing aim to achieve fairness.

Work-sharing techniques exploit common operators across queries. They use a shared query plan [11, 19, 21, 29, 35] to process all queries, and, in recent systems, the Data-Query model [11, 13, 19, 27–29, 35] to efficiently share work between queries with the same joins and different filters. The shared operators compute the union of the results for the participating queries and route their output to one or more parent operators. Choosing a shared plan has been studied extensively in the context of multi-query optimization [32–34], sharing-aware optimization [20, 35], and heuristics [11, 13, 28]. Prior work discusses sharing or not sharing queries based on selectivity [20] or overlap [26], but it does not take into account the individual query requirements. [38] decides to share or not incremental queries based on final work constraints. We make a case for cross-optimizing applications and tasks with diverse requirements and characteristics while respecting isolation guarantees.

Sharing violates performance isolation: Using a shared global plan to execute all queries might violate individual query requirements. For the queries of Figure 2, a shared query plan would use a shared join operator processing $R \bowtie S$ and route its results to a different parent for each query. This plan decreases total processing time by approximately two times. However, all queries finish when the shared join finishes, and so, the latency of q_2 is drastically increased compared to if it was executed independently using 1/3 of the CPU, violating performance isolation. This is because most of the processed intermediates belong to q_1 and q_3; these two queries benefit from sharing and have the lowest latency in this case.

Functional isolation for sharing: Let us consider fair scheduling as the baseline for performance isolation. Sharing is only permissible when it results in all queries finishing at least as fast as they would with fair scheduling. From the user's point of view, the system must perform at least as well as it would using resource isolation. We define the concrete conditions for functional isolation.

Assume that each query q_i has requested a time-share $share_i$ of the CPU. We partition the queries in non-overlapping groups $G_1, \ldots, G_m$ such that $\cup_{j=1}^{m} G_j = \{q_1, \ldots, q_n\}$ and run each group's shared plan using a time-share $share_j'$. We denote each group's latency, which is also the latency of all participating queries, as $L(G_j)$. In this setup, sharing achieves functional isolation if, for each group, G_j, the following conditions hold:

(1) $L(G_j) \leq L(q_{j_k}), \forall q_{j_k} \in G_j$.
(2) $share_j' \leq \sum_{q_{j_k} \in G_j} share_{q_{j_k}}, \forall q_{j_k} \in G_j$

The objective is twofold: i) queries achieve the same or lower latency compared to isolated execution, and ii) resource consumption for each group is the same or lower than the aggregated resources of individual tasks. Then, the processing time of a query in the shared case is lower or equal to the isolated case:

$$T(G_j) = L(G_j) \times share_j^{'} \leq L(G_j) \times \sum_{q_{j_k} \in G_j} share_{q_{j_k}}$$
$$\leq \sum_{q_{j_k} \in G_j} (L(q_{j_k}) \times share_{q_{j_k}}) \leq \sum_{q_{j_k} \in G_j} T(q_{j_k}) \quad (1)$$

Groupings for which at least one of the inequalities holds are superior to resource isolation. We call such groups *sharing groups*. The surplus resources may be used to further improve performance, reduce infrastructure costs, or boost properties such as availability and elasticity. In the last case of the example in Figure 2, q_1 and q_3 form one sharing group, while q_2 forms its own group. The groups reduce both latency and total processing time.

3 SHARING WITHOUT REGRETS

We introduce GroupShare, a hybrid scheduling-optimization algorithm that partitions and executes queries into sharing groups, providing at least as fast progress as fair scheduling and, thus, lower latency and reducing the total CPU time. We first discuss the practical challenges that drive GroupShare's design and then present the algorithm.

3.1 Challenges

Finding groups of queries that can share work while respecting isolation guarantees has two key challenges: *scalability* and *accuracy*.

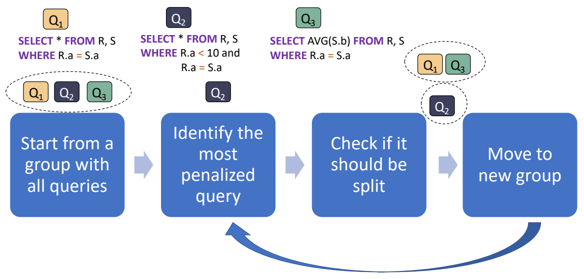
Scalability: As there can be hundreds or thousands of concurrent queries, any scheduling algorithm needs to be efficient at that scale. Exhaustively searching for the best grouping for a large number of queries is prohibitive both due to the large search space of possible partitions and the overhead of validating whether each group's optimal shared plan accelerates the progress of all queries. The number of all possible ways of partitioning n queries into groups is:

$$\sum_{k=1}^{n} S(n,k) = \sum_{k=1}^{n} \sum_{i=0}^{k} \frac{(-1)^{k-1} i^n}{(k-i)! i!}$$

where $S(n,k)$ the Stirling number of the second kind and k the number of groups. For example, even when processing 10 queries, there are 115975 ways to partition the queries into groups, and finding the optimal shared plan for each candidate group can take several milliseconds or seconds. Running the queries without co-optimization would require much less processing time than finding the optimal grouping and plans.

GroupShare instead is opportunistic: by design, it avoids searching for the optimal grouping of queries among all possible partitions. Instead, it focuses on finding one partition that consists of sharing groups, i.e., groups of queries where co-optimization is guaranteed to bring a net benefit. Converging to a grouping with guarantees is critical for production environments where predictability and explainability are paramount.

Accuracy: Evaluating sharing opportunities a priori is challenging as it necessitates predicting query overlap, which relies on estimating cost and intermediate cardinalities, known for their inaccuracy. GroupShare instead employs runtime information from trial-and-error execution. It measures the progress rate of the individual queries and the shared plans and uses the measurements to make grouping decisions. This way, similar to solutions for runtime fair scheduling, it bypasses the need for a priori estimates.


```
SELECT * FROM R, S
WHERE R.a = S.a

SELECT * FROM R, S
WHERE R.a < 10 and
      R.a = S.a

SELECT AVG(S.b) FROM R, S
WHERE R.a = S.a
```

Q1
Q2
Q3
Start from a group with all queries
Identify the most penalized query
Check if it should be split
Move to new group

Figure 3: Overview of re-grouping queries in GroupShare

3.2 Identifying Sharing Groups

GroupShare's key idea is that any set of queries can be partitioned into one or more sharing groups. Starting from a group containing all queries, GroupShare continuously regroups queries until it converges into a partition that satisfies isolation guarantees for all queries. Subsequently, it uses this grouping until all queries finish.

We first present GroupShare's components: (a) grouping, (b) monitoring, (c) stride scheduling, and (d) delta queries, and then, we describe the algorithm itself.

Groups: At any given moment, queries are partitioned into a set of groups. The groups are ordered by their rate of progress: the first group makes the slowest progress (e.g., it contains more or heavier queries), and the last group makes the fastest. Each group's queries progress in lockstep (at the group's rate) because, at every step, GroupShare schedules a group's shared plan to process a slice of data. In the trivial case, a group contains a single query.

Testing if a group is a sharing group is critical. Let group $G_j = \{q_{j_1}, \ldots, q_{j_m}\}$ be a set of queries, t_{G_j} the processing rate of G_j's shared plan, and t_{q_i} the processing rate of query q_i in isolation – *monitoring* estimates both. G_j is a sharing group if and only if:

$$t_{q_{j_k}} \leq a \times t_{G_j} \times |G_j|, \ \forall k \in \{1, \ldots, m\} \quad (2)$$

where $a < 1$ is a constant that controls the sensitivity of grouping. If a is low, GroupShare uses sharing only when the benefit is high.

GroupShare examines the existing groups and adjusts partitioning until it converges to having only sharing groups. Figure 3 shows this iterative process for the queries of Figure 2. For each group, GroupShare finds the most penalized query (defined in *stride scheduling*) and tests whether inequality 2 holds. If so, the query progresses fast enough to satisfy its isolation guarantees and remains in the group; otherwise, it moves to the next group in line. If there is no next group, the query forms a new group on its own. To amortize bookkeeping, GroupShare removes and attaches queries at the granularity of chunks with a few tens of thousands of rows.

Monitoring: Monitoring collects runtime information about the processing rate of each query and each group. The rate is the ratio of the number of rows in each slice of data over the required processing time. Monitoring has two phases: (a) normal and (b) isolated. During the normal phase, GroupShare processes each group's shared plan. By contrast, during the isolated phase, it runs each query belonging to the scheduled group separately and computes the processing rate for each of these queries. The decision on which phase to use is randomized: GroupShare chooses the normal phase with (high) probability p and the isolated phase with (low) probability $1 - p$.

Stride scheduling: At each step, GroupShare chooses a group to schedule for a slice of data using a variant of stride scheduling [42], a well-known scheduling algorithm for allocating a target fraction of the CPU time to a work unit/query. It is also used in state-of-the-art query scheduling. It measures the CPU allocation for each query q_i using a quantity called the pass P_i. For simplicity, we assume equal priorities across all queries, but priorities could be easily added to our formulation. After processing a query for time $f \times T$, where T is the target duration for each slice of data, stride scheduling updates the pass as follows:

$$P_i \leftarrow P_i + f$$

In the long term, stride scheduling guarantees that CPU time will be split equally among queries.

GroupShare adapts stride scheduling: after processing the shared plan for G_j, it updates the pass of each query q_{j_k} using:

$$P_{j_k} \leftarrow P_{j_k} + f \times \frac{t_{G_j}}{t_{q_{j_k}}}$$

Here, $f \times T \times \frac{t_{G_j}}{t_{q_{j_k}}}$ is the processing time that q_{j_k} would take to make the same progress. Thus, we credit each query with the equivalent CPU time in isolated execution. The group of the query that has been credited the least CPU time is scheduled at each step. CPU allocation is thus driven by the requirements of the query with the least allocation from each group. This query still progresses fast enough due to the sharing-group test.

Delta queries: When attaching a query to the next group, the new group may have already processed some chunks ahead. To maintain correctness, the newly attached query needs to process the chunks between the cursors of its old and new groups. For these chunks, GroupShare issues delta queries. Each delta query is represented as a virtual group that is scheduled for processing every time the respective query is the most penalized one. After its delta query finishes, scheduling the query results in scheduling its group.

Overall, GroupShare's execution flow is as follows: Initially, the algorithm starts from a group that contains all running queries. At every step of execution, the algorithm identifies the most penalized query and its associated group or virtual group. If the group is in between chunks, GroupShare checks if the query needs to move to the next group; if so, it moves the query to a queue for the next group, registers the delta queries, and restarts the scheduling step. Similarly, a group starting a new chunk checks for queries that need to be attached to it and only then starts processing the next chunk. While the chunk has more data, the group takes a vector of data and processes it. GroupShare chooses between the normal and isolated phases for execution. In both cases, it runs the chunk, updates throughput measurements, and updates each query's stride.

The benefit of applying GroupShare to a batch of queries depends on the benefit of sharing work within each sharing group. This, in turn, depends on the overlap in the sharing group's best shared plan; the best shared plan is not necessarily the result of stitching together the plans of individual queries, as reordering operators may expose more overlap. The higher the benefit of sharing, the more likely it is that GroupShare can share subexpressions without penalizing any query, and hence, the more GroupShare can decrease the total amount of work and the queries' execution time. In the worst case, GroupShare decides to execute queries individually and performs similarly to fair scheduling, while in the best case, all queries can be cross-optimized in a single group sharing all common subexpressions.

Convergence: We prove that GroupShare converges to sharing groups, assuming that the queries run for long enough (until it converges) and that the processing rate of each query or group of queries is constant and known. For ease of presentation, we assume that stride scheduling schedules each query or group of queries for exactly time T and do not take monitoring steps into consideration.

Lemma 1. *Let there be groups of queries $G_1, G_2, \ldots, G_m$, where $G_1, \ldots, G_{k-1}$ sharing groups. After GroupShare schedules G_k at most $|G_k|$ times, G_k will be a sharing group as well.*

Proof. We prove the lemma through induction on $|G_k|$, i.e., the number of queries in group G_k. Assume Q is the set containing all queries.

For $|G_k| = 1$, G_k is trivially a sharing group.

Let the lemma hold for $|G_k| = p$. Then, we show that the lemma also holds for $|G_k| = p + 1$.

Let's assume that G_k is not a sharing group. As groups $G_1, \ldots, G_{k-1}$ are already sharing groups and remain so because the processing rates are constant, GroupShare keeps these groups intact, and no new query is added to G_k.

Furthermore, G_k does not starve; it is eventually scheduled after a finite amount of time. This is guaranteed by stride scheduling. Stride scheduling chooses at each step the group G_j that contains the query with the minimum pass and increments the pass of each query q_i in G_j by $\frac{t_{G_j}}{t_{q_i}}$. Thus, as all query passes are greater or equal to the pass of G_j's query with the minimum pass. The maximum difference between the minimum passes of queries in other groups and the minimum pass of G_j at the end of the quantum is $max_{q_i \in G_j} \frac{t_{G_j}}{t_{q_i}} = \frac{t_{G_j}}{min_{q_i \in G_j} t_{q_i}}$.

Group G_j will again be scheduled as soon as it contains, once again, the query with the minimum pass. At each subsequent step, one of the other $m - 1$ groups, $G_{j'}$ is scheduled, and the passes of all their queries will be incremented by at least

$$min_{q_i \in G_{j'}} \frac{t_{G_{j'}}}{t_{q_i}} \geq min_{l \neq j} (min_{q_i \in G_l} \frac{t_{G_l}}{t_{q_i}}) = min_{l \neq j} \frac{t_{G_l}}{max_{q_i \in G_l} t_{q_i}}$$

Thus, after scheduling group $G_{j'}$ for at most

$$n = \left\lceil \frac{\frac{t_{G_j}}{min_{q_i \in G_j} t_{q_i}}}{min_{l \neq j} \frac{t_{G_l}}{max_{q_i \in G_l} t_{q_i}}} \right\rceil$$

quanta, all of its queries will have a higher pass than the minimum pass of G_j. We note that this finite amount of time is

$$n \leq \left\lceil \frac{max^2_{q_i \in Q} t_{q_i}}{min^2_{q_i \in Q} t_{q_i}} \right\rceil$$

because

$$\frac{t_{G_j}}{min_{q_i \in G_j} t_{q_i}} \leq \frac{max_{q_i \in Q} t_{q_i}}{min_{q_i \in Q} t_{q_i}}$$

and

$$min_{l \neq j} \frac{t_{G_l}}{max_{q_i \in G_l} t_{q_i}} \geq \frac{min_{q_i \in Q} t_{q_i}}{max_{q_i \in Q} t_{q_i}}$$

After at most $n \times (|Q| - 1)$ scheduling steps (to account for possible reconfigurations in groups $G_{k+1}, \ldots, G_m$), G_k will have the query with minimum pass across all groups. Thus, after a finite amount of time since $G_1, \ldots, G_{k-1}$ became sharing groups, G_k is scheduled. GroupShare identifies it as a non-sharing group and moves its minimum-pass query to group G_{k+1}. After the minimum-pass query has been moved, G_k has p queries and thus will become a sharing group after being scheduled p more times (p+1 in total). □

Theorem 3.1. *GroupShare converges to a set of sharing groups for a set of queries Q after a finite amount of time.*

Proof. We prove the theorem with induction over the number of queries in Q.

For $|Q| = 1$, Q is trivially one sharing group, so GroupShare converges immediately.

Let's assume that GroupShare converges for a set of queries with $|Q| \leq p$. Then, it also converges when $Q = p + 1$.

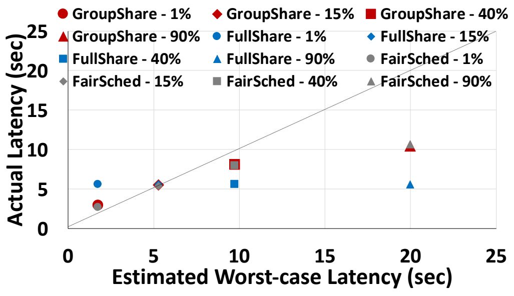
Using the previous lemma, we know that GroupShare produces a sharing group G_1 after a finite amount of time. From that point on, G_1 remains intact, and GroupShare only needs to turn the remaining $p + 1 - |G_1| \leq p$ queries into sharing groups. Thus, GroupShare converges. □

GroupShare iteratively evaluates the formed query groups until it reaches a convergence point that upholds fairness guarantees. Heuristics could be used to accelerate the convergence rate further if needed. For example, an offline pre-processing step can be introduced to partition queries into coarse groups based on the overlap of the queries and the data distribution.

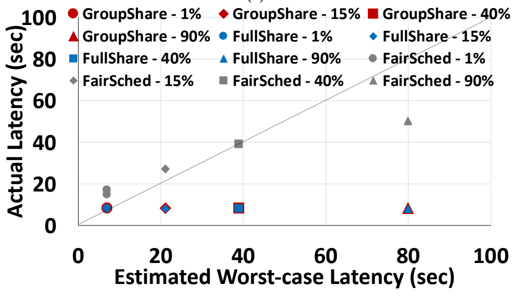
3.3 Implementation Details

We implement GroupShare on top of RouLette [35], an in-memory analytical query engine that shares work by incrementally optimizing the shared plan at runtime. Roulette's efficiency has been shown on Join Order Benchmark queries over real-world IMDB data, as well as query batches with varying complexity. RouLette changes query plans between data vectors, and thus, it is a natural fit for GroupShare. We modify RouLette as follows. First, we implement multiple cursors on scans, which enable different groups to progress independently. Second, we replace RouLette's simple scheduling logic, which assigns a vector to a worker, with GroupShare's algorithm.

In cases where GroupShare converges to a single group containing all the queries, the performance of GroupShare would be the same as the performance of Roulette. By contrast, in cases where



(a)



(b)

Figure 4: Comparison between each query's achieved and expected latency (latency in isolation × concurrency) when running a) 4 queries, b) 16 queries.

GroupShare cannot use a single group for all queries because that would violate the isolation guarantees, the total execution time of GroupShare can be higher than that of Roulette, but the execution time of all participating queries will be at least the same compared to isolated execution. This behavior is illustrated in Figures 4 and 5.

We tune the system as follows: we set the probability for the normal phase to 0.95 and the parameter a to 0.9, and use 1024-row vectors and 2^{16}-row chunks.

3.4 Applicability and Limitations

GroupShare is designed for the in-memory scale-up setup discussed in Section 2. Also, it assumes that the underlying system can switch between plans at runtime. This is possible for systems using adaptive processing but is not a common capability across all databases.

GroupShare is meant as an initial effort to achieve functional isolation in shared execution. It uses a simplified model for in-memory processing and thus does not cover excessive demand for memory, ad-hoc query arrivals, and caching optimizations. Furthermore, it assumes that data distribution is the same across the table and may rule out groupings that become beneficial in later chunks. We aim to study these challenges in subsequent work.

4 EXPERIMENTAL EVALUATION

The experiments presented in this section show that: (i) Eagerly sharing execution for all queries penalizes short-running queries and benefits long-running queries. (ii) A group of queries can provide fairness using sharing if the queries have the appropriate characteristics (e.g. correlation between queries). For high concurrency, the full batch of queries is one such group. (iii) GroupShare exploits

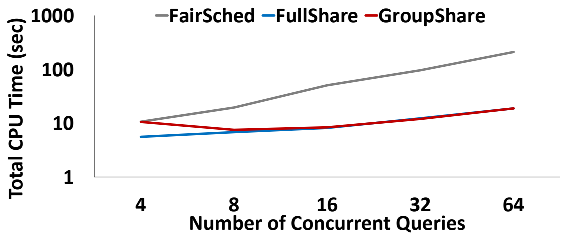


Figure 5: Comparison of total processing time as a function of the number of concurrent queries.

sharing as group size is increased and achieves both fairness and lower total processing time in all experiments.

Hardware All experiments took place on a server that has an Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz with 2 sockets, 12(×2) threads per socket, and 376GB. The experiments are in-memory, in a single NUMA node, and used 12 threads.

Data and Workload We generate synthetic data and queries. We use a fact table with 8 integer columns (1 for filter, 6 for join keys, and 1 for aggregation) and $256M$ rows, and, for joins, 6 dimensions with $100k$ rows each. The range of join keys is $[0, 100k)$ and the range of filters is $[0, 100)$. All queries contain a filter on the fact table and 6 joins and only differ in their selectivity.

Execution and Scheduling Strategies We evaluate three different approaches: (i) **FairSched**, a fair scheduling approach that uses stride scheduling; (ii) **FullShare**, which uses one shared query plan for all queries utilizing all CPU time; and (iii) **GroupShare**, the algorithm we introduce in this work.

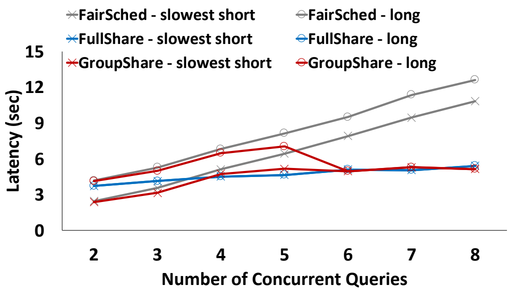
4.1 Sharing-Fairness Trade-off

First, we demonstrate the sharing-fairness trade-off and show that GroupShare reduces total processing time without penalizing any query. We use 4 query classes with different selectivity (1%, 15%, 40%, and 90%). The workload contains an equal number of queries from each class. We submit all queries at once, as a batch.

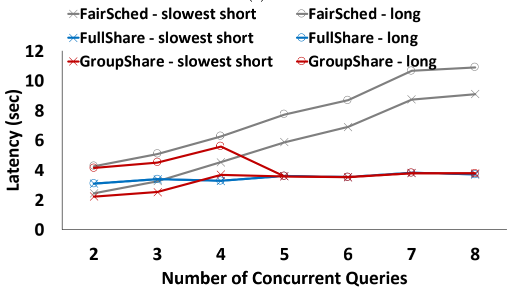
Figure 4 compares the actual latency for each query to the expected latency for fair scheduling, which we estimate as *latency in isolation* × *concurrency*. The estimate is pessimistic for long-running queries because as short-running queries finish, the former receives more CPU time. A query's CPU allocation requirement is respected when the respective data point is below the $y = x$ line. Figures 4(a) and (b) show runs with 4 and 16 queries, respectively. The experiments represent the two extremes: in (a) FullShare hinders fairness, whereas in (b) FullShare reduces latency for all queries compared to FairSched.

In Figure 4(a), GroupShare converges to FairSched. With FullShare, the shortest and longest-running query share work despite having an 11.5 : 1 processing time ratio, thus, the shortest-running's latency is increased by 2×. Note that FairSched also slightly penalizes latency. This is an implementation artifact: we use the same shared filters for all execution strategies, and thus filtering cost increases logarithmically with concurrency.

In Figure 4(b), GroupShare uses sharing for all queries. The concurrency is high enough that sharing reduces latency for all queries, by 1.7 – 6×, compared to FairSched. Thus, sharing overcompensates the isolation guarantees of the queries, even short-running ones.



(a)



(b)

Figure 6: Latency comparison for each execution strategy when queries have: (a) uncorrelated and (b) correlated filters.

Figure 5 plots the total time for processing all queries with each strategy. This is the workload's CPU consumption. We run up to 64 concurrent queries. FullShare decreases both the total processing time and the rate of increase. GroupShare starts with performance equal to FairSched, to respect isolation guarantees, but switches to sharing in one group as soon as there are enough queries to both achieve isolation guarantees and minimize processing time.

Takeaway: GroupShare respects isolation guarantees, whereas FullShare may violate them. Also, by judiciously using sharing, GroupShare achieves both lower latency for all queries and lower total processing time compared to FairSched.

4.2 Intra-group Sharing

Next, we show the impact of query characteristics on grouping. We use 2 query classes with the same template and 10% and 50% selectivity, respectively. The workload contains one query from the 50% class, whereas the rest of the queries are from the 10% class.

Figure 6 shows the latency for each query class as concurrency increases. It plots the latency of the long-running and the slowest short-running query for each strategy. The long-running query's latency is also the end-to-end CPU time. In Figure 6(b), filters cover half the filter attribute's domain, and, in Figure 6(a), the full domain.

GroupShare converges to FairSched for low and to FullShare for high concurrency. In between, it shares work between short-running queries and runs the long-running one separately. It reduces the long-running query's latency as well because, by reducing processing time for short-running queries, sharing leaves more CPU time for the long-running one. In Figure 6(a), GroupShare achieves up to 2.1× lower latency for the short-running and 2.45× for the long-running query compared to FairSched.

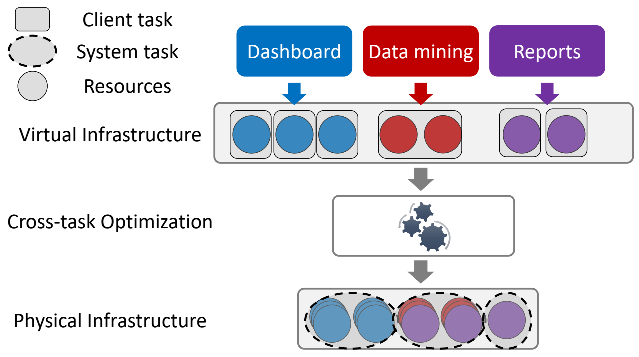


Figure 7: Functional isolation can be supported through virtualization. Rectangles show data-intensive logical tasks submitted by the user. Ovals depict system tasks that process multiple client tasks at once using cross-task optimizations. Circles show allocated resources for the task that encapsulates them. The color of the circles represents the application that the task serves, and for the case of system tasks, the number and color of circles show the logical tasks that the internal task processes.

Figure 6(b) shows that intra-group sharing is more effective when queries are correlated: the decrease in latency is higher – namely up to 2.4× for short-running and 2.9× for long-running queries. Also, sharing across all queries becomes beneficial for lower concurrency. This result motivates that grouping needs to be workload-driven based on the characteristics of the queries involved.

Takeaway: GroupShare uses sharing within smaller groups where isolation guarantees can be satisfied. This improves latency even for non-shared queries as it allows allocating more CPU. Overall, appropriate grouping depends on the queries' properties.

5 ADOPTING FUNCTIONAL ISOLATION

Native support for functional isolation requires each system component that is involved in cross-task optimizations to be aware of the guarantees they are expected to provide. Components should be able to decide how and when to cross-optimize tasks while respecting their requirements.

An architectural approach that would be easier to adopt is to support functional isolation through virtualization, as depicted in Figure 7. An intermediate layer receives client tasks and produces a set of internal tasks that the system will process. Each system task performs work for a group of client tasks or subtasks and is formed such that it leverages cross-optimization using native primitives of the system, e.g., writing shared join queries using native SQL operations [30] and materialized views. The mapping of client tasks to system tasks should be done by a cross-optimizer in the virtualization layer. This way, the internal system components do not need to be aware of the task requirements. In some applications, different types of cross-task optimization techniques (e.g., jointly examining reuse and work sharing [36]) interfere with one another, and thus examining them holistically when choosing sharing groups can lead to higher performance per query. However, the mapping of client-to-system tasks may vary across system components. For instance, two queries may belong to the same system task for memory management, benefiting from shared caches, but to different ones for the scheduling component, allowing independent scheduling.

Resource managers could be incorporated into this architecture to enable the scheduling of heterogeneous applications with potential co-optimization opportunities. However, resource managers cannot discover the opportunities themselves. This can only be done by the backend data processing framework, which may be shared between one or more applications. The virtualization layer would be responsible for translating the client tasks to the allocations that must be accommodated by the resource manager.

In both cases of native support and virtualization, it is necessary to support performance isolation in the first place. Additionally, if incorrect optimizer decisions are possible, the optimizer should be able to predict requirement violations (e.g., via a feedback loop) and adapt the mapping of client to system tasks or fall back to isolated execution, while the system must be able to adapt at runtime to accommodate the re-optimizations. Techniques for runtime adaptation have already been studied in the context of adaptive query processing [17].

Functional isolation requires no extra client-related considerations. Widely used cloud providers [4, 6, 8] typically establish their pricing models around SLAs that quantify the amount of resources to be reserved. Functional isolation seamlessly aligns with such resource SLAs but also with performance SLAs. An oligolithic system can get the SLA as input, translate it into resource isolation requirements, leverage cross-task optimization, and find an execution schedule so that the SLAs are met with minimum cost. The only possible user-facing difference is exposing as cost savings the resource savings due to under-the-hood cross-optimization.

6 CONCLUSION

Existing systems guarantee performance isolation by enforcing resource barriers between concurrent tasks, but doing so, limits the optimization space. We envision cost- and resource-efficient systems that exploit cross-query optimizations while satisfying each task's requirements. We showcase the feasibility of our vision for the use case of work sharing: we propose GroupShare, an algorithm that partitions queries into groups that can share sub-expressions and reduces total processing while meeting isolation guarantees. This is a first step towards a paradigm for cross-optimizing diverse co-existing applications in order to meet mission-critical per-application requirements without painful tuning and at less cost.

The functional isolation principle is not limited to either the use case of work sharing or the discussed solution. It can cover use cases with a variety of optimizations, resources, and optimization goals. For example, we can conceive a use case where multiple applications run, each with their dedicated memory for caches and auxiliary structures, to meet their respective stringent latency requirements. Then, sharing caches and auxiliary structures to eliminate redundancy and save memory while meeting performance requirements is a case of functional isolation. Similarly, in a different use case where applications use techniques such as approximate query processing and specify accuracy requirements, the reclaimed memory can be used to boost accuracy rather than improve latency.

REFERENCES

[1] [n. d.]. Apache Doris - User Stories. https://doris.apache.org/users/ Accessed: 2023-11-07.

[2] 2022. IBM DB2. https://www.ibm.com/docs/en/db2-for-zos/11?topic=threads-how-db2-allocates. Accessed: 2023-11-07.

[3] 2023. AWS - Concurrent Scaling. https://docs.aws.amazon.com/redshift/latest/dg/concurrency-scaling.html Accessed: 2023-11-07.

[4] 2023. AWS Pricing Calculator. https://calculator.aws/#/ Accessed: 2023-11-30.

[5] 2023. Databricks Customer Requests. https://community.databricks.com/t5/warehousing-analytics/sql-warehouse-high-number-of-concurrent-queries/td-p/7140 Accessed: 2023-11-07.

[6] 2023. Google Cloud SQL Pricing. https://cloud.google.com/sql/pricing Accessed: 2023-11-30.

[7] 2023. Microsoft SQL Server. https://learn.microsoft.com/en-us/sql/relational-databases/thread-and-task-architecture-guide?view=sql-server-ver16. Accessed: 2023-11-30.

[8] 2023. Oracle Cloud Infrastructure Pricing. https://www.oracle.com/cloud/pricing/ Accessed: 2023-11-30.

[9] 2023. PostgreSQL - Resource Consumption. https://www.postgresql.org/docs/current/runtime-config-resource.html#RUNTIME-CONFIG-RESOURCE Accessed: 2023-11-30.

[10] 2023. PostgreSQL - Resource Consumption. (2023). https://ourworldindata.org/grapher/historical-cost-of-computer-memory-and-storage Accessed: 2023-11-30.

[11] 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 *SIGMOD*.

[12] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade. *Queue* 14, 1 (jan 2016), 70–93. https://doi.org/10.1145/2898442.2898444

[13] George Candea, Neoklis Polyzotis, and Radek Vingralek. 2009. A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses. *Proceeding of the VLDB Endowment* 2, 1 (2009).

[14] George Candea, Neoklis Polyzotis, and Radek Vingralek. 2011. Predictable performance and high query concurrency for data analytics. *VLDB J.* 20 (04 2011), 227–248. https://doi.org/10.1007/s00778-011-0221-2

[15] E. F. Codd. 1960. Multiprogram Scheduling: Parts 1 and 2. Introduction and Theory. *Commun. ACM* 3, 6.

[16] Fernando J. Corbató, Marjorie Merwin-Daggett, and Robert C. Daley. 1962. An Experimental Time-Sharing System. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*.

[17] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. 2007. Adaptive Query Processing. *Foundations and Trends® in Databases* 1, 1 (2007), 1–140.

[18] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database–An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.

[19] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2012. SharedDB: Killing One Thousand Queries with One Stone. *Proceedings of the VLDB Endowment.*

[20] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, and Donald Kossmann. 2014. Shared Workload Optimization. *Proceedings of the VLDB Endowment.*

[21] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. 2005. QPipe: A Simultaneously Pipelined Relational Query Engine. In *SIGMOD*.

[22] John L. Hennessy and David A. Patterson. 2012. *Computer Architecture - A Quantitative Approach* (5 ed.). Morgan Kaufmann.

[23] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting Subexpressions to Materialize at Datacenter Scale. *Proceedings of the VLDB Endowment.*

[24] Alekh Jindal, Lalitha Viswanathan, and Konstantinos Karanasos. 2019. Query and Resource Optimizations: A Case for Breaking the Wall in Big Data Systems. https://doi.org/10.48550/ARXIV.1906.06590

[25] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. 2015. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. 485–497.

[26] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2019. AJoin: Ad-Hoc Stream Joins at Scale. *Proceedings of the VLDB Endowment* 13, 4.

[27] Sailesh Krishnamurthy, Michael J. Franklin, Joseph M. Hellerstein, and Garrett Jacobson. 2004. The Case for Precision Sharing. *Proceedings of the VLDB Endowment.*

[28] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. 2002. Continuously Adaptive Continuous Queries over Streams. In *SIGMOD*.

[29] Darko Makreshanski, Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2016. MQJoin: Efficient Shared Execution of Main-memory Joins. *Proceedings of the VLDB Endowment.*

[30] Renato Marroquín, Ingo Müller, Darko Makreshanski, and Gustavo Alonso. 2018. Pay One, Get Hundreds for Free: Reducing Cloud Costs through Shared Query Execution. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. 439–450.

[31] Lin Qiao, Vijayshankar Raman, Frederick Reiss, Peter J. Haas, and Guy M. Lohman. 2008. Main-Memory Scan Sharing for Multi-Core CPUs. *Proceeding of the VLDB Endowment* 1, 1 (2008).

[32] Prasan Roy, Srinivasan Seshadri, S Sudarshan, and Siddhesh Bhobe. 2000. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*.

[33] Timos K Sellis. 1988. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*.

[34] Kyuseok Shim, Timos Sellis, and Dana Nau. 1994. Improvements on a Heuristic Algorithm for Multiple-Query Optimization. *Data Knowl. Eng.* 12, 2.

[35] Panagiotis Sioulas and Anastasia Ailamaki. 2021. Scalable Multi-Query Execution Using Reinforcement Learning. In *SIGMOD*. ACM, New York, NY, USA.

[36] Panagiotis Sioulas, Ioannis Mytilinis, and Anastasia Ailamaki. 2023. Real-Time Analytics by Coordinating Reuse and Work Sharing. arXiv:2307.08018 [cs.DB]

[37] I. Stoica, H. Abdel-wahab, Kevin Jeffay, S.K. Baruah, J.E. Gehrke, and C.G. Plaxton. 1997. A Proportional Share Resource Allocation Algorithm for Real-time Time-shared Systems.

[38] Dixin Tang, Zechao Shang, William W. Ma, Aaron J. Elmore, and Sanjay Krishnan. 2021. Resource-Efficient Shared Query Execution via Exploiting Time Slackness. In *SIGMOD*.

[39] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing, SoCC 2013*. Association for Computing Machinery (ACM), United States.

[40] Benjamin Wagner, André Kohn, and Thomas Neumann. 2021. Self-Tuning Query Scheduling for Analytical Workloads. In *SIGMOD*.

[41] Carl Waldspurger. 1996. Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management. (1996).

[42] C. A. Waldspurger and E. Weihl. W. 1995. *Stride Scheduling: Deterministic Proportional- Share Resource Management.* Technical Report. USA.

[43] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. 2007. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. VLDB Endowment.