Resource-efficient Shared Query Execution via Exploiting Time Slackness

Dixin Tang*
UC Berkeley
totemtang@berkeley.edu

Zechao Shang*
Snowflake Computing
zechao.shang@snowflake.com

William W. Ma
University of Chicago
williamma@uchicago.edu

Aaron J. Elmore
University of Chicago
aelmore@cs.uchicago.edu

Sanjay Krishnan
University of Chicago
skr@uchicago.edu

ABSTRACT

Shared query execution can reduce resource consumption by sharing common sub-expressions across concurrent queries. We show that this is not always the case when regularly querying a dataset under change. Depending on latency goals, how eagerly to incrementally process the new data differs. Naively sharing the execution of queries with different latency goals will push the whole shared plan to meet the lowest latency goal and execute more eagerly than each participating query. The overhead introduced by the eager execution can even offset the benefit of shared query execution. We propose an optimization framework iShare to exploit the benefit of shared execution and avoid the overhead of eager execution. iShare judiciously shares queries with different latency goals and selectively executes parts of the share plan lazily. iShare can significantly reduce resource consumption compared to eagerly executing share plans from the state-of-the-art multi-query optimizer or approaches that execute queries separately.

CCS CONCEPTS

• **Information systems** → **Database query processing**; **Query optimization**; **Query planning**.

KEYWORDS

shared query execution; multi-query optimization; incremental view maintenance; scheduled queries; cloud database

ACM Reference Format:
Dixin Tang, Zechao Shang, William W. Ma, Aaron J. Elmore, and Sanjay Krishnan. 2021. Resource-efficient Shared Query Execution via Exploiting Time Slackness. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457282

1 INTRODUCTION

Scheduled queries are prevalent in today's database applications [3, 5], such as executing regular ETL jobs or maintaining recurring dashboard reports over a dynamic dataset (e.g. continuously loaded data) [50]. Here, it is common that many queries are scheduled over the same set of data [4, 26, 50] (e.g. analyzing the daily loaded data at 6 am), which provides opportunities for exploiting common sub-expressions of the scheduled queries to reduce redundant query work (e.g. sharing the scan over the same dataset) and, consequently, save resources (e.g. CPU cycles). Reducing resource consumption is increasingly critical to today's database designs due to the fast growth of data and the wide adoption of pay-per-use models in the cloud. For example, Google's Cloud Scheduler allows users to schedule queries [3] and charges each query by the resources it consumes [2] (e.g. the computing resources are estimated as the number of milliseconds it takes to finish a query).

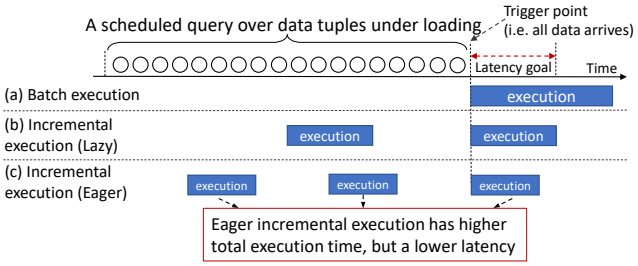


Figure 1: The trade-off between resource consumption and query latency for lazy and eager incremental execution

A common practice of exploiting the opportunities of shared query work is adopting shared query execution [16, 22, 33] or multi-query optimization (MQO) [17, 24, 40]. However, we find that shared query execution is not always beneficial when the scheduled queries have different deadlines of returning the results [4, 26] (e.g. some daily reports are due at 7 am and some others are due at 10 am). In this paper, we use *latency goals* to represent this deadline. It is defined as the maximally allowed time a query takes to return the result after the data for the query is complete.

Consider an example of a scheduled query in Figure 1. If we use batch execution, which starts the query when all data is ready (i.e. trigger point in Figure 1), the query misses its latency goal. To meet its latency goal, we can start processing data early and incrementally maintain the query result (i.e. *incremental execution* in Figure 1). While incremental execution reduces the query latency, it may increase the total execution time and CPU consumption for certain queries [23, 44, 50]. This is because tuples that are output in earlier executions can be removed by later executions. Consider an aggregate operator that maintains the number of orders each customer has placed over a stream of newly inserted orders. The

*The work is mainly done when the two authors were at the University of Chicago.

SIGMOD '21, June 20–25, 2021, Virtual Event, China.

ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3457282

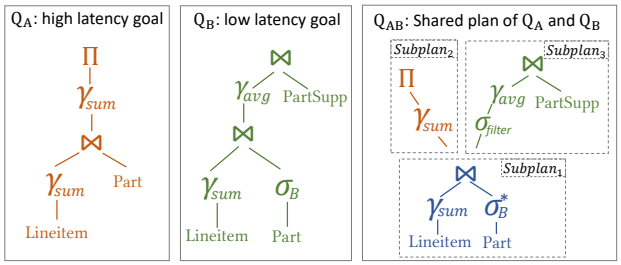


Figure 2: Example query plans w/(o) MQO

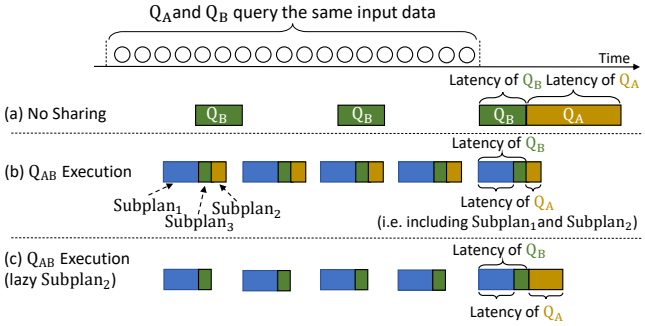


Figure 3: Query execution for no sharing and the shared plan

initial execution of this operator computes the aggregated results and outputs them to its parent operator. The later executions for newly inserted orders can update the aggregated results for some customers. Therefore, this aggregate operator needs to remove prior output tuples (i.e. via delete operations) and then insert the updated aggregated results. To control the query latency, we can adjust the frequency of executing the query. Figure 1 shows that if we execute a query more *eagerly*, which means that we start each execution for smaller amount of data, compared to *lazy* execution, we can reduce the query latency. However, this increases the computing resources necessary because more tuples outputted from earlier executions are be removed in later executions.

In this context, when multiple queries have different latency goals, the shared execution may not be beneficial and consumes more resources compared to executing queries separately. The main reason is that the shared query plan needs to execute more eagerly to meet the lowest latency goal. The extra resource consumption introduced by eager incremental execution may offset the benefit of sharing. While recent research [27, 30, 40] judiciously decides the parts of queries to be shared, an essential goal of MQO is minimizing overall resource consumption. No existing MQO approach considers the overhead of different latency goals and the consequential eager execution on shared parts.

We illustrate the problem with an example. Consider query Q_A and Q_B in Figure 2. An MQO optimizer generates a plan Q_{AB} that shares two almost identical joins (with the difference of σ_B). Note that the new σ_B^* in Q_{AB} only marks tuples that belong to Q_B, and does not drop any tuples, which are all needed by Q_A. When $Subplan_3$ pulls data from $Subplan_1$, it filters out the tuples that do not belong to Q_B (i.e. using σ_{filter}). Consider the case that Q_A has a high latency goal and Q_B has a low latency goal. If we execute the two queries separately, Q_A is executed lazily and Q_B is executed eagerly as shown in Figure 3(a). However, if we choose the shared plan Q_{AB}, it needs to meet the lower goal (i.e. Q_B's) and executes the whole plan eagerly (i.e. Figure 3(b)). In this case, Q_{AB} has higher total execution time and CPU consumption than executing the two queries separately. Here, the total execution time is the summation of the elapsing time of all incremental executions, and the latency of a query in a shared plan is defined as the summation of its subplans' final execution time. For example, Figure 3(b) shows that the latency of Q_A is the sum of the final execution time of $Subplan_1$ and $Subplan_2$.

This shared plan has two cases of overhead due to overly eager execution. First, $Subplan_1$ is executed eagerly to meet Q_B's latency goal and can consume more computing resources compared to not sharing $Subplan_1$. Assume the selectivity of σ_B is 1%. Without sharing, all data can be executed lazily for Q_A (e.g. using one batch) and only 1% data is executed eagerly for Q_B. In the shared $Subplan_1$, all data is executed eagerly to meet Q_B's latency goal. As discussed above, eager incremental execution for an aggregate operator (i.e. γ_{sum} in $Subplan_1$) repeatedly removes prior output tuples and inserts new tuples when the aggregated results are updated. Therefore, eagerly processing all data for $Subplan_1$ can consume more computing resources compared to executing this subplan separately for the two queries. Second, $Subplan_2$ has overly eager execution because Q_{AB} is executed eagerly to meet the goal of Q_B, but Q_A has a high latency goal and $Subplan_2$ could have executed lazily. This example optimization is shown in Figure 3(c), where we delay the execution of $Subplan_2$ to when all data is complete.

We propose *iShare* for scheduled queries. iShare is more favorable than existing solutions when the scheduled queries have different latency goals and partially overlap on shared subplans. Instead of using a single frequency for a shared plan, iShare selectively untangles a shared (sub)plan in two ways: 1) executing different subplans in different frequencies by considering the latency goals; 2) breaking the shared subplans into separate ones based on the latency goals (i.e. unshare) and run them at different frequencies.

However, such optimization is time-consuming due to the complex search space in finding the execution frequency for each subplan [44] and the possible ways to decompose the shared plan. We propose several techniques to address this challenge and make the following contributions:

- First, we extend the metric of incrementability [44] to quantify the cost-effectiveness of incremental execution for a subplan, and design an optimized greedy algorithm to quickly find the execution frequencies.
- Second, we propose a heuristic metric, *sharing benefit*, which can estimate whether it is worthwhile to share a subplan for two sets of queries, and a greedy algorithm to decompose a shared subplan based on *sharing benefit*.
- Third, we design a algorithm to quickly compute the execution frequencies for the decomposed subplan without computing the execution frequencies for the whole plan from scratch.
- Finally, we perform extensive experiments to show that iShare has low optimization overhead and can significantly reduce CPU consumption compared to executing share plans (from the state-of-the-art MQO optimizer) in a single frequency and two approaches that execute queries separately.

2 PROBLEM STATEMENT AND OVERVIEW

In this section, we discuss the context and definition of our optimization problem, present the definitions used in iShare, and the underlying shared query execution engine.

2.1 Problem context and definition

We consider a scenario where a stream of tuples is being loaded into the database, and users want to analyze this data stream via scheduled queries. The queries are scheduled based on pre-defined events (e.g., time/count-based). We name this pre-defined event trigger condition. We focus on optimizing the scheduled queries with the same trigger conditions (e.g., daily loaded data). We assume knowledge of the data arrival rate (i.e., number of new tuples per hour for each base relation). Historical statistics [42] can estimate this information. We use this information to estimate the cost of query execution and query latency. For simplicity, we assume a fixed data arrival rate in this paper.

As in prior work [44], we use the **total work** as a proxy for the total execution time of all queries (i.e. CPU consumption), and the **final work** as a proxy for the latency of each query. *Both total work and final work are quantified based on the DBMS's cost model.* For example, it could be the estimated cost of CPU cycles or the number of tuples processed by all operators. As shown in Figure 3, the total execution time is the summation of the elapsing time of all incremental executions. Therefore, total work estimates the total units of work done by all incremental executions for all queries. The latency shown in Figure 3 is the remaining time a query takes to return the result after the trigger point. Therefore, we define the final work as the remaining units of work to be done for each query after the trigger point. Recall that the latency of a query in a shared plan is the summation of its subplans' final execution time. For example, the latency of Q_A in Figure 3(b) involves the final execution time of $Subplan_1$ and $Subplan_2$. Similarly, the final work of a query sums the units of work of its subplans' final execution.

In iShare, users additionally submit a *final work constraint* for each query as a proxy for the latency goal. The final work constraint of a query can be specified as an absolute number of units of work based on a cost model (i.e. *absolute final work constraint*) or a relative value defined as the ratio between the final work users want to achieve and the final work of separately executing the query in one batch (i.e. relative final work constraint). For example, a relative constraint of 0.1 means that users want to reduce the final work to 10% of the final work of executing the query in one batch. This constraint allows users to make a trade-off between CPU consumption and query latency. To achieve a desired latency for a recurring query, users can adjust the final work constraint based on this query's prior executions. Therefore, our optimization problem is given a set of scheduled queries with the same trigger conditions, how to find a query plan to minimize the total work of all queries while meeting each query's final work constraint.

2.2 Definitions and Optimization Overview

We find directly using the shared plan from existing MQO optimizers has high total work because the shared plan is executed in a single frequency. Therefore, our key idea is to break the shared plan into subplans and execute each subplan using a separate frequnecy.

Subplan A subplan in iShare represents a subtree of operators that are shared by the same set of queries. We break the shared plan into subplans at the operators that have more than one parent operator. Consider the shared plan in Figure 2. iShare breaks it into three subplans, where all shared operators belong to a subplan (i.e., $SubPlan_1$) and the unique plans for Q_A and Q_B are two separate subplans. When the root operator of one subplan has two or more parent operators, it materializes its output into a buffer such that the parent subplans can consume the intermediate results at individual frequencies [40]. Similarly, we treat all base relations or delta logs as buffers as well. Therefore, each incremental execution of one subplan processes all new data from the buffers of its child subplans or base tables. Then, it materializes the result tuples into this subplan's buffer or outputs them as the query results. We note that there are multiple parent subplans consuming the same child subplan's buffer. Therefore, each parent subplan will track the offsets of the tuples it has processed. We assume a shared query execution engine that requires the query set of a subplan subsume the query set of its parent subplans (e.g. the query set $\{Q_A, Q_B\}$ of $Subplan_1$ in Figure 2 contains the query set $\{Q_A\}$ of $Subplan_2$) We note that it is possible to break the shared plan in a more fine-grained way [44], which comes with a higher optimization cost. We use subplans as the granularity of control as it significantly reduces the optimization time, which we show in Section 3.2.

Pace The execution frequency of a subplan can be defined as time-based (every 5s), count-based (every 1000 tuples), or heuristic-based. For simplicity, we borrow the concept *pace* [44] to represent the execution frequency of a subplan. A pace is defined on the finite amount of data for a trigger condition (e.g. daily loaded data). A pace k means that the subplan starts one execution whenever the system has received $\frac{1}{k}$ of the total estimated tuples for that trigger condition. The higher the pace is, the more eagerly we execute the subplan. For example, Figure 3(b) shows that the pace for all three subplans in Q_{AB} is 5 (i.e. 5 executions). In Figure 3(c), we see that the pace for $Subplan_2$ is 1 since we execute this subplan one time when all data arrives (i.e. batch execution for $Subplan_2$). A *pace configuration* represents the set of paces $P = (p_1, p_2, \ldots, p_M)$ for all M subplans. The pace configuration $P_{\mathbb{1}} = (1, 1, \ldots, 1)$ represents the batch execution for all subplans.

Optimization overview In iShare, we take a shared plan generated by an existing MQO optimizer [17] as input and adopt the following two techniques to reduce its total work. First, we use a greedy algorithm to find a pace configuration to minimize the total work and also meet the final work constraints, which is discussed in Section 3. Based on the shared plan annotated with paces, we consider decomposing each shared subplan into multiple separate subplans. This way, we can execute different subplans at different paces to further reduce the total work. We discuss the subplan decomposition in Section 4. We choose to optimize the plan generated by an MQO optimizer because we want our optimization techniques to be independent of any existing MQO optimizers . In addition, this approach significantly reduces the optimization space and overhead. We test a method of enumerating all query plans using a MQO optimizer [17] and finding the pace configuration holistically for each plan. We find that its optimization time is up to 4.6 hours for the TPC-H benchmark, and its generated plan has

similar CPU consumption and query latencies compared to iShare. We omit this experiment due to space limits.

2.3 Query execution

iShare combines the ideas of SharedDB [16] and prior work in incremental view maintenance [13] to support shared incremental execution of scan, select, project, aggregate, and inner join operators with respect to insert, delete, and update operations. Two physical subplans are considered sharable if they have exactly the same structure and operators, with the exception of allowing their select and project operators to be different. To quickly identify the sharable plans, we encode each subplan using a string signature [17, 40]. Two subplans are sharable if their string signatures are exactly the same. Merging two different project operators unions their projection expressions to generate a new project operator. If two select operators are different, they are not merged but directly copied from the original subplans. The key idea for enabling the shared execution of the subplan is to annotate each intermediate tuple with a bitvector $B = (b_1, b_2, \ldots, b_n)$, where one bit indicates whether this tuple is valid for a query [16], and each operator is also associated with a bitvector where one bit is set if a query shares this operator. To support delete operations, each intermediate tuple is additionally associated with a bit that indicates the insertion or deletion [13]. An update operation is implemented as a delete plus an insert. The physical implementation of shared query execution for our supported operators is discussed in prior work [16].

3 FINDING THE PACE CONFIGURATION

iShare allows each subplan to have a different pace to reduce the total work with respect to the final work constraints of participating queries. The system is allowed to lazily execute subplans in the queries that have higher final work constraints. Therefore, we extend prior work [44] to reduce the total work by considering the different final work constraints and the structure of shared plans.

Specifically, we redefine the metric *incrementability* [44]. Incrementability quantifies the cost-effectiveness of incremental executions and is a key metric for efficient incremental execution for a single query. We redefine this metric for shared query execution, and propose an optimized algorithm with a low running time to find a nonuniform pace configuration using incrementability.

3.1 Incrementability definition in iShare

If we execute a query more eagerly, we have a lower query latency but a higher resource consumption. The intuition of incrementability is to quantify how much query latency we can reduce given the same additional resources we invest. The original paper [44] defines incrementability as the ratio between the reduced final work and the increased total work.

For iShare's incrementability, we define the benefit of decreased final work differently by considering the final work constraints of different queries. Intuitively, if a pace configuration has already met the final work constraints of some queries, further increasing the paces for those queries' subplans does not yield additional benefit. Therefore, the benefit of a query here should be the reduced missed final work with respect to its final work constraint rather than the absolute reduction. With this observation, we now define the benefit for N queries $Q = (q_1, q_2, \ldots, q_N)$ between two pace configurations P_A and P_B, where P_A should be eagerer than P_B. This means that any pace in P_A is no smaller than the corresponding pace in P_B and there is at least one subplan's pace in P_A larger than the one in P_B. The formula of the benefit between the two is:

$$\texttt{Benefit}(P_A, P_B) = \sum_{\forall q_i \in Q} max(0, C_F(P_B, q_i) - C'_F(P_A, q_i)) \quad (1)$$

where $C'_F(P, q_i) = max(L(q_i), C_F(P, q_i))$.

Here, $L(q_i)$ represents a query's final work constraint and $C_F(P, q_i)$ means the final work of a query given a pace configuration P. Therefore, $C'_F(P, q_i)$ represents the bounded final work that is no lower than the constraint and $max(0, C_F(P_B, q_i) - C'_F(P_A, q_i))$ is the benefit of reducing the missed final work with respect to query q_i's constraint. Finally, Equation 1 sums the per-query benefit to compute the overall benefit.

The overhead of the eager execution from P_B to P_A is $C_T(P_A) - C_T(P_B)$, where $C_T(\cdot)$ represents the total work of a pace configuration. The incrementability definition for iShare is:

$$\texttt{InC}(P_A, P_B) = \frac{\texttt{Benefit}(P_A, P_B)}{C_T(P_A) - C_T(P_B)} \quad (2)$$

3.2 Pace configuration via incrementability

In this subsection, we first discuss the algorithm of estimating incrementability and then we present the algorithm of leveraging incrementability to find the pace configuration. Finding the pace configuration essentially uses incrementability to prune the search space of different pace configurations. Therefore, the cost of estimating incrementability is the bottleneck of finding the pace configuration. Computing incrementability requires estimating the final work $C_F(P, \cdot)$ for each query and total work $C_T(P)$ given a pace configuration P. As the number and complexity of subplans grow, directly adapting the algorithm from an existing project [44] to compute the total work and the final work of a pace configuration is time-consuming. Our experiments in Section 5.5 shows that the original algorithm [44] cannot finish within 30 mins for the full TPC-H query set when the max pace of each subplan is larger than 50. This is because the original algorithm computes the cost of a pace configuration by simulating its the execution from scratch. Therefore, we propose a memoization-based algorithm to quickly compute the cost of a pace configuration.

Memoization algorithm By definition, a pace k means that the subplan starts one incremental execution to process its new data when the system receives $\frac{1}{k}$ of the total estimated tuples. To estimate the cost of a pace configuration, the original algorithm simulates the execution of each subplan with respect to the progress of how much new data arrives. To enable more reuse opportunities our memoization algorithm estimates the cost of a pace configuration by redefining the pace of a subplan to be dependent on the subplan's input data rather than the system's input data. To estimate the cost of a subplan with a pace k, we take the estimated total input data of this subplan and starts k incremental executions where each processes $\frac{1}{k}$ of its total input data.

We use Figure 4 to explain the algorithm of estimating the total work and the final work given a pace configuration. We use Q_{AB} in Figure 2 and consider a pace configuration $(3, 2, 1)$. Given the

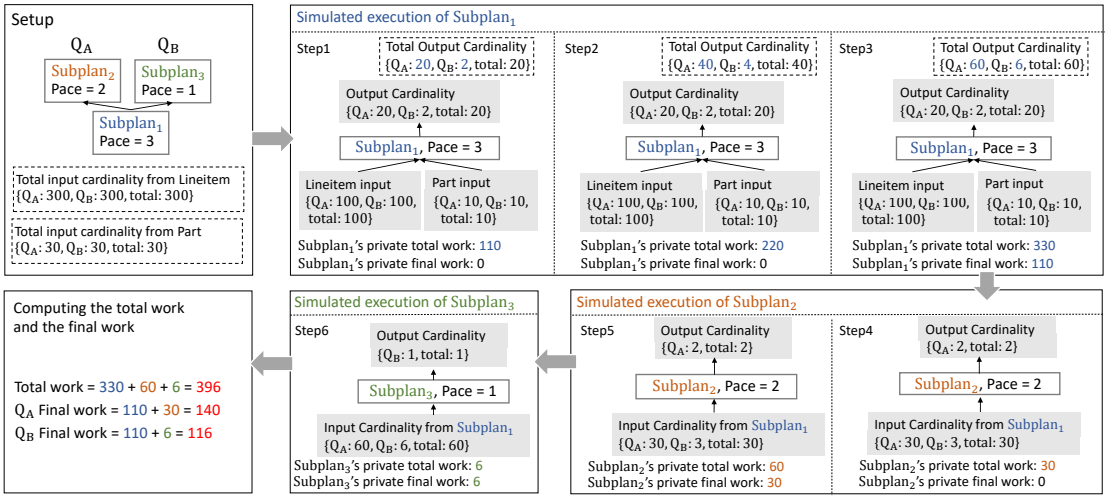


Figure 4: A running example of estimating the total work and the final work given a pace configuration

Algorithm 1: Estimate $C_T(P)$ and $C_F(P, \cdot)$

```
1  G_sorted ← Sorting subplans topologically from child
2                to parent subplans
3  for g_i ∈ G_sorted do
4      key ← Find the private pace configuration for g_i in P
5      if memo_i.contains(key) then
6          (pT, pF) ← memo_i(key)
7      else
8          (pT, pF, outCard) ← Estimating the cost and
9                  output cardinality of p_i simulated executions
10         Add (key -> (pT, pF, outCard)) to memo_i
11     end
12     for q_i ∈ Q do
13         if q_i includes g_i then
14             Add pF to C_F(P, q_i)
15     end
16     Add pT to C_T(P)
17 end
```

estimated total input cardinality of `Lineitem` and `Part`, this algorithm estimates the cost from bottom to top. It first simulates 3 incremental executions for $Subplan_1$ (i.e. Step 1 to 3 in Figure 4). The input cardinality for each simulated execution is $\frac{1}{3}$ of the total input and the cost of each execution is added to the *private total work* of this subplan, which represents the estimated total cost of this subplan processing its input data. We further define the *private final work* of a subplan as the cost of its final execution (i.e. Step 3). For each simulated incremental execution, we update the statistics of intermediate states (e.g. estimated hash table size for symmetric hash join) and estimate the output cardinality of each execution, which is added to the `total output cardinality` in Figure 4. Now, we have the total output cardinality of $Subplan_1$, which is the input cardinality of $Subplan_2$ and $Subplan_3$. $Subplan_2$ and $Subplan_3$ take this output cardinality, and simulate 2 and 1 incremental executions respectively (i.e. Step 4 to 6). Finally, the total work is the summation of the private total work of all subplans and the final work of a query includes the private final work of this query's subplans. Figure 4 shows that the final work of Q_A is the summation of the private final work of $Subplan_1$ and $Subplan_2$.

Now we discuss reusing the prior estimated results to quickly estimate the cost of a pace configuration. The estimated results we can reuse include output cardinality, private total work, and private final work of each subplan. We note that these estimated results depend on the paces of the subplan and its descendant subplans. We call these paces *private pace configuration* for a subplan. To reuse prior results, each subplan maintains a key-value memo table, where the key is a private pace configuration and the value includes output cardinality, private total work, and private final work.

Algorithm 1 shows our memoization algorithm. We estimate the private total work and private final work of each subplan (i.e. pT and pF in Algorithm 1) from the bottom to top. For each subplan, we first probe its memo table to look for prior estimated results. If not found, we start one simulation to estimate the private total work, the private final work, and its output cardinality (i.e. $outCard$). This information is then stored in the memo table. Finally, we add pT to the total work and add pF to the final work of the queries that include this subplan. We note that the estimation of the total work and final work might not be accurate due to the inaccurate cardinality estimation [43, 44]. For the recurring queries, we can calibrate the cardinality estimation based on previous query executions. We test the inaccurate cardinality estimation and find iShare has lower CPU consumption and similar query latencies compared to the baselines. The results are omitted due to space limits.

Algorithm of finding pace configuration Now we have an optimized algorithm for computing incrementability. We then adapt the algorithm from the existing project [44] to find the pace configuration in the shared setting. It starts at $P_{\mathbb{1}}$, which is the case of batch execution, and repeatedly increase the pace of one subplan having the highest incrementability. The loop includes two steps:

- Check whether all queries have met the final work constraints (i.e. $\forall q_i \in Q : C_F(P, q_i) <= L(q_i)$) and whether all paces have reached the max pace J (i.e. $\forall p_i \in P : p_i >= J$). If either is true, the optimization stops.
- For each $subplan_i$, its incrementability is $Inc(P, P_{[p_i \backslash p_i+1]})$, where $P_{[p_i \backslash p_i+1]}$ means that we increase $subplan_i$'s pace p_i by

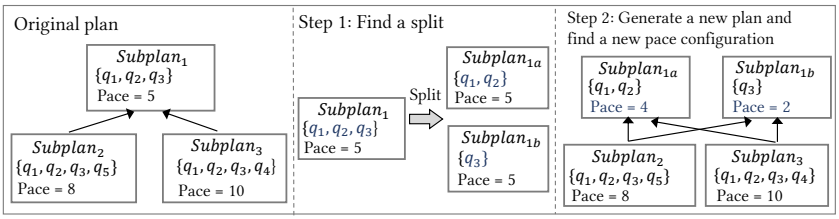


Figure 5: An overview of decomposing a shared subplan

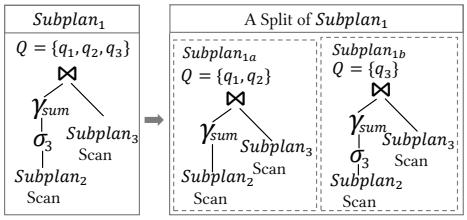


Figure 6: One split of $Subplan_1$

one. Assuming $subplan_{i^*}$ has the highest incrementability, the pace configuration P is updated to $P_{[p_{i^*} \setminus p_{i^*}+1]}$.

We note that the pace of a parent subplan should be no larger than its child subplan. The step two in the algorithm will filter out a candidate pace configuration $P_{[p_i \setminus p_i+1]}$ if it violates this requirement.

4 DECOMPOSING A SHARED SUBPLAN

iShare exploits the time slackness in the diverse final work constraints by selectively executing parts of the shared plan lazily to reduce the total work. After finding nonuniform paces for different subplans, iShare considers "unsharing" or decomposing each shared subplan for lazier execution. For example, Figure 2 shows that we can decompose $Subplan_1$ into two separate subplans such that Q_A and Q_B can be executed with different paces. Our following discussion is focused on decomposing a subplan as a whole (e.g. "unsharing") and more fine-grained decomposition (i.e. decomposing parts of a subplan) is discussed in Section 4.3.

Decomposing a shared plan loses opportunities for sharing. Therefore, we systematically consider the shared opportunities and the benefit of lazy incremental executions. We use the example of decomposing $Subplan_1$ in Figure 5 to show an overview of our decomposition algorithm:

- First, given a shared subplan, we split the queries that share this subplan. For example, Figure 5 shows that the query set $\{q_1, q_2, q_3\}$ is split into $\{q_1, q_2\}$ and $\{q_3\}$. Each subset of queries shares a single subplan (i.e. $Subplan_{1a}$ and $Subplan_{1b}$). The challenge here is that there is an exponential number of possible ways of splitting the queries. Therefore, we propose a clustering algorithm to heuristically split the queries. This algorithm uses a metric *sharing benefit* that can quickly decide whether it is worthwhile to share two sets of queries. We discuss splitting a subplan in Section 4.1.
- Second, we replace this decomposed subplan with the original subplan to generate a new query plan and find its pace configuration. For example, Figure 5 shows that we replace $Subplan_1$ with $Subplan_{1a}$ and $Subplan_{1b}$ and find new paces. We compute the new plan's total work, compare it with the total work of the original plan, and choose the one with the lowest total work. We discuss this step in Section 4.2.

We talk about fine-grained decomposition in Section 4.3 and applying our decomposition algorithm to the full plan in Section 4.4.

4.1 Finding a split for a shared plan

We define a `split` as a partitioning of the queries that share this subplan. Consider the example in Figure 6. The $Subplan_1$ is shared by three queries $Q = \{q_1, q_2, q_3\}$, which is split into two query sets $\{q_1, q_2\}$ and $\{q_3\}$. The new plan for one query set (e.g. $Subplan_{1a}$)

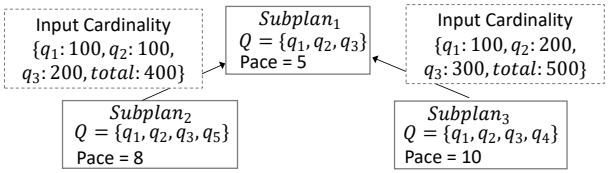


Figure 7: Input cardinalities of $Subplan_1$ using a pace configuration

copies all operators from the original subplan, except the select operators that do not belong to this query set (e.g. σ_3), and derives the same parent and child operators from the original subplan. Not shown in the figure are the project operators, which are copied from the original subplan and modified to include all attributes required by its ancestor operators.

There is an exponential number of ways of splitting a query set. Computing the total work from scratch for all splits can be time-consuming. Therefore, we define a local optimization problem of finding the split that best reduces the work of the subplan itself. The intuition here is that if a split can reduce the work of the subplan, it should also be able to reduce the total work of the whole plan. To solve this problem, we use a clustering algorithm to heuristically find the split that reduces the work of this subplan. In this subsection, we first talk about the definition of this local optimization problem and then discuss the clustering algorithm.

4.1.1 Defining the local optimization problem. Before we formally define this problem, we use an example to conceptually explain it.

Explaining the optimization problem with an example We first explain the optimization objective: **local total work**. Consider the example of decomposing $Subplan_1$ in Figure 7. The input cardinality of $Subplan_1$ represents the estimated total number of tuples that $Subplan_1$ needs to process given the pace configuration in Figure 7. For example, the input cardinality from $Subplan_3$ is 500, where 100, 200, and 300 tuples are valid for q_1, q_2, and q_3, respectively. If $Subplan_1$ uses pace 5, we simulate 5 incremental executions for every one-fifth of the input data. The local total work of this subplan is the sum of the five executions' work.

Then, we discuss estimating the local total work for a split. If we split $Subplan_1$ shown in Figure 6, we have two partitions $\{q_1, q_2\}$ and $\{q_3\}$. Assuming that they use paces 2 and 4 respectively, the local total work of $Subplan_1$ is computed as follows. We simulate 2 incremental executions for $\{q_1, q_2\}$ to its process the input data of $Subplan_1$. The total work of this partition is the sum of the 2 executions' work. Similarly, we can simulate 4 executions for $\{q_3\}$ to compute the total work of this partition. We call the total work of a partition **partial local total work**. Then, the local total work for is split is the summation of each partition's partial local total work. The goal of our optimization problem is to find a split that reduces the local total work.

We now explain the constraints of the optimization problem. We first define **local final work** of each query. Consider the partition $\{q_1, q_2\}$ in Figure 6. If its pace is 2, it means this partition uses 2 incremental executions to process it input data. Thus, the local final work of q_1 and q_2 is the work of the final execution (i.e. the 2nd) of their partition. Note that the local final work differs from private final work (Sec. 3.2) in that local is defined on a partition of a split, but final is for an entire subplan.

Our optimization problem is constrained on the local final work of each query. We compute the **local final work constraints** as follows. We first compute the absolute final work constraint for each query. Then, we proportionally scale each query's absolute final work constraint to its local final work constraint for each subplan. Consider $Subplan_1$ in Figure 6. The query q_1 has two operators in this subplan (i.e. the join and the aggregate operators). Assume that the two operators occupy 20% of the work of executing q_1 separately in one batch. Then, the local final work constraint for the two operators is also 20% of the constraint on q_1. We pre-compute the local final work constraints before we start the decomposition.

We call the paces for all partitions in a split the **local pace configuration**. To estimate the input data for each subplan's local optimization problem, we simulate the execution of the nonuniform pace configuration found in Section 3.2. Therefore, the optimization problem is *finding a split along with its local pace configuration to minimize the local total work and meet local final work constraints.*

Formal definition We use $\mathcal{W}_T(O, R)$ to denote the local total work given a split $O = (O_1, \cdots, O_D)$ and a local pace configuration $R = (R_1, \cdots, R_D)$. D represents the number of partitions and R_i represents the pace of partition O_i. The partial local total work of a partition O_i and a pace R_i is denoted as $\mathcal{W}_{PT}(O_i, R_i)$. Therefore, we have

$$\mathcal{W}_T(O, R) = \sum_{i=1}^{D} \mathcal{W}_{PT}(O_i, R_i) \tag{3}$$

In addition, the local final work for partition O_i, and each of its queries, with pace R_i is $\mathcal{W}_F(O_i, R_i)$.

Assuming that we have H queries sharing a subplan with local final work constraints $S = (S_1, \cdots, S_H)$, the local optimization problem is formally defined as

$$\begin{aligned} \underset{(O,R)}{\text{minimize}} \quad & \mathcal{W}_T(O, R) \\ \text{subject to} \quad & \mathcal{W}_F(O_i, R_i) \le \min_{j \in O_i} S_j \\ & \forall i \in [1, D] \end{aligned}$$

Here, the local final work of each partition $\mathcal{W}_F(O_i, R_i)$ needs to meet the lowest local final work constraint among the partition's queries (i.e. $\min_{j \in O_i} S_j$)

4.1.2 Finding the best split. Solving the above optimization problem needs to consider the split and its local pace configuration. Simply searching the space is time-consuming due to its exponential complexity. Thus, we leverage the following observation to prune the search space.

Observation Consider two partitions O_1 and O_2 in a split. We define the **selected pace**, R_i^*, of a partition, O_i, as the smallest pace that allows O_i to meet its local final work constraints (i.e. $\mathcal{W}_F(O_i, R_i^*) \le \min_{j \in O_i} S_j$). R_i^* means the laziest possible execution that reduces the most local total work. The observation is if we merge O_1 and O_2 into a single partition (i.e. O_{12}), the selected pace

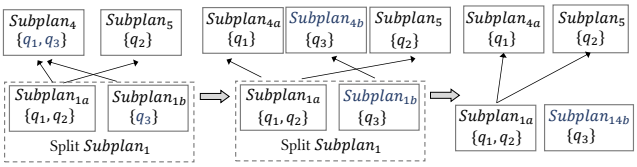


Figure 8: Generating a new plan using the decomposed $Subplan_1$

R_{12}^* of O_{12} should be no smaller than that of O_1 and O_2. The reason is that the work O_{12} needs to do is the union of O_1 and O_2, and this union of work will be no smaller than an individual partition. In addition, O_{12} needs to meet the lower constraint of the two partitions. So O_{12} will be no lazier than O_1 and O_2 respectively. The monotonicity of the selected pace motivates us to cluster the queries from the bottom up (i.e. merging partitions) and monotonically increasing the selected pace for the merged partitions.

Sharing benefit As we are clustering the queries from the bottom up, we need to choose which partitions to merge. To do so, we develop a metric, sharing benefit, to quantify the reduced total work for the merged partition. Consider merging two partitions O_i and O_j into a new partition O_{ij} The benefit is:

$$\begin{aligned} \texttt{Sharingbenefit}(Q_i, Q_j) = & \mathcal{W}_{PT}(O_i, R_i^*) + \mathcal{W}_{PT}(O_j, R_j^*) \\ & - \mathcal{W}_{PT}(O_{ij}, R_{ij}^*) \end{aligned} \tag{4}$$

Here, $\mathcal{W}_{PT}(O_i, R_i^*)$ represents the lowest partial total work of partition O_i given its selected pace R_i^*.

The clustering algorithm The clustering algorithm keeps merging two partitions with the highest sharing benefit until there is no positive benefit or there is only one partition left. It starts with a split where each query is in a separate partition. The initial pace configuration is $P_{\mathbb{1}}$. Before we merge partitions, we increase the pace of each partition to find the selected pace that meets the partition's local final work constraints. After, we merge the pair of partitions that has the highest benefit. As the observation shows, the newly merged partition adopts the larger selected pace of the two old partitions and increases this pace to find the new selected pace. Thus, the search for the selected pace does not start from 1, but monotonically from the larger selected paces of old partitions.

4.2 Generating a new plan & pace configuration

The first step proposes a decomposed subplan. Then, we replace the old subplan with the decomposed one and check whether this new plan can reduce the total work. We note that our local optimization is to find a decomposed subplan that has the potential of enabling lazy execution to reduce CPU consumption, that is, the local pace configuration has smaller paces compared to pace of the original subplan. While this local pace configuration can reduce CPU consumption, it might violate the requirement that the pace of a parent subplan should be no larger than the pace of its child subplans. In addition, the local paces might not be small enough to exploit the full potential of the decomposed subplan when we consider the full shared plan as a whole. So we perform an optimization to find a corrected version of the local pace configuration of this decomposed subplan to make sure the paces we find are small enough to reduce CPU consumption but not violate the aforementioned requirement.

Generating a new plan Recall that we assume an execution engine that requires the query set of a subplan subsume the query

sets of its parent subplans. However, the new decomposed subplan may not meet this requirement. Consider the subplan in Figure 8 as an example. We see that the query set $\{q_3\}$ of $Subplan_{1b}$ does not subsume the query set of its parent $Subplan_4$ (i.e. $\{q_1, q_3\}$). In this case, we split the parent subplans to align them with their child subplans (e.g. the middle graph in Figure 8). We recursively do this for the parent subplans until we meet the requirement.

After that, we consider merging the newly generated subplans when a new subplan has only one parent subplan. Consider the $Subplan_{1b}$ and $Subplan_{4b}$ of the newly generated plan in Figure 8. Since $Subplan_{1b}$'s the only parent subplan is $Subplan_{4b}$, they should be merged (i.e. $Subplan_{14b}$ in the right graph of Figure 8).

Finding a new pace configuration Recall that the motivation of the decomposition is that the decomposed subplan enables us to execute the whole shared plan lazily by leveraging the diverse final work constraints. Therefore, we need to find a lazier pace configuration (i.e. smaller pace). The key idea is to initialize the newly generated plan with a pace configuration that is eagerer than or equal to the original one (i.e. equal or larger pace) and incrementally decrease the paces. Therefore, we use two steps to generate the initial pace configuration:

- Step 1. For each newly generated subplan, we use the pace of the original subplan that the new subplan is derived from. For example, since $Subplan_{1a}$ and $Subplan_{1b}$ in Figure 8 are derived from $Subplan_1$, the two new plans then adopt the pace of $Subplan_1$.
- Step 2. If a newly generated subplan should be merged with another subplan (e.g. $Subplan_{1b}$ and $Subplan_{4b}$ in Figure 8) and the two subplans have different paces, we choose the larger pace for the merged subplan.

Starting from this pace configuration, we use a modified algorithm in Section 3.2 to incrementally find a new nonuniform pace configuration. The difference is that at each step instead of increasing the pace of the subplan with the highest incrementability, we decrease the pace of the subplan that has the lowest incrementability. That is, we choose the subplan that can best lower the total work for the same final work increase.

4.3 Partial decomposition

Partially decomposing a subplan selects a subtree that shares the root of the subplan and then splits the subtree. For example, consider $Subplan_1$ in Figure 6. We can choose to split the join operator (i.e. $\bowtie$) and leave its child operators unchanged. The key idea is that we first break the subplan into three subplans: the join operator itself, and the left/right child subtree of the join operator. Afterwards, we split the join operator using the clustering algorithm.

We note that there is an exponential number of subtrees sharing the root of a subplan. Therefore, our partial decomposition considers a subset of them. Specifically, we generate the subtree candidates by starting with the root operator and gradually expanding the subtree to include its child operators using a breath-first like search. Each new subtree includes one additional child operator that is the closest to the root operator. Therefore, the number of subtree candidates is no larger than the number of operators in a subplan, which greatly reduces the optimization time while keeping the opportunities of decomposing the subplan with a fine-granularity.

4.4 Applying decomposition to the full plan

We now discuss applying the decomposition algorithm of a shared subplan to the full plan. After we find the nonuniform pace configuration for the full plan, we also collect statistics information required by the decomposition algorithm. We simulate the execution of the nonuniform pace configuration to generate the input cardinalities for each subplan and run each query separately in one batch to collect the local final work constraints. Then, we sort all subplans topologically from the parent to the child and apply the decomposition algorithm for each subplan in this order to generate a new plan with a smaller total work.

5 EXPERIMENTS

Our experiments address the following questions:

- Compared to a state-of-the-art shared plan [17] that uses a single pace and two other approaches that execute queries separately, how much does iShare reduce the total execution time given the same final work constraints? (Sec. 5.3)
- How much more efficient is our decomposition algorithm in reducing computing resources in iShare and how is it compared to a brute-force approach? (Sec. 5.4)
- What is the optimization overhead of iShare, and how much optimization overhead our memoization and clustering algorithms reduce compared to other algorithms? (Sec. 5.5)
- How does different levels of incrementability and varied final work constraints impact the resource consumption of the baseline approaches and iShare? (Sec. 5.6)

All experiments are run on a server that has 196 GB of main memory and two Intel Xeon Silver 4116 processors, with 24 total physical cores. We use 20 cores for all experiments and leave the rest for the OS (Ubuntu 18.04) and other supporting processes (e.g. HDFS).

5.1 Prototype Implementation

iShare is implemented in Spark 2.4.0 [6]. We extend Spark SQL to support shared query execution based on SharedDB [16] and incremental execution of deletes and updates based on existing IVM algorithms [13]. We adopt techniques to reduce the cost of starting Spark jobs for incremental execution [47] . We use a Kafka [1] cluster as the data source that streams new data into Spark queries. The Kafka cluster is also used to materialize intermediate output tuples from a subplan that has two or more parent subplans. Each parent subplan pulls the new data of this subplan from the Kafka cluster. Additionally, the cluster is run on a different machine with the same hardware configuration. The two machines are placed on the same rack and have 10 Gbps Ethernet connection to each other.

Users submit a set of SQL queries along with final work constraints to iShare. Recall that we support both relative and absolute final work constraints. For easy presentation, we choose the relative one. Users can tune the relative constraints to explore the trade-off between resource consumption and query latency.

iShare uses a state-of-the-art MQO optimizer [17] to generate a shared query plan from the submitted queries. Note that we extend this optimizer to account for the materialization cost of intermediate tuples as suggested by an existing MQO optimizer [40]. iShare optimizes this shared plan by finding the pace configuration and decomposing subplans if necessary. Each incremental execution

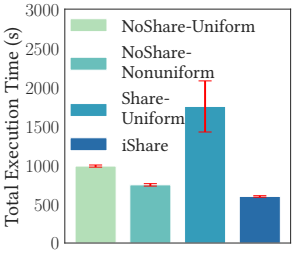


Figure 9: Tests of random relative constraints

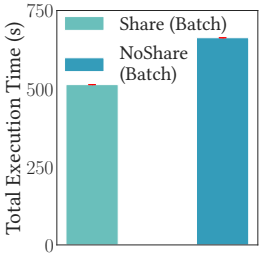


Figure 10: Batch execution (22 queries)

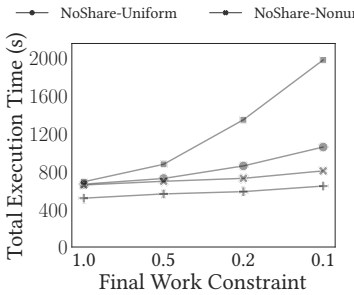


Figure 11: Tests of uniform relative constraints (22 queries)

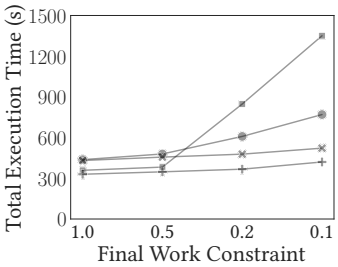


Figure 12: Tests of uniform relative constraints (10 queries)

	Random				Uniform			
	Mean %	Mean Sec.	Max %	Max Sec.	Mean %	Mean Sec.	Max %	Max Sec.
NoShare-Uniform	37.24	1.37	854.69	30.48	21.36	0.98	884.47	31.54
NoShare-Nonuniform	6.37	0.42	123.41	9.06	5.66	0.38	192.59	7.60
Share-Uniform	44.24	1.72	895.19	31.92	29.48	2.02	802.19	33.53
iShare	6.39	0.22	153.66	4.68	7.17	0.34	207.24	7.14

Table 1: Missed latencies of random and uniform relative constraints.

of a subplan uses all 20 CPU cores. When an execution is finished, we execute the next subplan based on the pace configuration. If multiple subplans start their incremental executions at the same time (e.g. they have the same pace), the child subplans are executed earlier than their parent subplans.

Recall that iShare is optimized to reduce CPU consumption and achieve similar latencies compared to the baseline approaches. Therefore, we use the *total execution time* to represent the CPU consumption for a set of scheduled queries. Total execution time is the summation of the elapsing time of all incremental executions. Lower total execution time means lower CPU utilization. The query *latency* is defined as the summation of the final execution time of all subplans in this query. We report missed latency with respect to the *latency goal* in the experiment. Here, we compute the latency goal of a query by multiplying the query's relative final work constraint by the latency of its batch execution. We note that for different queries they may have different latency goals even for the same relative final work constraints. This is because the latency of their respective batch execution is different. We report two types of missed latency, *absolute missed latency* and *relative missed latency*. The *absolute missed latency* represents difference between the tested latency and the latency goal, which is $max(0, \text{tested latency} - \text{latency goal})$. The *relative missed latency* represents the percentage of the absolute missed latency compared to the latency goal, which is $\frac{\text{absolute missed latency}}{\text{latency goal}}$.

5.2 Experiment setup

Benchmark We use the TPC-H benchmark in our experiments and our prototype supports all 22 TPC-H queries. Furthermore, we test the two example queries Q_A and Q_B from Figure 2:

```
Q_A: SELECT SUM(agg_l.sum_quantity) as total_sum_quantity
     From part p,
          (SELECT SUM(l_quantity) as sum_quantity
           FROM Lineitem
           GROUP BY l_partkey) agg_l
     WHERE p_partkey == l_partkey

Q_B: SELECT ps_partkey
     FROM partsupp ps,
          (SELECT AVG(agg_l.sum_quantity) as avg_quantity
           From part p,
                (SELECT SUM(l_quantity) as sum_quantity
                 FROM Lineitem
                 GROUP BY l_partkey) agg_l
           WHERE p_partkey == l_partkey
           AND   p_brand" == "Brand#23" AND p_size == 15)
     WHERE ps.ps_availqty < avg_quantity
```

We preload the full dataset into Kafka and let iShare pull data from Kafka at a rate of 100MB/min. This data pull rate allows the system to process all the data steadily (i.e. no back pressure) even if we run all 22 TPC-H queries concurrently. We use a dataset with a scale factor of 5 (i.e. 5GB data) to make sure that Spark does not run out of memory even for all TPC-H queries. The max pace is capped at 100. Therefore, the time of loading 5GB data at the rate of 100MB/min is 3000s. If we use the max pace 100, we will start one execution every 30s. We assume that the system knows the data arrival rate based on previous executions. We run each test three times and report the average unless otherwise specified.

Baselines We compare iShare against three baselines. **NoShare-Uniform** executes each query separately with a separate pace for each query. The pace is set to meet each query's final work constraint using the algorithm from Section 3.2. Second, **NoShare-Nonuniform** is adapted from previous work [44]. Here, each query is broken into smaller parts, where each part is executed at a different pace. Therefore, each query is assigned a set of *nonuniform* paces. We implement this idea by breaking a query into subplans at blocking operators (e.g. aggregate). The root of a subplan is either a blocking operator or the root of the query. To generate a subplan, we expand the subplan's root to gradually include its descendant operators until another blocking operator or a base relation. The pace configuration for each query with respect to the query's absolute final work constraint is found with the algorithm from Section 3.2. **Share-Uniform** uses an existing MQO optimizer [17] to generate several separate shared plans, where each plan is assigned a separate pace. We have separate plans because they have no sharable sub-expressions or the MQO optimizer finds the sharing cost too

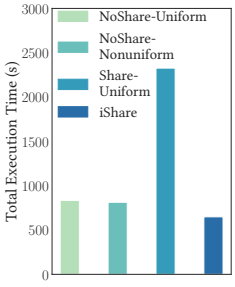


Figure 13: Manually tuned paces

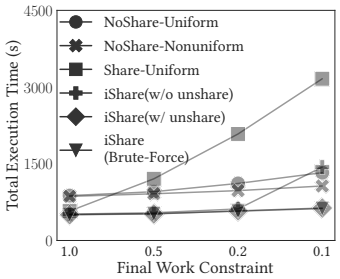


Figure 14: Tests for decomposition algorithm

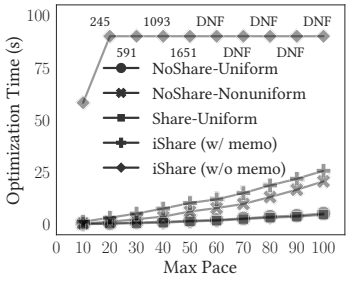


Figure 15: Overhead of end-to-end optimization

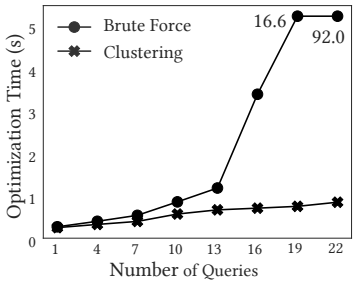


Figure 16: Optimization overhead of Clustering algorithm

	Mean %	Mean Sec.	Max %	Max Sec.
NoShare-Uniform	4.40	0.16	96.85	3.45
NoShare-Nonuniform	0	0	0	0
Share-Uniform	32.34	1.15	711.57	25.37
iShare	0	0	0	0

Table 2: Missed latencies for manually tuned paces

	Mean %	Mean Sec.	Max %	Max Sec.
NoShare-Uniform	16.62	0.79	910.19	32.45
NoShare-Nonuniform	2.71	0.29	104.49	7.79
Share-Uniform	28.76	1.79	670.8	26.08
iShare (w/o unshare)	27.01	0.98	996.32	35.53
iShare (w/ unshare)	0.39	0.01	32.82	0.96
iShare (Brute-Force)	0.48	0.03	25.23	1.67

Table 3: Missed latencies for the decomposition algorithm

high (e.g. due to the high materialization cost). For each separate plan, we use the algorithm from Section 3.2 to find a single pace that minimizes the total work and makes the plan meet the final work constraints of all queries in this shared plan. We also compare iShare to a simple approach that starts one execution before the trigger point and a final execution at the trigger point to process all data. Our test tunes the point of starting the first execution to find the best case for this approach. We find that this approach significantly misses query latencies (i.e. up to 25.7s and 1046% even for its best case in our test), but the missed latencies are zero for iShare in the same test. Therefore, we omit the results due to the space limit.

5.3 Low CPU consumption with the same final work constraints

In this subsection, we examine how much iShare reduces the total execution time with similar or lower absolute and relative missed latencies compared to the baseline approaches. We test 22 TPC-H queries with two types of relative final work constraints. First, we generate a set of relative final work constraints for all queries by randomly picking relative constraints from $(1.0, 0.5, 0.2, 0.1)$ for each query. Second, we use a uniform relative final work constraint from $(1.0, 0.5, 0.2, 0.1)$ for all queries.

Tests of random relative constraints For the first experiment, we test three sets of randomly generated relative constraints. We report the mean, minimum, and maximum total execution time for all approaches, and their missed latencies. The results in Figure 9 show that iShare has 60.5%, 80.1%, and 34.1% of mean total execution time compared to NoShare-Uniform, NoShare-Nonuniform, and Share-Uniform. This is because iShare reduces redundant work from overlapping sub-expressions compared to NoShare approaches, and iShare lazily executes some subplans to avoid the overly eager execution compared to Share-Uniform. Share-Uniform has a larger variance due to its need to meet the lowest constraint, which is highly variable from the random selection of relative constraints. For completeness, we show the total execution time reduction of executing the shared plan from Share-Uniform in one batch relative to executing each query independently in Figure 10. Thus, the overhead of overly eager execution is what makes Share-Uniform have higher CPU consumption than other approaches.

The `Random` column in Table 1 shows the results of missed latencies. `Mean Sec.` and `Max Sec.` represent the mean and max absolute missed latencies, and `Mean %` and `Max %` represent the mean and max relative missed latencies. The minimum and median missed latencies, not shown, for all approaches are zero. We see that iShare has similar or less absolute and relative missed latencies compared to the baselines. The main reason for missed latency is the inaccuracy of the cost model. Additionally, the maximum absolute and relative missed latencies for NoShare-Uniform and Share-Uniform are large because parts of some queries are not incrementable. Thus, using a single pace to eagerly execute these queries does not reduce the query latency, resulting in a high missed latency. One such example is Q_{15}. It maintains two aggregate operators, where one aggregate operator, `max` is parent of another aggregate operator, `sum`. When the aggregated values in the the `sum` operator are changed, this operator will output a delete and an insert operation to the parent `max` operator. If a max value is deleted, the `max` operator needs to rescan all arrived values to find the new max one. Eagerly maintaining the whole query does not reduce the cost of finding a new max value, which is why NoShare-Uniform and Share-Uniform have high query latencies. However, NoShare-Nonuniform and iShare use different paces for different parts of the query plan and can maintain the `max` operator lazily to avoid deleting the max value and, thus, the cost of finding a new max value.

Tests of uniform relative constraints Our second test uses uniform relative final work constraints for all queries. We use relative constraints of 1.0, 0.5, 0.2, and 0.1, and report the total execution time and missed latencies. Figure 11 shows that iShare reduces CPU consumption compared to the baselines for all relative constraints. We also observe that the Share-Uniform has similar CPU

consumption compared to NoShare approaches even when the relative constraint is 1.0. We note that the absolute constraint for each query is different given the same relative constraint 1.0 for all queries. To meet the lowest absolute constraint, Share-Uniform has overly eager executions, which offsets the benefit of shared query execution. To show the benefit of Share-Uniform, we perform an additional test of 10 TPC-H queries, Q_4, Q_5, Q_7, Q_8, Q_9, Q_{15}, Q_{17}, Q_{18}, Q_{20}, and Q_{21}, which have significant amounts of overlapping work and, for the same relative constraint, they have similar absolute final work constraints. We see, in Figure 12, that Share-Uniform has lower CPU consumption compared to NoShare approaches, because the absolute constraints are less diverse and Share-Uniform has smaller overhead of overly eager execution. This leads to better shared performance. For all constraints, iShare has lower CPU consumption compared to all other approaches.

The `Uniform` column in Table 1 shows the mean and maximum missed latencies for all queries tested in Figure 11 and Figure 12. Again, the minimum and median missed latencies are not shown because they are zero. We have the same observation as the test of random relative constraints. iShare has similar absolute and relative missed latencies compared to other approaches. Share-Uniform and NoShare-Uniform have higher maximum missed latencies because one tested query (i.e. Q_{15}) is not incrementable.

Tests for manually tuned pace configuration In this test, we manually tune the pace configuration to make all approaches meet the latency goals (a relative constraint of 0.1). If there are queries that cannot meet the latency goal for some approaches, we make sure that these queries have the smallest missed latencies.

For NoShare-Uniform, we test all paces for each query; for Share-Uniform, we tune the pace for the whole plan; and for NoShare-Nonuniform and iShare, we tune the pace configurations by setting smaller relative final work constraints for queries that otherwise have missed latencies.

Figure 13 shows that iShare uses uses 77.7%, 80.0%, and 27.9% of the CPU seconds compared to NoShare approaches and Share-Uniform, respectively. Table 2 shows the results of the mean and maximum missed latencies, with the minimum and median excluded as they are all zero. We see that both NoShare-Uniform and Share-Uniform still have missed latencies because the query Q_{15} is not incrementable and increasing a single pace for the query plan could not achieve the desired query latency.

5.4 Performance impact of decomposition

We show our decomposition algorithm can reduce the total execution time compared to using the nonuniform pace configuration only. Here, `iShare (w/o unshare)` means that iShare does not use the decomposition algorithm, and `iShare (w/ unshare)` is the variant with all optimizations. We create a query set that has much overlapping work and show the benefit of the decomposition algorithm for this "sharing-friendly" query set. Specifically, we take the 10 TPC-H queries in Figure 12, modify their predicates to generate new 10 TPC-H queries, and combine the original and new queries to create a new query set. For each query, we modify the two types of predicates: equality predicate (e.g. $name = "Tom"$) and range-based predicates (e.g. $A > 10$ and $A < 20$). For 50% of the equality predicates, we use a different value (e.g. $name = "Jerry"$), and for a range-based predicate, we generate a new predicate that with an overlap up to 50% (e.g. $A > 15$ and $A < 25$). We test uniform relative final work constraints of 1.0, 0.5, 0.2, and 0.1.

Figure 14 shows that when we use the relative constraint 0.1, iShare (w/o unshare) uses more CPU seconds than the NoShare approaches, because there is significant overhead of overly eager execution introduced by shared subplans. For example, consider Q_{15} and its variant Q'_{15}. iShare (w/o unshare) shares the subplan of maintaining the *max* aggregate operator for the two queries. Since Q_{15} and Q'_{15} have different (but overlapping) predicates, the *max* aggregate of the shared plan needs to do more work than the individual *max* aggregate in each query. Therefore, a lower relative final work constraint (e.g. 0.1) pushes the shared plan to execute more eagerly. Eagerly maintaining this *max* operator is expensive. iShare (w/ unshare) avoids this cost by decomposing the shared subplan between Q_{15} and Q'_{15} and executing each subplan lazily. Thus, iShare (w/ unshare) uses 52.3%, 71.8% , 31.8%, and 58.6% of the CPU seconds compared to NoShare-Uniform, NoShare-Nonuniform, Share-Uniform, and iShare (w/o unshare), respectively. We also test a variant of iShare that decomposes a subplan by searching all possible ways of splitting the subplan (denoted by `iShare (Brute-Force)`). Figure 14 and Table 3 show that this approach has similar total execution time and missed latencies compared to iShare (w/ unshare), which uses a greedy clustering algorithm for the subplan decomposition. In the next subsection, we show that the optimization overhead of iShare (Brute-Force) is much higher than iShare (w/ unshare).

5.5 Optimization overhead

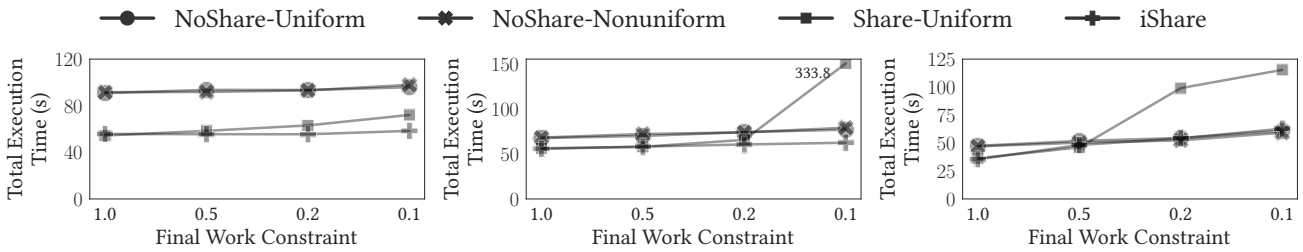
We test the optimization time of iShare, the baselines, and an iShare variant that computes incrementability using a simulation algorithm [44] without using the memoization. We denote this approach `iShare (w/o memo)` and the one with the memoization `iShare (w/ memo)`. We use all TPC-H queries, vary the value of the max pace from 10 to 100, and set a low relative final work constraint for all queries (i.e. 0.01).

Figure 15 shows the optimization time, where iShare (w/ memo) has a much lower running time compared to iShare (w/o memo). We mark a test case as `DNF` if it does not finish within 30 minutes, where iShare (w/o memo) fails when the max pace is larger than 50. In the worst case, iShare (w/ memo) uses 25.6s to finish the optimization. While it is higher than the baselines, we believe the significant reduction in CPU consumption justifies the cost.

We also compare our clustering algorithm for decomposing a subplan to an algorithm of searching all possible ways of splitting the subplan (denoted as `Brute-force`). We use a max pace 100 and vary the number of queries we need to optimize. Figure 16 shows that the running time of our clustering algorithm is significantly smaller than that of the Brute-force method, which increases exponentially as we increase the number of queries to optimize.

5.6 Impact of incrementability and final work constraints

In this subsection, we test how incrementability and relative final work constraints impact CPU consumption with three pairs of queries: 1) PairA: Q_5 and Q_8; 2) PairB: Q_7 and Q_{15}; and 3) PairC:



(a) CPU consumption of Q_5 and Q_8 **(b) CPU consumption of Q_7 and Q_{15}** **(c) CPU consumption of Q_A and Q_B**

Figure 17: Micro benchmarks for queries with varied levels of incrementability and relative final work constraints

Q_A and Q_B. PairA consists of two queries that are amenable to incremental executions. PairB includes an incrementable query (Q_7) and a query that is not amenable to incremental executions (Q_{15}). Finally, PairC has two queries that are less incrementable. For each pair, we fix one query's relative constraint to 1.0 (i.e. Q_5, Q_{15}, and Q_A) and change the relative constraint of the other query.

Figure 17a shows that the overhead of overly eager execution for Share-Uniform is small, since Q_5 and Q_8 are amenable to incremental executions. Thus, Share-Uniform has lower CPU consumption than NoShare approaches and iShare has a slightly lower CPU consumption than Share-Uniform due to its nonuniform pace configuration. When we mix a less incrementable query (i.e. Q_{15}) with an incrementable query Q_7 and eagerly execute the incrementable query, Q_7, Share-Uniform is no longer better than NoShare approaches. Figure 17b shows that, when the relative constraint is 0.1, Share-Uniform consumes more CPU seconds than NoShare approaches. However, iShare has lower CPU consumption compared to all baselines. Finally, mixing two less incrementable queries Q_A and Q_B in Figure 17c, we also see that Share-Uniform becomes sub-optimal when Q_B's relative final work constraint is decreased. Here, iShare shares Q_A and Q_B for the constraints 1.0 and 0.5. When the constraint is 0.2 and 0.1, it executes Q_A and Q_B separately, and has similar performance to NoShare approaches. In addition, all approaches have small missed latencies except the Share-Uniform for PairB because Share-Uniform executes the non-incrementable query Q_{15} eagerly. We omit the results due to the space limit.

6 RELATED WORK

Multi-query optimization and shared query execution Many MQO and shared query execution approaches focus on ad-hoc queries. Some work [16, 17, 33, 39, 40, 54] considers batching several queries to exploit the common sub-expressions and builds a single plan to maximally share the work of batched queries. Other approaches consider specific operators, such as sharing scans [37, 38, 41] and joins [8, 32]. In addition, some approaches also consider reusing intermediate results of running queries to process new queries on-the-fly [22, 36]. This idea of shared query execution is widely used in continuous query processing or stream computing [21, 24, 30, 45, 46, 49, 52]. For example, shared arrangements [34] considers sharing the intermediate states across standing queries and supports different indexed views over the same states. Other approaches [28, 29] consider sharing the execution of standing and ad-hoc queries. Prior research works also studied whether queries should be shared by considering the overhead introduced by parallel execution [27] or materializing intermediate results [40].

iShare is different from these approaches in that it considers heterogeneous latency goals and judiciously shares query execution by considering the overhead of eager execution.

Incremental view maintenance and continuous queries Incremental view maintenance (IVM) studies how to efficiently and incrementally incorporating new data into prior materialized results. Existing approaches propose many IVM algorithms for maintaining select-project-join (SPJ) views, supporting negation and aggregate operators [19, 20], incrementally processing recursive views and nested subqueries [20, 35, 51], and optimizing incremental executions for semi-join, outer join, and acyclic joins [7, 18, 25, 31, 48].

Beyond efficient IVM algorithms, some approaches explore the trade-offs between view maintenance cost and query latency [14, 15, 23, 44, 53]. Colby et al. [15] proposes several policies for IVM to decide how eagerly or lazily to maintain a view. Prior studies observe the asymmetric maintenance cost for different parts of a query [23, 44]. They execute different parts of a query at different frequencies to meet a latency goal. Many continuous query and stream systems use IVM algorithms to provide low query latency [6, 9–12]. They allow users to adjust the number of tuples processed for each incremental execution [6, 9, 10]. This knob provides a trade-off between query latency and resource consumption.

iShare is different from existing approaches because it considers exploiting the benefit of shared query execution and avoiding the overhead of eager query execution at the same time.

7 CONCLUSION

We present iShare as a new optimization framework for scheduled queries with different latency goals. It judiciously decides what parts of a query to share and how eagerly or lazily to execute different parts of the shared plan. Our experiments show that iShare can significantly reduce CPU consumption compared to the shared query execution using a single pace and two approaches that execute queries separately.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their valuable feedback. This work was supported by NSF Award CCF-1139158, a Google DAPA Research Award, the CERES Center for Unstoppable Computing, a US Army TATRC Research Award, and Intel.

REFERENCES

[1] Apache kafka. https://kafka.apache.org/.
[2] Google cloud function pricing. https://cloud.google.com/functions/pricing?hl=vi.
[3] Google cloud scheduled query. https://cloud.google.com/bigquery/docs/scheduling-queries.
[4] Hoodie. https://eng.uber.com/hoodie/.
[5] Popsql. https://popsql.com/scheduled-queries.
[6] Spark structured streaming. https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html.
[7] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *Proc. VLDB Endow.*, 5(10):968–979, 2012.
[8] G. Candea, N. Polyzotis, and R. Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *PVLDB*, 2(1):277–288, 2009.
[9] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.
[10] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proc. VLDB Endow.*, 8(4):401–412, 2014.
[11] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR 2003, First Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 5-8, 2003, Online Proceedings*, 2003.
[12] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 379–390, 2000.
[13] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.
[14] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 469–480, 1996.
[15] L. S. Colby, A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, and K. A. Ross. Supporting multiple view maintenance policies. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA.*, pages 405–416, 1997.
[16] G. Giannikis, G. Alonso, and D. Kossmann. Shareddb: Killing one thousand queries with one stone. *Proc. VLDB Endow.*, 5(6):526–537, 2012.
[17] G. Giannikis, D. Makreshanski, G. Alonso, and D. Kossmann. Shared workload optimization. *Proc. VLDB Endow.*, 7(6):429–440, 2014.
[18] T. Griffin and B. Kumar. Algebraic change propagation for semijoin and outerjoin queries. *SIGMOD Record*, 27(3):22–27, 1998.
[19] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, pages 328–339, 1995.
[20] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*, pages 157–166, 1993.
[21] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer, editors, *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, pages 297–308. Morgan Kaufmann, 2003.
[22] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 383–394, 2005.
[23] H. He, J. Xie, J. Yang, and H. Yu. Asymmetric batch incremental view maintenance. In K. Aberer, M. J. Franklin, and S. Nishio, editors, *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, pages 106–117. IEEE Computer Society, 2005.
[24] M. Hong, M. Riedewald, C. Koch, J. Gehrke, and A. J. Demers. Rule-based multi-query optimization. In M. L. Kersten, B. Novikov, J. Teubner, V. Polutin, and S. Manegold, editors, *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings*, volume 360 of *ACM International Conference Proceeding Series*, pages 120–131. ACM, 2009.
[25] M. Idris, M. Ugarte, and S. Vansummeren. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1259–1274, 2017.
[26] S. Jacobs, M. Y. S. Uddin, M. J. Carey, V. Hristidis, V. J. Tsotras, N. Venkatasubramanian, Y. Wu, S. Safir, P. Kaul, X. Wang, M. A. Qader, and Y. Li. A BAD demonstration: Towards big active data. *Proc. VLDB Endow.*, 10(12):1941–1944, 2017.
[27] R. Johnson, N. Hardavellas, I. Pandis, N. Mancheril, S. Harizopoulos, K. Sabirli, A. Ailamaki, and B. Falsafi. To share or not to share? In C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C. Kanne, W. Klas, and E. J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 351–362. ACM, 2007.
[28] J. Karimov, T. Rabl, and V. Markl. Ajoin: Ad-hoc stream joins at scale. *Proc. VLDB Endow.*, 13(4):435–448, 2019.
[29] J. Karimov, T. Rabl, and V. Markl. Astream: Ad-hoc shared stream processing. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 607–622. ACM, 2019.
[30] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 623–634, 2006.
[31] P. Larson and J. Zhou. Efficient maintenance of materialized outer-join views. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 56–65, 2007.
[32] D. Makreshanski, G. Giannikis, G. Alonso, and D. Kossmann. Mqjoin: Efficient shared execution of main-memory joins. *PVLDB*, 9(6):480–491, 2016.
[33] D. Makreshanski, J. Giceva, C. Barthels, and G. Alonso. Batchdb: Efficient isolated execution of hybrid OLTP+OLAP workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 37–50, 2017.
[34] F. McSherry, A. Lattuada, M. Schwarzkopf, and T. Roscoe. Shared arrangements: practical inter-query sharing for streaming dataflows. *Proc. VLDB Endow.*, 13(10):1793–1806, 2020.
[35] M. Nikolic, M. Dashti, and C. Koch. How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 511–526, 2016.
[36] I. Psaroudakis, M. Athanassoulis, and A. Ailamaki. Sharing data and work across concurrent analytical queries. *Proc. VLDB Endow.*, 6(9):637–648, 2013.
[37] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-memory scan sharing for multi-core cpus. *PVLDB*, 1(1):610–621, 2008.
[38] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 60–69, 2008.
[39] R. Rehrmann, C. Binnig, A. Böhm, K. Kim, W. Lehner, and A. Rizk. Oltpshare: The case for sharing in OLTP workloads. *Proc. VLDB Endow.*, 11(12):1769–1780, 2018.
[40] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 249–260. ACM, 2000.
[41] M. Switakowski, P. A. Boncz, and M. Zukowski. From cooperative scans to predictive buffer management. *PVLDB*, 5(12):1759–1770, 2012.
[42] R. Taft, N. El-Sayed, M. Serafini, Y. Lu, A. Aboulnaga, M. Stonebraker, R. Mayerhofer, and F. J. Andrade. P-store: An elastic database system with predictive provisioning. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 205–219. ACM, 2018.
[43] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin. Intermittent query processing. *Proc. VLDB Endow.*, 12(11):1427–1441, July 2019.
[44] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin. Thrifty query execution via incrementability. In *SIGMOD 2020*, pages 1241–1256. ACM, 2020.
[45] K. Tangwongsan, M. Hirzel, S. Schneider, and K. Wu. General incremental sliding-window aggregation. *Proc. VLDB Endow.*, 8(7):702–713, 2015.
[46] J. Traub, P. M. Grulich, A. R. Cuellar, S. Breß, A. Katsifodimos, T. Rabl, and V. Markl. Efficient window aggregation with general stream slicing. In M. Herschel, H. Galhardas, B. Reinwald, I. Fundulaki, C. Binnig, and Z. Kaoudi, editors, *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, pages 97–108. OpenProceedings.org, 2019.
[47] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 374–389, 2017.
[48] Q. Wang and K. Yi. Maintaining acyclic foreign-key joins under updates. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 1225–1239. ACM, 2020.

[49] S. Wang, E. A. Rundensteiner, S. Ganguly, and S. Bhatnagar. State-slice: New paradigm of multi-query optimization of window-based stream queries. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 619–630, 2006.

[50] Z. Wang, K. Zeng, B. Huang, W. Chen, X. Cui, B. Wang, J. Liu, L. Fan, D. Qu, Z. Hou, T. Guan, C. Li, and J. Zhou. Tempura: A general cost-based optimizer framework for incremental data processing. *Proc. VLDB Endow.*, 14(1):14–27, Sept. 2020.

[51] K. Zeng, S. Agarwal, and I. Stoica. iOLAP: Managing uncertainty for efficient incremental OLAP. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1347–1361, 2016.

[52] R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple aggregations over data streams. In F. Özcan, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 299–310. ACM, 2005.

[53] J. Zhou, P. Larson, and H. G. Elmongui. Lazy maintenance of materialized views. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 231–242, 2007.

[54] J. Zhou, P. Larson, J. C. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In C. Y. Chan, B. C. Ooi, and A. Zhou, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 533–544. ACM, 2007.