# Planning Progressive Execution: Resource Smoothing in Cloud-Scale Data Warehouses

Zuozhi Wang[1], Kai Zeng[2], Botong Huang[2], Wei Chen[2], Xiaozong Cui[2], Bo Wang[2], Ji Liu[2], Liya Fan[2], Dachuan Qu[2], Zhenyu Ho[2], Tao Guan[2], Chen Li[1], Jingren Zhou[2]
[1]{zuozhiw, chenli}@ics.uci.edu, [2]{zengkai.zk, botong.huang, wickeychen.cw, xiaozong.cxz, yanyu.wb, niki.lj, liya.fly, dachuan.qdc, zhenyuhou.hzy, tony.guan, jingren.zhou}@alibaba-inc.com
[1]UC Irvine, [2]Alibaba Group

## ABSTRACT

In modern cloud-scale data warehouses, ever increasing data analysis is predominately handled by batch processing due to its resource efficiency and query generality. This processing pattern causes severe resource skew in clusters and incurs excessive resource waste. Progressively processing analysis jobs provide a much more flexible resource usage pattern and can smooth resource skews. However, despite rich research work in various progressive computation methods, there is still no optimizer framework for planning progressive execution that can not only explore all existing incremental computation techniques, but also support cost-based optimization. In this paper, we propose a novel theory called time-varying relation (TVR) model that can formally model progressive execution in a general form, and provide a unified cost-based optimizer framework. We build a Cascade-style optimizer called `Beanstalk` that incorporates the TVR model and enables cost-based optimization of progressive plans. We verify this technique on daily analysis jobs in a cloud-scale data warehouse to show that it can save the peak resource usage by almost 30%.

## 1 INTRODUCTION

Modern business is pervasively data-driven. As keystone systems among the big data infrastructure of a modern enterprise, cloud-scale data warehouses have seen an unprecedented growth in automated routine data analysis. For instance, at a company running cloud-scale big data analytic service, up to 65% of daily workload is automated recurring routine jobs, which usually have a stringent schedule and deadline determined by various business logic, e.g., scheduling a daily report query after 12 am when the previous day's data has been fully collected and delivering the results by 6 am sharp before the bill settlement time. By now, the automated routine analysis is still predominately handled using batch query processing, in favor of efficient resource consumption and better query generality. On one hand, these companies observe dreadful "rush hour" scheduling patterns of routine analysis jobs, putting pressure on resources during traffic hours. The cluster size needs to grow constantly to keep up with the growth of peak resource capacity. On the other hand, cluster resources become over-provisioned off the traffic hours, causing excessive waste. Fig. 1 shows the resource skew of a typical day in such a company.

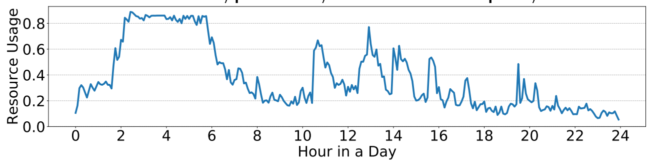


**Figure 1: Cluster resource skew of a typical day.**

Fortunately, advancement in big data systems makes data ingestion to warehouses more real time, usually with a latency of minutes or even seconds. This real-time ingestion enables *progressive processing* for routine analysis jobs, which have a more flexible resource usage pattern and can smooth potential resource skew. Specifically, instead of waiting until the scheduled time, we can process analysis queries progressively as data arrives, e.g., as promptly as each data batch ingested, or for certain elapsed time, or even whenever the cluster has free resources. Progressive processing essentially moves part of the computation off the traffic hours to a less congested time. However, it is a nontrivial problem to figure out the optimal progressive execution plan, considering query characteristics and the dynamic cluster environment. In this paper, we focus on developing a unified cost-based query optimization framework, which allows users to express and integrate various incremental computation techniques and provides a turn-key solution to decide optimal progressive execution plans subject to various objectives.

*Woodstock '18, June 03–05, 2018, Woodstock, NY*
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

To elaborate the complexity of progressive query planning, let us consider a periodic big data analytic job in Example 1.1 that mimics many TPC-DS workloads. In the pipeline, the upstream *sales_status* query computes the final revenue of all orders by consolidating the sales orders with the returned ones; then a downstream *summary* query computes the gross revenue per order category. Tables *sales* and *returns* are continuously synchronized from the external transactional data sources. Without waiting for data to be fully synchronized after a day, progressive execution incrementally answers the queries periodically for small batches of newly ingested data.

*Example 1.1 (A simple analytics pipeline).*

```
sales_status =
  SELECT sales.o_id, category, price, cost
  FROM sales LEFT OUTER JOIN returns
    ON sales.o_id = returns.o_id
summary =
  SELECT category,
    SUM(IF(cost IS NULL, price, -cost)) AS gross
  FROM sales_status
  GROUP BY category
```

Now consider two basic incremental computation methods from view maintenance and stream computing. Both methods can be adapted to be used to progressively compute the analytics pipeline, in a batch manner. But which method is optimal depends on the data and many other factors. The planning process needs to search through the plan space of jointly different incremental methods, and the decisions of whether to progressively compute each subpart of a query. The view maintenance approach treats *sales_status* and *summary* as views. After each ingested data batch, it eagerly maintains the results of the two queries as if they were directly computed from the data of *sales* and *returns* seen so far. Therefore, even if a *sales* order is returned in the future, its revenue will be counted into the gross revenue temporarily. On the contrary, the stream computing method would avoid such retraction. Instead, it holds on *sales* orders that it is not sure if will be returned in the future, and only counts them in until data is fully synchronized. Clearly, if returned orders are rare, the view maintenance approach can maximize the amount of early computation and thus deliver better resource-usage plans. Otherwise, if returned orders are often, the stream computing approach will save unnecessary recomputation caused by retraction and thus will be better. Even for a single query, different incremental computation methods, including those very sophisticated ones such higher-order view maintenance [3], can be jointly applied to achieve an optimal plan.

Furthermore, progressive processing introduces complex dependencies both along the time dimension and between upstream/downstream queries in a pipeline. For instance, in a progressive plan, previous runs may save intermediate states to be shared by later runs. What and how intermediate states are saved can significantly impact the cost. Second, modern data warehouses typically consist of wide and deep analysis pipelines. How a downstream query can use an early output can affect how the upstream queries produce early outputs. We will illustrate the details in § 2.

The above discussion clearly demonstrates the complexity of planning progressive processing. It needs to jointly plan along three dimensions: (1) different incremental computation techniques, (2) data characteristics such distribution and arrival rate, and (3) complex dependencies along the time line and across queries. Such planning has to be done in a cost-based manner. However, existing theory and systems for query optimization are usually only able to explore the plan space of a query at a single time point, using a single fixed execution mode or computation method, either normal execution or a fixed incremental computation method. A progressive processing paradigm needs new theories and systems for query planning.

In this paper, we propose a theory called *time-varying relation model* for progressive execution. It naturally extends the relational model by considering the temporal aspect, modeling changing data as a time-varying relation (TVR), with snapshots and deltas as two different views of encoding a TVR. It also includes various data-manipulation operations on TVR's, as well as TVR rewriting rules. It provides a general framework for a query optimizer to define various incremental computation techniques, and unifies them to explore a search space to generate an efficient execution plan. This model allows various incremental computation techniques to work cooperatively, and enables cost-based search among all possible incremental plans.

We demonstrate how to support cost-based progressive planning by incorporating the TVR model on a Cascade-style optimizer, and build a prototype optimizer named `Beanstalk`. Our key contributions are:

- We propose a new theory called time-varying relation model that can formally model progressive execution in its most general form (§ 3), and provide a general framework for a query optimizer to define and unify various incremental computation techniques to jointly explore a search space for an efficient progressive plan (§ 4).
- We build a Cascade-style optimizer named `Beanstalk` that incorporates the TVR model, and supports cost-based optimization for progressive planning. We discuss how to explore the plan space (§ 5) and select an optimal progressive plan (§ 6) in `Beanstalk`.
- We study experimentally progressive execution in cloud-scale data warehouses on both real-life and benchmark workloads. The experimental results show the effectiveness of well-planned progressive execution on help smooth resources in large clusters (§7).

## 2 PROBLEM FORMULATION

In this section we formally define the problem of cost-based progressive execution of queries for a given schedule, and use an example to illustrate different execution plans and search-related challenges.

### 2.1 Schedule-Based Progressive Execution

Consider a warehouse with data arriving continuously, and a set of queries that need to be executed at certain times. Each query is expected to run at a specific time $T$ on the data collected in a time window that ends at $T$. The data warehouse has a computing cluster whose resource usage fluctuates over time, i.e., sometimes it is heavily used and sometimes there are many idle resources. The warehouse monitors its cluster usage and the data-ingestion rate. Whenever it finds that the cluster has free resources, or some queries have enough arrived data, it selects one or more queries (referred to as a workflow $W$) from the registered queries, and proposes a *schedule* $S$ to run this workflow. The schedule includes a set of time points on which we want to run $W$ progressively. Our goal is to do a cost-based analysis and generate a progressive plan to run the query incrementally at those times. Formally, we are given the following:

- A workflow $W$ consisting one or more queries.
- A schedule $S = \{t_0, t_1, \ldots, t_n, t^*\}$, where $t_0$ is the current time, and $t^*$ is the final time of finishing $W$, i.e., the max of the specified execution time $T$ of each query in $W$.
- The input data available at each time $t_i$, e.g., the newly available data at time $t_i$, or early output generated from upstream queries.

Our goal is to generate a progressive plan for the schedule. A progressive plan consists of a sequence of physical plan $\mathbb{P} = [P_0, P_1, \cdots, P_n]$, where $P_i$ defines the task to execute at time $t_i$. Compared to a traditional physical plan, $P_i$ can include decisions of materializing intermediate states at $t_i$ to share for later runs. A proposed schedule can be tentative and subject to change. Once the schedule changes, we can re-plan for the new schedule by taking into account the partially executed progressive plan of the old schedule. We will discuss these advanced planning issues in § 6.4.

### 2.2 Example

Consider the analysis workflow and tables in Example 1.1. Suppose we are given a schedule $S$ with two times $t_1$ and $t_2$, where $t_2$ is the specified execution time of both queries in the workflow, and $t_1$ is an earlier time. Assume the data visible at $t_1$ and $t_2$ in *sales* and *returns* are those in Fig. 2(a). Our goal is to generate a progressive plan that consists an execution task for each of the two times.

Progressive computation has been studied in the contexts such as view maintenance, stream computing, mini-batch

*sales*

| *o_id* | *cat* | *price* | |
|---|---|---|---|
| $o_1$ | $c_1$ | 100 | $t_1$ |
| $o_2$ | $c_2$ | 150 | $t_1$ |
| $o_3$ | $c_1$ | 120 | $t_1$ |
| $o_4$ | $c_1$ | 170 | $t_1$ |
| $o_5$ | $c_2$ | 300 | $t_2$ |
| $o_6$ | $c_1$ | 150 | $t_2$ |
| $o_7$ | $c_2$ | 220 | $t_2$ |

*returns*

| *o_id* | *cost* | |
|---|---|---|
| $o_1$ | 10 | $t_1$ |
| $o_2$ | 20 | $t_2$ |
| $o_6$ | 15 | $t_2$ |

(a)

*sales_status*

| *o_id* | *cat* | *price* | *cost* |
|---|---|---|---|
| $o_1$ | $c_1$ | 100 | 10 |
| $o_2$ | $c_2$ | 150 | 20 |
| $o_3$ | $c_1$ | 120 | null |
| $o_4$ | $c_1$ | 170 | null |
| $o_5$ | $c_2$ | 300 | null |
| $o_6$ | $c_1$ | 150 | 15 |
| $o_7$ | $c_2$ | 220 | null |

*summary*

| *cat* | *gross* |
|---|---|
| $c_1$ | 265 |
| $c_2$ | 500 |

(b)

*sale_status* at $t_1$

| *o_id* | *cat* | *price* | *cost* |
|---|---|---|---|
| $o_1$ | $c_1$ | 100 | 10 |
| $o_2$ | $c_2$ | 150 | null |
| $o_3$ | $c_1$ | 120 | null |
| $o_4$ | $c_1$ | 170 | null |

*sale_status* at $t_2$

| *o_id* | *cat* | *price* | *cost* | # |
|---|---|---|---|---|
| $o_2$ | $c_2$ | 150 | null | −1 |
| $o_2$ | $c_2$ | 150 | 20 | +1 |
| $o_5$ | $c_2$ | 300 | null | +1 |
| $o_6$ | $c_1$ | 150 | 15 | +1 |
| $o_7$ | $c_2$ | 220 | null | +1 |

(c)

*sale_status* at $t_1$

| *o_id* | *cat* | *price* | *cost* |
|---|---|---|---|
| $o_1$ | $c_1$ | 100 | 10 |

*sale_status* at $t_2$

| *o_id* | *cat* | *price* | *cost* | # |
|---|---|---|---|---|
| $o_2$ | $c_2$ | 150 | 20 | +1 |
| $o_3$ | $c_1$ | 120 | null | +1 |
| $o_4$ | $c_1$ | 170 | null | +1 |
| $o_5$ | $c_2$ | 300 | null | +1 |
| $o_6$ | $c_1$ | 150 | 15 | +1 |
| $o_7$ | $c_2$ | 220 | null | +1 |

(d)

**Figure 2: (a) Data arrival patterns of *sales* and *returns*, (b) results of query *sales_status* and *summary* at $t_2$, (d) incremental results of *sales_status* produced by view maintenance at $t_1$ and $t_2$, and (d) incremental results of *sales_status* produced by stream computing at $t_1, t_2$.**

execution and so on [3, 4, 12, 18]. The following are two commonly used incremental computation methods.

**View maintenance approach.** It treats *sales_status* and *summary* as views, and uses incremental computation to keep the views always up to date with respect to the data seen so far. The incremental computation is done on the delta input. For example, the delta input to *sales* at $t_2$ are tuples $\{o_5, o_6, o_7\}$. Fig. 2(c) depicts *sales_status*'s incremental outputs at $t_1$ and $t_2$, respectively, where $\# = +/-1$ denote insertion or deletion respectively. Note that a *returns* record (e.g., $r_2$ at $t_2$) can arrive much later than its corresponding order (e.g., the shaded $o_2$ at $t_1$). Therefore, an *order* record may be output early as it cannot join with a *returns* record at $t_1$, but retracted later at $t_2$ when the *returns* record arrives, such as tuple $o_2$ of *sales_status* at $t_2$ in Fig. 2(c).

**Stream computing approach.** It can avoid such retraction during incremental computation. Specifically in the outer join of *sales_status*, tuples in *orders* that do not join with tuples from *returns* for now (e.g., $o_2$, $o_3$, and $o_4$) may join in the future, and thus will be held back at $t_1$. Essentially the outer join is computed as an inner join at $t_1$. The incremental outputs of *sales_status* are shown in Fig. 2(d).

### 2.3 Plan Space and Challenges

In addition to these two progressive-computation methods, there are many other methods. Deciding a one with a good performance is non-trivial due to the following reasons.

*First, the optimal progressive plan is data dependent, and should be determined in a cost-based way.* In the running example, the view maintenance approach computes 9 tuples (5 tuples in the outer join and 4 tuples in the aggregate) at $t_1$, and 10 tuples at $t_2$. Suppose the cost per unit at $t_1$ is 0.2 (due to fewer queries), and the cost per unit at $t_2$ is 1. Then its total cost is $9 \times 0.2 + 10 \times 1 = 11.8$. The stream computing approach computes 6 tuples at $t_1$, and 11 tuples at $t_2$, with a total cost of $6 \times 0.2 + 11 \times 1 = 12.2$. The latter approach is more efficient, since it can do more early computation in the outer join, and more early outputs further enable *summary* to do more early computation. On the contrary, if the retraction is often, say, with one more order tuple $o_4$ at $t_2$, then the stream computing approach will become more efficient, as it costs 12.2 versus the cost 13.8 of the first approach. The reason is that retraction wastes early computation and causes more re-computation overhead. Notice that the performance difference of these two approaches can be arbitrarily large.

*Second, the entire space of possible plan alternatives is huge.* Different queries of a workflow and different parts within a query can choose different progressive methods. Even if early computation of the entire query does not pay off, we can still progressively execute a subpart of the query. For instance, for the left outer join in *sales_status*, we can progressively shuffle the input data once it is ingested without waiting for the last time. Progressive planning needs to search the entire plan space ranging from the traditional batch plan at one end to a fully-incrementalized plan at the other.

*Third, complex across-query and temporal dependencies can also impact the plan decision.* Across queries, a small change in a downstream query can lead to very different plans for the upstream. For instance, if a MAX aggregate is applied on *sales_status*, retraction from *sales_status* and thus the view maintenance approach are no longer allowed, since retraction can cause MAX to recompute from scratch. The temporal aspect can further complicate the planning. For instance, as more data gets ingested, query *sales_status* may prefer a broadcast join at $t_1$ when the *returns* table is small, but a shuffled hash join at $t_2$ when the *returns* table gets bigger. But such a decision may not be optimal, as shuffled hash join needs data to be distributed according to the join key, which broadcast join does not provide. Thus, different join implementations between $t_1$ and $t_2$ incur reshuffling overhead. Progressive planning needs to jointly consider across the entire schedule.

Such complex reasoning is challenging, if not impossible, even for very experienced experts. To solve this problem, we offer a cost-based solution to systematically search the entire plan space, which can generate an efficient plan. Our solution can combine and exploit different incremental computation techniques in a single plan.

## 3 TIME-VARYING-RELATION MODEL

The core of progressive execution at different times is to deal with relations changing over time, and understand how the computation on these relations can be expanded along the time dimension. In this section, we introduce a concept of time-varying relation (TVR), and a formal method called *time-varying-relation (TVR) model.* The model naturally extends the relational model by considering the temporal aspect to formally describe progressive execution. It also includes various data-manipulation operations on TVR's, as well as rewriting rules of TVR's in order for a query optimizer to define and explore a search space to generate an efficient progressive execution plan. Note that although the concept of TVR was studied earlier [4, 6], to our best knowledge, our proposed TVR model is the first one that not only unifies different progressive execution methods, but also can be used to develop a principled cost-based optimization framework for progressive execution.

We focus on definitions and basic operations of TVR's in this section, and dwell on TVR rewriting rules in § 4.

### 3.1 Time-Varying Relations (TVR)

*Definition 3.1 (Time-varying relation).* A *time-varying relation (TVR) R* is a mapping from a time domain $\mathcal{T}$ to a bag of tuples belonging to a schema.

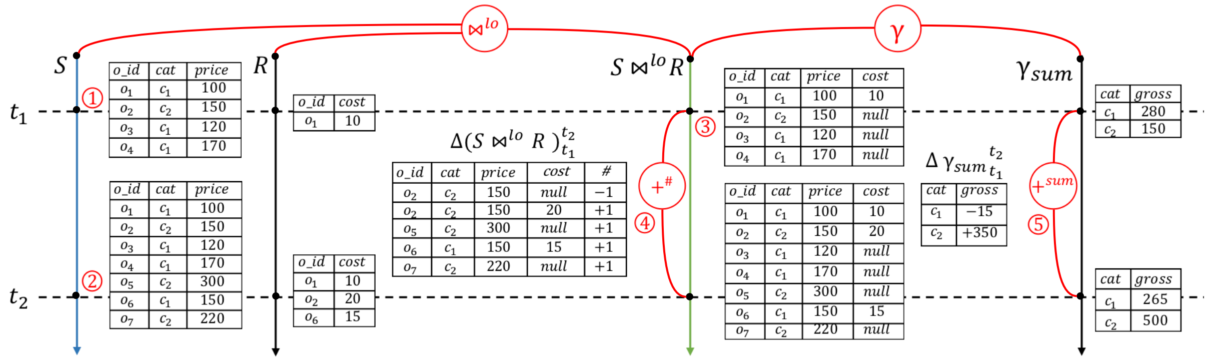
A *snapshot* of $R$ at a time $t$, denoted $R_t$, is an instance of $R$ at time $t$. For example, due to continuous ingestion, table *sales* ($S$) in Example 1.1 is a TVR, depicted as the blue line in Fig. 3. On the line, tables ① and ② show the snapshots of $S$ at $t_1$ and $t_2$, i.e., $S_{t_1}$ and $S_{t_2}$, respectively. Traditional data warehouses run queries on relations at a specific time, while progressive execution runs queries on TVR's.

*Definition 3.2 (Querying TVR).* Given a TVR $R : \mathcal{T} \to schema(R)$, applying a query $Q$ on $R$ defines another TVR $Q(R)$ on time domain $\mathcal{T}$, where $[Q(R)]_t = Q(R_t), \forall t \in \mathcal{T}$.

In other words, the snapshot of $Q(R)$ at $t$ is the same as applying $Q$ as a query on the snapshot of $R$ at $t$. For instance, in Fig. 3, joining two TVR's *sales* ($S$) and *returns* ($R$) yields a TVR ($S \bowtie^{lo} R$), depicted as the green line. Snapshot $(S \bowtie^{lo} R)_{t_1}$ is shown as table ③, which is equal to joining $S_{t_1}$ and $R_{t_1}$. Here we denote left semi-join as $\bowtie^{lo}$, left anti-join as $\bowtie^{la}$, and left semi-join as $\bowtie^{ls}$. For brevity, we use $Q$ to refer to the TVR $Q(R)$ throughout the paper when there is no ambiguity.

### 3.2 Basic Operations on TVR's

Besides as a sequence of snapshots, a TVR can be encoded from a delta perspective using the changes between two snapshots. We denote the difference between two snapshots of TVR $R$ at $t, t' \in T$ $(t < t')$ as the *delta* of $R$ from $t$ to $t'$, denoted $\Delta R_t^{t'}$. Formally, $\Delta R_t^{t'}$ defines a second-order TVR.



| o_id | cat | price |
|---|---|---|
| $o_1$ | $c_1$ | 100 |
| $o_2$ | $c_2$ | 150 |
| $o_3$ | $c_1$ | 120 |
| $o_4$ | $c_1$ | 170 |

| o_id | cat | price |
|---|---|---|
| $o_1$ | $c_1$ | 100 |
| $o_2$ | $c_2$ | 150 |
| $o_3$ | $c_1$ | 120 |
| $o_4$ | $c_1$ | 170 |
| $o_5$ | $c_2$ | 300 |
| $o_6$ | $c_1$ | 150 |
| $o_7$ | $c_2$ | 220 |

| o_id | cost |
|---|---|
| $o_1$ | 10 |

| o_id | cost |
|---|---|
| $o_1$ | 10 |
| $o_2$ | 20 |
| $o_6$ | 15 |

| o_id | cat | price | cost | # |
|---|---|---|---|---|
| $o_2$ | $c_2$ | 150 | null | −1 |
| $o_2$ | $c_2$ | 150 | 20 | +1 |
| $o_5$ | $c_2$ | 300 | null | +1 |
| $o_6$ | $c_1$ | 150 | 15 | +1 |
| $o_7$ | $c_2$ | 220 | null | +1 |

| o_id | cat | price | cost |
|---|---|---|---|
| $o_1$ | $c_1$ | 100 | 10 |
| $o_2$ | $c_2$ | 150 | null |
| $o_3$ | $c_1$ | 120 | null |
| $o_4$ | $c_1$ | 170 | null |

| o_id | cat | price | cost |
|---|---|---|---|
| $o_1$ | $c_1$ | 100 | 10 |
| $o_2$ | $c_2$ | 150 | 20 |
| $o_3$ | $c_1$ | 120 | null |
| $o_4$ | $c_1$ | 170 | null |
| $o_5$ | $c_2$ | 300 | null |
| $o_6$ | $c_1$ | 150 | 15 |
| $o_7$ | $c_2$ | 220 | null |

| cat | gross |
|---|---|
| $c_1$ | −15 |
| $c_2$ | +350 |

| cat | gross |
|---|---|
| $c_1$ | 280 |
| $c_2$ | 150 |

| cat | gross |
|---|---|
| $c_1$ | 265 |
| $c_2$ | 500 |

**Figure 3: Example TVR's and their relationships.**

*Definition 3.3 (TVR difference).* $\Delta R_t^{t'}$ defines a mapping from a time interval to a bag of tuples belonging to the same schema, such that there is a merge operator "+" satisfying

$$R_t + \Delta R_t^{t'} = R_{t'}.$$

Table ④ in Fig. 3 shows $\Delta(S \bowtie^{lo} R)_{t_1}^{t_2}$, the delta of snapshots $(S \bowtie^{lo} R)_{t_1}$ and $(S \bowtie^{lo} R)_{t_2}$. Here multiplicities $\# = +1$ and $\# = -1$ represent insertion or deletion of the corresponding tuple, respectively. The merge operator + is defined as additive union on bag-based relations, which adds up the multiplicities of tuples in bags.

Interestingly, a TVR can have different delta views. For instance, the delta $\Delta(S \bowtie^{lo} R)_{t_1}^{t_2}$ can be defined differently as Table ⑤ in Fig. 3. Here the merge operator + directly sums up the partial SUM values (the *gross* attribute) per *category*. For *category* $c_1$, summing up the partial SUM's in $\gamma(S \bowtie^{lo} R)_{t_1}$ and $\Delta\gamma(S \bowtie^{lo} R)_{t_1}^{t_2}$ yields the value in $\gamma(S \bowtie^{lo} R)_{t_2}$, i.e., $280 + (-15) = 265$. To differentiate these two merge operators, we denote the merge operator for $\Delta(S \bowtie^{lo} R)_{t_1}^{t_2}$ as $+^{\#}$, and the merge operator for $\gamma(S \bowtie^{lo} R)_{t_2}$ as $+^{sum}$.

This observation shows that the way to define TVR deltas and the merge operator + is not unique. In general, as studied in previous research [20, 27], the difference between two snapshots $R_t$ and $R_{t'}$ can have two types:

- $R_t$ and $R_{t'}$ may be different on the multiplicities of tuples. $R_t$ may miss tuples or have extra tuples than $R_{t'}$;
- $R_t$ and $R_{t'}$ may be different on the attribute values of tuples. $R_t$ may have different attribute values for some tuples compared to $R_{t'}$.

If we view the TVR difference from a multiplicity perspective, the merge operator combines the same tuples by adding up their multiplicities. From an attribute perspective, the merge operator groups tuples with the same primary key, and combines the delta updates on the changed attributes into one value. The above $+^{\#}$ and $+^{sum}$ merge operators correspond to the two perspectives respectively. Furthermore, for some merge operator +, there is an inverse operator −, such that $R_{t'} - R_t = \Delta R_t^{t'}$. For instance, the inverse operator $-^{sum}$ for $+^{sum}$ is defined as taking the difference of SUM values per *category* between two snapshots.

## 4 TVR REWRITING RULES

Most traditional query optimizers rely on rewriting rules based on relational algebra equivalence to explore the possible plan space. As the snapshots and deltas of TVR's are just traditional static relations, these relational rewriting rules still hold for the TVR model. However, for planning of progressive queries, traditional rewriting rules are not enough. Progressive-query planning introduces a new type of rewriting rules that we refer to as *TVR rewriting rules.* Compared to traditional rules, TVR rewriting rules are different since they show algebra equivalence along both the temporal aspect and across different views (i.e., snapshots/deltas) of one or multiple TVR's.

In this section, we propose a trichotomy of TVR rewriting rules, namely *TVR-generating rules*, *intra-TVR equivalence rules*,TVR's and *inter-TVR equivalence rules.* We will show how to model previous incremental computation techniques using these rules. This modeling enables us to unify existing incremental computation techniques when exploring the plan space, rather than previous research works that mostly study each technique in isolation.

### 4.1 TVR-Generating and Intra-TVR Rules

Let us start with an observation about existing research on incremental computation. As shown in Example 1.1, the view maintenance approach delivers partial results at $t_1$ by computing on the snapshots $S_{t_1}$ and $R_{t_1}$. At $t_2$, instead of computing the results for the query from scratch on the snapshots $S_{t_2}$ and $R_{t_2}$, it computes deltas on the delta inputs $\Delta S_{t_1}^{t_2}$ and $\Delta R_{t_1}^{t_2}$, and merges the deltas with the partial results at $t_1$ to have the results at $t_2$.

This observation shows a key notion—*delta query*—for incremental view maintenance. It is formally described as:

$$Q(R_{t'}) = Q(R_t) + \eth Q(R_t, \Delta R_t^{t'}). \quad (1)$$

Most existing work on incremental view maintenance revolves around delta queries and their use according to Eq. 1. As $Q(R_{t'})$ and $Q(R_t)$ are snapshots $Q_{t'}$ and $Q_t$, the query $\eth Q$ essentially gives a way to directly compute delta $\Delta Q_t^{t'}$, and the operator + essentially defines the merge operator between snapshots and deltas of TVR $Q$. Formally, these can be summarized into two categories, namely *TVR-generating rules* and *intra-TVR equivalence rules* as below. Note that as TVR rewriting rules are also based on relational algebra equivalence, they are expressed in terms of snapshots/deltas of TVR's that are all traditional relations. Next we discuss these three types of rules.

**TVR-Generating Rules**. These are rules for computing the snapshots/deltas of a TVR from its input TVR's. As examples, Def. 3.2 gives a set of rules for computing the snapshots of a TVR from the snapshots of its input TVR's. The delta query in Eq. 1 gives a set of rules for computing the deltas of a TVR from the snapshots/deltas of its input TVR's. Below, we list the delta queries for a few common relational operators. There are many studies in delta queries under different semantics [8, 9, 12, 17, 18]. The blue lines in Fig. 4(a) demonstrate a set of TVR-generating rules.

- **SELECT**: $\Delta[\sigma_\theta(Q)]_t^{t'} = \sigma_\theta(\Delta Q_t^{t'})$;
- **PROJECT**: $\Delta[\pi_{\bar{A}}(Q)]_t^{t'} = \pi_{\bar{A}}(\Delta Q_t^{t'})$;
- **JOIN**: $\Delta(Q \bowtie P)_t^{t'} = (\Delta Q_t^{t'} \bowtie P_t) \cup (Q_t \bowtie \Delta P_t^{t'}) \cup (\Delta Q_t^{t'} \bowtie \Delta P_t^{t'})$;
- **UNION**: $\Delta(Q \cup P)_t^{t'} = \Delta Q_t^{t'} \cup \Delta P_t^{t'}$;
- **AGGREGATE**: $\Delta(\gamma_{\bar{A};\Psi=sum} Q)_t^{t'} = \gamma_{\bar{A};\Psi=sum}(\Delta Q_t^{t'})$[1].

**Intra-TVR Equivalence Rules**. They define the conversion between snapshots and deltas of a single TVR. An example rule merges snapshots and deltas, i.e., $R_t + \Delta R_t^{t'} = R_{t'}$ and $\Delta R_t^{t'} + \Delta R_{t'}^{t''} = \Delta R_t^{t''}$. Another example rule takes the difference between snapshots/deltas if the merge operator "+" has an inverse operator −, e.g., $R_{t'} - R_t = \Delta R_t^{t'}$. The red lines in Fig. 4(a) demonstrate a set of such rules.

## 4.2 Inter-TVR Equivalence Rules

There are other incremental computation techniques that do not solely rely on delta queries. Here we have another important observation. Example 1.1 demonstrates a second progressive approach, namely stream computing. Different from view maintenance, at $t_1$, stream computing does not directly deliver the snapshot of $S \bowtie^{lo} R$. Instead, it delivers only the subset of results of $S \bowtie^{lo} R$ that are guaranteed not to be retracted in the future, essentially $S \bowtie R$. At $t_2$ when the data is known to be complete, stream computing computes the rest part of $S \bowtie^{lo} P$, essentially $S \bowtie^{la} P$, then pads with nulls to match the schema $schema(S) \cup schema(R)$[2].

This observation shows a family of approaches often used by incremental computation: without computing $Q$ directly, one can incrementally compute a set of different queries $\vec{Q'}$ and then transform the results of $\vec{Q'}$ to get that of $Q$.

$$Q(R) = Q''(\vec{Q'}(R)). \tag{2}$$

For instance, the stream computing approach decomposes $Q = S \bowtie^{lo} P$ into a non-retractable positive part $Q^P = S \bowtie R$, and a retractable part $Q^N = S \bowtie^{la} R$. This approach incrementally computes $Q^P$ using methods similar to incremental view maintenance. At $t_2$, $Q^P$ and $Q^N$ are combined to $Q$, i.e., $Q_{t_2} = {Q^P}_{t_2} + {Q^N}_{t_2}$. Similar decomposition holds for $\gamma$ in *summary* too, just with a different merge operator $+^{sum}$. This can be summarized into *inter-TVR equivalence rules.*

These rules express the equivalence relationship between multiple TVR's. For instance, decomposing $Q = Q^P + Q^N$ in stream computing is an example of inter-TVR equivalence rule, as shown as the blue lines in Fig. 4(b). Inter-TVR equivalence rules are very general, and can describe many existing incremental computation methods, as shown below.

*(1) Outer-join view maintenance*: [21] proposed a method to incrementally maintain outer-join views when one of its inputs is updated. The main idea can be summarized as two types of inter-TVR equivalence rules. The first type consists of a rule that essentially decomposes a query into three parts given updates to an input: a directly affected part ($Q^D$), an indirectly affected part ($Q^I$), and an unaffected part ($Q^U$), where $Q^D$, $Q^I$, and $Q^U$ are defined formally using terms in the join-disjunctive normal form of $Q$. At updates, the delta of $Q$ can be computed by merging the deltas of $Q^D$ and $Q^I$, i.e., $\Delta Q_t^{t'} = \Delta {Q^D}_t^{t'} + \Delta {Q^I}_t^{t'}$. Take query *sales_status* as an example. As the algorithm in [21] requires updating one relation at a time, we insert a virtual time point $t'$ between $t_1$ and $t_2$, assuming $R$ and $S$ are updated separately at $t'$ and $t_2$. At $t'$, $Q = S \bowtie^{lo} R$ is decomposed into $Q^D = S \bowtie R$ and $Q^I = S \bowtie^{la} R$. Similarly at $t_2$, $Q$ is decomposed into $Q^D = S \bowtie^{lo} R$ with no $Q^I$. Then we can apply TVR-generating rules on each $Q^I$ and $Q^D$ to incrementally compute them (e.g., $\Delta {Q^D}_{t_1}^{t'} = S_{t_1} \bowtie \Delta R_{t_1}^{t'}$), getting $\Delta {Q^D}_{t_1}^{t'} = (o_2, c_2, 150, 20)$, $\Delta {Q^I}_{t_1}^{t'} = (o_2, c_2, 150, \text{null})$, and $\Delta {Q^D}_{t'}^{t_2} = \{o_5, o_6, o_7\}$. Combining them yields the delta of $Q$ as in Table ④ of Fig. 3. [21] also proposed an inter-TVR equivalence rule for computing the delta of $Q^I$ using the delta of $Q^D$ and the snapshot of $Q$. For instance, $\Delta {Q^I}_{t_1}^{t'}$ can be computed

[1] $\gamma_{\bar{A};\Psi} Q$ denotes applying aggregate $\Psi$ on $Q$ grouped by $\bar{A}$.

[2] For brevity, throughout the discussion we omit this padding operation if it is clear from the context. This padding can simply be implemented using a project operator.

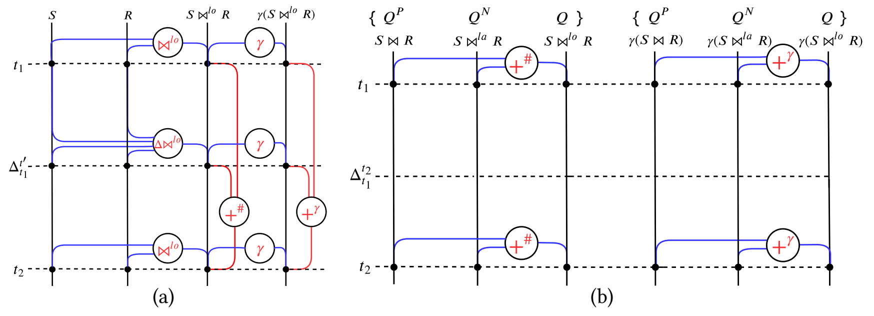


**Figure 4: (a) Examples of TVR-generating and intra-TVR rules, (b) examples of inter-TVR equivalence rules in stream computing, and (c) examples of inter-TVR equivalence rules in outer-join view maintenance.**

by replacing tuples in $\Delta {Q^D}^{t'}_{t_1}$ with *cost* = null. The blue lines in Fig. 4(c) show the above inter-TVR equivalence rules.

*(2) Higher-order view maintenance.* [3, 23] proposed a method that uses inter-TVR equivalence rules on deltas of TVR's, focusing on incrementally maintaining a query at updates to one of its inputs each time. Assuming a query $Q$ and updates to its inputs $\vec{R}$, the main idea is to transform the delta of a TVR $Q$ into an equivalent query $Q'$ evaluated over a set of materialized subqueries $M_1, \cdots, M_k$ and $\Delta\vec{R}^{t'}_t$, where $M_i$ does not involve $\Delta R^{t'}_t$ (update-independent), i.e.,

$$\Delta Q^{t'}_t = \eth Q(\vec{R}_t, \Delta\vec{R}^{t'}_t) = Q'(M_{1t}, \cdots, M_{kt}, \Delta\vec{R}^{t'}_t).$$

Let us take the *summary* query and updates to $S$ as an example. To maintain $\gamma(S \bowtie^{lo} R)$, it materializes an update-independent part

$$M = \gamma_{o\_id;total=\mathtt{SUM}(cost)}(R)$$

as an auxiliary view. It preprocesses $R$ by computing the total cost per *o_id*. When fed with the delta update to $S$, the delta to $\gamma(S \bowtie^{lo} R)$ can be computed by

$$Q' = \gamma_{category;\mathtt{SUM}(r)}(\Delta S^{t_2}_{t_1} \bowtie^{lo} M_{t_1}),$$

where $r = \mathtt{IF}(total\ \mathtt{IS\ NULL}, price, -total)$. $Q'$ computes the gross revenue per category by summing up the precomputed total cost in $M$ or the prices of the new orders added to $S$. The auxiliary view $M$ can be further incrementally maintained by repeatedly applying TVR-generating rules and the above inter-TVR equivalence rule to generate higher-order views.

## 4.3 Putting Everything Together

The above concepts and observations lay a good theoretical foundation for our progressive planning framework. Different TVR rules can be extended individually and they can work with each other automatically. For example, TVR-generating rules can be applied on any kind of decomposition introduced by inter-TVR equivalence rules. By applying these TVR rewriting rules along with all the existing relational algebra rewriting rules, we can explore the entire

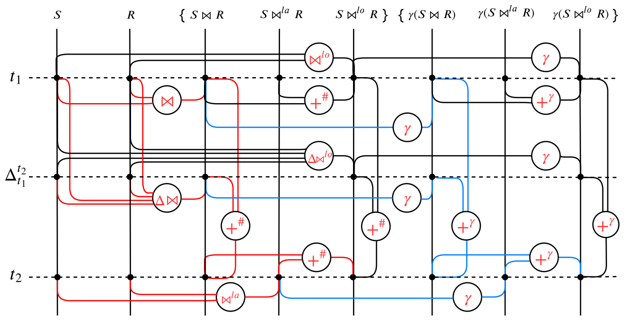


**Figure 5: The progressive plan space of Example 1.1**

progressive plan space. For instance, if we overlay Fig. 4(a) and 4(b), we can achieve the plan space as shown in Fig. 5. Any tree rooted at $\gamma(S \bowtie^{lo} R)_{t_2}$ is a valid progressive plan of the *summary* query, e.g., the red lines indicate the progressive plan used by stream computing.

In the next two sections, we discuss how to build an optimizer called Beanstalk based on the TVR model, including plan-space exploration (§ 5) and selecting an optimal progressive plan (§ 6).

## 5 PLAN-SPACE EXPLORATION

In this section we study how Beanstalk explores the space of progressive plans. Existing query optimizers do the exploratio only for a specific time. To support query optimization in progressive execution, we need to explore a much bigger plan space by considering not only relations at different times, but also transformations between them. In this section we study how to extend existing query optimizers to support cost-based optimization for progressive execution based on the TVR model. As an example, we consider one of the state-of-the-art solutions, the Cascades-style cost-based optimizer [15, 16]. We illustrate how to incorporate the TVR model into such an optimization framework to develop the corresponding optimizer called Beanstalk.

Beanstalk follows the general architecture of a Cascades-style optimizer to explore the plan space. It consists of two main modules. (1) *Memo*: It keeps track of the explored plan space, i.e., all plan alternatives generated, in a succinct data structure, typically represented as an AND/OR tree, for detecting redundant derivations and fast retrieval. (2) *Rule engine*: It manages all the transformation rules, which specify algebraic equivalence laws and suitable physical implementations of logical operators, monitors new plan alternatives generated in the memo. Whenever changes detected, the rule engine fires applicable transformation rules on the newly-generated plans to add more plan alternatives to the memo for further exploration.

The memo and rule engine of a traditional Cascades optimizer lack the capability to support the TVR model and thus progressive planning. In the following discussion, we focus on the key adaptations we make on the two modules to incorporate the TVR model.

## 5.1 Memo: Capturing TVR Relationships

The traditional memo only captures two levels of equivalence relationship: *logical equivalence* and *physical equivalence*. A logical equivalence class groups operators that generate the same result set; within each logical equivalence class, operators are further grouped into physical equivalence classes by their physical properties such as sort order, distribution etc. The "Traditional Memo" part in Fig. 6(a) depicts the traditional memo corresponding to the *sales_status* query. For brevity, we omit the physical equivalence classes. For instance, *LeftOuterJoin*[0,1] has Groups 0 and 1 as children, and it corresponds to the plan tree rooted at $\bowtie^{lo}$. Group 2 represents all plans logically equivalent to *LeftOuterJoin*[0,1].

However, the above two equivalences are far from enough to capture the rich relationship along the time dimension and between different incremental computation methods in the TVR model. For example, the relationship between snapshots and deltas of a TVR cannot be modeled using the logical equivalence due to simple facts: two snapshots at different times produce different relations, and the snapshots and deltas do not even have the same schema (deltas has an extra # column). On top of logical/physical equivalence classes, we explicitly introduce TVR nodes into the memo, and track the following relationships.

- **Intra-TVR relationship** specifies the snapshot/delta relationship between logical equivalence classes of operators and the corresponding TVR's.
- **Inter-TVR relationship** specifies the user-defined relationship between TVR's introduced by inter-TVR equivalence rules.

The "Beanstalk Memo" part in Fig. 6(a) depicts the new memo structure. For example, the original memo only models scanning the full content of $S$, i.e., $S_{t_2}$, represented by Group 0, while the new memo models two more scans: scanning the partial content of $S$ available at $t_1$ ($S_{t_1}$), and scanning the delta input of $S$ newly available at $t_2$ compared to $t_1$ ($\Delta S_{t_1}^{t_2}$). These two new scans are represented by Groups 3 and 5, and the memo uses an explicit TVR node $o$ to keep track of these intra-TVR relationships. Similarly, Group 2 is the snapshot at $t_2$ of TVR 2. There are also inter-TVR relationships tracked in TVR nodes. For example, the stream computing approach discussed in § 3 decomposes TVR 3 ($S \bowtie^{lo} R$) into the non-retractable part ($Q^P$) and the retractable part ($Q^N$), represented by TVR 3 ($S \bowtie R$) and TVR 4 ($S \bowtie^{la} R$), respectively. Similarly, the stream computing decomposition of TVR's 0, 1, 3, and 4 is marked as shown in the figure. It is worth noting that the above relationships are transitive. For example, as Group 7 is the snapshot at $t_2$ of TVR 3 and TVR 3 is the $Q^P$ part of TVR 2, it is the snapshot at $t_2$ of the $Q^P$ part of TVR 2.

## 5.2 Rule Engine: Enabling TVR Rewritings

The rule engine of `Beanstalk` supports both traditional SQL rewriting rules and TVR rewriting rules. Due to the fact that traditional rewriting rules match and generate patterns of relational operators only, and the memo of `Beanstalk` strictly subsumes a traditional Cascades memo, traditional rewriting rules can be adopted without modifications and work as is. Differently, TVR rewriting rules usually consist of both relational operators and TVR nodes and need special support. Specifically, `Beanstalk` defines TVR rewriting rules as a graph pattern of two types of nodes: *operator operands* and *TVR operands*, where operator operands are to match operators and TVR operands are to match TVR nodes. There are three types of edges: (1) edges between operator operands specify traditional parent-child relationship of operators; (2) edges between operator operands and TVR operands specify intra-TVR relationships; (3) the edges between TVR operands specify inter-TVR relationships. All nodes and the intra/inter-TVR edges can have predicates. Furthermore, different from traditional rules that can only register logical equivalence of new operator trees to existing ones, TVR rewriting rules can register new TVR nodes and new intra/inter-TVR relationships.

Fig. 7 depicts two TVR rewriting rules: Rule 1 is the TVR-generating rule for computing the delta of an inner join. It matches a tree rooted at an inner join, whose children $O_1$ and $O_2$ are snapshots of TVR's $V_1$ and $V_2$ at $t^*$, and each has a delta sibling $O_1'$ and $Q_2'$. Once matched, the rule will generate a delta join node taking $O_1$, $O_2$, $O_1'$, and $Q_2'$ as inputs and computing $O_1 \bowtie O_2' \cup O_1' \bowtie O_2 \cup O_1' \bowtie O_2'$, and register it as a delta sibling of the inner join, shown as dotted lines. Rule 2 is the inter-TVR equivalence rule of the stream computing approach. The rule matches a tree rooted at a left outer join operator, which is a snapshot at $t$ of a TVR ($V_3$ in the figure). Each of its two children $O_1$ ($O_2$) is snapshot at $t$ of a TVR,

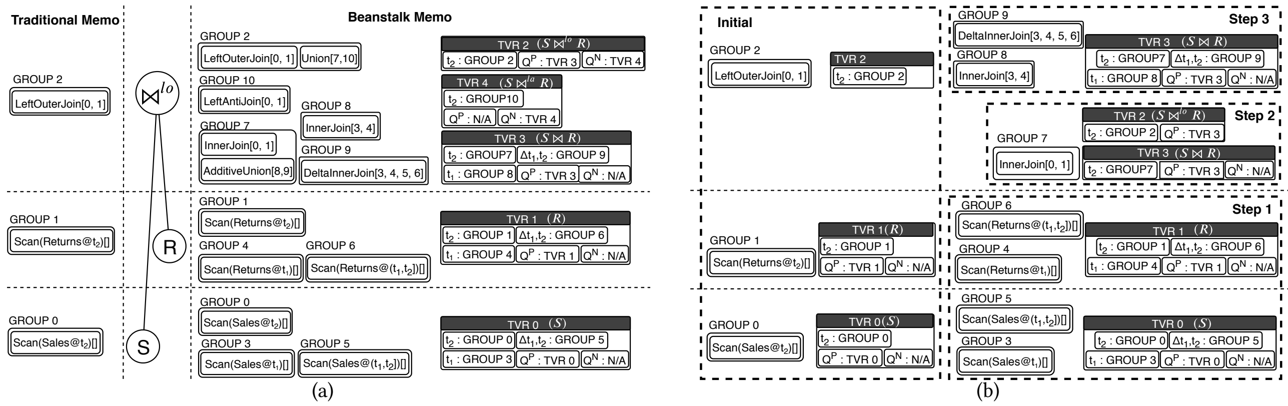


**Figure 6: (a) An example memo of query *sales_status*, and (b) an step-wise illustration of the growth of the memo.**

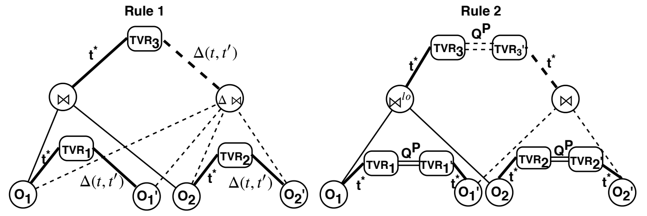


**Figure 7: Examples of TVR rewriting rule patterns.**

whose $Q^P$ part has a snapshot at $t$ referred to as $O_1'$ ($O_2'$). The rule will generate an inner join with $O_1'$ and $O_2'$ as children, and register this inner join to TVR $V_3$ as a snapshot at $t$ of $V_3$'s $Q^P$ part (i.e., $V_3'$).

To facilitate fast rule triggering, Beanstalk indexes the rule patterns by their operator operands and edges of intra/inter-TVR relationships. Whenever changes in Memo are detected, the corresponding rules are checked for firing. For example, if a new TVR node and its relationship edge with a logical equivalence class are added, Beanstalk only checks and fires the rules indexed by the relationship edge.

Fig. 6(b) demonstrates the growth of a memo in Beanstalk. In every step, we only draw the updated part due to space limitation. The memo starts with Group 0 to Group 2 and their corresponding TVR 0 to TVR 2. As given by the schedule, in step 1 we can first populate all the snapshots and deltas of the input relations, e.g., Group 3 to Group 6, and register the corresponding relationship in TVR 0 and TVR 1. Note that we can also populate their $Q^P$ and $Q^N$ relationships in the stream computing approach as for base input relations these relationships are trivial. In step 2, rule 2 matches the tree rooted at the left outer join in Group 2, generates the inner join of Group 7, and registers Group 7 to TVR 3 as the snapshot at $t_2$, and TVR 3 to TVR 2 as $Q^P$. In step 3, rule 1 matches the inner join in Group 7 and generates *DeltaInnerJoin*[3,4,5,6] as the delta of TVR 3. By following a similar process and applying other TVR rewriting rules, we can eventually populate the memo as shown in Fig. 6(a).

## 5.3 Speeding Up Exploration Process

Progressive planning explores a much bigger plan space than traditional query planning. Blindly generating all possible plan alternatives can waste a lot of search effort. We introduce heuristics to guide the plan-space exploration.

TVR rewriting rules are usually defined on logical operators. Therefore, we restrict TVR rewriting rules to only fire on patterns consisting of logical operators. There are different ways to compute a TVR delta. One is through taking the difference of two snapshots, and another could be through TVR-generating rules by computing directly from deltas of the inputs. Based on the experience of previous research on incremental computation, the plans generated by TVR-generating rules are usually more efficient. Therefore, for operators that are known to be easily incrementally maintained, such as filter and project, we assign a lower importance to intra-TVR rules for generating deltas to defer their firing. Once we find a delta that can be generated through TVR-generating rules, we skip the corresponding intra-TVR rules altogether.

Inside a TVR, snapshots and deltas that are consecutive in time can be merged together, leading to combinatorial explosion of rule applications. However, the merge order of these snapshots and deltas usually do not affect the cost of the final plan. Thus, we limit the exploration to a left-deep merge order. Specifically, we disable merging of consecutive deltas, but instead only searching for plans that merge a snapshot with its immediately consecutive delta. In this way, we always use a left-deep merge order.

## 6 SELECTING AN OPTIMAL PLAN

In this section we discuss how `Beanstalk` selects an optimal progressive plan in the explored space. The problem of selecting an optimal plan has two distinctions than existing query optimization:

- Costing the plan space and searching through the space need to consider the temporal execution of a plan.
- The optimal plan needs to take in consideration various sharing opportunities, both between different time points within a single query, as well as across upstream/downstream queries.

### 6.1 Time-Point Annotations of Operators

Costing the plan alternatives properly is crucial for correct optimization. However, as the temporal dimension is involved in query planning, costing is not as trivial as in a traditional query optimizer. Fig. 8(a) depicts one physical plan alternative derived from the plan rooted at $S \bowtie^{lo} R_{t_2}$ shown in red in Fig. 5. This plan only specifies the concrete physical operations taken on the data, but does not specify when these physical operations are executed. Actually, each operator in the plan usually has multiple choices of execution time. In Fig. 8(a), the time points annotated alongside each operator shows the possible temporal domain that each operator can be executed. For instance, snapshots $S_{t_1}$ and $R_{t_1}$ are available at $t_1$, and thus can execute at any time after that, i.e., $\{t_1, t_2\}$. Deltas $\Delta R_{t_1}^{t_2}$ and $\Delta S_{t_1}^{t_2}$ are not available until $t_2$, and thus any operators taking it as input, including the *IncrHashInnerJoin*, can only be executed at $t_2$. The temporal domain of each operator $O$, denoted t-dom($O$), can be defined inductively:

- For a base relation $R$, t-dom($R$) is the set of execution times that are no earlier than the time point that $R$ is available.
- For an operator $O$ with inputs $I_1, \cdots, I_k$,t-dom($R$) is the intersection of its inputs' temporal domains. t-dom($R$) = $\bigcap_{1 \leq j \leq k}$t-dom($I_j$).

To fully describe a physical plan, one has to assign each operator in the plan execution time from the corresponding temporal domain properly. We denote a specific execution time of an operator $O$ as $\tau(O)$. We have the following definition of a valid temporal assignment.

*Definition 6.1 (Valid Temporal Assignment).* An assignment of execution times to a physical plan is valid if and only if for each operator $O$, its execution time $\tau(O)$ satisfies $\tau(O) \in$ t-dom($O$) and $\tau(O) \geq \tau(O')$ for all operators $O'$ in the subtree rooted at $O$.

Fig. 8(b) demonstrates a valid temporal assignment of the physical plan in Fig. 8(a). As $S_{t_1}$ and $R_{t_1}$ are already available at $t_1$, the plan chooses to compute *HashInnerJoin* of $S_{t_1}$ and $R_{t_1}$ at $t_1$, as well as shuffling $S_{t_1}$ and $R_{t_1}$ in order to prepare for *IncrHashInnerJoin*. At $t_2$, the plan shuffles the new deltas $\Delta S_{t_1}^{t_2}$ and $\Delta R_{t_1}^{t_2}$, finishes *IncrHashInnerJoin*, and unions the results with that of *HashInnerJoin* computed at $t_1$. Note that if an operator $O$ and its input $I$ have different execution times, then the output of $I$ needs to be saved first at $\tau(I)$, and later loaded and fed into $O$ at $\tau(O)$, e.g., *Union* at $t_2$ and *HashInnerJoin* at $t_1$. The cost of *Save* and *Load* needs to be properly included in the plan cost. It is worth noting that some operators save and load the output as a by-product, for which we can spare *Save* and *Load*, e.g., *Exchange* of $S_{t_1}, R_{t_1}$ at $t_1$ for *IncrHashInnerJoin*.

### 6.2 Time-Point-Based Cost Functions

The cost of a progressive plan is defined under a specific assignment of execution times. Therefore, the optimization problem of searching for the optimal progressive plan is formulated as: given a plan space, find *the physical plan* and *its valid temporal assignment* that achieve the lowest cost. In this section, we introduce our cost model and optimization algorithm for this problem.

We inherit the general cost model used in traditional query optimizers. That is, the cost of a plan is the sum of the cost of all its operators. However, existing cost functions need to be extended to reflect resource cost at different times. Below we show a few typical extensions of the cost function. We denote the existing cost function as $c$, which is annotated with a subscript $i$ to distinguish the cost at different time $t_i$, whereas the total cost of a plan is denoted as $\tilde{c}$. As before, $c$ can be a simple number, e.g., estimated monetized resource cost, or a structure, e.g., a vector of CPU time and I/O.

(1) $\tilde{c}_w(O) = \sum_{i=1..T} w_i \cdot c_i(O)$. The extended cost of an operator is a weighted sum of its cost at each time $t_i$. For example, for the example setting in § 2.2, $w_1 = 0.2$ for $t_1$ if $w_2$ for $t_2$ is set to 1.
(2) $\tilde{c}_v(O) = [c_1(O), \cdots, c_T(O)]$. The extended cost is a vector combining costs at different times. $\tilde{c}$ can be compared using a lexical order entry-by-entry in the vector.

The dynamic programming algorithm used predominantly in existing query optimizers also need to be adapted to handle the cost model extensions. In existing query optimizers, the state space of the dynamic programming is the set of all operators in the plan space, represented as $\{O\}$. Each operator $O$ records the best cost of all the subtrees rooted at $O$. We extend the state space by considering all combinations of operators and their execution times, i.e., $\{O\} \times$ t-dom($\{O\}$). Also instead of recording a single optimum, each operator $O$ records multiple optimums, one for each execution time $\tau(O)$, which represents the best cost of all the subtrees rooted at $O$ if $O$ is executed at $\tau$. During optimization, the state-transition function is as Eq. 3. That is, the best cost of $O$ if executed at $\tau$ is the best cost of all possible plans of computing $O$ with

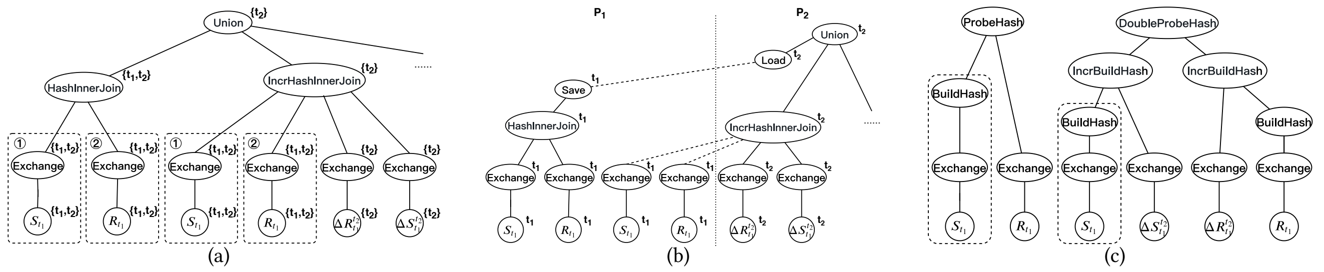


**Figure 8: (a) An example temporal physical plan space and (b) an example temporal assignment of the physical plan for query *sales_status*; (c) the sharable sub-plans under our physical operator model.**

all possible valid temporal assignments compatible with $\tau$.

$$\tilde{c}[O,\tau] = min_{\forall \text{ valid } \tau_j}(\sum_j \tilde{c}[I_j,\tau_j] + c_\tau(O)). \quad (3)$$

We have the following correctness result of the above dynamic programming algorithm. In general, we can apply dynamic programming to the optimization problem for any cost function satisfying the property of optimal substructure.

Theorem 6.2. *The optimization problem under cost functions $\tilde{c}_w$ and $\tilde{c}_v$ satisfies the property of optimal substructure, and dynamic programming is applicable.*

## 6.3 Sharing States among Time Points

There are many sharing opportunities in the plan space, which can be crucial to the query performance. A shared sub-plan between $P_i$ and $P_j$ in a progressive plan is essentially an intermediate state that can be saved by $P_i$ and reused by $P_j$. E.g., in Fig. 8(a), since both *HashInnerJoin* and *IncrHashInnerJoin* require shuffling $S_{t_1}$ and $R_{t_1}$, the two relations can be shuffled only once and reused for both joins. The parts ① and② circled in dashed lines depict the sharable sub-plans.

Finding the optimal common sub-plans to share is a multi-query optimization (MQO) problem, which has been extensively studied [19, 24, 28]. In this work, we extend the latest MQO algorithm in [19], which proposes a greedy framework on top of Cascade-style optimizers for MQO. For the sake of completeness, we list the algorithm in Algo. 1, by highlighting the extensions for progressive planning. The algorithm runs in an iterative fashion. In each iteration, it picks one more candidate from all possible shareable candidates, which if materialized can minimize the plan cost (line 4), where *bestPlan*($\mathbb{S}$) means the best plan with $\mathbb{S}$ materialized and shared. The algorithm terminates when all candidates are exhausted or adding candidates can no longer improve the plan cost. However, as progressive planning needs to consider the temporal dimension, the shareable candidates are no longer solely the set of shareable sub-plans, but pairs of a shareable sub-plan $s$ and a choice of its execution time $\tau(s)$. Pair $\langle s,\tau(s)\rangle$ means computing and materializing the sub-plan $s$ at time $\tau(s)$, which can only benefit the computation that happens after $\tau(s)$. For instance, considering the physical plan space in Fig. 8(a), the sharable candidates are $\{\langle ①, t_1\rangle, \langle ①, t_2\rangle, \langle ②, t_1\rangle, \langle ②, t_2\rangle\}$. The speed-up optimizations in [19] are still applicable to Algo. 1.

**Algorithm 1** Greedy Algorithm for MQO

```
1: $\mathbb{S} = \emptyset$
2: $\mathbb{C}$ = **shareable candidate set consiting of all shareable nodes and their potential execution times** $\{\langle s, \tau(s)\rangle\}$
3: **while** $\mathbb{C} \neq \emptyset$ **do**
4:     Pick $\langle s,\tau(s)\rangle \in \mathbb{C}$ that minimizes $\tilde{c}(bestPlan(\mathbb{S}'))$ where $\mathbb{S}' = \{\langle s,\tau(s)\rangle\} \cup \mathbb{S}$
5:     **if** $\tilde{c}(bestPlan(\mathbb{S}')) < \tilde{c}(bestPlan(\mathbb{S}'))$ **then**
6:         $\mathbb{C} = \mathbb{C} - \{\langle s,\tau(s)\rangle\}$
7:         $\mathbb{S} = \mathbb{S}'$
8:     **else**
9:         $\mathbb{C} = \emptyset$
10:    **end if**
11: **end while**
12: **return** $\mathbb{S}$
```

As expanded with execution time options, the enumeration space of the shareable candidate set becomes much larger than the original algorithm in [19]. Interestingly, we find that under certain cost models we can reduce the enumeration space down to a size comparable to the original algorithm, formally summarized in Theorem 6.3. Theorem 6.3 relies on the fact that materializing a shareable sub-plan at its earliest possible time subsumes other materialization choices. Due to space limit, we omit the proof.

Theorem 6.3. *For cost function $\tilde{c}_w$ satisfying $w_i < w_j$ if $i < j$, or cost function $\tilde{c}_v$ satisfying the property that entry i has a lower priority than entry j if i < j in the lexical order, we only need to consider the earliest valid execution time for each shareable sub-plan. That is, for each s, we only need to consider the shareable candidate $\langle s, min(t\text{-}dom(s))\rangle$ in Algorithm 1.*

However, traditional ways of modeling physical operators treat the inner implementation of operators as black boxes, which are not able to fully utilize the sharing opportunities. To understand that, consider again *HashInnerJoin* and *IncrHashInnerJoin* in Fig. 8(b). Both joins build hash tables on one or multiple inputs. By exploiting this, we can save not only the shuffling of the input data, but also the hash-building operations. In Beanstalk, we explicitly model all inner data structures of physical operators. For instance, we model hash-based join operators by two separate operators *BuildHash* and *ProbeHash*. *BuildHash* reads an input and builds a hash table in memory using the join predicates as keys. *ProbeHash* reads the other input, probes the built hash table, and produces join tuples. Fig. 8(c) shows an example implementation of *HashInnerJoin* and *IncrHashInnerJoin* using *BuildHash* and *ProbeHash*. The *BuildHash* subtrees circled in dashed lines show another sharing opportunity. Similar modeling can be applied to other operators such as hash-based aggregates.

## 6.4 Discussions

In this section, we discuss several issues related to optimization of progressive execution.

**Re-optimization of progressive plan**. A proposed schedule can be tentative and subject to change whenever the environment (such as the cluster resource usage) changes. In this case, the progressive plan for the original schedule may be only half way done, and have already generated and saved some intermediate states. Blindly redoing progressive planning from scratch for a new schedule can waste the saved intermediate states. Instead, during re-planning, we can treat the saved states as materialized views, and provide them to the optimizer for a joint consideration of the half-way finished old schedule and the new one.

**Workload selection for a schedule**. A full job pipeline can contain many queries. Taking the entire pipeline as a single workload for progressive planning may lead to an excessively large plan space, and may be too rigid for a tentative schedule. This problem can be avoided by planning a big workload piece-wise. Specifically, we can divide a big workload $W$ into a sequence of smaller disjoint workloads $[W_1, \ldots, W_k]$, where $W_i \cap W_j = \emptyset$. These $W_i$'s are topologically ordered according to their dependency relation, i.e., $W_{i+1}$ takes the outputs of $\{W_1, \ldots, W_i\}$ as its input. We plan each $W_i$ separately in order. To make sure the progressive plan of $W_i$ is compatible with those of its downstream queries, we can include the queries dependent on $W_i$ up to a few steps, denoted $N(W_i)$. We can plan $W_i$ and $N(W_i)$ jointly, and only retain the physical plan corresponding to $W_i$ in the end.

## 7 EXPERIMENTS

In this section, we evaluate the effectiveness and performance of Beanstalk using the TPC-DS benchmark and 3 real-world analysis workloads from the internal big data warehouse of CorpX. We chose 3 query pipelines of different scales and complexities: *Pipeline A* had 13K queries with a max dependency depth of 28, *Pipeline B* had 1K queries with a max dependency depth of 3, and *Pipeline C* had 3K queries with a max dependency depth of 15. We implemented several progressive computation techniques, including the view maintenance algorithms in [12, 17, 18], the outer-join view maintenance algorithm in [21], the higher-order view maintenance algorithm in [3], and the traditional stream computation approach. Beanstalk was implemented on Apache Calcite 1.17.0 [10], using a single-threaded core engine.

The CorpX internal query pipelines are all recurrent daily jobs. Originally they were scheduled to run every day at midnight on the data of the previous day, and TPC-DS queries were assumed to run in the same way. All experiments were conducted with an execution schedule of three time points, namely 14:00, 19:00, and 24:00, unless specified otherwise. The two early-execution time points (14:00 and 19:00) were chosen to simulate a cluster usage pattern based on the observation that the cluster was often under-utilized at those times. For every pipeline, we experimentally evaluated progressive plans generated by Beanstalk by comparing them with baseline plans generated by a traditional query optimizer, which ran in one shot at the last time point when all its input data becomes available. We used the cost function $\tilde{c}_v(O)$ described in § 6.2, corresponding to the scenario where the cluster had spare resources at the moment, but was expected to be busier in the near future. The cost estimation of CPU, IO, memory, and network was using standard techniques based on a distributed batch execution engine.

## 7.1 Effectiveness of Progressive Planning

We first compared the estimated cost of a progressive plan with the baseline plan. Note that the baseline plan had only one cost incurred at the last time point, while a progressive plan had a cost per time point in the schedule.

First we evaluated the peak-hour resource usage of both plans for each pipeline. We computed the last cost of the progressive plan divided by the baseline cost for each query in the pipeline, and presented the ratio as a cumulative distribution across all queries. The results are shown in Fig. 9(a). As we can see, queries were progressive and half of the queries had about 45 to 65 percent of its computation task at the last time point. Early computation was able to utilize the idle resources in the cluster. The cluster became less busy later and the query latency of the last run was also shortened.

Next we evaluated the overhead of progressive plans in terms of the total cost of tasks at all the time points compared with the baseline cost. A similar cost-ratio distribution is shown in Fig. 9(b). We can see that the cost ratio ranged from 1.0 to 2.0 for the majority of queries. This overhead was mainly due to the extra I/O and computation required to progressively compute the queries over time.

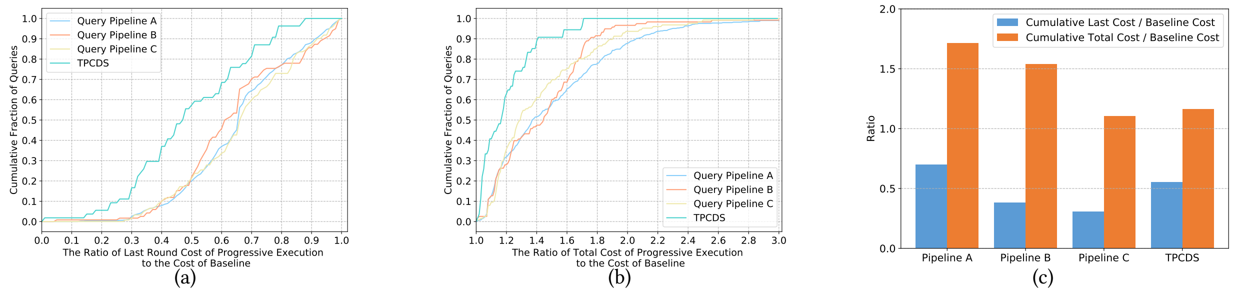


**Figure 9: Estimated cost comparison between progressive and baseline plans. (a) Last cost comparison per query; (b) total cost comparison per query; and (c) cluster-wise cumulative cost comparison across all queries.**

To evaluate the effect of progressive computing on the cluster usage, we accumulated both the costs of the progressive plans and those of the baseline plan. Fig. 9(c) shows the results. We can see that the resources used at the last time point were significantly smaller compared to the original plans, while the overall resource overhead was not high.

## 7.2 Performance of Progressive Planning

**Table 1: Statistics of Selected Representative Queries**

| Query | Q22 | Q20 | Q43 | Q67 | Q27 | Q99 | Q85 | Q91 | Q5 | Q33 |
|---|---|---|---|---|---|---|---|---|---|---|
| # Joins | 2 | 2 | 2 | 3 | 4 | 4 | 6 | 6 | 7 | 9 |
| # Aggregates | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 4 | 4 |
| # Sub-Queries | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 7 | 7 |

In this section, we study the performance aspect of progressive planning using the TPC-DS benchmark.

**Overall Planning Speed**. We first studied the overall planning performance by comparing progressive planning with traditional planning. Fig. 10(a) shows the total planning time on all TPC-DS queries. As shown, progressive planning is in general slower than the traditional planning by 57X on average. Q31 takes the longest optimization time, around 186 seconds. Note that this is still within tolerable. This is mainly because that progressive planning explores a much bigger plan space than traditional query planning, due to jointly considering multiple incremental computation methods and multiple schedule time points. Nevertheless this is worthwhile considering the potential benefit of cluster resource saving and smoothing progressive plans can achieve.

**Query Complexity** To study the impact of query complexity on performance, we picked ten queries from TPC-DS with various number of joins, aggregates and subqueries, and listed them in Table 1. Fig. 10(b) further breaks down their optimization time into time spent on plan space exploration (§ 3 and § 4) and multi-query cost optimization (§ 6.3). As we can see when query complexity increases, the plan space becomes larger and thus more time spent on exploration. The time of MQO grows even faster because the algorithm is non-linear to the size of the plan space.

**Number of Time Points in the Schedule**. Fig. 10(c) illustrates the impact of schedule size by changing from two to four time points, and plots the total number of rules fired (linear to size of the plan space explored). As expected, the size of the plan space grows linearly with the schedule size.

In practice, since at planning time we typically do not know much about future information any better than statistical estimations, we don’t see much benefit of further increase the number of time points. In general, what matters is what to run and save at current time that will benefit the future. What to run at future time can be re-optimized again with the exact information available then.

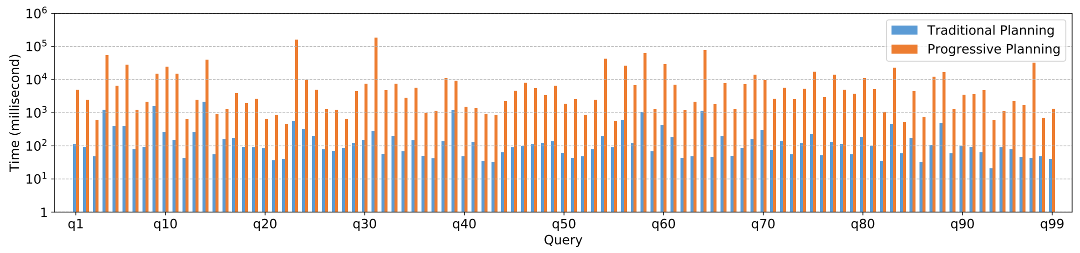
**Optimization Breakdown**. With a four time point schedule, Fig. 10(d) shows the performance impact when certain space trimming techniques (recall in § 5.3) are turned off.

With `Beanstalk` as the baseline, *without TVR rule importance* fires the diff rule (the rule of generating a delta by taking the difference of two snapshots) normally upon rule match as opposed to delaying and possibly skipping. Many useless deltas are thus generated and propagated downstream in the plan space, resulting in a big performance hit.

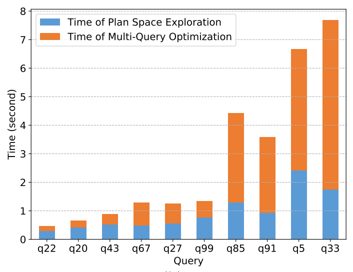
A similar situation happened with *without guided merge order* when we allow all merges of snapshots and deltas that are adjacent in time, rather than left-deep merge only. The number of tables generated per TVR becomes quadratic rather than linear to the schedule size.

## 8 RELATED WORK

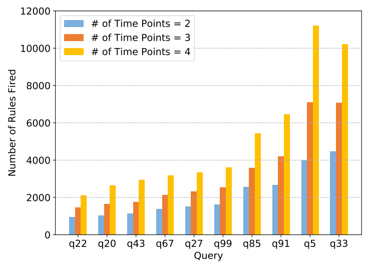
**Incremental Computation**. There is a rich body of research on various forms of incremental computation, ranging from incremental view maintenance (IVM), stream computing, to approximate/interactive query answering and so on. Incrementally maintaining query answers in the form of views has been intensively studied before. This problem has been considered under both the set [8, 9] and bag [12, 18] relational algebra, for queries with outer joins [17, 21], and using higher-order maintenance methods [3]. Previous studies mainly focused on the creation of delta queries. Stream



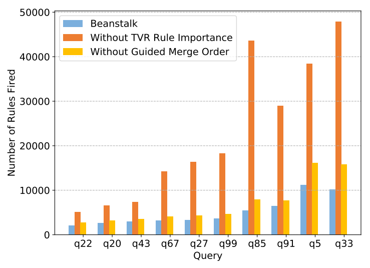
(a)



(b)



(c)



(d)

**Figure 10: (a) Planning performance comparison on the TPC-DS benchmark between the traditional query planning and our progressive planning with a schedule consisting of three time points. (b) Planning time break-down for representative TPC-DS queries of different complexities. (c) # of rules fired during progressive planning w.r.t. # of time points in the schedule. (d) Effect of plan-space exploration speed-up optimizations.**

computing [1, 14, 22, 26] adopts incremental processing (e.g., sliding windows) and sublinear-space algorithms to process updates and deltas. Many approximate query answering studies [2, 5, 11] focused on constructing optimal samples to improve query accuracy. Approximate/interactive query answering studies [25, 27] used either proactive or trigger-based progressive computation to achieve low query latency. [27] proposed a probability-based pruning technique for incremental computing complex queries including those with nested queries. [25] focused on selecting optimal states to materialize for different incremental data arrivals. These studies proposed incremental computation techniques in isolation, and do not have a general cost-based optimization framework. In addition, they can be integrated into our `Beanstalk` optimizer, and contribute to a unified plan space for searching the optimal progressive plan.

**Query Planning for Incremental Computation**. Some previous work on incremental computation also proposed query optimization solution. [3] proposed rewriting rules and a cost-based approach to determining higher-order view sets to materialize. The core technique in [25] is a dynamic programming algorithm, which given a physical incremental plan, selects intermediate states for processing deltas with a memory budget, by considering future data arrival patterns. These optimization techniques all focus on a limited plan space and does not support planning for general progressive computation techniques. For example, DBToaster [3] does not consider other advanced incremental techniques such as [21] and complex temporal and across-query dependencies during planning. [25] relies on an existing database optimizer to generate a physical plan first, thus cannot explore the plan space joining multiple incremental computation techniques. Big data systems such as [13] use [7] as the optimizer and provide basic optimization for stream queries. They only considered basic stream computing techniques, and do not support planning for multiple queries under a multi-time schedule. Instead, our proposed `Beanstalk` optimizer provides a general framework for users to integrate various incremental computation techniques, and explores the unified plan space using cost-based search. This optimizer can also consider complex dependencies both along the time dimension and across queries. In short, the plan space of our optimizer subsumes those of earlier studies.

**Streaming Semantic Models**. [4] proposes the CQL language and semantic model for stream processing. It uses time-varying relations to uniformly model relations and streams. Their model does not include operations and rewriting rules on top of TVR's, and thus cannot fully model progressive computation. Recent work [6] proposed to integrate streaming into the SQL standard, and briefly mentioned in their proposal that TVR's can have very rich representation forms to serve as the unified basis of both relations and streams. Our TVR model does a thorough study of TVR's and their operations, as well as rewriting rules of TVR's.

## 9 CONCLUSION

We presented the time-varying relation model and an optimizer framework Beanstalk based on the theory for planning progressive execution. Beanstalk allows users to explore the plan space that unifies all kinds of progressive computation techniques, and provides a cost-based turn-key solution to decide the optimal progressive plan. Besides optimizing cluster resource usage, our optimizer framework can serve progressive planning in its most general form under various application scenarios.

## REFERENCES

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. 2005. The design of the borealis stream processing engine.. In *Cidr*, Vol. 5. 277–289.

[2] Swarup Acharya, Phillip B Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. 1999. The Aqua approximate query answering system. In *ACM Sigmod Record*, Vol. 28. ACM, 574–576.

[3] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB* 5, 10 (2012), 968–979.

[4] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal* 15, 2 (2006), 121–142.

[5] Brian Babcock, Surajit Chaudhuri, and Gautam Das. 2003. Dynamic sample selection for approximate query processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data.* ACM, 539–550.

[6] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. 2019. One SQL to Rule Them All. *CoRR* abs/1905.12133 (2019). arXiv:1905.12133 http://arxiv.org/abs/1905.12133

[7] 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 (SIGMOD '18).* ACM, New York, NY, USA, 221–230. https://doi.org/10.1145/3183713.3190662

[8] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. 1986. Efficiently Updating Materialized Views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data (SIGMOD '86).* ACM, New York, NY, USA, 61–71. https://doi.org/10.1145/16894.16861

[9] O. Peter Buneman and Eric K. Clemons. 1979. Efficiently Monitoring Relational Databases. *ACM Trans. Database Syst.* 4, 3 (Sept. 1979), 368–382. https://doi.org/10.1145/320083.320099

[10] Apache Calcite. 2019. (2019). https://calcite.apache.org

[11] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. 2007. Optimized stratified sampling for approximate query processing. *ACM Transactions on Database Systems (TODS)* 32, 2 (2007), 9.

[12] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. 1995. Optimizing Queries with Materialized Views. In *Proceedings of the Eleventh International Conference on Data Engineering (ICDE '95).* IEEE Computer Society, Washington, DC, USA, 190–200. http://dl.acm.org/citation.cfm?id=645480.655434

[13] Apache Flink. 2019. (2019). https://flink.apache.org

[14] Thanaa M Ghanem, Ahmed K Elmagarmid, Per-Åke Larson, and Walid G Aref. 2010. Supporting views in data stream management systems. *ACM Transactions on Database Systems (TODS)* 35, 1 (2010), 1.

[15] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *Data Engineering Bulletin* 18 (1995).

[16] Goetz Graefe and William J McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering.* IEEE, 209–218.

[17] Timothy Griffin and Bharat Kumar. 1998. Algebraic Change Propagation for Semijoin and Outerjoin Queries. *SIGMOD Rec.* 27, 3 (Sept. 1998), 22–27. https://doi.org/10.1145/290593.290597

[18] 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 (SIGMOD '95).* ACM,

New York, NY, USA, 328–339. https://doi.org/10.1145/223784.223849

[19] Tarun Kathuria and S. Sudarshan. 2017. Efficient and Provable Multi-Query Optimization. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '17)*. ACM, New York, NY, USA, 53–67. https://doi.org/10.1145/3034786.3034792

[20] Willis Lang, Rimma V. Nehme, Eric Robinson, and Jeffrey F. Naughton. 2014. Partial Results in Database Systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 1275–1286. https://doi.org/10.1145/2588555.2612176

[21] Per-Åke Larson and Jingren Zhou. 2007. 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*. 56–65. https://doi.org/10.1109/ICDE.2007.367851

[22] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. 2003. Query processing, resource management, and approximation in a data stream management system. CIDR.

[23] Milos Nikolic, Mohammad Dashti, and Christoph Koch. 2016. 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 '16)*. ACM, New York, NY, USA, 511–526. https://doi.org/10.1145/2882903.2915246

[24] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. 2000. Efficient and Extensible Algorithms for Multi Query Optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*. ACM, New York, NY, USA, 249–260. https://doi.org/10.1145/342009.335419

[25] 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

[26] Hetal Thakkar, Nikolay Laptev, Hamid Mousavi, Barzan Mozafari, Vincenzo Russo, and Carlo Zaniolo. 2011. SMM: A data stream management system for knowledge discovery. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 757–768.

[27] Kai Zeng, Sameer Agarwal, and Ion Stoica. 2016. iOLAP: Managing Uncertainty for Efficient Incremental OLAP. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1347–1361. https://doi.org/10.1145/2882903.2915240

[28] Jingren Zhou, Per-Ake Larson, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. 2007. Efficient Exploitation of Similar Subexpressions for Query Processing. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD '07)*. ACM, New York, NY, USA, 533–544. https://doi.org/10.1145/1247480.1247540