Agamotto: Scheduling of Deadline-Oriented Incremental Query Execution under Uncertain Resource Price

Botong Huang
Lianggui Weng
Wei Chen
Kai Zeng
Yihui Feng
Bolin Ding

Jingren Zhou
Alibaba Group
Hangzhou, China
{botong.huang, lianggui.wlg, wickeychen.cw, zengkai.zk, yihui.feng, bolin.ding, jingren.zhou}@alibaba-inc.com

Zuozhi Wang
Chen Li
University of California, Irvine
Irvine, United States
{zuozhiw, chenli}@ics.uci.edu

ABSTRACT

Incremental query processing is widely used in data warehouses and streaming systems. While many optimization techniques are developed to generate incremental query plans, the scheduling support for incremental processing remains preliminary. Typically, execution is triggered with fixed frequencies specified by the user. In this paper, we propose a novel scheduling problem for incremental query execution under a deadline, assuming the resource has a fluctuating and unforeseen price. We propose two naive solutions as well as a prophet scheduler that foresees the future. We present an end-to-end system `Agamotto` that models future probabilities offline with a Markov Decision Process (MDP) and makes cost-based and dynamic scheduling decisions online. We show how `Agamotto` can be extended to handle a workflow of dependent queries, so that they can all incrementally execute in an asynchronous fashion. Experiments show that `Agamotto` consistently outperforms the naive solutions, and the achieved cost is on average 10x closer to the theoretical lower bound provided by the prophet scheduler.

PVLDB Reference Format:
Botong Huang, Lianggui Weng, Wei Chen, Zuozhi Wang, Kai Zeng, Chen Li, Yihui Feng, Bolin Ding, and Jingren Zhou. Agamotto: Scheduling of Deadline-Oriented Incremental Query Execution under Uncertain Resource Price. PVLDB, 18(6): 1852 - 1864, 2025.
doi:10.14778/3725688.3725711

1 INTRODUCTION

Incremental query processing is widely used in data processing, including streaming systems [8, 10] and view maintenance [15, 16] in databases. When input data gradually becomes available over time, either as a continuous stream or as intermittent batches, incremental processing can be triggered multiple times, each processing the new delta of the input. While there are works on incremental optimizers [12, 30] that produce incremental plans for user queries, the scheduling support for incremental processing remains preliminary. Executions are typically triggered with fixed frequencies, or whenever certain amount of new input data arrives [2, 8, 22, 26]. Furthermore, the trigger conditions are usually user-specified. Here are two scenarios that entail a more sophisticated scheduler.

Routine Analytics in a Public Spot Market. A spot market in a public cloud (e.g., Amazon [5] or Alibaba Cloud [3]) is a market where the machine price fluctuates over time. The cloud provider decides and announces the machine price as time goes. Users acquire the machines and pay-as-you-go at the market price.

Suppose we need to run two dependent daily reporting queries Q_1 and Q_2 in Figure 1 to analyze business data generated over the day. If they are run as traditional batch jobs when input data becomes fully ready, the monetary cost can be high if the market price of machines spikes at that time. Instead, incremental processing can be used to progressively compute the queries as data becomes available during the day. We can save monetary cost by scheduling the incremental runs whenever the market price is low. Also as Figure 1 shows, dependent queries Q_1 and Q_2 can be scheduled to run incrementally in an asynchronous fashion.

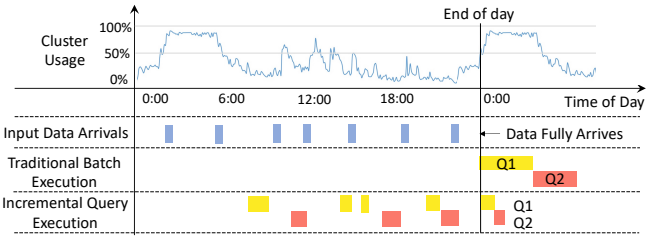


Figure 1: Traditional batch vs. incremental query execution under uncertain resource price (cluster utilization).

Progressive Execution in Alibaba Cloud Data Warehouse [29]. In enterprise data warehouses where compute resources are usually not explicitly priced, cluster capacities are typically managed via user groups and queues. Each user group (e.g., a department) is prebooked with certain amount of capacities according to their budgets. Queries are queued when there is no more available resource for the group, resulting in longer wait time and higher latency.

If one is to run the same reporting queries as batch jobs when data becomes fully ready at midnight, there is a chance of missing the deadline when there are lots of similar queries submitted at that time. In this case, incremental processing is beneficial because it reduces the amount of computation at peak hours. If we interpret the resource usage percentage of the user group as the "price", the scheduler can then schedule the incremental runs during the day

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 18, No. 6 ISSN 2150-8097.
doi:10.14778/3725688.3725711

whenever the utilization is low, also minimizing the possibility of impacting other jobs running then. In general, this scheduler will alleviate the overall resource skew in the cluster.

In this paper, we propose a novel scheduling problem for incremental query execution in a cluster where resource has a fluctuating and unforeseen price. Given a deadline D and a workflow of queries whose input data arrival amount and time are also unknown, the goal is to select k ($k \geq 1$) time points $t_1 \dots t_k \in (0, D]$ and the set of queries to incrementally run at each time point, so that the estimated total weighted cost $\sum_{i=1}^{k} p_i c_i$ is minimized. Here p_i is the resource price at the t_i, and c_i is the estimated execution cost at t_i.

This problem is challenging, mainly due to price uncertainty. Naive solutions such as setting a price threshold and scheduling a run whenever the price falls below the threshold can hardly be good enough: 1) in one day if the price mostly stays below the threshold, we end up with a lot of incremental runs at not-so-low prices, incurring heavy incremental processing overhead at the same time, or 2) in another day the price stays above the threshold, we end up not running at all and leaving all work at the deadline, and the price then can be even higher. If one wants to do better by dynamically making decisions on the fly, then the problem becomes: should we 1) "run," i.e., exploiting the current decent price, or 2) "wait," i.e., betting on a lower price later before the deadline.

Another challenge lies in query characteristics and there is a need for a cost-based scheduler. Some queries are naturally more suitable for incremental execution than others. The scheduler can hardly make wise decisions without understanding the cost characteristics of the dependent queries in the workflow.

At the core of our proposed solution `Agamotto`[1], the scheduling problem is formulated as a Markov Decision Process (MDP) that probabilistically describes different future possibilities with discretized states and transitions. It jointly considers price evolvement probabilities and the cost dynamics of the incremental query execution, and thus is able to reason about questions like: how likely is the price going to drop further and can we afford to wait? Is this worth it to run the query incrementally and incur overhead? To what extent can the early output of upstream queries benefit the early execution of downstream queries? With all these aspects considered, the MDP model produces the best cost-based scheduling decisions that minimize the expectation of the total weighted cost.

In this paper we make the following contributions:

- We propose a novel scheduling problem for deadline-oriented incremental query execution under uncertain resource price in Section 2. Two naive solutions are given in Section 3.
- In a simplified problem where the exact future is known, we propose the *Prophet* scheduler in Section 4 that provides the optimal scheduling. It also provides a theoretical cost lower bound for the full problem with unknown future.
- We present `Agamotto` in Section 5, an end-to-end scheduling solution that dynamically makes cost-based scheduling decisions without knowing the future. `Agamotto` consists of an offline optimization module featuring a Markov Decision Process that models different future possibilities and produces the optimal scheduling policy, and an online agent that dynamically makes scheduling decisions at run time.
- `Agamotto` and its MDP model can be extended to handle a workflow of multiple queries (Section 6).
- We systematically evaluate `Agamotto` in Section 7, by comparing it with the two naive approaches and the *Prophet*.

[1]The eye of Agamotto enpowers Dr. Strange to see through infinite alternative futures.

1.1 Related Work

Scheduling problems focusing on latencies [1, 13, 23] are well studied. While in our problem, we try to minimize the compute cost of incremental query execution. The results are only required at the deadline, without the need to deliver latest results with low latency.

There are limited studies on the scheduling of incremental query processing. Incremental executions are typically scheduled with fixed frequencies, or are triggered by input data arrival or the ending of the execution of an upstream query [2, 8, 22, 26]. Furthermore, the trigger conditions usually have to be specified by the user.

In Spark streaming, [11, 19, 20] try to adjust the size of the micro-batch and the execution frequency, while not violating the latency or SLA requirement. [14] tries to save resources by scheduling certain jobs of the workflow less frequently, as long as the predicted changes of the output are bounded. [21] introduces a more relaxed form of incremental processing allowing temporal asynchrony, and tries to find the schedule that minimizes cost given data freshness and consistency requirements.

Our work is different from all these works on scheduling incremental execution because it solves a new scheduling problem where resource has an unknown price. Nevertheless, [11, 14, 21, 22] are similar to us in that they all consider asynchronous execution of upstream and downstream queries in a workflow.

[26] studies how to choose intermediate states for incremental query execution under a memory constraint and minimize the refresh latency upon new input data. [24, 27] study the incrementability of each query path inside a query, and how to pick the best execution frequency of each path to minimize the total execution cost given a final-run-cost constraint. [28] considers multiple queries with different frequency requirements, and picks the best sharing plan with refresh frequencies for different parts of the plan. In all these works, incremental runs are scheduled with fixed frequencies or when a certain number of tuples arrive. [25] develops scheduling algorithm for incremental maintenance of Datalog programs, and focuses on the schedule order of the affected nodes in the computational DAG of the Datalog program when inputs have changed. Cümülön-D [18] tries to minimize the monetary cost of running statistical workloads using Amazon EC2 spot instances [5]. Its scheduler problem is similar to this paper, but the workload and problem setting are quite different. Matrix workload is a fixed amount of malleable work which can be finished early, while the input data in our setting will not be fully ready until the deadline.

In Grosbeak [29] we envisioned the scenario where incremental execution is used to reduce peak traffic in the data warehouses. This paper is the full research version that solves the scheduling problem which is the key and the most challenging module in Grosbeak.

2 PROBLEM FORMULATION

We start with background settings on incremental query processing in a data lake, then describe the concept of an incremental plan template, and define the scheduling problem studied in this paper.

Background. Image that we are doing incremental query processing in a data lake. Live sales data is ingested into Apache Hudi [6] over time, which manages different time versions of tables and allows us to access certain version of a table at any given time (time travel), and the changes (aka. delta) of a table between any two time points. Apache Spark [7] is used to incrementally process a two-query workflow shown in Figure 2, where Q_1 filters input table *Sales* and writes to table *Items*, and Q_2 aggregates over *Items* and reports the average amount per group.

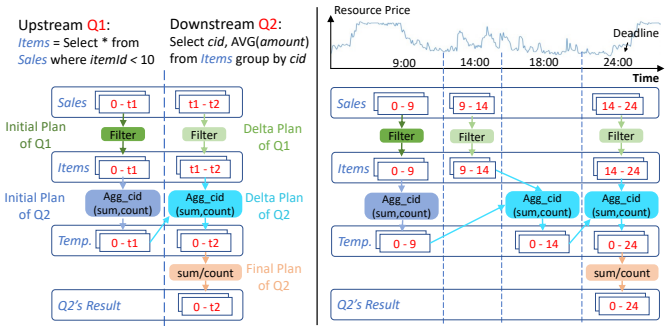


Figure 2: The Incremental plan template (left) vs. a full execution plan (right) of an example two-query workflow.

Q_1 can be incrementally computed as is. While to process the aggregation in Q_2 over partial input, we need to compute the sum and count instead of the average, and save the partial aggregation result as a temporary table *Temp*. When data becomes fully ready, the final average amount can be computed by dividing the sum by the count over table *Temp*.

As the left of Figure 2 shows, the incremental plan involves two time points t_1 and t_2. The initial plans of both queries run at t_1, they consume the entire input tables available then and produce the result tables so far. While in the delta plan at t_2, Q_1 reads from *Sales* only the new input rows arrived between t_1 and t_2, computes the filter and appends the results to table *Items*. Q_2 reads the new rows in *Items*, combines them with the previous partial result in *Temp*, and update the new sum and count per group back into *Temp*. Lastly, Q_2 has a final plan to compute and report the average.

Incremental Plan Template. In general, the incremental plan template of each query Q_i in the workflow has three parts:

$$\begin{aligned} &\text{The initial plan} && P_i^I(t_1): && D_{t_1} \rightarrow S_{t_1} \\ &\text{The delta plan} && P_i^D(t_1,t_2): && \Delta D_{t_1,t_2}, S_{t_1} \rightarrow S_{t_2} \\ &\text{The final plan} && P_i^F(t_2): && S_{t_2} \rightarrow R_{t_2} \end{aligned}$$

The initial plan consumes all input data D_{t_1} at t_1 and produces a partial state result S_{t_1}; the delta plan consumes the delta from t_1 to t_2, and updates the partial result; the final plan at t_2 converts the partial result into the query result. In the above example, Q_1 has an empty final plan while Q_2 has all three parts. Note that we call it a template because it will be repeatedly instantiated by the scheduler at run time, replacing t_1 and t_2 with real execution times.

A full and valid incremental execution of each query starts with exactly one run of the initial plan followed by zero to multiple runs of the delta plan. Whenever the query result is required, the final plan can be appended to the end of the initial or delta plan.

Figure 2 illustrates how the incremental plan template (on the left) can be assembled into a full incremental execution (on the right) throughout a day. The initial runs of Q_1 and Q_2 are scheduled at 9:00. Then Q_1 and Q_2's delta runs are scheduled to run at 14:00 and 18:00 respectively. At 24:00, both plans with the final plan added are scheduled for both queries to produce the final results.

In this paper, we assume the incremental plan template is given. It can be manually written (e.g. with Spark API), or generated by an incremental query optimizer. This optimizer can be rule-based as commonly used in streaming systems [8, 10] and incremental view maintenance in traditional DBMS [15, 16]. It can also be cost-based such as Tempura [30] and Enzyme [12], which choose a best plan with the lowest cost among various incremental algorithms.

Scheduling Problem Definition. Now we formally define the scheduling problem studied in this paper. We want to incrementally execute a workflow of N dependent queries $Q_1, \cdots, Q_N$, whose input data gradually arrives between time zero and a given deadline D (e.g. 24 hours). We assume compute resources have a fluctuating and unforeseen price, and the input data arrival time and statistics are also unknown in advance.

The scheduler's goal is to pick n_i ($n_i \geq 1$) execution time $t_1^i \cdots t_{n_i}^i \in (0, D]$ for each query Q_i so that the estimated total weighted cost

$$C_w = \sum_{i=1}^{N} \left(cost(P_i^I(t_1^i)) \times p(t_1^i) + \sum_{s=2}^{n_i} cost(P_i^D(t_{s-1}^i, t_s^i)) \times p(t_s^i) + cost(P_i^F(t_{n_i}^i)) \times p(t_{n_i}^i) \right)$$

is minimized. Here $p(t)$ is the resource price at time t.

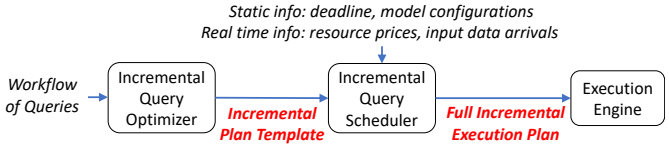


Figure 3: The scheduling problem setup.

Figure 3 shows the system setup for the scheduler. After the incremental plan template for each query is generated, the scheduler takes in static information like the deadline and real-time information such as resource price and input data arrival so far, makes scheduling decisions (which queries to run at the moment) and submit the executions into the underlying engine.

3 BASIC APPROACHES AND LIMITATIONS

Intuitively, there are two important aspects when minimizing C_w:

(1) It is preferable to schedule the incremental runs at the times when the price is low, but the imminent problem is that we do not know the exact future prices.
(2) Incremental execution incurs computation overhead. Frequent incremental runs lead to wasted computation. As a result, we prefer to process sufficient amount of new input data at each run, and reduce the total number of runs.

In this section, we provide two naive solutions and discuss their limitations. Assuming the input data comes in continuously and

steadily, Figure 4 demonstrates these two approaches using the two-query workflow. We will introduce the *Prophet* in Section 4.

A fixed-time scheduler runs at pre-determined frequencies, regardless of price evolvement and data arrival status. For instance in Figure 4, it runs Q_1 six times and Q_2 three times. When both queries need to run, e.g., at 8:00, Q_1 and Q_2 are executed sequentially. As we can see, this approach can hardly achieve a low cost because it runs at random prices and can also incur significant incremental overhead because of the large number of runs.

A fixed-threshold scheduler is a greedy approach that schedules a new run whenever the price falls below a threshold θ and the amount of new input accumulated goes above a threshold δ. Figure 4 shows its decisions if both queries use the same threshold values. It scheduled both queries three times when resource price stays below θ between 2 to 11 o'clock. Note that at the deadline, it has to schedule a final run for both queries to deliver the final result, even though the price is above the threshold.

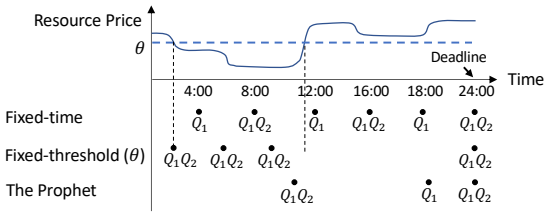


Figure 4: Example scheduling decisions made by three scheduling approaches for the two-query workflow under a specific price trace. For each query, the first run would use the initial plan and all the rest runs use the delta plan

The fixed-threshold approach may perform better than the fixed-time scheduler, but it is hard to determine the best threshold for each workflow. Also there are missed opportunities, mainly due to the large variances in price trends. The price usually tends to be higher/lower in different clusters and at different times. Even within the same cluster at the same time of the day, the price can be high on one day and low on another day. In general, if the fixed price threshold is set to be too high, we end up with a lot of runs (thus a high incremental overhead) at not-so-low prices. On the other hand, if the threshold is set to be too low, we end up not running any job at all, leaving more work at the deadline when the price can be high. Figure 4 gives an example where it hits both problems (e.g., the threshold is too high from 0:00 to 13:00 and too low from 13:00 to 24:00) even on the same day.

4 THE PROPHET SCHEDULER IN A REPETITIVE ENVIRONMENT

Before tackling the full problem in Section 2, we start by considering a simplified problem where future prices and input data arrivals are exactly known beforehand. This setting applies in scenarios where the price is pre-determined by fixed rules, or where both the price and the workload are highly repetitive with minimum variances. We also assume there is only one query in the workflow. Later we will show how the insights learned in this setting can help us solve the harder problem with an unknown future.

It turns out that we can statically find the optimal scheduling plan that minimizes the total weighted cost using dynamic programming. We call it the *Prophet* scheduler. Continuing the example in Figure 4, it typically chooses to run at local minimums of the price trace and has a low number of runs overall. During low-price windows, e.g., around 6-11 o'clock, it always schedules the run at the end of the time windows in order to consume more input data at the same low price. Any other scheduler that cannot foresee the future obviously cannot do as well as the *Prophet*, thus the *Prophet*'s performance provides a lower bound of the cost for the problem.

Next we present the solution of the *Prophet* in two steps.

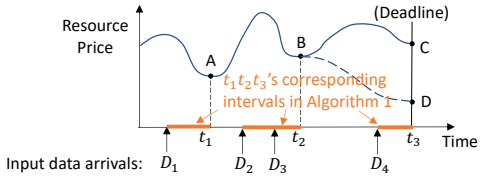


Figure 5: The Prophet's perspective, with known future prices and data arrivals (D_1 to D_4).

4.1 Step One: Finding the Time Candidates

Given the future price trace and data arrival time points, we can determine the set of candidate time points that the optimal scheduling would possibly choose from, without looking at the specific query and data characteristics. For example, in Figure 5 if the price trace evolves as $A \rightarrow B \rightarrow C$, it makes sense to possibly run at t_1, t_2, and t_3 due to the following reasons:

(1) If there is unprocessed data at t, then it is best to process the data at the time with the minimum price between t and the deadline, e.g., D_1 at t_1, D_2, D_3 at t_2, and D_4 at t_3.
(2) Another execution would be of interest only if there is unprocessed data. If we choose to process D_1 at t_1, then it does not make sense to run again between t_1 and D_2.

Algorithm 1 Deriving the set of execution-time candidates.
Input: Deadline D, price trace $price(t), t \in (0, D]$, and the set of time points with new input data arrivals.

1: $\mathcal{S} \leftarrow \emptyset$ // the set of execution-time candidates.
2: $x \leftarrow 0$
3: **while** $x \neq D$ **do**
4: $x \leftarrow$ first time point in $(x, D]$ with a new data arrival
5: $y = \arg\min_{t \in [x,D]} price(t)$ (Pick the largest t if there are multiple t's with the same minimal price(t)).
6: Add y into $\mathcal{S}$. // candidate y's corresponding interval is $[x, y]$
7: $x \leftarrow y$
8: **end while**
9: **return** $\mathcal{S}$

Based on this observation, we develop Algorithm 1 for finding the execution time candidates. Starting from time zero, it greedily collects the minimum price point from the last candidate (adjusted to the time with new input data, line 4) to the deadline as the next candidate, until it reaches the deadline. Note that we assume the actual execution is relatively fast with respect to the time scope of the deadline, so that it will finish before the next scheduled run.

Theorem 4.1. *We can achieve the minimum total cost by limiting executions at the time points given by Algorithm 1.*

Proof. For any optimal scheduling plan $\mathcal{P}$ that achieves minimum total cost C_w, we complete the proof by constructing another plan $\overline{\mathcal{P}}$ that only schedules at the candidate time points in $\mathcal{S}.keys$ from Algorithm 1 and that has cost no larger than that of $\mathcal{P}$.

Starting from $\mathcal{P}$, we construct $\overline{\mathcal{P}}$ by doing the following for each execution time $\hat{t}$ in $\mathcal{P}$ that is not in $\mathcal{S}.keys$. Consider the k disjoint intervals $[x_i, y_i]$ in $\mathcal{S}.values$:

(1) If $\hat{t}$ lies within one of the intervals $[x_i, y_i]$. Since $price(y_i)$ is the price minimum in $[x_i, D]$ (line 5 in Algorithm 1), we have $price(y_i) \leq price(\hat{t})$, then scheduling this execution at time y_i instead of $\hat{t}$ is no worse than $\mathcal{P}$.
(2) If $\hat{t}$ lies between intervals $[x_{i-1}, y_{i-1}]$ and $[x_i, y_i]$. Similarly since $y_{i-1} < \hat{t} \leq D$, we have $price(y_{i-1}) < price(\hat{t})$. Since there is no data arrival during (y_{i-1}, x_i) (line 4 in Algorithm 1), Scheduling this execution earlier at time y_{i-1} instead of $\hat{t}$ generates a better plan with lower cost.

□

Note that some of these candidate time points may not appear in the optimal schedule. This is because a lower number of runs incurs less incremental processing overhead. For instance, in Figure 5, although t_1 has a lower price than t_2, for certain query it might be better to process D_1, D_2, and D_3 altogether at t_2 instead of t_1.

4.2 Step Two: Finding the Optimal Plan

Given the execution time candidates (e.g., t_1, t_2, and t_3 in Figure 5), we can collect cost information of the query and statically compute the optimal scheduling plan.

This problem can also be solved using dynamic programming. Notice that a query's subsequent incremental runs share identical plans, instantiated from the plan template (see Section 2), thus information about the last incremental run suffices to capture the query's execution status and its implications for later runs, e.g., the total amount of input already processed and the amount of output already produced so far. Knowing all prices and input data statistics at all time points, we can use the time of the last run to uniquely identify the incremental processing state of the query.

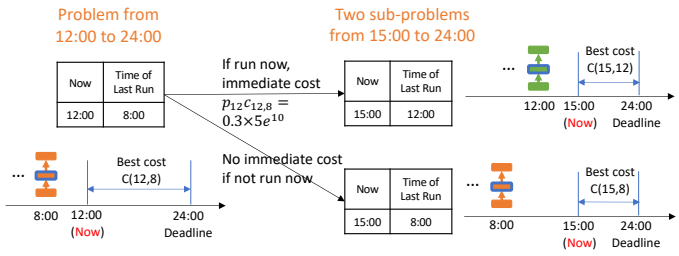


Figure 6: Dynamic programming state transitions for solving the optimal scheduling plan.

For instance, assume in Figure 6 that 24:00 is the deadline, and 8:00, 12:00, and 15:00 are the execution time candidates derived by Algorithm 1. Consider the problem that we are at 12:00 and the last run is at 8:00. We want to find the optimal scheduling between 12:00 and 24:00 that minimizes the total cost of the rest of the executions. There are two choices we can make at 12:00, i.e., to run the query or to wait. If we run the query, an immediate cost is incurred weighted by the price at 12:00, and then have the sub-problem at 15:00 with the last run at 12:00. If we choose not to run the query, then we have the sub-problem at 15:00 with the last run at 8:00. Note that both sub-problems are of smaller sizes.

Formally, given the candidates time points $t_1 \ldots t_k$, let $C(i, j)$ denote the minimal cost of finishing the rest of the execution from t_i, when the time of the last run is t_j ($j < i$, $j = 0$ means no earlier execution). We have the following DP transition function:

$$C(i,j) = \begin{cases} min\{C(i+1,j), p_i c_{ij} + C(i+1,i)\} & \text{if } 1 \leq i < k; \\ p_i c_{ij} & \text{if } i = k. \end{cases}$$

In the function, c_{ij} is the estimated cost of the incremental run at t_i given the last run is at t_j. At time t_i, we choose the better option from (i) not running the query and transitioning to $C(i+1, j)$, and (ii) running the query and incurring a cost of $p_i c_{ij}$ and transitioning to $C(i+1, i)$. Whereas at t_k, we must schedule the last run to process the last piece of data and produce the final result.

From the last time point backwards, this DP process can be solved in $O(k^2)$ time, where k is the number of the candidate time points.

5 AGAMOTTO: AN MDP-BASED SCHEDULER

Now we return to the full problem where future prices and data arrivals are unknown beforehand. We introduce Agamotto, an MDP-based dynamic cost-based scheduler. We discuss its rationale in Section 5.1 and an overview in Section 5.2. Starting with single-query workflows, we describe the MDP model in Section 5.3, and derive the optimal policy offline in Section 5.4. In Section 5.5 we discuss how the online agent turns the policy into dynamic scheduling decisions at run time. Later in Section 6, we discuss how to extend the model to handle multi-query workflows.

5.1 Rationale of the MDP Approach

Note that without knowing the future, it is simply impossible to achieve the same performance as the *Prophet*. In Figure 5's example, if the price trace evolves along points $A \rightarrow B \rightarrow C$, the *Prophet*'s plan is to execute at t_1, t_2, and t_3. However, if the price evolves along $A \rightarrow B \rightarrow D$ instead (the dotted line), then the *Prophet*'s plan becomes to run everything in one shot at t_3. For the two traces that divert at t_2, their optimal schedulings differ at an earlier time t_1. Without knowing the future, at t_1 we cannot know which future it will be at t_2. Whatever scheduling decision we make at t_1, we will achieve a worse performance than the *Prophet* in the other case.

Nevertheless, it is not hopeless. The main idea behind Agamotto is that although we do not know which future it will be, we can model different future outcomes with probabilities, and try to minimize the expectation of the total future cost. Following this idea, the DP process in Section 4.2 naturally upgrades into a *Markov Decision Process* (MDP) that models future possibilities with a set of discretized states and the transitions in between.

As later shown in Section 7.3, Agamotto can be interpreted as an improvement from the fixed-threshold scheduler (Section 3) towards the *Prophet* (Section 4), where the MDP model pre-calculates

the best threshold price to use at all possible future time and execution progress, and the plan is dynamically applied at run time.

5.2 Overview of Agamotto

Figure 7 gives an overview of Agamotto, which consists of an offline MDP optimization module, and an online scheduling agent that dynamically makes scheduling decisions at run time.

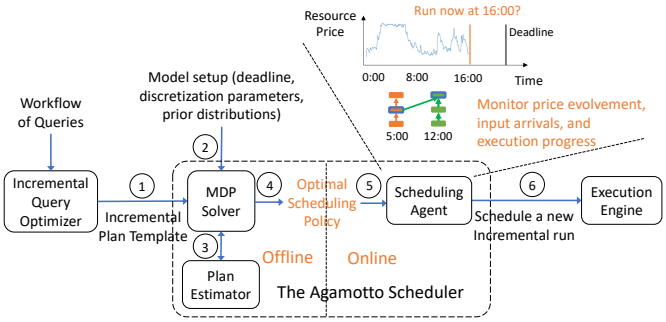


Figure 7: An overview of the Agamotto Scheduler.

Besides the plan template (①) generated by the incremental query optimizer, the offline model also takes in additional configurations (②), including the deadline D as well as model setups such as discretization parameters and prior distributions (later in Section 5.3). Next, an MDP model is built to describe the space of all possible futures possibilities with a set of discretized **states** and the **transitions** in between. It makes a series of calls to a plan estimator (③) to obtain estimated cost and output cardinality of the incremental plans instantiated from the template under different input data sizes, and incorporate the information into the MDP network. The plan estimator can be the cost/cardinality estimation component of any typical cost-based query optimizer (e.g., Apache Calcite [9]). We omit the details due to space limitation.

The model is then solved offline, where each state chooses a best **action** so that the expected remaining cost of the state is minimized. The produced result (④) is the **optimal scheduling policy**, a mapping from states to their best actions.

At run time, an online scheduling agent is created for each instance of the workflow, e.g., one workflow instance per day. It remains active from the time of the first input data arrival to the deadline, during which it constantly monitors the price evolvement, input data arrival status, as well as the execution progress of the workflow instance. It also consults the optimal policy (⑤), and makes dynamic scheduling decisions on the fly. Whenever the decision is to run certain queries, it instantiates a new plan from the plan template, either the initial plan or the delta plan, and schedules the new incremental run (⑥) for the selected queries.

Reuse of the Optimal Policy. The lifecycles of the offline and online modules of Agamotto are different. For workflows that need to be repeatedly executed (e.g., daily routine queries), the offline MDP solver only needs to be trained once, and the produced optimal scheduling policy can be reused by all future online scheduling agents created for the workflow instances. Only if there is new information or changes on the prior distributions of the price evolvement and/or data arrival patterns, a new scheduling policy needs to be re-trained from the offline MDP model.

5.3 MDP Modeling

Now we formally describe the MDP model. Firstly, the time before the deadline was discretized into T time steps $\tau_1, \ldots, \tau_T$, and the range of possible prices into P values $\delta_1, \cdots, \delta_P$. Input data characteristics to be modeled, e.g. estimated row count, also need to be discretized. Let **data version** vector $\overrightarrow{D}$ represent the data characteristics of the query's input tables at a certain time.

Secondly, we define **prior distributions** of the price evolvement ($\mathbb{P}_p$) and input data arrival ($\mathbb{P}_d$). Both are inputs to the MDP model. Specifically, $\mathbb{P}_p(\delta_j|t, \delta_i)$ denotes the conditional probability that the price at τ_{t+1} is δ_j given that the price at τ_t is δ_i, and $\mathbb{P}_d(\overrightarrow{\Delta D}|t)$ denotes the probability of receiving a new delta of size $\overrightarrow{\Delta D}$ (difference from $\overrightarrow{D_{\tau_{t-1}}}$ to $\overrightarrow{D_{\tau_t}}$) in the input tables from τ_{t-1} to τ_t.

As mentioned in Section 5.1, the information of the last incremental run alone suffices to capture and identify the status of the incremental processing. Specifically, we include $\overrightarrow{D^L}$ the data version vector at the time of the last run in the MDP state, describing the total amount of input data available at the last run.

Now we can define the **state** in the MDP as a quadruple

$$S = \{t, \overrightarrow{D}, \delta_i, \overrightarrow{D^L}\},$$

which includes the time step t (e.g. time point τ_t), the resource price (δ_i), the input data version available then ($\overrightarrow{D}$), and the input data version at the last run ($\overrightarrow{D^L}$). If there is no previous run at all, e.g., at the beginning of the process, we have $\overrightarrow{D^L} = \mathbf{0}$.

At state S, we can choose an **action** $A \in \{0, 1\}$ indicating whether to schedule an incremental run at the moment, receive an **reward** $c_t(S, A)$ which is the estimated cost of the incremental execution, and then with probability $\mathbb{P}(S'|S, A)$ **transition to** a new state S' from a set of states at time step $t + 1$. Formally, at state S,

- If we do nothing and wait ($A = 0$), then the immediate cost $c_t(S, 0) = 0$. We will transition to a next state $S' = \{t + 1, \overrightarrow{D} + \overrightarrow{\Delta D}, \delta_j, \overrightarrow{\boldsymbol{D^L}}\}$ with a probability $\mathbb{P}(S'|S, 0) = \mathbb{P}_d(\overrightarrow{\Delta D}|t + 1) \times \mathbb{P}_p(\delta_j|t, \delta_i)$. Here $\overrightarrow{\Delta D}$ and δ_j are random variables.
- If we run ($A = 1$), we incur an estimated cost $c_t(S, 1)$ under the current price δ_i, and transition to $S' = \{t + 1, \overrightarrow{D} + \overrightarrow{\Delta D}, \delta_j, \overrightarrow{\boldsymbol{D}}\}$ with a probability $\mathbb{P}(S'|S, 1) = \mathbb{P}_d(\overrightarrow{\Delta D}|t + 1) \times \mathbb{P}_p(\delta_j|t, \delta_i)$.

Figure 8 gives an example of the state transitions in the MDP model, where in state $\{t, \overrightarrow{D}, \delta_i, \overrightarrow{D^L}\} = \{8, 7e^8, 0.3, 2e^8\}$, $7e^8$ is the estimated row count of the query's single input table at 8 am. Depending on the action taken at 8 am, we incur certain immediate cost, and probabilistically transition to a set of states at 9 am. Note that the estimated cost of the incremental run ($5e^{10}$) needs to be multiplied by the current price 0.3 when accounted for C_w.

As mentioned in Section 4.2, since delta plans instantiated from the template are identical, the incremental processing satisfies the Markov property, that the information (i.e., time and amount of input data) of the last run suffices to capture the processing status and there is no need to look back further in the execution history.

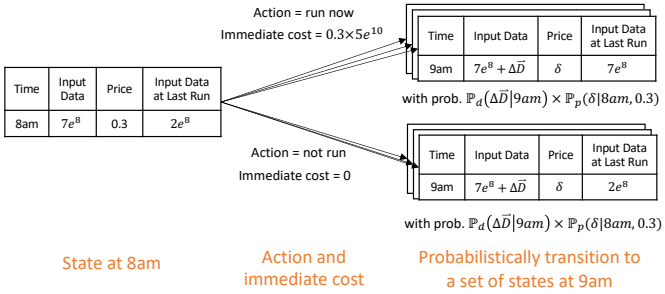


Figure 8: An illustration of the MDP state transitions.

5.4 Finding the Optimal MDP Policy Offline

Both the prior distributions of price ($\mathbb{P}_p$) and input data arrival ($\mathbb{P}_d$) can be trained using historical data, or manually constructed without prior knowledge. See more discussions in Section 5.6 and experiments on prior sensitivity in Section 7.

Actions at all states are summarized into an MDP **policy** $\mathcal{L} : S \rightarrow A$, which maps any given state S to an action A. Let $C(S, \mathcal{L}|A)$ denote the additional cost to finish the remaining process if we take action A at state S and then follow policy $\mathcal{L}$ afterwards. Let $\mathbb{C}(S, \mathcal{L}|A)$ denote the expectation of this random variable. We have:

$$\mathbb{C}(S, \mathcal{L}|A) = \delta_i \times c_t(S, A) + \sum_{S'} \mathbb{P}(S'|S, A) \times \mathbb{C}(S', \mathcal{L}|\mathcal{L}(S')).$$

`Agamotto` looks for the **optimal scheduling policy** $\overline{\mathcal{L}}$ that takes the action that minimizes the state's expected remaining cost:

$$\overline{\mathcal{L}}(S) = \operatorname{argmin}_A \mathbb{C}(S, \overline{\mathcal{L}}|A).$$

Note that for the states at the last time step, we should always take the action of run, which concludes the incremental execution.

This is an MDP model with finite number of states and time steps, which can be **statically solved via DP**, similar as in Section 4.2. Starting from the last time step, we work backwards in time. For each state we compute the remaining expected cost for all actions and choose the best one that minimizes the cost expectation.

Let V denote the maximum number of discretized values of $\overrightarrow{\Delta D}$ during each time step in $\mathbb{P}_d$. Then with T time steps, $\overrightarrow{D}$ and $\overrightarrow{D^L}$ can take $O(TV)$ distinct values. If there are P discretized price values, then the size of the state space is $O(PT^3V^2)$. For each state, we compute the summation of complexity $O(PV)$ in $\mathbb{C}(S, \mathcal{L}|A)$ twice, for $A = 1$ and 0 respectively. As a result, the overall complexity of solving the MDP is $O(P^2T^3V^3)$.

The bottleneck of solving the MDP lies in calls to the plan estimator to collect the immediate cost $c_t(S, 1)$ for every state S. For the given initial plan or delta plan in the plan template, the cost only changes with $\overrightarrow{D}$ and $\overrightarrow{D^L}$, so they can be pre-collected with $O(T^2V^2)$ calls to the plan estimator.

5.5 `Agamotto`'s Online Scheduling Agent

Interpolating the optimal policy. The optimal policy is not enough in practice, because it only encodes the suggested actions at discretized states (i.e., time, price, amount of input data), while the real world is continuous and events can happen at any time.

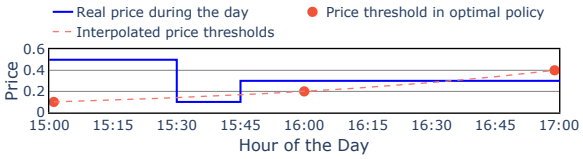


Figure 9: An example trace of price evolvement during a day.

Figure 9 gives an example price trace where the price drops to 0.1 from 15:30 to 15:45. However the policy has no direct information on whether we should run at that time, because the MDP model only considered 15:00 and 16:00 during training. The same problem also applies in the dimensions of the price and input data vectors.

Although the MDP model solves the problem in a sampled future space, we can interpret its resulting policy as a rough sketch of the high-level trend of how decisions should be made in the continuous actual space. We resort to **interpolation** to turn the discretized policy into a continuous one. For instance in Figure 9, the interpolated line shows we should run at 15:30 and 16:35 respectively.

Algorithm 2 Execution Logic of the Online Scheduling Agent.

1: **while** current time < deadline D **do**
2: Block and wait for a qualifying event (e.g., a price change, new data arrival, execution failure, sufficient time since the last event)
3: Collect latest status (i.e., current time, price, $\overrightarrow{D}$ and $\overrightarrow{D^L}$) and ask the interpolated optimal policy whether to run now.
4: If positive, instantiate a plan from the template and schedules a run.
5: **end while**
6: Schedule the last final run, if there is unprocessed input (i.e. $\overrightarrow{D} \neq \overrightarrow{D^L}$)

Applying the Policy at Runtime. `Agamotto`'s online scheduling agent (recall in Figure 7) is built to run Algorithm 2, where it constantly observes the real time situations, consults the interpolated policy and dynamically makes decisions at run time. Specifically, it keeps waiting for qualifying events, which can be a price change, sufficient amount of input data accumulated, execution failure of an incremental run, or simply that certain amount of time has passed since the last qualifying event. Whenever such an event happens, the agent attempts a new decision cycle for the query, where it collects the latest status (time, price, $\overrightarrow{D}$ and $\overrightarrow{D^L}$), asks the interpolated MDP optimal policy whether to run now. If the answer is yes, it instantiates a new plan from the initial or delta plan template (Section 2), and schedules a new incremental execution.

5.6 Discussions

Cost-based Approach. Despite the challenges of imprecise or unknown data statistics, cost-based query optimizers have achieved great success thanks to decades of research on cardinality estimation and cost models. Recently [30] shows that the cost-based Cascades framework can be extended to generate high quality and complex incremental plans. `Agamotto` follows the cost-based approach and solves the scheduling problem for incremental processing.

Repetitive Scenarios. In large-scale data-processing platforms, many routine jobs have repetitive patterns. The cluster-wise daily

resource usage pattern is usually even more obvious. Priors built using historical data in these scenarios often have a high quality.

Low Sensitivity and Robustness. In cases when historical data is not available, one can manually construct a prior from experience, or simply use a neutral prior of a uniform distribution. Later in experiments we will show that thanks to the MDP model's adaptive nature and Agamotto's design to constantly observe, adjust, and adapt on the fly, it is not as sensitive to the quality of the prior and cost model as one would imagine. Its performance remains competitive across high quality priors trained from historical data, neutral priors and misleading priors. How to further construct a better prior and/or cost model is not the focus of this paper.

Runtime Flexibility. Lastly, Agamotto is flexible and dynamic at runtime. The MDP model training, with updated priors and/or new queries, can be done offline in parallel any time. The resulting policy can be updated into the online agent any time during the execution process. We will showcase this flexibility in Section 7.5.

Cluster-wise Impact on Price. We assume that Agamotto's schedule decisions will not impact the resource price. As future work, we will relax the assumption by considering the case where a significant number of workflows are concurrently scheduled by Agamotto.

6 AGAMOTTO: MULTI-QUERY WORKFLOWS

In this section, we show how Agamotto can be extended to handle a workflow of multiple dependent queries, for instance, the one in Figure 10 with two base tables and four queries. When opportunities appear, both upstream and downstream queries can be scheduled to run asynchronously, as shown in Figure 1. As a result, the scheduler needs to jointly consider all query characteristics and the workflow structure in order to make wise decisions.

We describe the MDP model extension in Section 6.1 and show how to reduce the model complexity in Section 6.2. In Section 6.3 we discuss the online agent's nuances for multi-query workflows.

6.1 MDP Model Extension

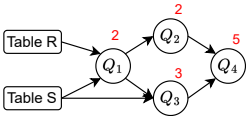


Figure 10: A workflow of four queries and two input tables. Each query has a complexity score (Section 6.2) in red.

Although the incremental dynamics of each query can be individually analyzed by the plan estimator, the offline MDP model needs to be extended to describe the overall behavior of the entire workflow, in order to minimize the workflow's total cost.

Recall in Section 5.3 that the data version vector $\overrightarrow{D^L}$ in the MDP state represents the data characteristics of all the input tables at the time of the last run. It needs to be extended to fully capture the progress status of all queries, rather than one query.

In Figure 10, other than R and S, the data versions of output tables of Q_1, Q_2, and Q_3 (denoted by O_1, O_2, and O_3) should also be included in $\overrightarrow{D^L}$ because they are also input to certain queries and thus are necessary to represent the last run's status of those queries. Note that table S is consumed by both Q_1 and Q_3, and their last run could have consumed two different data versions of table S. As a result, S needs to appear twice in $\overrightarrow{D^L}$ for the status of Q_1 and Q_3, respectively. The same reasoning applies to table O_1 (output of Q_1) because the table is consumed by both Q_2 and Q_3. Consequently, the extended $\overrightarrow{D^L}$ for this example workflow becomes:

$$\overrightarrow{D^L} = [\overrightarrow{D_1^L}, \overrightarrow{D_2^L}, \overrightarrow{D_3^L}, \overrightarrow{D_4^L}] = [\underbrace{R, S}_{Q_1}, \underbrace{O_1}_{Q_2}, \underbrace{O_1, S}_{Q_3}, \underbrace{O_2, O_3}_{Q_4}],$$

Formal Extension. Suppose there are N queries $Q_1, \ldots, Q_N$ in the workflow. Let $\overrightarrow{D_i}$ and $\overrightarrow{D_i^L}$ denote the data version vector of Q_i's input tables at the moment and at the last run, respectively. The overall state in the MDP becomes:

$$\{t, \overrightarrow{D_1}, \ldots, \overrightarrow{D_N}, \delta_i, \overrightarrow{D_1^L}, \ldots, \overrightarrow{D_N^L}\}.$$

At each state, the action becomes $\overrightarrow{A} = [a_1, \ldots, a_N]$, $a_i \in \{0, 1\}$, representing whether to schedule an incremental run for each query at τ_t. The action will incur immediate execution costs estimated by the plan estimator. Next we transition to a set of states at τ_{t+1}:

$$\{t + 1, \overrightarrow{X_1}, \ldots, \overrightarrow{X_N}, \delta_j, \overrightarrow{Y_1}, \ldots, \overrightarrow{Y_N}\},$$

where $\overrightarrow{Y_i} = \overrightarrow{D_i^L}$ if $a_i = 0$, or $\overrightarrow{Y_i} = \overrightarrow{D_i}$ otherwise ($i = 1 \ldots N$). The $\overrightarrow{X_i}$'s are the new input version vectors for the queries at the next time step. The $\overrightarrow{X_i}$ entries have two cases:

- The ones corresponding to the base input tables (e.g., S and R) are random variables drawn from the input data's prior distributions (see Section 5.3).
- The ones corresponding to upstream output tables (e.g., O_1) are updated depending on whether the upstream query is scheduled to run at τ_t. There are deterministic values, estimated by the plan estimator.

Other parts of the offline MDP model in Sections 5.3 and 5.4, including prior distributions and solving the MDP to derive the optimal scheduling policy, remain the same.

6.2 Reducing Model Complexity by Compressing Data Versions

For each query in the workflow, the plan estimator produces the estimated cost and output-data statistics given different input data configurations, without worrying about other queries in the workflow. As a result, this process remains local to each query and can be done for each query independently. However, the number of data versions in $\overrightarrow{D_i}$ and $\overrightarrow{D_i^L}$, and thus the number of states in the MDP network, can grow with the complexity of the workflow.

In Figure 10, if both S and R have three data versions, Q_1's output will have nine different versions. Q_2 will have 9 versions and Q_3 will have 27 versions. Q_4's output will have $27 \times 9 = 243$ versions. In general, whenever a query takes multiple inputs, the number of data versions grows and the MDP state space increases.

For this purpose, we define a **complexity score** for each query, measuring the number of repeating base tables in this query's $\overrightarrow{D_i}$. For example, Q_1 in Figure 10 takes two tables as its input, thus has a complexity score of two. Q_3 gets the two from Q_1 and one from table S directly, thus has a score of three. The complexity score of a workflow is defined as the maximum score of the query in the workflow, e.g., 5 for the example workflow. `Agamotto` employs a data version compressing technique over each table's version evolvement graph to reduce MDP model complexity.

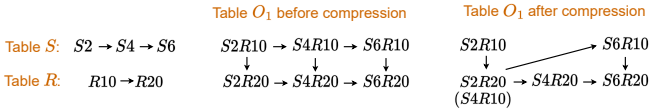


Figure 11: The version evolvement graph of R, S and Q_1's output table O_1 in Figure 10 before and after data version compression. $R10$ means R's data version with row count 10.

Version evolvement graph of a table is a graph where each node represents a data version and a directed edge means a possible table version evolvement over time. Take query Q_1 and its input tables S and R in Figure 10 as an example. Figure 11 shows their evolvement graphs. Assume S has three data versions ($S2$, $S4$, and $S6$) and R has two versions ($R10$ and $R20$). Here 10 and 20 denote discretized estimated row counts of table R. In table S's evolvement graph, $S2$ has outgoing edges to $S4$ and $S6$, which means if table S is at version $S2$ now, then it can evolve into version $S4$ or $S6$ later. Since the version evolvement relationship is transitive and forms a partial order, for clarify of presentation we only draw the immediate edges such as $S2 \rightarrow S4$, $S4 \rightarrow S6$, and omit long ones such as $S2 \rightarrow S6$.

If we collect for Q_1 all version pairs from both inputs, we arrive at the evolvement graph of the output table O_1 with 6 versions as shown in Figure 11. Long edges like $S2R10 \rightarrow S6R20$ are omitted.

Data version compressing. We can reduce the number of data versions of a table by merging nearby nodes in its evolvement graph. This sacrifices model quality so that the model is manageable.

We define version similarities by comparing their costs. For instances, if the query is to join both input tables, then producing $S2R20$ and $S4R10$ might take about the same amount of work, and their costs in the MDP network and implications for future states can be similar. Merging these two versions into one compromises least information in the MDP network.

Figure 11 shows the resulting simplified graph if $S2R20$ and $S4R10$ are merged. Note that in general, if cycles are formed after a merge, then all nodes in the cycle need to be further merged into a single node as well to maintain the partial order property. We need to keep merging until no more cycles are found in the graph.

We define a global threshold N_{DV}. For each query in the workflow with more than one input tables, we keep merging its data versions with the highest similarity until the number of data versions in its version evolvement graph drops below N_{DV}. In this way, every intermediate table in the workflow will have at most N_{DV} data versions exposed to its downstream queries.

6.3 Online Agent for Multi-Query Workflows

When the MDP model in Section 6.1 suggests to run dependent upstream and downstream queries at the same time step, it assumes that both queries are charged at the same current price. However in practice, downstream queries might not have any new data to process until the upstream finishes the new run. As a result, the price may have changed when the downstream query starts.

For the online agent, one option (denoted by **Agamotto-Strict**) is to strictly follow the suggestions from the optimal scheduling policy, and run dependent queries one at a time in a row, regardless of the possible price change later during the execution. Another option (denoted by **Agamotto-Flex**) is that when the upstream query finishes, we re-evaluate the overall status, consult the policy again, and see if it is still worthwhile to run the downstream and/or any other query under the possibly changed price then.

Recall in the single-query case in Section 5.5, a new decision cycle is triggered whenever a qualifying event happens, which can be a price change, arrival of sufficient amount of new data, execution failure of an incremental run scheduled earlier, or simply that certain amount of time has passed since the last decision cycle. Now in *Agamotto-Flex*, an execution-finished event of any upstream query is also considered a qualifying event, signaling that new input data becomes available for downstream queries. In general, whenever a decision cycle suggests to run a set of queries S, we concurrently schedule those queries in S that do not have an immediate upstream query in S. The rest of the queries in S can simply be ignored, because the execution-finished events of the scheduled queries later will trigger new decision cycles that will potentially run them.

7 EXPERIMENTS

We report our experimental evaluation of the proposed solutions.

7.1 Setting

Using Tempura [30] as the incremental query optimizer and plan estimator, Alibaba's Cloud MaxCompute Platform [4] as the execution platform, we systematically evaluate **Agamotto** against the two naive solutions in Section 3, the fixed-time scheduler (**Fixed-***) and fixed-threshold scheduler (**Thres-***), as well as the **Prophet** scheduler (Section 4) that foresees the future and serves as a theoretical lower bound for all other schedulers. For example, *Fixed-6* runs at six equally spaced time points before the deadline, and *Thres-0.3* runs whenever resource price drops below 0.3 and sufficient new input data has accumulated. We also include the traditional **Batch** (can be seen as *Thres-0.0*) that processes everything at the deadline.

Complexity Score	2	3	4	5	6	7	10
Number of workflows	4	4	5	1	3	2	1

Table 1: Statistics on the Alibaba multi-query workflows.

For multi-query workflows, when the decision is to run, both *Fix-** and *Thres-** run dependent queries sequentially, similar to *Agamotto-Strict* (Section 6.3). We improve *Thres-0.3* to **Thres2.0-0.3**, that if the price when upstream execution finishes goes above 0.3, the downstream execution is delayed to the next time the price drops below 0.3. The *Prophet* can also be extended to handle multi-query workflows (details omitted due to space limit), but also suffers

from the data version explosion problem (Section 6.2). We apply the same data-compressing technique into **Prophet-Compress**, which provides an approximation to the true lower bound.

Three workloads are used: the TPC-H, TPC-DS benchmark, and selected multi-query routine workflows in Alibaba cloud shown in Table 1. For metrics, we focus on the estimated total weighted cost C_w as in our problem definition in Section 2.

Model Setup. Both the training and test price traces come from real historical cluster utilization data from Alibaba cloud, where the price is high at midnight with a high probability. The default prior distribution on price is trained using the stochastic model from [17]. To evaluate Agamotto's sensitivity to prior quality, we use a neutral prior with no prior knowledge on price, assuming a uniform price distribution at each time step. We use **Agamotto** and **Agamotto-N** to denote the Agamotto using the default and neutral prior, respectively. Two priors are used for input data arrivals: 1) $\mathcal{D}_{cont}$ (default), a continuous steam of data arriving at a fixed rate, and 2) $\mathcal{D}_{bulk}$, occasional bulk arrivals of input data, where it is possible to receive no new data at each time step.

By default, we used 10 discretized values for price and 24 time points for a 24-hour deadline. Evaluation is challenging because in each 24-hour window we can only run the workflow in the cluster once and collect one data point. It is hard to collect sufficient data to arrive at a statistically meaningful conclusion on the scheduler's overall performance, and we can easily miss the long tail. So we resort to simulation instead, where the test traces for a 24-hour window are "played" against the scheduler, and the simulated execution costs for each trace can be collected.

The model-solving times reported are measured on a laptop with 2.5GHz Intel Core i7 and 16GB DDR3 memory. The data compression for multi-query workflows is on by default.

7.2 Scheduling Behavior Under a Sample Trace

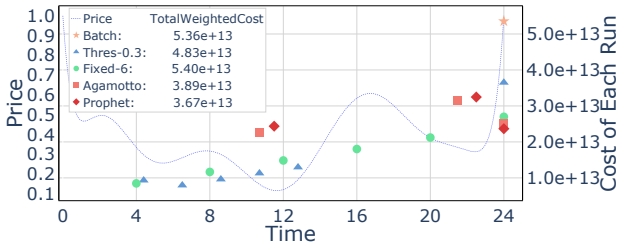


Figure 12: Scheduling results under a sample price trace.

Figure 12 shows the scheduling decision of these schedulers under a specific test price trace over a 24-hour window for query Q39 in TPC-DS. Note the cost plotted is not multiplied by the price, reflecting the amount of work done for each execution. The total weighted cost C_w is reported in the legend.

Running the traditional batch mode at the deadline yields a high cost of $C_w = 5.36E13$ because of the high price of 1.0 at 24:00. *Fixed-6* runs at fixed time points and has an even higher total cost of $5.40E13$. This is because the six random prices can be high and at the same time considerable incremental overhead is incurred because of the number of runs. Note that the cost of processing the same amount of new data increases in each run. This is because even with the same new delta size (ΔS, ΔR), the cost of an incremental join (e.g., $\Delta S \bowtie R$) grows with the total table size (e.g., R) so far.

For *Thres-**, the threshold value is hard to set because the price can be low in one day and high in another. In this example, *Thres-0.3* hits both problems where the threshold 0.3 is a bit high before 13:00 (it runs five times at no-so-low prices) and too low for the second half of the day (it fails to exploit the opportunity around 22:00). Still, it manages to bring down the cost to $4.83E13$.

Agamotto achieves a much better cost of $3.89E13$ because it abstemiously runs in the first half when the price is low, and knows to run at the decent price when the price becomes high later.

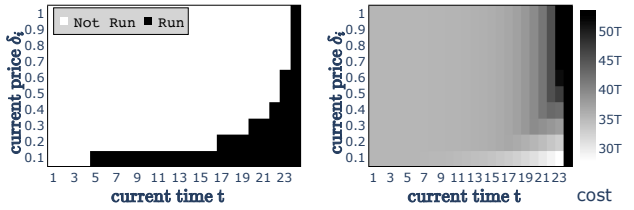


Figure 13: a) Scheduling policy produced by the offline MDP model, showing whether to schedule a new incremental run (black is run) at each state with $D^L = \mathbf{0}$. (b) The corresponding expected remaining cost of each state calculated by the MDP.

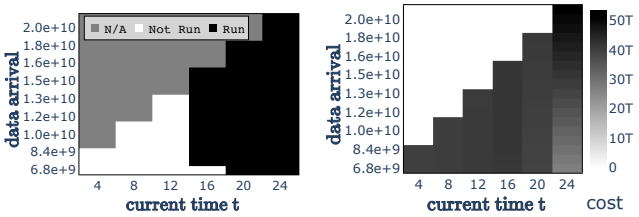


Figure 14: Scheduling policy similar to Figure 13, showing the dimension of input data arrivals instead.

7.3 Optimal MDP Scheduling Policy of Agamotto

To better understand Agamotto's behavior, let us zoom into its optimal scheduling policy of the MDP model. Recall that the MDP state for single query is a quadruple $\{t, \overrightarrow{D}, \delta_i, \overrightarrow{D^L}\}$, and the optimal policy tells us the best action (whether to run) to take at every state.

Impact of Price. Figure 13 visualizes a slice of the state space with $\overrightarrow{D^L} = \mathbf{0}$ for Q39 in TPC-DS. Under different combinations of the current time t and price δ_i, it plots the action on the left (black means run) and the expected remaining cost of the state ($\mathbb{C}(S, \overline{\mathcal{L}}|A)$ in Section 5.4) on the right. Assuming data of the non-dimensional tables comes in at a steady rate, $\overrightarrow{D}$ grows in proportion with time t.

As we can see, Agamotto prefers to run on low prices, and the threshold price to run keeps increasing as we get closer to the deadline. Thanks to the cost-based reasoning between exploiting the current price against the possibility of getting a lower price later, the suggested behavior aligns perfectly with the intuition that

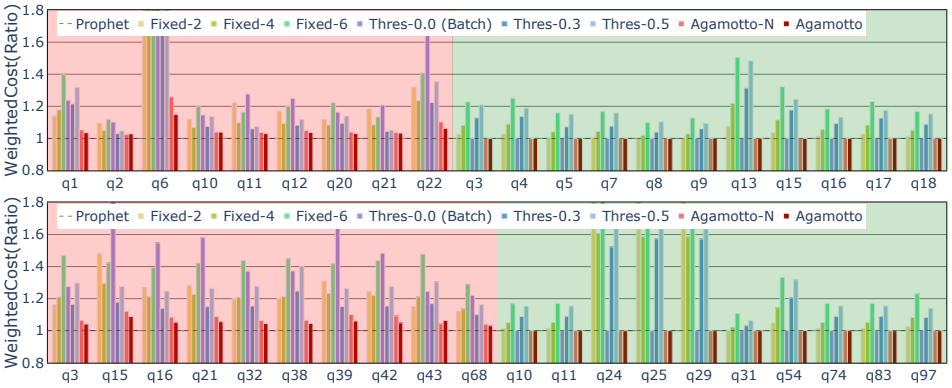


Figure 15: Estimated weighted total cost of TPC-H (upper) and TPC-DS (lower) queries under various schedulers, normalized by the *Prophet*. Ratio of one is the lower bound. Queries with red (green) background are ones (not) suitable for incremental execution.

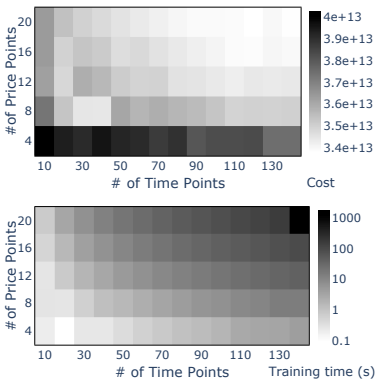


Figure 16: Impact of MDP model discretization parameters (number of time/price points).

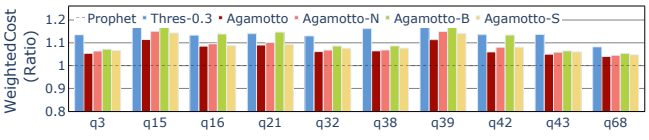


Figure 17: Cost of the left part of TPC-DS queries under different prior distributions on resource price. `Agamotto-S` switches scheduling policy from the bad to default prior at mid-day.

one should be more inclined to wait for a possibly low price earlier and get more aggressive when approaching the deadline.

As to the remaining cost on the right, since in most earlier states the decision is simply to wait, the expected costs of those states stay the same regardless of the price then. Whenever we hit and run at a low price, the remaining cost decreases. If we are unlucky that the price stays high towards the end without much progress, the calculated expected cost starts rising. This is where `Agamotto` gets into the panic mode and starts to run at higher prices.

In general, this scheduling behavior can be interpreted as an improvement of the naive fixed-threshold scheduler towards the *Prophet*, where the pre-calculated threshold value is dynamically applied on the fly based on the real time information then (e.g., progress, price, input data, and up-to-date belief on the future trend).

Impact of Input Data. Using the data prior $\mathcal{D}_{bulk}$ instead, Figure 14 shows a different slice of the state space by fixing $\delta_i = 0.4$ and $\overrightarrow{D^L} = \mathbf{0}$. The y-axis becomes the current data size $\overrightarrow{D}$. Assuming only zero to certain maximum amount of data can arrive at each step, the states above the stair shape (in grey) are impossible to reach. With a price of 0.4, `Agamotto` chooses to not run before 16:00. At 16:00, `Agamotto`'s decision also depends on the amount of input data at hand. It is worth running only when the amount is large enough, due to the impact of the incremental execution overhead.

7.4 Cost Performance on Single Query

Now we systematically compare `Agamotto` with other approaches by simulating them with real world test traces from Alibaba cloud. Figure 15 shows the sum of C_w across all traces normalized by that of the *Prophet*, for selected TPC-H and TPC-DS queries. Some queries are naturally more suitable for incremental execution than others. We split the queries into two groups: ones suitable (red background) and not suitable (green) for incremental progressing, according to whether the *Prophet* decides to only run once at the deadline for all test traces. With a ratio of one being the lower bound provided by the *Prophet*, `Agamotto` achieves very close to one ratio across the board, without knowing the future, and consistently outperforms all *Fixed-** and *Thres-** by 16% and 18% on an average C_w ratio for TPC-H and TPC-DS, respectively.

In general, `Agamotto`'s outstanding performance has two main reasons: 1) the ability to evaluate and adapt to the situations dynamically, thanks to the MDP model illustrated earlier; 2) incorporation of the cost characteristics of queries into the model. For instance, for queries not suitable for the incremental processing (green background), the overhead is so high that even with the high price at the deadline, it is still not worthwhile to process the data early.

Since the cost characteristics of the queries are reflected in the MDP model, `Agamotto` knows that it is not beneficial to run these queries early even at a very low price, so that it wisely makes the same decision as the *Prophet* and *Batch*: simply wait and run once at the deadline. On the contrary, both *Fix-** and *Thres-** blindly run the queries incrementally and result in higher costs.

Sensitivity to Prior Quality.

We also added `Agamotto-N` using the neutral prior that does not encode any real price trend. Although not as good as `Agamotto`, it remains the best among others and close to the *Prophet*. This is due to the dynamic nature of `Agamotto`: even if all decisions are a bit off due to imprecise prior information, it always re-evaluates the situation and adjusts when making the next decision, so that it partially compensates for the earlier mistake.

7.5 Sensitivity, Robustness, and Flexibility

Now we take one step further by asking what if the usage pattern in a repetitive cluster all of a sudden becomes very different in one day due to various reasons, say on Black Friday or Singles' day?

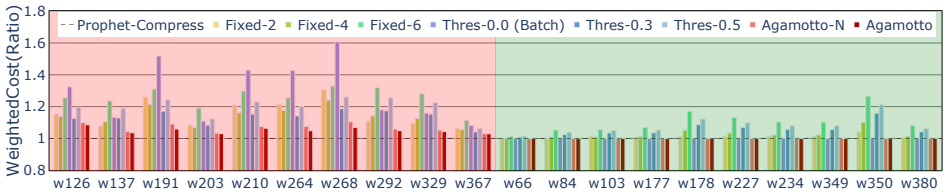


Figure 18: Estimated total cost of Alibabamulti-query workflows, under various schedulers, normalized by the *Prophet-Compress*, an approximation to the true lower bound. Queries with red (green) background are ones (not) suitable for incremental execution.

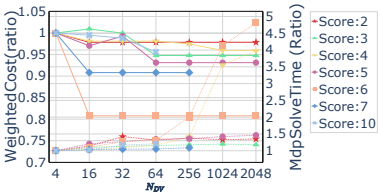


Figure 19: Effect of data version compression on cost (solid line) and model solving time (dotted line).

We handpick another cluster with a rather different daily pattern from the ones used for the default prior and test traces, and trained Agamotto-B with this prior. We call it "bad" because it is worse than the neutral prior, it provides a stronger (concentrated probabilities) but wrong trend, and misleads the MDP model even more.

In this unfortunate case, we can actually do much better than Agamotto-B because of its runtime flexibility and online nature. When realizing that today's pattern is going to be very different from the prior used, one can decide to switch to a new policy trained from a new prior during the execution. To illustrate this behavior, we add Agamotto-S that switches from Agamotto-B to Agamotto (trained with the default prior) at the 12h midpoint of the day.

Figure 17 shows the results for the left (red) portion of the queries in TPC-DS. As we can see, Agamotto-B downgrades more than Agamotto-N, but still beats *Thres-0.3*, the best baseline scheduler, while Agamotto-S performs much closer to Agamotto, demonstrating great robustness and flexibility of the Agamotto solution.

7.6 The Effect of Discretization Parameters

Coming back to Q39 in TPC-DS, Figure 16 shows the impact of MDP discretization parameters (number of time/price points) on performance as well as the model training time. $T24P10$ stands for the default setting with 24 time steps and 10 price values.

As we can see, more discretized points lead to lower cost. This is because a more fine-grained model better captures the dynamics of the future space and produces a higher-quality scheduling policy. As the upper right corner of the first figure shows, since the prior information does not contain more fine-grained information, further increasing the model complexity has diminishing benefit on cost performance. On the other hand, the increasing model complexity leads to a longer running time spent solving the MDP.

7.7 Performance on Multi-query Workflows

Now we turn to selected multi-query workflows (Table 1) from Alibabacloud. Figure 18 shows the C_w of various approaches across all test traces normalized by that of the *Prophet-Compress*.

Similar to the single-query case, Agamotto still achieves the best performance consistently. Note that *Prophet-Compress* here is *Prophet* with data version compression on, so it is an approximation of the actual lower bound of each specific test trace. For workflows in the green background, *Prophet-Compress* decides to not run early for any query in the workflow, and Agamotto is smart enough to make the same scheduling decisions. For other workflows suitable for incremental processing (red background), Agamotto chooses to run the upstream and downstream queries asynchronously according to the price evolvement and outperforms other schedulers.

7.8 Effect of Data Version Compression

Next, we evaluate the effectiveness of data version compression for the multi-query workflows. Data version compression makes the model tractable by reducing its complexity.

Figure 19 shows the MDP model's solving time and the achieved total cost C_w for multiple workflows with various complexity scores (see Section 6.2). We normalized the cost and time of each query by that achieved with the highest compression setting of $N_{DV} = 4$.

As we can see, when we tighten the compression rate by reducing N_{DV}, the effect of compression kicks in at different threshold values. There are two reasons: 1) for the majority of low-fan-in queries (with a low number of input tables), the number of output data versions is low and compression will not take effect when N_{DV} is high; 2) there are typically one or two queries in each workflow that contribute the most and dominate the overall cost. Compression effect on other queries is thus negligible in the total cost.

Overall, compression via combining similar data versions only mildly reduces the model quality and harm the performance, but at the same time drastically reduces the model complexity and enables Agamotto to handle complex real-world workflows efficiently.

8 CONCLUSIONS

In this paper, we propose a novel scheduling problem for incremental query execution under a given deadline, assuming cluster resource has a fluctuating and unforeseen price. Given the incremental plan template produced by a given incremental query optimizer, the goal of the scheduler is to decide when and which query to run, so that the overall total cost weighted by the resource price is minimized. We first propose two naive solutions, and a *Prophet* scheduler that statically solves the simplified problem where the future is fully known. Next, using a Markov Decision Process to model future probabilities, we present Agamotto, an end-to-end scheduling solution that dynamically makes cost-based decisions without knowing the future. We show that Agamotto can be further extended to handle a workflow of dependent queries, so that they can all incrementally execute in an asynchronously fashion. Experiments show that Agamotto consistently outperforms the naive solutions, and the achieved cost is on average 10x closer to the theoretical lower bound provided by the *Prophet* scheduler.

REFERENCES

[1] Zulfiqar Ahmad, Ali Imran Jehangiri, Mohammed Alaa Ala'anzy, Mohamed Othman, Rohaya Latip, Sardar Khaliq Uz Zaman, and Arif Iqbal Umar. 2021. Scientific Workflows Management and Scheduling in Cloud Computing: Taxonomy, Prospects, and Challenges. *IEEE Access* 9 (2021), 53491–53508. https://doi.org/10.1109/ACCESS.2021.3070785

[2] Tyler Akidau, Paul Barbier, Istvan Cseri, Fabian Hueske, Tyler Jones, Sasha Lionheart, Daniel Mills, Dzmitry Pauliukevich, Lukas Probst, Niklas Semmler, Dan Sotolongo, and Boyuan Zhang. 2023. What's the Difference? Incremental Processing with Change Queries in Snowflake. *Proc. ACM Manag. Data* 1, 2, Article 196 (jun 2023), 27 pages. https://doi.org/10.1145/3589776

[3] Alibaba Cloud ECS Spot Instances n.a.. https://www.alibabacloud.com/help/zh/batch-compute/latest/bidding-resources

[4] Alibaba MaxCompute n.a.. https://www.alibabacloud.com/product/maxcompute

[5] Amazon EC2 Spot Instances n.a.. https://aws.amazon.com/ec2/spot/

[6] Apache Hudi n.a.. https://hudi.apache.org/.

[7] Apache Spark n.a.. https://spark.apache.org/.

[8] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 601–613. https://doi.org/10.1145/3183713.3190664

[9] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. ACM, New York, NY, USA, 221–230. https://doi.org/10.1145/3183713.3190662

[10] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf

[11] Dazhao Cheng, Xiaobo Zhou, Yu Wang, and Changjun Jiang. 2018. Adaptive Scheduling Parallel Jobs with Dynamic Batching in Spark Streaming. *IEEE Transactions on Parallel and Distributed Systems* 29, 12 (2018), 2672–2685. https://doi.org/10.1109/TPDS.2018.2846234

[12] DataBricks Enzyme, a Cost-Based Incremental Query Optimizer 2022. https://www.databricks.com/blog/2022/06/29/delta-live-tables-announces-new-capabilities-and-performance-optimizations.html

[13] Debasis Dwibedy and Rakesh Mohanty. 2022. Semi-Online Scheduling: A Survey. *Comput. Oper. Res.* 139, C (mar 2022), 23. https://doi.org/10.1016/j.cor.2021.105646

[14] Sérgio Esteves, Helena Galhardas, and Luís Veiga. 2018. Adaptive Execution of Continuous and Data-Intensive Workflows with Machine Learning. In *Proceedings of the 19th International Middleware Conference* (Rennes, France) *(Middleware '18)*. Association for Computing Machinery, New York, NY, USA, 239–252. https://doi.org/10.1145/3274808.3274827

[15] Timothy Griffin and Leonid Libkin. 1995. Incremental Maintenance of Views with Duplicates. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* (San Jose, California, USA) *(SIGMOD '95)*. Association for Computing Machinery, New York, NY, USA, 328–339. https://doi.org/10.1145/223784.223849

[16] Ashish Kumar Gupta and Inderpal Singh Mumick. 1995. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Eng. Bull.* 18 (1995), 3–18.

[17] Botong Huang, Nicholas W. D. Jarrett, Shivnath Babu, Sayan Mukherjee, and Jun Yang. 2015. Cümülön: Matrix-Based Data Analytics in the Cloud with Spot Instances. *Proc. VLDB Endow.* 9, 3 (nov 2015), 156–167. https://doi.org/10.14778/2850583.2850590

[18] Botong Huang and Jun Yang. 2017. Cümülön-D: Data Analytics in a Dynamic Spot Market. *Proc. VLDB Endow.* 10, 8 (apr 2017), 865–876. https://doi.org/10.14778/3090163.3090165

[19] Suyeon Lee, Yeonwoo Jeong, Minwoo Kim, and Sungyong Park. 2021. Q-Spark: QoS Aware Micro-batch Stream Processing System Using Spark. In *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. 38–43. https://doi.org/10.1109/ACSOS-C52956.2021.00027

[20] Wenxin Li, Di Niu, Yinan Liu, Shuhao Liu, and Baochun Li. 2017. Wide-Area Spark Streaming: Automated Routing and Batch Sizing. In *2017 IEEE International Conference on Autonomic Computing (ICAC)*. 33–38. https://doi.org/10.1109/ICAC.2017.36

[21] Christopher Olston. Technical report, 2011. Modeling and Scheduling Asynchronous Incremental Workflows. http://i.stanford.edu/~olston/publications/asynchronousWorkflowsTR.pdf

[22] Christopher Olston, Greg Chiou, Laukik Chitnis, Francis Liu, Yiping Han, Mattias Larsson, Andreas Neumann, Vellanki B.N. Rao, Vijayanand Sankarasubramanian, Siddharth Seth, Chao Tian, Topher ZiCornell, and Xiaodan Wang. 2011. Nova: Continuous Pig/Hadoop Workflows. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece) *(SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 1081–1090. https://doi.org/10.1145/1989323.1989439

[23] C. N. Potts and V. A. Strusevich. 2009. Fifty Years of Scheduling: A Survey of Milestones. *The Journal of the Operational Research Society* 60 (2009), s41–s68. http://www.jstor.org/stable/40206725

[24] Zechao Shang, Xi Liang, Dixin Tang, Cong Ding, Aaron J. Elmore, Sanjay Krishnan, and Michael J. Franklin. 2020. CrocodileDB: Efficient Database Execution through Intelligent Deferment. In *CIDR*.

[25] Shikha Singh, Sergey Madaminov, Michael A. Bender, Michael Ferdman, Ryan Johnson, Benjamin Moseley, Hung Q. Ngo, Dung Nguyen, Soeren Olesen, Kurt Stirewalt, and Geoffrey Washburn. 2020. A Scheduling Approach to Incremental Maintenance of Datalog Programs. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, May 18-22, 2020*. IEEE, 864–873. https://doi.org/10.1109/IPDPS47924.2020.00093

[26] Dixin Tang, Zechao Shang, Aaron J. Elmore, Sanjay Krishnan, and Michael J. Franklin. 2019. Intermittent Query Processing. *Proc. VLDB Endow.* 12, 11 (July 2019), 1427–1441. https://doi.org/10.14778/3342263.3342278

[27] Dixin Tang, Zechao Shang, Aaron J. Elmore, Sanjay Krishnan, and Michael J. Franklin. 2020. Thrifty Query Execution via Incrementability. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1241–1256. https://doi.org/10.1145/3318464.3389756

[28] 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* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1797–1810. https://doi.org/10.1145/3448016.3457282

[29] Zuozhi Wang, Kai Zeng, Botong Huang, Wei Chen, Xiaozong Cui, Bo Wang, Ji Liu, Liya Fan, Dachuan Qu, Zhenyu Hou, Tao Guan, Chen Li, and Jingren Zhou. 2020. Grosbeak: A Data Warehouse Supporting Resource-Aware Incremental Computing. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2797–2800. https://doi.org/10.1145/3318464.3384708

[30] Zuozhi Wang, Kai Zeng, Botong Huang, Wei Chen, Xiaozong Cui, Bo Wang, Ji Liu, Liya Fan, Dachuan Qu, Zhenyu Hou, Tao Guan, Chen Li, and Jingren Zhou. 2020. Tempura: A General Cost-Based Optimizer Framework for Incremental Data Processing. *Proc. VLDB Endow.* 14, 1 (sep 2020), 14–27. https://doi.org/10.14778/3421424.3421427