The VLDB Journal
https://doi.org/10.1007/s00778-023-00785-1

REGULAR PAPER

# Tempura: a general cost-based optimizer framework for incremental data processing (Journal Version)

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 Hou[2] · Tao Guan[2] · Chen Li[1] · Jingren Zhou[2]

Received: 20 January 2022 / Revised: 14 January 2023 / Accepted: 25 January 2023

**Abstract**
Incremental processing is widely adopted in many applications, ranging from incremental view maintenance, stream computing, to recently emerging progressive data warehouse and intermittent query processing. Despite many algorithms developed on this topic, none of them can produce an incremental plan that always achieves the best performance, since the optimal plan is data dependent. In this paper, we develop a novel cost-based optimizer framework, called Tempura, for optimizing incremental data processing. We propose an incremental query planning model called TIP based on the concept of time-varying relations, which can formally model incremental processing in its most general form. We give a full specification of Tempura, which can not only unify various existing techniques to generate an optimal incremental plan, but also allow the developer to add their rewrite rules. We study how to explore the plan space and search for an optimal incremental plan. We evaluate Tempura in various incremental processing scenarios to show its effectiveness and efficiency.

**Keywords** Incremental query processing · Volcano cost-based optimizer · Time-varying relations

## 1 Introduction

Incremental processing is widely used in data computation, where the input data to a query is available gradually, and the query computation is triggered multiple times each processing a delta of the input data. Incremental processing is central to database views with incremental view maintenance (IVM) [3,14,20,33] and stream processing [1,5,16,37,47]. It has been adopted in various application domains such as active databases [4], resumable query execution [12], and approximate query processing [13,29,54]. New advancements in big data systems make data ingestion more real-time and analysis increasingly time sensitive, which boost the adoption of the incremental processing model. Here, are a few examples of emerging applications.

*Progressive Data Warehouse* [49]. Enterprise data warehouses usually have a large amount of automated routine analysis jobs, which have a stringent schedule and dead-

✉ Botong Huang
botong.huang@alibaba-inc.com

Zuozhi Wang
zuozhiw@ics.uci.edu

Kai Zeng
zengkai.zk@alibaba-inc.com

Wei Chen
wickeychen.cw@alibaba-inc.com

Xiaozong Cui
xiaozong.cxz@alibaba-inc.com

Bo Wang
yanyu.wb@alibaba-inc.com

Ji Liu
niki.lj@alibaba-inc.com

Liya Fan
liya.fly@alibaba-inc.com

Dachuan Qu
dachuan.qdc@alibaba-inc.com

Zhenyu Hou
zhenyuhou.hzy@alibaba-inc.com

Tao Guan
tony.guan@alibaba-inc.com

Chen Li
chenli@ics.uci.edu

Jingren Zhou
jingren.zhou@alibaba-inc.com

[1] University of California, Irvine, USA

[2] Alibaba Group, Hangzhou, China

Published online: 20 March 2023

line determined by various business logic. For example, at Alibaba, daily report queries are scheduled after 12 am when the previous day's data has been fully collected, and the results must be delivered by 6 am sharp before the bill-settlement time. These routine analysis jobs are predominately handled using batch processing, causing dreadful "rush hour" scheduling patterns. This approach puts pressure on resources during traffic hours, and leaves the resources over-provisioned and wasted during the off-traffic hours. Incremental processing can answer routine analysis jobs progressively as data gets ingested, and its scheduling flexibility can be used to smoothen the resource skew.

*Intermittent Query Processing* [45]. Many modern applications require querying an incomplete dataset with the remaining data arriving in an intermittent yet predictable way. Intermittent query processing can leverage incremental processing to balance latency for maintaining standing queries and resource consumption by exploiting knowledge of data-arrival patterns. For instance, when querying dirty data, the data is usually first cleaned and then fed into a database. The data cleaning step can quickly spill the clean data but needs to conduct a time-consuming processing on the dirty data. Intermittent query processing can use incremental processing to quickly deliver informative but partial results to the user, before delivering the final results on the fully cleaned data.

A key problem behind these applications is how to generate an efficient incremental plan for a query. Previous studies focused on various aspects of the problem, e.g., incremental computation algorithms for a specific setting such as [3,14,33], or algorithms to determine which intermediate states to materialize [39,45,55]. The following example based on two commonly used algorithms shows that none of them can generate an incremental computation plan that is always optimal, since the optimal plan is *data dependent*.

***Example 1*** (Reporting consolidated revenue)

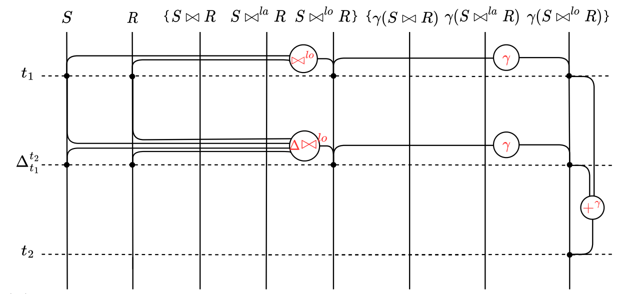
```
summary =
  WITH sales_status AS (
    SELECT sales.o_id, category, price, cost
    FROM sales LEFT OUTER JOIN returns
    ON sales.o_id = returns.o_id )
  SELECT category, SUM(IF(cost IS NULL, price, -cost))
  FROM sales_status GROUP BY category
```

In the progressive data warehouse scenario, consider a routine analysis job in Example 1 that reports the gross revenue by consolidating the sales orders with the returned ones. We want to incrementally compute the job as data gets ingested, to utilize the cheaper free resources occasionally available in the cluster. We want to find an incremental plan with the optimal resource-usage pattern, i.e., carrying out as much early computation as possible using cheaper free resources to keep the overall resource bill low. This query can be incrementally computed in different ways as the data in tables sales and returns becomes available gradually. For instance, consider two basic methods used in IVM and stream computing.

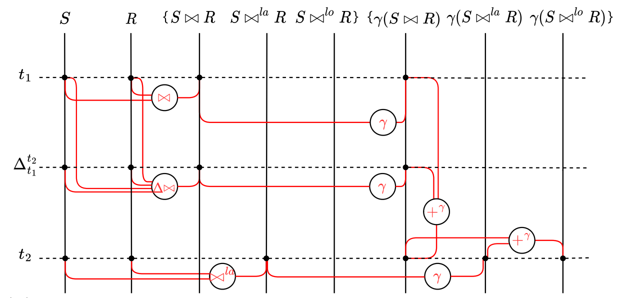
(1) A typical view maintenance approach (denoted as **`IM-1`**) treats summary as views [14,20,21,54]. It always maintains summary as if it is directly computed from the data of sales and returns seen so far. Therefore, even if a sales order will be returned in the future, its revenue is counted into the gross revenue temporarily. Figure 1a shows the execution plan using this approach. In Fig. 1, each vertical line represents a relation that evolves over time and each horizontal line represents a specific time point. At time $t_1$, the plan computes the left-outer join and aggregation results. After the new data from $t_1$ to $t_2$ arrive, the query plan computes the incremental left-outer join and aggregation results. Finally, the results at $t_1$ and the incremental results from $t_1$ to $t_2$ of the aggregation are combined to produce the final results.

(2) A typical stream-computing method (denoted as **`IM-2`**) avoids such retraction [24,34,36,46]. It holds back sales orders that do not join with any returns orders until all data is available. Figure 1b shows the execution plan using this approach. At $t_1$, the plan computes only the aggregation results of an inner join instead of a left-outer join. After the data from $t_1$ to $t_2$ arrive, the plan incrementally updates the aggregation results of the inner join. Next, the complete aggregation results of a left-anti join are computed using all input data up to $t_2$. Finally, the results of inner join aggregation and left-anti join aggregation are combined to produce the final results. Clearly, if returned orders are rare, `IM-1`can maximize the amount of early computation and thus deliver better resource-usage plans. Otherwise, if returned orders are often, `IM-2`can avoid unnecessary re-computation caused by retraction and thus be better. (See Sect. 2.2 for a detailed discussion.) This analysis shows that different data statistics can lead to different preferred methods.

Since the optimal plan for a query given a user-specified optimization goal is data dependent, a natural question is how to develop a principled cost-based optimization framework to support efficient incremental processing. To our best knowledge and also to our surprise, there is no such a framework in the literature. In particular, existing solutions still rely on users to empirically choose from individual incremental techniques, and it is not easy to combine the advantages of different techniques and find the plan that is truly cost optimal. When developing this framework, we face more challenges compared to traditional query optimization [18,43] (see Sect. 2.2): (1) Incremental query planning requires trade-off analysis on more dimensions than traditional query planning, such as different incremental



(a) Incremental query plan produced by approach `IM-1`. This plan actively maintains the most recent view results on each update.



(b) Incremental query plan produced by approach `IM-2`. This plan avoids retractions by only computing the delta of inner joins. When all data is available, left anti join results are added to compute the final results.

**Fig. 1** Comparison of incremental query plans produced by approach `IM-1`and approach `IM-2`

computation methods, data arrival patterns, which states to materialize, etc. (2) The plans for different incremental runs are correlated and may affect each other's optimal choices. It is essential to jointly consider the runs across the entire timeline.

In this paper, we propose 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 incremental execution plans subject to various objectives. We make the following contributions.

- We propose a new theory called the *TIP model* on top of time-varying relation (TVR) that formulates incremental processing using TVR. The TIP modeldescribes a formal algebra for TVRs, which includes a definition of TVRs on top of relations, semantics of querying on TVRs, and basic operations on TVRs such as TVR difference and merge operations (Sect. 3). This model serves as a theoretical foundation of our optimization framework.
- We provide a rewrite-rule framework under the TIP modelto describe different incremental computation techniques, and unify them to explore in a single search space for an optimal incremental plan (Sect. 4). This framework allows these techniques to work cooperatively, and enables cost-based search among possible plans.
- We build a Cascade-style optimizer named Tempura. It supports cost-based optimization for incremental query planning based on the TIP model. We discuss how to explore the plan space (Sect. 5) and search for an optimal incremental plan (Sect. 6).
- We give a detailed specification on how to integrate Tempurainto a traditional optimizer (Sect. 7).
- We propose multiple techniques to improve the query planning speed, such as template copying (Sect. 8.1), plan space pruning (Sect. 8.2), and optimizations of the rule engine (Sect. 8.3).
- We conduct a thorough experimental evaluation of the Tempuraoptimizer in various application scenarios. The results show the effectiveness and efficiency of Tempura(Sect. 10).

This paper is an extended version of a conference paper [50], with the following additional contributions:

- We give an in-depth description of how to integrate two advanced incremental computation algorithms in Tempura, namely outer-join view maintenance and higher-order view maintenance (Sect. 4.2).
- We elaborate the description of how to choose the optimal intermediate states to materialize, which is a critical step in selecting an optimal plan (Sect. 6.3).
- We add a detailed specification on how to integrate Tempurainto a traditional Cascades-style optimizer (Sect. 7).
- We add multiple techniques to improve the query planning speed to make Tempurahave a comparable speed as traditional optimizers while exploring a much bigger plan space (Sect. 8).
- We expand the experiments to evaluate both the effectiveness and query optimization speed of Tempurain more scenarios (Sect. 10).

## 2 Problem formulation

In this section, we formally define the problem of cost-based optimization for incremental computation. We elaborate on the running example to show that execution plans generated by different algorithms have different costs. We then illustrate the challenges.

### 2.1 Incremental query planning

Despite the different requirements in various applications, a key problem of cost-based incremental query planning (IQP) can be modeled uniformly as a quadruple $(\vec{T}, \vec{D}, \vec{Q}, \tilde{c})$, where:

- $\vec{T} = [t_1, \ldots, t_k]$ is a vector of time points when we can carry out incremental computation. Each $t_i$ can be either a concrete physical time, or a discretized logical time.
- $\vec{D} = [D_1, \ldots, D_k]$ is a vector of data, where $D_i$ represents the input data available at time $t_i$, e.g., the delta data newly available at $t_i$, and/or all the data accumulated up to $t_i$. For a future time point $t_i$, $D_i$ can be expected data to be available at that time.
- $\vec{Q} = [Q_1, \ldots, Q_k]$ is a vector of queries. $Q_i$ defines the expected results that are supposed to be delivered by the incremental computation carried out at $t_i$. If there is no required output at $t_i$, then $Q_i$ is a special empty query $\emptyset$.
- $\tilde{\mathfrak{c}}$ is a cost function that we want to minimize.

The goal is to generate an *incremental plan* $\mathbb{P} = [P_1, \ldots, P_k]$ where $P_i$ defines the task (a physical plan) to execute at time $t_i$, such that (1) $\forall 1 \le i \le k$, $P_i$ can deliver the results defined by $Q_i$, and (2) the cost $\tilde{\mathfrak{c}}(\mathbb{P})$ is minimized. Next we use a few IQP scenarios to demonstrate how they can be modeled using the above definition.

*Incremental View Maintenance* (IVM-PD). Consider the problem of incrementally maintaining a view defined by query $Q$. Instead of using any concrete physical time, we can use two logical time points $\vec{T} = [t_i, t_{i+1}]$ to represent a general incremental update at $t_{i+1}$ of the result computed at $t_i$. We assume that the data available at $t_i$ is the data accumulated up to $t_i$, whereas at $t_{i+1}$ the new delta data (insertions/deletions/updates) between $t_i$ and $t_{i+1}$ is available, denoted by $\vec{D} = [D, \Delta D]$. At both $t_i$ and $t_{i+1}$, we want to keep the view up to date, i.e., $\vec{Q}$ is defined as $Q_i = Q(D)$, $Q_{i+1} = Q(D + \Delta D)$. As the main goal is to find the most efficient incremental plan, we set $\tilde{\mathfrak{c}}$ to be the cost of $P_{i+1}$, i.e., the execution cost at $t_{i+1}$. (For a formal definition, see $\tilde{c}_v$ in Sect. 6.2.) Note that if $Q$ involves multiple tables and we want to use different incremental plans for updates on different tables, we can optimize multiple IQP problems by setting $\Delta D$ to the delta data on only one of the tables at a time.

*Progressive Data Warehouse* (PDW-PD). We model this scenario by choosing $\vec{T}$ as physical time points of the planned incremental runs. Note that we only require the incremental plan to deliver the results defined by the original analysis job $Q$ at the last run, that is, at the scheduled deadline of the job, without requiring output during the early runs. Thus, $\vec{Q} = [\emptyset, \ldots, \emptyset, Q]$. We set $\tilde{\mathfrak{c}}$ as a weighted sum of the costs of all plans in $\mathbb{P}$ (see $\tilde{c}_w(O)$ in Sect. 6.2).

## 2.2 Plan space and search challenges

We elaborate different plans to answer the query in Example 1 using the PDW-PD definition. Suppose the query summary is originally scheduled at $t_2$, but the progressive data warehouse

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 |

Changes to 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 |

Changes to 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)

**Fig. 2** **a** Data-arrival patterns of sales and returns, **b** results of sales_status and summary at $t_2$, **c** incremental results of sales_status produced by IM-1at $t_1$ and $t_2$, and **d** incremental results of sales_status produced by IM-2at $t_1$, $t_2$

decides to schedule an early execution at $t_1$ on partial inputs. Assume the records visible at $t_1$ and $t_2$ in sales and returns are those in Fig. 1a. In this IQP problem, we have $\vec{T} = [t_1, t_2]$ and $\vec{Q} = [\emptyset, q]$, where $q$ is the summary query, $\vec{D}$ is shown in Fig. 1a, and $\tilde{\mathfrak{c}}$ is the cost function that takes the weighted sum of the resources used at $t_1$ and $t_2$. Many existing incremental methods (e.g., view maintenance, stream computing, mini-batch execution [3,5,14,20]) can be used here. Consider two common methods IM-1and IM-2.

**Method IM-1** treats sales_status and summary as views, and uses incremental computation to keep the views 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 $t_2$ includes tuples $\{o_5, o_6, o_7\}$. Figure 1c 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., $o_2$ at $t_2$) can arrive much later than its corresponding sales record (e.g., the shaded $o_2$ at $t_1$). Therefore, a sales 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 the shaded tuple $o_2$ in Fig. 1c.

**Method IM-2** can avoid such retraction during incremental computation. Specifically, in the outer join of sales_status, tuples in sales 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 are 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. 1d.

In addition to these two, there are many other methods as well. Generating one plan with a high performance is non-trivial due to the following reasons.

(1) *The optimal incremental plan is data dependent, and should be determined in a cost-based way.* In the running example, `IM-1`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 at that time), and the cost per unit at $t_2$ is 1. Then, its total cost is $9 \times 0.2 + 10 \times 1 = 11.8$. `IM-2`computes 6 tuples at $t_1$, and 11 tuples at $t_2$, with a total cost of $6 \times 0.2 + 11 \times 1 = 12.2$. `IM-1`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 retraction is often, say, with one more tuple $o_4$ at $t_2$, then `IM-2`is more efficient, as it costs 12.2 versus 13.8 of `IM-1`. This is because retraction wastes early computation and causes more re-computation. Notice that the performance difference of these two approaches can be arbitrarily large.

(2) *The entire space of possible plan alternatives is very large.* Different parts within a query can choose different incremental methods. Even if early computing the entire query does not pay off, we can still incrementally execute a subquery. For instance, for the left-outer join in sales_status, we can incrementally shuffle the input data once it is ingested without waiting for the last time. IQP needs to search the entire plan space ranging from the traditional batch plan at one end to a fully incrementalized plan at the other.

(3) *Complex temporal dependencies between different incremental runs can also impact the plan decision.* For instance, during the continuous ingestion of data, 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 the decision may not be optimal, as shuffled hash join needs data to be distributed by the join key, which broadcast join does not provide. Thus, two join implementations between $t_1$ and $t_2$ incur reshuffling overhead. IQP needs to jointly consider all runs across the entire timeline.

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 to generate an optimal plan. Our solution can unify different incremental computation techniques in a single plan.

## 3 The TIP model

The core of incremental computation 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 formal theory based on the concept of *time-varying relation* (TVR) [5,8,41], called the *TVR-based incremental query planning (TIP) model*. The model naturally extends the relational model by considering the temporal aspect to formally describe incremental execution. It also includes various data-manipulation operations on TVRs, as well as rewrite rules of TVRs in order for a query optimizer to define and explore a search space to generate an efficient incremental query plan. To the best of our knowledge, the proposed TIP modelis the first one that not only unifies different incremental computation methods, but also can be used to develop a principled cost-based optimization framework for incremental execution. We focus on definitions and algebra of TVRs in this section, and dwell on TVR rewrite rules in Sect. 4.

### 3.1 Time-varying relations

**Definition 1** A *time-varying relation (TVR) $R$* is a mapping from a time domain $\mathcal{T}$ to a bag of tuples belonging to a schema.

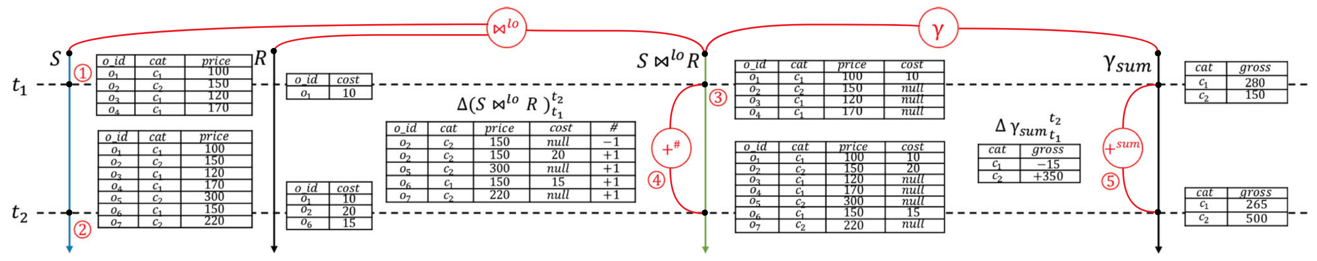
A *snapshot* of $R$ at a time $t$, denoted $R_t$, is the instance of $R$ at time $t$. For example, due to continuous ingestion, table sales ($S$) in Example 1 is a TVR, depicted as the blue line in Fig. 3. On the line, Tables 1, 2 show the snapshots $S_{t_1}$ and $S_{t_2}$, respectively. Traditional data warehouses run queries on relations at a specific time, while incremental execution runs queries on TVRs.

**Definition 2** (Querying TVR) Given a TVR $R$ on time domain $\mathcal{T}$, applying a query $Q$ on $R$ defines another TVR $Q(R)$ on $\mathcal{T}$, 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 TVRs 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 $St_1 \bowtie^{lo} R_{t_1}$. We denote left-outer join as $\bowtie^{lo}$, left-anti join as $\bowtie^{la}$, left-semi-join as $\bowtie^{ls}$, and aggregate as $\gamma$. For brevity, we use "$Q$" to refer to the "TVR $Q(R)$" when there is no ambiguity.

### 3.2 Basic operations on TVRs

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



| 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 |

**Fig. 3** Example TVRs and their relationships

TVR $R$ at $t, t' \in T$ $(t < t')$ as the *delta* of $R$ from $t$ to $t'$, denoted $\Delta R_t^{t'}$, which defines a second-order TVR.

**Definition 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 $\Delta(S \bowtie^{lo} R)_{t_1}^{t_2}$, which is the delta of snapshots $(S \bowtie^{lo} R)_{t_1}$ and $(S \bowtie^{lo} R)_{t_2}$. Here, multiplicities (#) represent insertion and deletion of the corresponding tuple, respectively. The merge operator $+$ is defined as additive union on relations with bag semantics, which adds up the multiplicities of tuples in bags.

Interestingly, a TVR can have different snapshot/delta views. For instance, the delta $\Delta\gamma_{\text{sum}}{}_{t_1}^{t_2}$ 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 $\gamma_{\text{sum}_{t_1}}$ and $\Delta\gamma_{\text{sum}}{}_{t_1}^{t_2}$ yields the value in $\gamma_{\text{sum}_{t_2}}$, i.e., $280 + (-15) = 265$. To differentiate these two merge operators, we denote the merge operator for $S \bowtie^{lo} R$ as $+^{\#}$, and the merge operator for $\gamma_{\text{sum}}$ as $+^{\text{sum}}$. This observation shows that the way to define TVR deltas and the merge operator $+$ is not unique. In general, as studied in previous research [32,54], the difference between two snapshots $R_t$ and $R_{t'}$ can have two types:

(1) *Multiplicity Perspective*. $R_t$ and $R_{t'}$ may have different multiplicities of tuples. $R_t$ may have less or more tuples than $R_{t'}$. In this case, the merge operator (e.g., $+^{\#}$) combines the same tuples by adding up their multiplicities.
(2) *Attribute Perspective*. $R_t$ may have different attribute values in some tuples compared to $R_{t'}$. In this case, the merge operator (e.g., $+^{\text{sum}}$) groups tuples with the same primary key, and combines the delta updates on the changed attributes into one value. Aggregation operators usually produce this type of snapshots and deltas. Formally, distributed aggregation in data-parallel computing platforms is often modeled using four methods [53]:

1. `Initialize`: It is called once before any data is supplied with a given key to initialize the aggregate state.
2. `Iterate`: It is called every time a tuple is provided with a matching key to combine the tuple into the aggregate state.
3. `Merge`: It is called every time when combining two aggregate states with the same key into a single aggregate state.
4. `Final`: It is called at the end on the final aggregate state to produce a result.

The snapshots/deltas are the aggregate states computed using `Initialize` and `Iterate` on partial data; the merge operator $+^{\gamma}$ is defined using `Merge`; at the end, the attribute-perspective snapshot is converted by `Final` to produce the multiplicity-perspective snapshot, i.e., the final result.[1] Note that for aggregates such as `MEDIAN` whose state needs to be the full set of tuples, `Iterate` and `Merge` degenerate to no-op.

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 $-^{\text{sum}}$ for $+^{\text{sum}}$ is defined as taking the difference of SUM values per *category* between two snapshots.

## 4 TVR rewrite rules

Rewrite rules expressing relational algebra equivalence are the key mechanism that enables traditional query optimizers to explore the entire plan space. As TVR snapshots and deltas are simply static relations, traditional rewrite rules still hold within a single snapshot/delta. However, these rewrite rules are not enough for incremental query planning, due to their inability to express algebra equivalence between TVR concepts.

To capture this more general form of equivalence, in this section, we introduce *TVR rewrite rules* in the TIP model, focusing on logical plans. We propose a trichotomy of TVR rewrite rules, namely *TVR-generating rules*, *intra-TVR rules*, and *inter-TVR rules*, and show how to model existing incre-

[1] Note that `Final` also needs to filter out empty groups with zero contributing tuples. We omit this detail for simplicity.

mental techniques using these three types of rules. This modeling enables us to unify existing incremental techniques and leverage them uniformly when exploring the plan space; it also allows IQP to evolve by adding new TVR rewrite rules.

### 4.1 TVR-generating and intra-TVR rules

Most existing work on incremental computation revolves around the notion of delta query that can be described as Eq. (1) below.

$$Q\left(R_{t'}\right) = Q\left(R_t + \Delta R_t^{t'}\right) = Q(R_t) + \mathfrak{d}Q\left(R_t, \Delta R_t^{t'}\right) \quad (1)$$

Intuitively, when an input delta $\Delta R_t^{t'}$ arrives, instead of re-computing the query on the new input snapshot $R_{t'}$, one can directly compute a delta update to the previous query result $Q(R_t)$ using a new delta query $\mathfrak{d}Q$. Essentially, Eq. (1) contains two key parts—the delta query $\mathfrak{d}Q$ and the merge operator $+$, which correspond to the first two types of TVR rewrite rules, namely *TVR-generating rules* and *intra-TVR rules*.

*TVR-Generating Rules*. Formally, TVR-generating rules define for each relational operator on a TVR, how to compute its deltas from the snapshots and deltas of its input TVRs. In other words, TVR-generating rules define $\mathfrak{d}Q$ for each relational operator $Q$ such that $\Delta Q_t^{t'} = \mathfrak{d}Q(R_t, \Delta R_t^{t'})$. Many previous studies on deriving delta queries under different semantics [10,11,14,20,21] fall into this category. Some example TVR-generating rules used by `IM-1`in Example 1 are shown as follows:

$$\begin{aligned}
&\Delta(S \bowtie^{lo} R)_{t_1}^{t_2} \\
&\quad = \Delta S^{+} \bowtie^{lo} R_{t_2} + S_{t_2} \bowtie \Delta R^{+} \\
&\qquad + (S_{t_1} - \Delta S^{-}) \bowtie^{ls} (\Delta R^{-} \bowtie^{la} R_{t_2}) - \Delta S^{-} \bowtie^{lo} R_{t_1} \\
&\qquad + (S_{t_1} - \Delta S^{-}) \bowtie^{ls} (\Delta R^{-} \bowtie^{la} R_{t_2}) - \Delta S^{-} \bowtie^{lo} R_{t_1}, \quad (2)
\end{aligned}$$

$$\Delta\gamma(S \bowtie^{lo} R)_{t_1}^{t_2} = \gamma(\Delta(S \bowtie^{lo} R)_{t_1}^{t_2}). \quad (3)$$

The rules for left-outer join (Eq. 2) and aggregate (Eq. 3) are from [21] and [20], respectively. In the rules, we use $\Delta S^-$ and $\Delta R^-$ to denote deletions in the delta, and $\Delta S^+$ and $\Delta R^+$ to denote insertions in the delta for simplicity. In Eq. (2), for brevity, padding nulls to match outer join's schema is omitted in Figs. 4, 6. This padding can simply be implemented using a project operator. The blue lines in Fig. 4 demonstrate these TVR-generating rules in a plan space.

*Intra-TVR Rules*. Intra-TVR rules define the conversions between snapshots and deltas of a single TVR. As in Eq. (1), the merge operator $+$ defines how to merge $Q$'s snapshot $Q_t$ and delta $\Delta Q_t^{t'}$ into a new snapshot $Q_{t'}$. Other examples of intra-TVR rules include rules that take the difference between

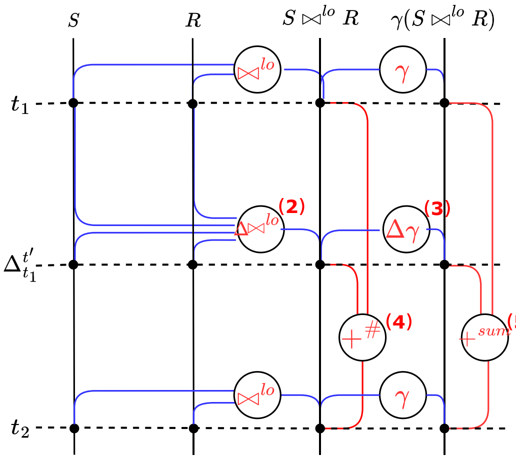


**Fig. 4** Examples of TVR-generating and Intra-TVR rules. Equation (2): incrementally compute the delta of $S \bowtie^{lo} R$. Equation (3): incrementally compute the delta of $\gamma(S \bowtie^{lo} R)$ from the delta of $S \bowtie^{lo} R$. Equations (4) and (5): merge a snapshot at $t_1$ and a delta to a generate a new snapshot at $t_2$

snapshots/deltas if the merge operator $+$ has an inverse operator $-$, e.g., $R_{t'} - R_t = \Delta R_t^{t'}$. In Fig. 4, the intra-TVR rules by `IM-1`in Example 1 are marked as red lines. These rules are shown as follows:

$$(S \bowtie^{lo} R)_{t_2} = (S \bowtie^{lo} R)_{t_1} +^{\#} \Delta(S \bowtie^{lo} R)_{t_1}^{t_2} \quad (4)$$

$$\gamma(S \bowtie^{lo} R)_{t_2} = \gamma(S \bowtie^{lo} R)_{t_1} +^{\text{sum}} \Delta\gamma(S \bowtie^{lo} R)_{t_1}^{t_2} \quad (5)$$

Note that when merging the snapshot/delta of $S \bowtie^{lo} R$, we use $+^{\#}$ (Eq. 4), whereas when merging the snapshot/delta of $\gamma(S \bowtie^{lo} R)$ (query summary), we use $+^{\text{sum}}$ (Eq. 5).

### 4.2 Inter-TVR rules

There are incremental methods that cannot be modeled using the two aforementioned types of rules alone. The `IM-2`approach in Example 1 is such an example. Different from `IM-1`, approach `IM-2`does not directly deliver the snapshot of $S \bowtie^{lo} R$ at $t_1$. Instead, it delivers only the tuples that will not be retracted in the future, essentially the results of $S \bowtie R$. At $t_2$ when the data is known to be complete, `IM-2`computes the rest part of $S \bowtie^{lo} R$, essentially $S \bowtie^{la} R$, then pads with nulls to match the output schema.

This observation shows another family of incremental methods: without computing $Q$ directly, one can incrementally compute a set of queries $\{Q'_1, \ldots, Q'_k\}$, and then apply another query $P$ on their results to get $Q$, formally described as Eq. (6). The intuition is that $\{Q'_1, \ldots, Q'_k\}$ may be more amenable to incremental computation:

$$Q(R) = P(Q'_1(R), \ldots, Q'_k(R)). \quad (6)$$

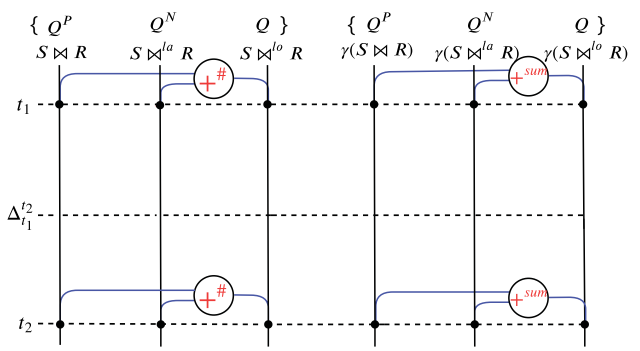


**Fig. 5** Examples of inter-TVR equivalence rules in IM-2. Each operator in original query $Q$ is decomposed into two parts: $Q^P$ (positive-only updates) and $Q^N$ (possibly negative updates)

Equation 6 describes a general family of methods: they all rely on certain rewrite rules describing the equivalence between snapshots/deltas of multiple TVRs. We summarize this family of methods into *inter-TVR rules*. Next we demonstrate using a couple of existing incremental methods how they can be modeled by inter-TVR rules.

(1) *IM-2*: Let us revisit IM-2using the terminology of inter-TVR rules. Formally, $Q = S \bowtie^{lo} R$ is decomposed into $Q^P$ and $Q^N$:

$$\begin{aligned} {Q^P}_t &= S_t \bowtie R_t,\ {Q^N}_t = S_t \bowtie^{la} R_t, \\ Q_t &= {Q^P}_t +^{\#} {Q^N}_t \end{aligned} \tag{7}$$

where $Q^P$ is a positive part that will not retract tuples if both $S$ and $R$ are append-only, whereas $Q^N$ represents a part that could retract tuples. The inter-TVR rule in Eq. (7) states that any snapshot of $Q$ can be decomposed into snapshots of $Q^P$ and $Q^N$ at the same time. Similar decomposition holds for the aggregate $\gamma$ in *summary* too, just with a different merge operator $+^{\text{sum}}$. Figure 5 depicts these rules in a plan space. As it is easier to incrementally compute inner join than left-outer join, $Q^P$ can be incrementalized more efficiently than $Q$ with rules in Sect. 4.1, whereas $Q^N$ cannot be easily incrementalized, and is not computed until the completion time.

(2) *Outer-join view maintenance (OJV)*: [33] proposed a method to incrementally maintain outer-join views.

*Query decomposition*. The main idea is to decompose a query into three parts given an update to a single input table: a directly affected part $Q^D$, an indirectly affected part $Q^I$, and an unaffected part $Q^U$. Intuitively, an insertion (deletion) in the input table will cause insertions (deletions) to $Q^D$ and deletions (insertions) to $Q^I$, but leave $Q^U$ unaffected. These parts are formally defined using the join-disjunctive normal form of $Q$ and its subsumption graph. We refer the readers

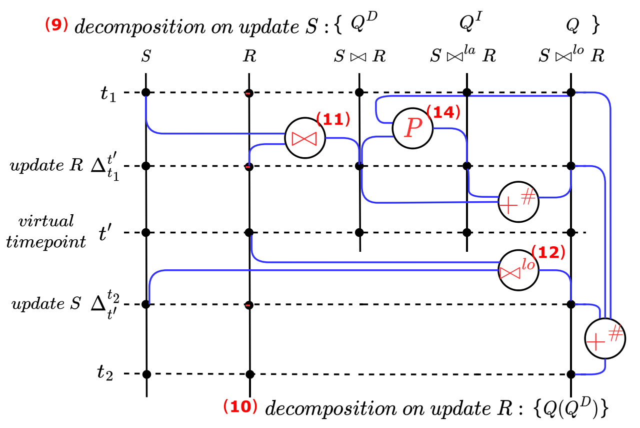

(9) decomposition on update $S$ : { $Q^D$
$Q^I$
$Q$ }
$S$
$R$
$S \bowtie R$
$S \bowtie^{la} R$
$S \bowtie^{lo} R$
$t_1$
update $R$ $\Delta_{t_1}^{t'}$
virtual timepoint $t'$
update $S$ $\Delta_{t'}^{t_2}$
$t_2$
(11)
(14)
(12)
$P$
$+^{\#}$
(10) decomposition on update $R$ : $\{Q(Q^D)\}$

**Fig. 6** Supporting outer-join view maintenance. A virtual timepoint $t'$ is inserted to model updating one base table at a time. Equations 9, 10: decompose the query into $Q^D$ and $Q^I$. Equations 11, 12: compute the delta of directly affected parts. Equation 14: compute the delta of indirectly affected parts

to [33] for details. This decomposition can be expressed using the following inter-TVR rule:

$$Q_t = {Q^D}_t +^{\#} {Q^I}_t +^{\#} {Q^U}_t. \tag{8}$$

Take query sales_status as an example. As the algorithm in [33] considers updating one input table at a time, we insert a virtual time point $t'$ between $t_1$ and $t_2$ to model that $R$ and $S$ are updated separately at $t'$ and $t_2$. The query sales_status is decomposed as follows:

$$\begin{aligned} &{Q^D}_{t'} = S_{t'} \bowtie R_{t'}, \quad {Q^I}_{t'} = S_{t'} \bowtie^{la} R_{t'}, \quad {Q^U}_{t'} = \emptyset, \\ &\qquad \text{when } R \text{ is updated at } t', \end{aligned} \tag{9}$$

$$\begin{aligned} &{Q^D}_{t_2} = S_{t_2} \bowtie^{lo} R_{t_2}, \quad {Q^I}_{t_2} = \emptyset, \quad {Q^U}_{t_2} = \emptyset \\ &\qquad \text{when } S \text{ is updated at } t_2. \end{aligned} \tag{10}$$

Note that there is no unaffected part in this example. Unaffected parts only exist when a query joins three or more tables according to the algorithm in [33].
*Delta computation*. The outer-join view maintenance algorithm maintains the directly affected parts and the indirectly affected parts separately.

To compute the delta of the directly affected parts for the query sales_status, OJVapplies the TVR-generating rules shown as follows:

$$\begin{aligned} \Delta {Q^D}_{t_1}^{t'} &= S_{t_1} \bowtie^{lo} \Delta R_{t_1}^{t'} \\ &= \{(o_2, c_2, 150, 20, +1)\}, \end{aligned} \tag{11}$$

$$\begin{aligned} \Delta {Q^D}_{t'}^{t_2} &= \Delta S_{t'}^{t_2} \bowtie R_{t'} \\ &= \{(o_5, c_2, 300, null, +1), \\ &\quad (o_6, c_1, 150, 15, +1), \\ &\quad (o_7, c_2, 220, null, +1)\}. \end{aligned} \tag{12}$$

Recall that insertions into the base table will cause insertions to the directly affected parts. Note that the delta tuples of each $Q^D$ part are all insertions.

To compute the delta of the indirectly affected parts, OJVcombines the delta of $Q^D$ and the previous snapshot of $Q$, as shown in Eq. (13). Compared to computing the detla directly from the base tables, this algorithm can reuse the already computed delta of the directly affected parts. This rule can be expressed using the following inter-TVR rule:

$$\Delta {Q^I}^{t'}_t = P(\Delta {Q^D}^{t'}_t, Q_t). \tag{13}$$

In the query sales_status, the delta of the indirectly affected part at $t'$ is computed using a filter and a semi-join, as shown as follows:

$$\begin{aligned}\Delta {Q^I}^{t'}_{t_1} &= -[\sigma_{cost=null}(Q_{t_1}) \ltimes^{ls} \Delta {Q^D}^{t'}_{t_1}] \\ &= \{(o_2, c_2, 150, null, -1)\}.\end{aligned} \tag{14}$$

This is equivalent to incrementally computing the left-anti join from the base tables. Recall that insertions into the base table will cause deletions to the indirectly affected parts. Note that the delta tuple $o_2$ of the $Q^I$ part is a deletion.

Note that for the query sales_status, both IM-2and OJVleverage the fact that an left-outer join can be decomposed into an inner join and a left-anti join. However, IM-2and OJVuses this decomposition in very different ways:

- IM-2considers updating all tables at the same time. It decomposes the query into two parts, $Q^P$ and $Q^N$.
- OJVconsiders updating one base table at a time. It decomposes the query into a finer granularity of three parts: ${Q^D}_{t'}$ on updating $R$, ${Q^I}_{t'}$ on updating $R$, and ${Q^D}_{t_2}$ on updating $S$.
- In IM-2, each $Q^P$ part contains the tuples that will never be retracted if the base tables are append-only. As an example, $\Delta {Q^P}^{t_2}_{t_1}$ contains only two tuples, $o_2$ and $o_5$.
- In OJV, each $Q^D$ part contains the tuples that are positive when its corresponding base table is updated. As an example, $\Delta {Q^D}^{t_2}_{t'}$ contains tuples $o_5$, $o_6$, and $o_7$. Tuples $o_5$ and $o_7$ could potentially be retracted in the future.
- IM-2computes the delta using the TVR-generating rules. OJVintroduces a new rule (Eq. 14) to compute the delta of indirectly affected parts.

(3) *Higher-order view maintenance*: [3,38] proposed a higher-order view maintenance algorithm, which can also be expressed by inter-TVR rules. The main idea is to treat the deltas of a query $Q$ as another TVR, and continue applying TVR rewrite rules to incrementally compute it. Formally, considering a query $Q$ and updates to one of its inputs $R$, the algorithm can be summarized as the following inter-TVR rule:

$$\Delta Q^{t'}_t = \eth Q\left(R_t, \Delta R^{t'}_t\right) = P\left(M_t, \Delta R^{t'}_t\right). \tag{15}$$

The rule decomposes the delta query into two parts: the delta update $\Delta R^{t'}_t$, and an update-independent subquery $M$ that does not involve $\Delta R^{t'}_t$. The two parts are combined using a query $P$ to get the delta of $Q$. If $M$ is a query involving input relations other than $R$, it can be further decomposed again with respect to updates to each of its input relations according to Eq. (15), until it becomes a constant. We refer the readers to [3] for a detailed algorithm. Take the summary query and updates to sales ($S$) as an example (we denote returns as R). Applying Eq. (15), we can decompose it as

$$\begin{aligned}\Delta Q^{t'}_t &= \gamma_{category;\texttt{SUM}(r)}(\Delta S^{t'}_t \ltimes^{lo} M_t), \\ \text{where } M_t &= \gamma_{o\_id;total=\texttt{SUM}(cost)}(R_t), \\ r &= \texttt{IF}(total \texttt{ IS NULL}, price, -total).\end{aligned} \tag{16}$$

$M$ essentially preprocesses returns by computing the total cost per *o_id*,[2] and $P$ computes the gross revenue per category by summing up the precomputed total cost in $M$ or the prices of the new orders added to $S$. Then, $M$ is materialized as a higher-order view and can be further incrementally maintained with respect to updates to returns by repeatedly applying the inter-TVR rule to generate higher-order views.

## 4.3 Putting everything together

The above TVR rewrite rules lay a theoretical foundation for our IQP framework. Different TVR rules can be extended individually and work together automatically. For example, TVR-generating rules can be applied on any TVR created by inter-TVR rules. By jointly applying TVR rewrite rules and traditional rewrite rules, we can explore a plan space much larger than any individual incremental method. Figure 7 shows an example plan space by overlaying Figs. 4, 5. Any tree rooted at $\gamma(S \ltimes^{lo} R)_{t_2}$ is a valid incremental plan for Example 1, e.g., IM-2's plan is shown in red.

In the next two sections, we discuss how to build an optimizer framework based on the TIP model, including plan space exploration (Sect. 5) and selecting an optimal incremental plan (Sect. 6).

[2] Here, we do not assume *o_id* as the primary key of returns. Say returns could contain multiple records for a returned order due to different costs such as shipping cost, product damage, and inventory carrying cost.

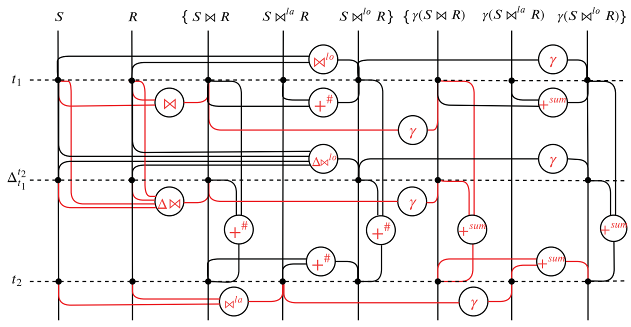


**Fig. 7** The combined incremental plan space of Example 1

## 5 Plan space exploration

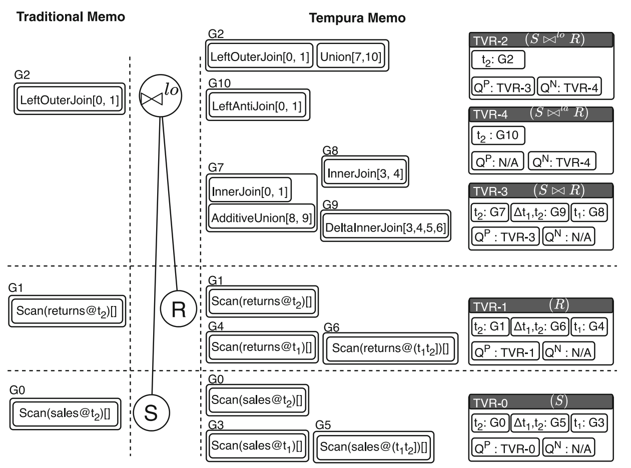
In this section, we study how Temporaexplores the incremental plan space. Existing query optimizers explore plans only for a specific time. For incremental processing, we need to explore a much bigger plan space by considering not only relations at different times, but also transformations between them. We illustrate how to incorporate the TIP modelinto a Cascades-style optimizer [18,19], and develop a cost-based optimizer framework for IQP called Tempura.

We focus on the key adaptations on 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 physical implementations of logical operators, and monitors new plans generated in the memo. Whenever there are changes, the rule engine fires applicable transformation rules on the newly generated plans to add more plan alternatives to the memo.

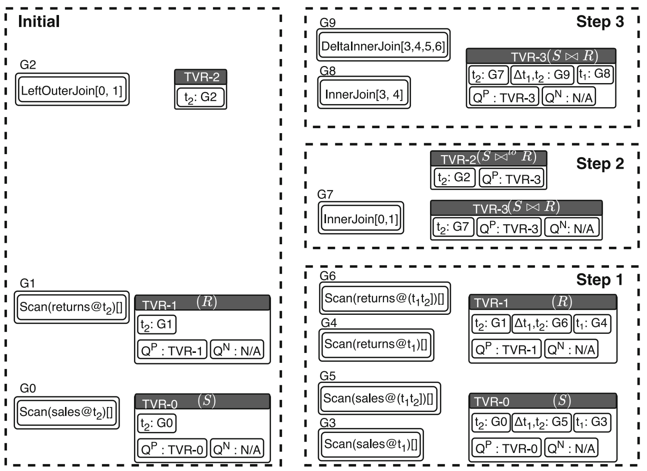
### 5.1 Memo: capturing TVR relationships

The memo in the traditional Cascades-style optimizer 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. 8a depicts the traditional memo of the sales_status query. For brevity, we omit the physical equivalence classes. For instance, *LeftOuterJoin*[0,1] has Groups G0 and G1 as children, and it corresponds to the plan tree rooted at $\bowtie^{lo}$. G2 represents all plans logically equivalent to *LeftOuterJoin*[0,1].

However, the above two equivalences are not enough to capture the rich relationships in the TIP model. For example, the relationship between snapshots and deltas of a TVR cannot be modeled using the logical equivalence due to the following reasons. Two snapshots at different times produce different relations, and the snapshots and deltas do not even have the same schema (deltas have an extra # column). To solve this problem, on top of logical/physical equivalence classes, we explicitly introduce TVR nodes into the memo, and keep track of the following relationships, shown as the "TempuraMemo" part in Fig. 8a: (1) **Intra-TVR relationship** specifies the snapshot/delta relationship between logical equivalence classes of operators and the corresponding TVRs. The traditional memo only models scanning the



(a) An example memo of subquery sales_status. Compared to the traditional memo, the Tempura memo: (1) maintains more equivalence groups of snapshots at earlier time points and deltas, and (2) has additional TVR nodes to keep track of the intra-TVR and inter-TVR relationships.



(b) A step-wise illustration of the growth of the memo. Step 1: for each source operator, its TVR, snapshots, and deltas are populated. Step 2: Rules of IM-2 decompose left outer join to an inner join. Step 3: TVR-generating rules generate the operators to incrementally compute the inner join.

**Fig. 8** Examples of the memo structure in Tempura

full content of $S$, i.e., $S_{t_2}$, represented by G0, while the Tempuramemo 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 $t_2$ ($\Delta S_{t_1}^{t_2}$), represented by G3 and G5. The memo uses an explicit TVR-0 to track these intra-TVR relationships. (2) **Inter-TVR relationship** specifies the relationship between TVRs described by inter-TVR equivalence rules. For example, the `IM-2`approach decomposes $S \bowtie^{lo} R$ (TVR-2) into two parts $Q^P$ (TVR-3) and $Q^N$ (TVR-4) as in Sect. 3. Note that the above relationships are transitive. For instance, as G7 is the snapshot at $t_2$ of TVR-3, and TVR-3 is in turn the $Q^P$ part of TVR-2, G7 is also related to TVR-2.

## 5.2 Rule engine: enabling TVR rewritings

As the memo of Tempurastrictly subsumes a traditional Cascades memo, traditional rewrite rules can be adopted and work without modifications. Besides, the rule engine of Tempurasupports TVR rewrite rules. Tempuraallows optimizer developers to define TVR rewrite rules by specifying a graph pattern on both relational operators and TVR nodes in the memo. A TVR rewrite-rule pattern consists of two types of nodes and three types of edges: (1) *operator operands* that match relational operators; (2) *TVR operands* that match TVR nodes; (3) *operator edges* between operator operands that specify traditional parent–child relationship of operators; (4) *intra-TVR edges* between operator operands and TVR operands that specify intra-TVR relationships; and (5) *inter-TVR edges* between TVR operands that specify inter-TVR relationships. All nodes and intra/inter-TVR edges can have predicates. Once fired, TVR rewrite rules can register new TVR nodes and intra/inter-TVR relationships.

Figure 9a–b depicts two TVR rewrite rules, where solid nodes and edges specify the patterns to match, and dotted ones are newly registered by the rules. In the figures, we also show an example match of these rules when applied on the memo in Fig. 8a. **Rule 1** is the TVR-generating rule to delta compute an inner join. It matches a snapshot of an *InnerJoin*, whose children $L$, $R$ each have a delta sibling $L'$, $R'$. The rule generates a *DeltaInnerJoin* taking $L$, $R$, $L'$, $R'$ as inputs, and register it as a delta sibling of the original *InnerJoin*. **Rule 2** is an inter-TVR rule of `IM-2`. It matches a snapshot of a *LeftOuterJoin*, whose children $L$, $R$ each have a $Q^P$ snapshot sibling $L'$, $R'$. The rule generates an *InnerJoin* of $L'$ and $R'$, and register it as the $Q^P$ snapshot sibling of the original *LeftOuterJoin*.

Figure 8b demonstrates the growth of a memo in Tempura. For each step, we only draw the updated part due to space limitation. The memo starts with G0 to G2 and their corresponding TVR-0 to TVR-2. In step 1, we first populate the snapshots and deltas of the *scan* operators, e.g., G3 to G6, and register the intra-TVR relationship in TVR-0 and TVR-1. We also populate their $Q^P$ and $Q^N$ inter-TVR relationships as in `IM-2`(for base tables these relationships are trivial). In step 2, in Fig. 9b. rule 2 matches the tree rooted at *LeftOuterJoin[0,1]* in G2, generates the inner join of G7, and registers G7 to TVR-3 as the snapshot at $t_2$, and TVR-3 to TVR-2 as $Q^P$. In step 3, rule 1 matches *InnerJoin[0,1]* in G7 in Fig. 9a and generates *DeltaInnerJoin*[3,4,5,6] as the delta of TVR-3. By applying other TVR rewrite rules, we eventually get the memo in Fig. 8a.

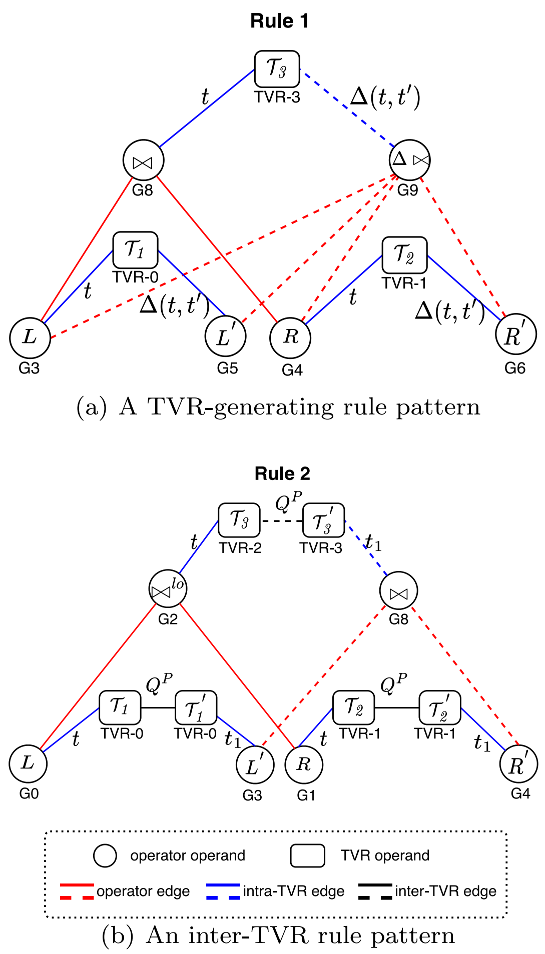


**Fig. 9** Example TVR rewrite-rule patterns in Tempura

## 6 Selecting an optimal plan

In this section, we discuss how Tempuraselects an optimal plan in the explored space. The problem differs from existing query optimizers in the following ways:

1. In a traditional query plan, all physical operators are executed at the same time point in a single query. In Tempura, physical operators in an incremental plan might be executed at different time points. In Sect. 6.1, we discuss

how to assign a valid execution time point of each physical operator.
2. Similarly, in a traditional query plan, the cost function represents the cost of a single time point. In Sect. 6.2, we discuss how to extend the cost function to consider the costs at different time points.
3. Finally, an incremental plan often needs to maintain intermediate states between the executions of different time points. In Sect. 6.3, we discuss how to find the optimal states to materialize.

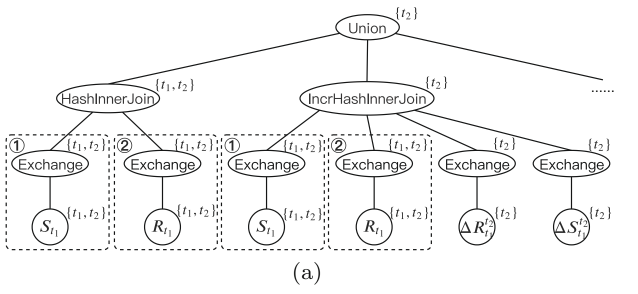
## 6.1 Time-point annotations of operators

Costing the plan alternatives is not trivial because the temporal dimension is involved. Figure 10a depicts one physical plan rooted at $(S \bowtie^{lo} R)_{t_2}$, as shown in red in Fig. 7. This plan only specifies the concrete physical operations taken on the data, but does not specify when they are executed. Actually, each operator in the plan usually has multiple choices of execution time. In Fig. 10a, the time points annotated alongside each operator shows the possible temporal domain of its execution. 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., $t_1$ or $t_2$. 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: (1) **For a base relation** $R$, t-dom$(R)$ is the set of execution time points that are no earlier than the time point when $R$ is available. (2) **For an operator** $O$ **with inputs** $I_1, \ldots, I_k$, t-dom$(R)$ is the intersection of its inputs' temporal domains: t-dom$(R) = \cap_{1 \leq j \leq k}$t-dom$(I_j)$.

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

**Definition 4** (Valid Temporal Assignment) An assignment of execution time points 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. 10b demonstrates a valid temporal assignment of the physical plan in Fig. 10a. At $t_1$, the plan computes *HashInnerJoin* of $S_{t_1}$ and $R_{t_1}$, and shuffles $S_{t_1}$ and $R_{t_1}$ 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 time points, 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



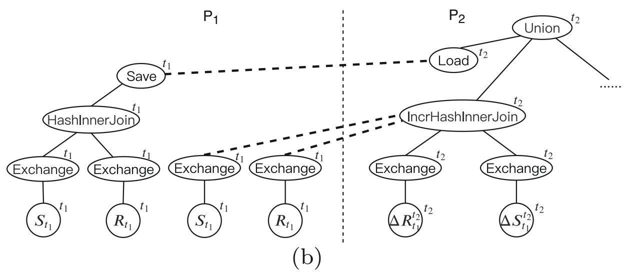


**Fig. 10** Examples of **a** the temporal plan space, and **b** a temporal assignment for subquery sales_status's plan

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 an incremental plan is defined under a specific assignment of execution time points. Therefore, the optimization problem is formulated as: given a plan space, find a physical plan and temporal assignment that achieve the lowest cost. In this section, we discuss the cost model and optimization algorithm for this problem without considering sharing common sub-plans. We will discuss the problem of how to decide which states to materialize in Sect. 6.3.

As an incremental plan can span across multiple time points, the cost function $\tilde{\mathfrak{c}}$ in an IQP problem (as in Sect. 2.1) is extended to a function taking into consideration of costs at different time points. For the cost at each time point, we inherit the general cost model used in traditional query optimizers, i.e., the cost of a plan is the sum of the costs of all its operators. Below we give two examples of $\tilde{\mathfrak{c}}$. We denote traditional cost functions as $c$, and $c_i$ is the cost at time $t_i$. As before, $c$ can be a 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 the example setting in Sect. 2.2, $w_1 = 0.2$ for $t_1$ and $w_2 = 1$ for $t_2$.

2. $\tilde{c}_v(O) = [c_1(O), \ldots, c_T(O)]$. The extended cost is a vector combining costs at different time points. $\tilde{c}_v$ can be compared entry-wise in a reverse lexical order. Formally, $\tilde{c}_v(O_1) > \tilde{c}_v(O_2)$ iff $\exists j$ s.t. $c_j(O_1) > c_j(O_2)$ and $c_i(O_1) = c_i(O_2)$ for all $i$, $j < i \le T$.

Consider the plan in Fig. 10a as an example. To get the result of *HashInnerJoin* at $t_2$, we have two options: (i) compute the join at $t_2$; or (ii) as in Fig. 10b, compute the join at $t_1$, save the result, and load it back at $t_2$. Assume the cost of computing *HashInnerJoin*, saving the result, and loading it are 10, 5, 4, respectively. Then for option (i) $(c_1, c_2) = (0, 10)$, for option (ii) $(c_1, c_2) = (15, 4)$. Say that we use $\tilde{c}_w$ as the cost function. If $w_1 = 0.6$ and $w_2 = 1$ then option (i) is better, whereas if $w_1 = 0.2$ and $w_2 = 1$, option (ii) becomes better.
**Dynamic programming** (DP) used predominantly in existing query optimizers [18,35,42] also need to be adapted to handle the cost model extensions. In existing query optimizers, the DP state space 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 time points, i.e., $\{O\} \times$ t-dom$(\{O\})$. Instead of recording a single optimum, each $O$ records multiple optima, one for each execution time $\tau(O)$, which represents the best cost of all the subtrees rooted at $O$ if $O$ is generated at $\tau$. During optimization, the state-transition function is the following:

$$\tilde{\mathfrak{c}}[O, \tau] = \min_{\forall \text{ valid } \tau_j} \left( \sum_j \tilde{\mathfrak{c}}[I_j, \tau_j] + c_\tau(O) \right). \qquad (17)$$

That is, the best cost of $O$ if executed at $\tau$ is the best cost of all possible plans of computing $O$ with all possible valid temporal assignments compatible with $\tau$.

We have the following observation of the above DP algorithm: the optimization problem under cost functions $\tilde{c}_w$ and $\tilde{c}_v$ without sharing common sub-plans satisfies the property of optimal substructure, and dynamic programming is applicable. In general, we can apply DP to the optimization problem for any cost function satisfying the property of optimal substructure.

### 6.3 Deciding states to materialize

In an incremental plan, a delta computation often requires intermediate states to be saved from earlier computation. As an example, in Fig. 10b, the incremental hash join at $t_2$ needs to use the saved intermediate states Shuffle$(S_{t_1})$ and Shuffle$(R_{t_1})$. However, an alternative plan is to re-compute these states from base tables instead of reusing materialized states. Whether to save the intermediate states or to re-compute the states needs to be decided in a cost-based manner.

We model the problem of choosing the optimal intermediate states to materialize as a multi-query optimization problem by treating the plan at each time point as an independent mini-query and finding sharing states between the mini-queries at different time points. In the example in Fig. 10, we treat the whole incremental query as two independent mini-queries at two time points: query1 computes the join result at $t_1$ and query2 computes the delta join result from $t_1$ to $t_2$. These two mini-queries both need the states Shuffle$(S_{t_1})$ and Shuffle$(R_{t_1})$: query1 uses the states to produce the hash join result at $t_1$, and query2 uses these states to compute the incremental hash join result at $t_2$. The parts ① and ② circled in dashed lines in Fig. 10a depict the shareable candidates. Therefore, computing the shuffle once and materializing these states once can benefit two reuse opportunities and reduce the overall cost of the incremental plan. Note that materializing a state is not always beneficial because the overhead of materialization might be higher than the cost of re-computing it. In order to choose the best sub-plans to materialize, we feed query1 and query2 together to a multi-query optimization algorithm [30,40,56]. In other words, a materialized shared sub-plan between two mini-queries $Q_i$ and $Q_j$ at two time points of an incremental plan is essentially an intermediate state that is saved by $Q_i$ and reused by $Q_j$.

In this paper, we extend the MQO algorithm in [30], which proposes a greedy framework on top of Cascade-style optimizers for MQO. For the sake of completeness, we list the algorithm in Algorithm 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 $best Plan(\mathbb{S})$ means the best plan with $\mathbb{S}$ materialized and shared. The algorithm terminates when all candidates are considered or adding candidates can no longer improve the plan cost. As IQP 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. 10a, 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 optimizations in [30] are still applicable to Algorithm 1.

As expanded with execution time options, the enumeration space of the shareable candidate set becomes much larger than the original algorithm in [30]. 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 1. This theorem relies on

**Algorithm 1** Greedy Algorithm for Choosing Optimal States to Materialize

```
1: $\mathbb{S} = \emptyset$
2: $\mathbb{C}$ = shareable candidate set consisting of all shareable nodes
   and their potential execution time points $\{\langle s, \tau(s)\rangle\}$
3: while $\mathbb{C} \neq \emptyset$ do
4:     Pick $\langle s, \tau(s)\rangle \in \mathbb{C}$ that minimizes $\tilde{\mathfrak{c}}(bestPlan(\mathbb{S}'))$ where $\mathbb{S}' = \{\langle s, \tau(s)\rangle\} \cup \mathbb{S}$
5:     if $\tilde{\mathfrak{c}}(bestPlan(\mathbb{S}')) < \tilde{\mathfrak{c}}(bestPlan(\mathbb{S}'))$ 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 $\mathbb{S}$
```

the fact that materializing a shareable sub-plan at its earliest possible time subsumes other materialization choices.

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

***Proof*** Materializing a shareable sub-plan at its earliest possible time subsumes other materialization choices, as any reuse opportunities can always choose between using or not using the materialized sub-plan. Therefore, the reuse cost of the shareable sub-plan does not increase. On the other hand, as the extended cost function strictly prefers an earlier execution time by assignment resources at an earlier time with a lower cost, the materialization overhead of the shareable sub-plan also does not increase. Combining these two points, one can see the shareable candidate $\langle s, min(\text{t-dom}(s))\rangle$ subsumes other candidates $\langle s, \cdot\rangle$. □

## 7 Integrating into traditional query optimizers

In this section, we give a detailed specification on how to integrate Tempurainto a traditional Cascades-style query optimizer. Specifically, we focus on how to represent TVRs in the memo structure.

We implemented Tempurabased on Apache Calcite. Without loss of generality, we use Calcite's terminologies in this section: an operator is called a *RelNode*, and a logically equivalent group of operators is called a *RelSet*. A *Trait* represents a physical property of physically equivalent classes. On top of these, Tempuraintroduces a new data structure called *TvrMetaSet* to store relevant information about a TVR: the

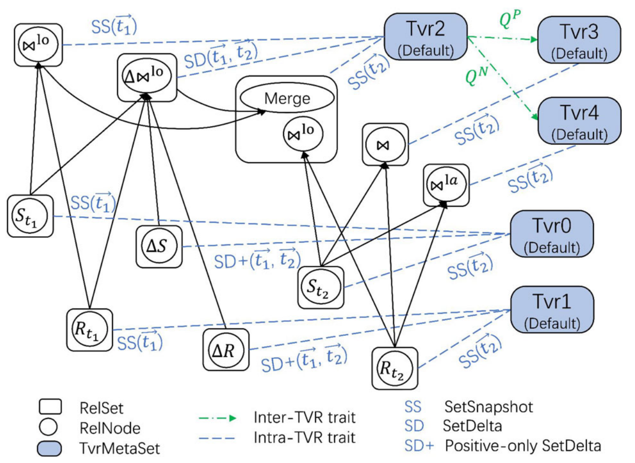


**Fig. 11** Partial memo of subquery sales_status from Example 1 in Tempura

time domain of the TVR, Intra-TVR relationships, and Inter-TVR relationships. Next we elaborate more on these using the subquery sales_status from Example 1 as an example.

*TVR Time Points and Intervals* By now we used single time points to identify data versions, in which all (intermediate) results are computed from input relations all at the version of the same time point. Whereas in many computing methods, one need to reason about results computed from input relations at different time points. For instance in both outer-join view maintenance and higher-order view maintenance (both described earlier in Sect. 4.2), to model a partial update of relation $S$ from $t_1$ to $t_2$ in $S \bowtie^{lo} R$ with $R$ unchanged, we need to represent the join result of $S$ at $t_2$ and $R$ at $t_1$, or vice versa.

Consequently, Tempurakeeps track of the time version of every input relations, respectively, to allow for incremental computation using combinations of input relations at different time points. For a query with $k$ input relations $[I_1, \ldots, I_k]$, we define a TVR time point to be a vector $\vec{t} = [t^1, \ldots, t^k]$. Each time point in the vector represents the time version of the $k$-th input relation. For example, for a query with two input relations $R$ and $S$, the TVR time point $[t_1^R, t_2^S]$ represents a state where the result is computed from $R$ in version $t_1$ and $S$ in version $t_2$. A TVR time interval is defined as $(\vec{t}, \vec{t'})$, the interval between two TVR time points $\vec{t} = [t^1, \ldots, t^k]$ and $\vec{t'} = [t'^1, \ldots, t'^k]$, where $\forall i\ t^i \leq t'^i$. When the context is clear, we use $\vec{t}$ to denote the TVR time point $\vec{t} = [t, \cdots, t]$. For instance in Fig. 11, all five TVRs contain two TVR time points $\vec{t_1} = [t_1^R, t_1^S]$ and $\vec{t_2} = [t_2^R, t_2^S]$.

*Time Domain of a TVR* The time domain $\mathcal{T}$ of a TVR (introduced earlier in Sect. 3.1) defines relevant time points of the TVR. Specifically, it consists of a list of valid TVR time points and intervals, specified in a data structure called *TvrMetaSetType*. Tempuraallows a TVR to have an incomplete time domain, which means not all TVR time points and intervals are required to be present in a TVR.

With two input relations $R$ and $S$ and three time points of $[t_1, t_2, t_3]$, Fig. 12 visualizes two types of TVR meta set: *Default* and *Partial*, where each blue point is a valid TVR time point and each yellow arrow is a valid TVR time interval in the time domain of the TVR.

1. The Default TvrMetaSetType only allows TVR time points where all input relations are at the same time version. On top of that, it only allows TVR time intervals involving adjacent TVR time points to avoid a combinational number of delta intervals. For example in Fig. 12a, there are three TVR time points $[t_1^R, t_1^S]$, $[t_2^R, t_2^S]$, and $[t_3^R, t_3^S]$, and two intervals $([t_1^R, t_1^S], [t_2^R, t_2^S])$ and $([t_2^R, t_2^S], [t_3^R, t_3^S])$. This policy has a complexity linear to number of time points, which helps limit exploration space and improve optimization speed.
2. The Partial TvrMetaSetType of certain input relation (e.g. $R$ or $S$) only allows TVR time intervals where only the corresponding input relation is updated. For example, assuming $R$ is always updated before $S$ in very time step, then Fig. 12b shows the partial TvrMetaSetType on updating relation $R$ only, where only two TVR time intervals $([t_1^R, t_1^S], [t_2^R, t_1^S])$ and $([t_2^R, t_2^S], [t_3^R, t_2^S])$ are allowed. Similarly, Fig. 12c shows the partial TvrMetaSetType on updating relation $S$ only. Note that although each partial type is incomplete, the two partial types can work together to constitute a valid update path from $[t_1^R, t_1^S]$ to $[t_3^R, t_3^S]$. This policy is useful for incremental computation algorithms that only consider updating one input relation at a time, such as outer-join view maintenance and higher-order view maintenance.

In the memo example in Fig. 11, all TvrMetaSets are of the Default TvrMetaSetType, with two TVR time points $\vec{t_1}$ and $\vec{t_2}$, and one interval $(\vec{t_1}, \vec{t_2})$.

*Intra-TVR traits* A TVR has a mapping from its time domain $\mathcal{T}$ to many relations, e.g. snapshots and deltas. A TvrMetaSet stores these Intra-TVR relationships using Intra-TVR traits, which are bidirectional edges between a TvrMetaSet and a RelSet. For example, Fig. 11 plots Intra-TVR traits in blue dotted lines, which connect TvrMetaSets to their related RelSets. Custom Intra-TVR traits can be defined and used by various incremental computing methods. Earlier in Sect. 3, we described the multiplicity and attribute perspectives of a TVR. They corresponds to the example Intra-TVR traits as follows.

1. *SetSnapshot.* This Intra-TVR trait represents that a RelSet is a multiplicity-perspective snapshot at a TVR time point of the connecting TvrMetaSet. The specific TVR time point is stored in the SetSnapshot trait.

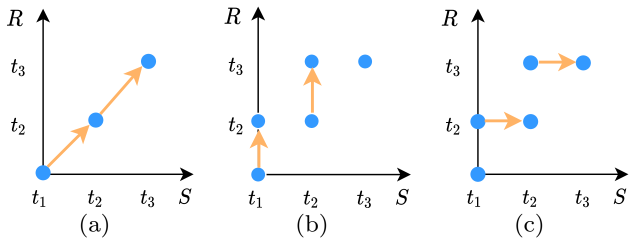


**Fig. 12** TvrMetaSetType Examples: **a** Default, **b** Partial updating R only, and **c** Partial updating S only. Each blue point is a valid TVR time point and each yellow arrow is a valid TVR time interval in the time domain of the TVR

2. *SetDelta.* This Intra-TVR trait represents that a RelSet is a multiplicity-perspective delta for a TVR time interval. The specific TVR time interval is stored in the Intra-TVR trait. Additionally, SetDelta has two variations, namely positive-only SetDelta and retractable SetDelta. Each variation has a different merge function (see Definition 3 in Sect. 3.2) for snapshots and deltas. For retractable delta, the information of the specific column that encodes insertion or deletion is stored in the SetDelta trait.
3. *ValueSnapshot.* This Intra-TVR trait represents that a RelSet is an attribute-perspective snapshot at a TVR time point. An attribute-perspective snapshot is produced by an aggregation operator. It can be converted to a SetSnapshot by applying the *Final* aggregation function (see Sect. 3.2). The information needed for the conversion is stored in the ValueSnapshot trait, including the group-by keys and the *Final* aggregation functions.
4. *ValueDelta.* This Intra-TVR trait represents that a RelSet is an attribute-perspective delta for a TVR time interval. Similar to SetDelta, there are positive-only ValueDelta and retractable ValueDelta. Similar to ValueSnapshot, necessary information on conversions to SetSnapshot is also stored in ValueDelta.

*Inter-TVR Trait.* A TvrMetaSet stores Inter-TVR relationships using Inter-TVR traits, which are directed edges between two TvrMetaSets. Custom Inter-TVR traits can be defined to annotate the information needed by an incremental computation algorithm. For example in Fig. 11, the green $Q^P$ and $Q^N$ edges are two Inter-TVR traits used in the `IM-2`approach.

*TVR Equivalence and Anchor Time Point.* Tracing equivalent operators and merging them into logically equivalent classes is an important step in Cascades-style optimizers. Similarly, we need to merge two TVRs if they are found equivalent.

If we know two TVRs are equivalent, then the snapshots on each time point are also equivalent. However, if we only know that two snapshots at a specific time point are equivalent, we cannot infer if their corresponding TVRs are equivalent. The following example shows such a scenario. Consider a simple query $\sigma(S)$ with two time points $t_1$ and $t_2$. The query has two TVRs: the TVR of the scan operator $S$ and the TVR of the filter operator $\sigma(S)$. Suppose the base table S is empty at $t_1$. Applying a filter on an empty table is also empty. The optimizer detects the logical equivalence between the two empty snapshots $S_{t_1}$ and $\sigma(S_{t_1})$. Apparently, this does not imply that the two TVRs $S$ and $\sigma(S)$ are equivalent at all time points. At $t_2$, data might arrive at the base table $S$ and the two snapshots $S_{t_2}$ and $\sigma(S_{t_2})$ are not equivalent anymore. The rule for discovering whether a table is empty at a specific time point is a time-dependent rule, which means it does not apply at all time points.

A strict way to detect if two TVRs are equivalent is to check that at all the time points, the corresponding snapshots of the two TVRs are equivalent. However, this detection mechanism is impractical in a real-world query optimizer implementation for two reasons. First, TVR equivalence can only be detected after regular logical rewriting rules are fired on all the time points. During this process, many TVRs could be created but their equivalence cannot be detected. The redundant TVRs can slow down the query optimization speed. Second, the optimizer cannot guarantee that all snapshots at all time points can be fully generated because some operators in the memo might be pruned during the search process. In this case, some TVR equivalence might never be detected.

We introduce a practical mechanism of detecting TVR equivalence by designating a special anchor time point. Tempuraonly allows time-independent rules to be fired on this special time point. In this way, any snapshots at the anchor time point can be generalized to all time points in the TVR. If two snapshots at the anchor points are equivalent, their corresponding TVRs are also equivalent. In the meantime, we still allow time-independent rules to fire at all other time points, increasing the potential to find a better plan.

Formally, two TVRs $R$ and $R'$ are equivalent if they have the same time domain and their snapshots are the same at all valid TVR time points.

$$R' = R \iff \mathcal{T}(R') = \mathcal{T}(R) \wedge R'_t = R_t, \forall t \in \mathcal{T}.$$

If $R'_t$ is equivalent to $R_t$ at a specific time point $t$, it does not imply that both TVRs are identical.

$$\exists t \in \mathcal{T}\ s.t.\ R'_t = R_t \not\Longrightarrow R' = R.$$

Tempuradesignates one time point as the special anchor time point $t_{\perp\!\!\!\smile}$ of a TVR. The anchor time point must ensure that all rules applied at the anchor time point are time-independent and can be generalized to all time points of a TVR.

$$R'_{t_{\perp\!\!\!\smile}} = R_{t_{\perp\!\!\!\smile}} \wedge \mathcal{T}(R') = \mathcal{T}(R) \implies R' = R.$$

Tempurachooses to use the last time point in the time domain as the anchor time point because it produces the final result. Two TVRs are considered equivalent if and only if (1) they share the same logical equivalent class for the anchor snapshot and (2) they have the same TvrMetaSetType. Note that for the same RelSet at the anchor time, Tempuraallows multiple TVRs with different TvrMetaSetTypes to co-exist.

## 8 Improving query optimization speed

As Tempuraexplores a much bigger plan space, if implemented naively, incremental planning can be much slower than traditional query planning. In this section, we discuss several techniques to speed up the optimization process, which help Tempuraachieve comparable optimization latency as traditional optimizers.

### 8.1 Translational symmetry of TVRs

Generating a plan for many time points imposes a challenge for the optimization speed. With an increasing number of time points, the memo size and the overhead of the rule pattern matching and firing grows larger. We have an observation that the TVR rules generate the same patterns when applied on operators of different time points of the same TVR. For instance, in Fig. 9b, if we let $t' = t_1$ instead, $L'$ $(R')$ matches G0 (G1) instead of G3 (G4), and we generate the *InnerJoin* in G7 instead of G8. In other words, *InnerJoin*[0,1] in G7 and *InnerJoin*[3,4] are translation symmetric, modulo the fact that G0, G1, and G7 (G3, G4, and G8) are all snapshot $t_1$ $(t_2)$ of the corresponding TVRs.

Most traditional rewriting rules, such as filter pushdown, are time-independent and have the same behavior on different time points. By leveraging this symmetry, instead of repeatedly firing these rules on all snapshots/deltas of the same set of TVRs, we can apply them on just one snapshot/delta and copy the structures to the rest times. This helps eliminate the expensive process of pattern matching and applying the same rule behavior on different time points in the memo. We first present the process of using translational symmetry to copy the memo, then discuss how Tempurahandles rules that are non-translational, e.g. time-dependent.

*Template Copying.* Before the copying starts, we need to first decide a template and a copy mapping. Out of all time points and intervals, we first choose one consecutive pair of a time point $t$ for snapshot and a time interval $(t, t')$ for delta as the copying template. Then, we define a copy mapping from the template time point/interval to the rest of time points/intervals. For example, if there are three time points $[t_1, t_2, t_3]$ and two time intervals $[(t_1, t_2), (t_2, t_3)]$, we could choose $t_1$ as the template time point and $(t_1, t_2)$ as the template time interval. The copying mapping for time point is

$t_1 \mapsto \{t_2, t_3\}$ and the mapping for time interval is $(t_1, t_2) \mapsto \{(t_2, t_3)\}$. Next, we explain the template-generation phase and the copying phase step-by-step.

1. *Template-Generation Phase.* We seed the TVRs of the leaf operators (usually *Scan* operators) with the snapshot/delta in the template time point/interval. We run the optimizer to populate the memo. Note that all rules except some non-translational symmetric rules discussed later are enabled. This includes the majority of TVR rewrite rules, traditional logical and physical rules. After the rule firing is completed, we record the template operator tree for copying in the next phase.
2. *Copying Phase.* Next, we disable the pattern matching and firing of the rules enabled earlier and copy the template operators to their corresponding mapped time points/intervals. We traverse the template operator tree bottom up in topological order. For each template operator, we find its time point/interval using the intra-TVR link, then copy the operator to all other time points/intervals according to the mapping. After each operator is copied, we record their copied instances at each time point so that the copy of its parent operator can locate the corresponding input.

*Non-Translational Symmetric Rules.* There are two kinds of non-translational rules that are not fired in the copying process: time-dependent rules and rules across more than two time points beyond the template.

Time-dependent rules generates operators that based on time-specific properties that vary across time. For example, an input relation being empty at time point $t_1$ does not imply that the relation will be empty for at all times. If an empty pruning rule is applied to an operator at $t_1$, it cannot be generalized to the entire TVR. As a result, time-dependent rules cannot be fired when constructing the template. Tempuradefers the firings of time-dependent rules after the copying phase has ended. Recall that the rule engine performs rule matching for every structrual change in the memo. Tempuraalways enables such rules for pattern matching during the template-generation and copying phase, but any successful match is put into a separate queue for deferred firing after the copying phase has ended.

Rules across many time points can match multiple operators beyond the template time point/interval. For example, an union merge rule that combines multiple union operators at more than two time points into a single union operator. Such rules might match and generate new patterns during the copy. These rules are also enabled for pattern matching, but deferred for firing after the copying phase.

By leveraging translational symmetry, Tempurais able to scale with many time points because most traditional and TVR rules only need to be matched and fired on one single time point and interval. Moreover, Tempuraensures the completeness and correctness of the memo by a special process of matching and deferred firing of non-translational symmetric rules.

## 8.2 Pruning plan exploration space

*Pruning non-promising alternatives.* There are multiple ways to compute a TVRs snapshot or delta, within which certain ways are usually more costly than others. We can prune the non-promising alternatives. For instance, to compute a delta, one can take the difference of two snapshots, or use TVR-generating rules to directly compute from deltas of the inputs. Based on the experience of previous research on incremental computation [31], we know that 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. To implement this optimization, we can give this subset of intra-TVR rules a lower priority than all other rules, and thus other TVR rewrite rules and traditional rewrite rules will always be ranked higher. Each intra-TVR rule also has an extra skipping condition, which is tested to see whether the target delta is already generated before firing the rule. If so, the rule is skipped.
*Guided exploration.* Inside a TVR, snapshots and deltas 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, and only allow patterns that merge a snapshot with its immediately consecutive delta. In this way, we always use a left-deep merge order.

## 8.3 Optimization of the rule engine

In a traditional cascades-style optimizer, the memo structure is an AND-OR tree. Most rewriting rules are of tree structures specifying the parent–child relationship of a few operators. However in Tempura, the memo becomes a complex graph. The TVR rules are also graphs that need to match multiple operators, TvrMetaSets, and several types of edges among them. Thus, we upgraded the rule engine from supporting tree matching to graph matching. Note that the upgraded rule API is fully backward compatible, all existing rules can work as is.

For example, the TVR-generating rule in Fig. 9a matches five operators, three TvrMetaSets, and seven edges. The inter-TVR rule in Fig. 9b matches five operators, five TvrMetaSets, and nine edges. Rule matching in Temprurais a subgraph isomorphism problem that matches the given rule pattern against the memo graph. The subgraph isomorphism problem is NP-complete and could bring a major performance overhead. In this section, we first explain the general rule-matching and firing process in Tempura, then show how Temprurasppeeds up the rule-matching process using techniques including indexing, pre-compliation, and multiple heuristics on match order.

In Tempura, the rule-matching process is triggered by any structural changes in the memo, for example, adding a new operator, TvrMetaSet, or edge, including intra/inter-TVR edges and edges between operators. Merging of RelSets or TvrMetaSets is also a structural change. For each rule, Tempuratries to match it starting from the location of the triggering change. As shown in Fig. 9b starting from the example triggering TvrMetaSet vertex, Tempurafollows a depth first search (DFS) matching order in the rule pattern. Whenever the matching fails at one point, it backtracks and moves on to the next candidate in the traversal. Upon finding a successful match, the rule with all matched vertices and edges are added to a rule queue, waiting to be applied.

*Pre-compilation of Rule Patterns*. Tempuraoffline analyzes the matching patterns of all user-provided rules and compiles them into data structures specific for subgraph matching. The compilation phase happens before optimizing a query and it consists of two major steps. In the first step, it determines a linear matching order with respect to each vertex and edge as the triggering point. Whenever a rule pattern is triggered during runtime, the matching process just follows the pre-determined matching order without the need to compute the matching order every time. We will cover how Tempuradetermines the match order later in this subsection. In the second step, it analyzes the predicates for all vertices and edges, as well as the predicates on multiple vertices or edges, and pre-process and simplify the matching conditions as much as possible.

*Determining Matching Order*. The matching order has a major impact on optimizer speed and we want to choose an order that can quickly prune the search space and abort as early as possible upon match failure. Next, we list the different options that can lead to different match orders when the backtracking process reaches each type of vertices and edges. We also present the heuristics of our choice to determine a matching order and give the rationale behind the heuristics.

1. *Operator Vertex*: At an operator vertex, we can either match connected operators or connected TVRs. Tempuraprioritizes on matching operators. It first follows Calcite's traversal order for matching the operator tree, which is to travel up to root and then a pre-order traversal for the rest of the operators. When the operator tree is fully matched, it then follow the intra-TVR edges to expand the search to connected TVRs. The rationale is that operators are associated with many predicates that are less likely to find a match and thus can abort early.
2. *TVR Vertex*: At a TVR vertex, we can either match operators belong to this TVR, or other connected TVRs via inter-TVR edges. Tempuraprioritizes checking the inter-TVR trait and the connected TVRs. After matching all inter-TVR edges and connected TVRs, we then traverse the inter-TVR edges to match operators. This is because inter-TVR traits are less likely to find a match, which can cause the rule matching to terminate earlier.
3. *Intra-TVR Edge*: At an Intra-TVR edge, which connects an operator and a TVR, we need to choose the match order when both connected operator and TVR are not matched yet, i.e., the Intra-TVR edge itself is the triggering point. In this case, we can either match the connected operator first, or the connected TVR first. Tempuraprioritizes on matching the connected TVR because this enables expanding the inter-TVR edges faster.
4. *Inter-TVR Edge*: At an Inter-TVR edge, which connects two TVRs, we need to prioritize which side to match first. Tempuracompares the number of inter-TVR edges of the two TVRs, and prioritizes the TVR with more inter-TVR edges.

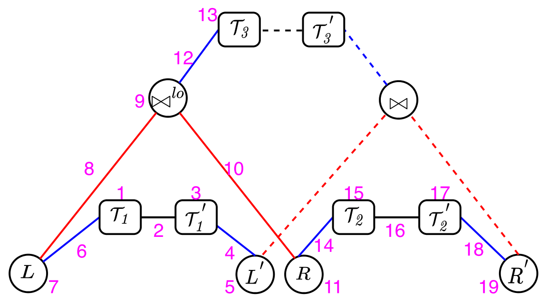


**Fig. 13** A possible matching order of the rule in Fig. 9b starting from a TVR vertex

Figure 13 shows a possible match order of the example rule starting from a TVR vertex.

## 9 Tempura in action

In this section, we discuss a few important considerations when applying Tempurain practice.

*Dynamic re-optimization of incremental plans.* We have studied the IQP problem assuming a static setting, i.e., in $(\vec{T}, \vec{D}, \vec{Q}, \tilde{\mathfrak{c}})$ where $\vec{T}$ and $\vec{D}$ are given and fixed. In many cases, the setting can be much more dynamic where $\vec{T}$ and $\vec{D}$ are subject to change. Tempuracan be adapted to a dynamic setting using re-optimization. Generally, an incremental plan $\mathbb{P} = [P_1, \ldots, P_{i-1}, P_i, \ldots, P_k]$ for $\vec{T} = [t_1, \ldots, t_{i-1}, t_i, \ldots, t_k]$ is only executed up to $t_{i-1}$, after which $\vec{T}$ and $\vec{D}$ change to $\vec{T}' = [t_{i'}, \ldots, t_{k'}]$ and $\vec{D}' = [D_{i'}, \ldots, D_{k'}]$. Tempuracan adapt to this change by re-optimizing the plan under $\vec{T}'$ and $\vec{D}'$. We want to remark that during re-optimization, Tempuracan incorporate the materialized states generated by $P_1, \ldots, P_{i-1}$ as materialized views. In this way, Tempuracan choose to reuse the materialized states instead of blindly re-computing everything.

*Data statistics estimation.* IQP scenarios usually involve planning for future logical times (e.g., IVM-PD) or physical times (e.g., PWD-PD) as described in Sect. 2.1, for which estimating the data statistics becomes very challenging. Since these scenarios typically involve recurring queries, we can use historical data-arrival patterns to estimate future data statistics. Having inaccurate statistics is not a new problem to query optimization, and many techniques have been proposed [52] to tackle this issue. Note that we can always re-optimize the plan when we find that the previously estimated statistics is not accurate. Also, techniques such as robust planning [7,17,51] can be adopted to IQP too. These are out of the scope of this paper.

## 10 Experiments

In this section, we study the effectiveness and efficiency of Tempura. We used the query optimizer of Alibaba Cloud MaxCompute [28], which was built on Apache Calcite 1.17.0 [23], as a traditional optimizer baseline. We implemented Tempuraon the optimizer of MaxCompute. We integrated four commonly used incremental methods into Tempurausing TVR rewrite rules: (1) **IM-1** in Sect. 2.2, (2) **IM-2** in Sect. 2.2 and Sect. 4.2, (3) **OJV** the outer-join view maintenance algorithm in Sect. 4.2, (4) **HOV** the higher-order view maintenance algorithm in Sect. 4.2. By default, Tempurajointly considered all four methods in planning. In the experiments, we used Tempurato simulate each method by turning off the inter-TVR rules of the other methods.

We used two incremental processing scenarios, **PDW-PD** and **IVM-PD** described in Sect. 2.1, to demonstrate Tempura. PDW-PD uses the cost function $\tilde{c}_w(O)$ (in Sect. 6.2), where $c_i$ was a linear function of the estimated CPU/IO/memory/network costs, and $w_i \in [0.25, 0.3]$ for early runs and $w_i = 1$ for the last run.

We used the TPC-DS benchmark [22] ($1TB$) to study the effectiveness (Sect. 10.1) and performance (Sect. 10.3) of Tempura. To further demonstrate the effectiveness of the plans, in Sect. 10.2 we used two real-world analysis workloads consisting of recurrent daily jobs from Alibaba's enterprise data warehouse, denoted as W-A and W-B.

Table 1 shows statistics of the two workloads.

Query optimization was carried out single-threaded on a machine with an Intel Xeon Platinum 8163 CPU @ 2.50GHz and 512GB memory, whereas the generated query was executed on a cluster of thousands of machines shared with other production workloads.

**Table 1** Statistics of two workloads at Alibaba

| | # Queries | Avg # Joins | Avg # Aggregates | # Queries (≥ 1 join) | # Queries (≥ 2 joins) |
|---|---|---|---|---|---|
| W-A | 274 | 1.14 | 1.77 | 167 | 83 |
| W-B | 554 | 1.18 | 1.99 | 357 | 144 |

### 10.1 Effectiveness of IQP

We first evaluated the effectiveness of IQP by comparing Tempurawith four individual incremental methods IM-1, IM-2, OJV, and HOV, in both the PDW-PD and IVM-PD scenarios. We controlled and varied two factors in the experiments: (1) Queries. We chose five representative queries covering complex joins (inner-, left-outer-, and left-semi-joins) and aggregates. (2) Data-arrival patterns. We controlled the amount of input data available in each incremental run by varying the ratio $r = |D_1|/|D_2|$, where $D_1$ is the amount of input data arriving at the first time point, and $D_2$ is the amount of newly arrived input data at the second time point. We chose four data-arrival patterns. Two data-arrival patterns have append-only input data: delta-big ($r = 1$) and delta-small ($r = 4$). We varied the amount of input data arriving at the second time point to test the effect of different delta sizes. Two data-arrival patterns have retractions: delta-R($r = 2$) and delta-RS($r = 2$). Delta-R has retractions in the sales table, whereas delta-RS has retractions in both sales and returns tables. Queries with retractions at the base tables are usually more expensive to incrementally compute because additional states need to be saved to handle retractions. Note that the IM-2method cannot support these two arrival patterns because it cannot handle retractions from the base tables. As the accuracy of cost estimation is orthogonal to Tempura, to isolate its interference, we mainly compared the estimated costs of plans produced by the optimizer, and reported them in relative scale (dividing them by the corresponding costs of IM-1) for easy comparison. We reported the real execution costs as a reference later, and the trend was consistent with the planner's estimation. Due to space limit, we only report the most significant entries in the cost vector of $\tilde{c}_v$ for IVM-PD.

**IVM-PD**. We first fixed the data-arrival pattern to delta-big and varied the queries. The optimal plan costs are reported in Fig. 14a. As shown, different queries prefer different incremental methods. For example, IM-1outperformed both OJVand HOVfor complex queries such as q35. This is because OJVcomputed $Q^I$ by left-semi joining the delta of $Q^D$ with the previous snapshot (Sect. 4.2), and a bigger delta incurred a higher cost of computing $Q^I$. Whereas for simpler queries

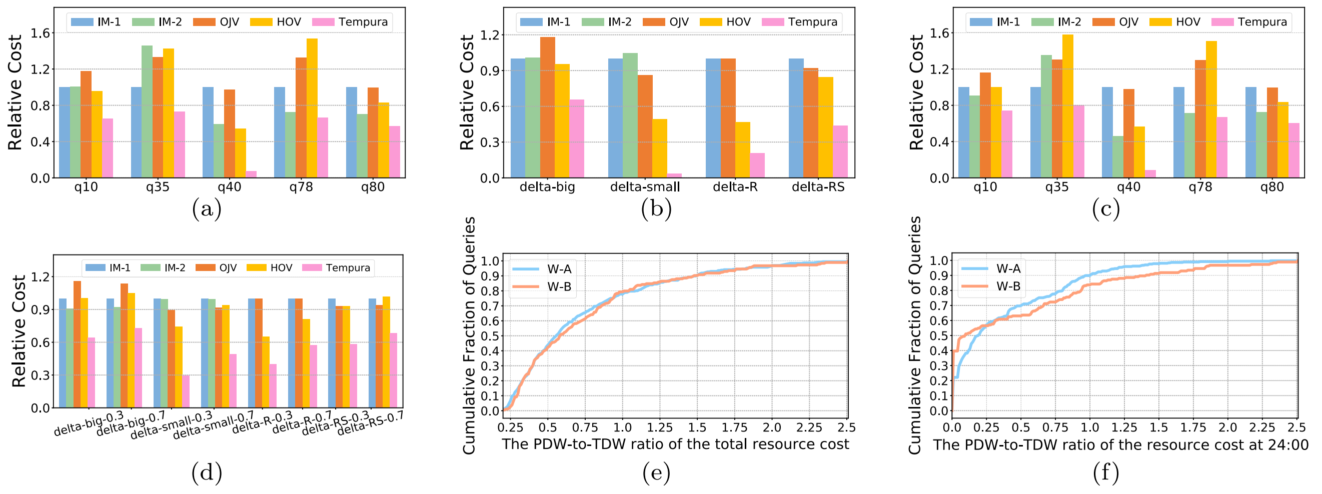


**Fig. 14** **a**, **b** The optimal estimated costs of incremental plans in IVM-PD for different queries and data-arrival patterns. **c**, **d** The optimal estimated costs of incremental plans in PDW-PD for different queries, data-arrival patterns, and cost weights. **e**, **f** The PDW-to-TDW ratio of the real total CPU cost and CPU cost at 24:00 for the data warehouse workloads, respectively

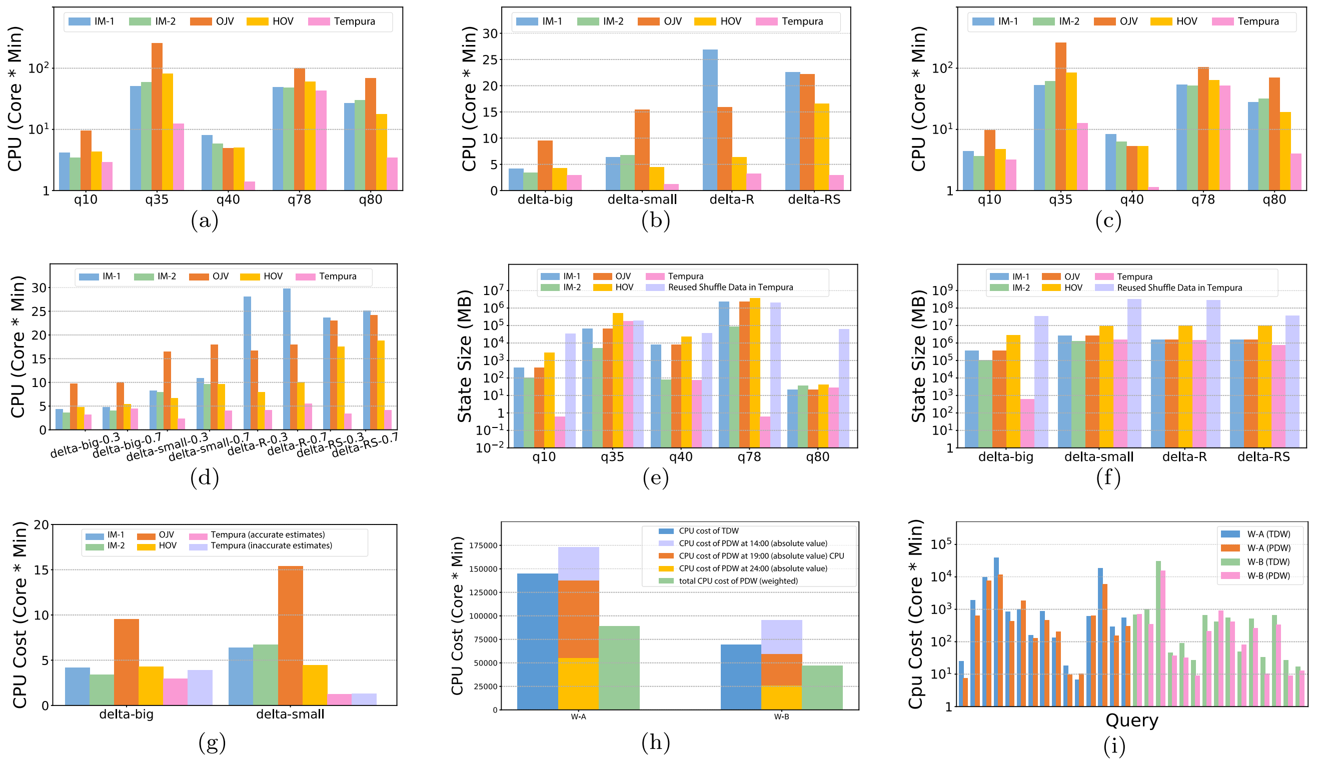


**Fig. 15** **a**, **b** The optimal real CPU costs of different incremental plans in IVM-PD for different queries and data-arrival patterns. **c**, **d** The optimal real CPU costs of different incremental plans in PDW-PD for different queries, data-arrival patterns and cost weights. **e**, **f** The state sizes of different incremental plans in IVM-PD for different queries and data-arrival patterns. **g** The plan quality of Tempuraunder inaccurate cardinality estimation. **h** The comparison between TDW and PDW on the CPU cost of all queries in W-A and W-B, and **i** a detailed comparison of 30 randomly sampled queries in W-A and W-B

such as q80, OJVdegenerated to a similar plan as IM-1, and thus had similar costs. Note that HOVcosts much less than both OJVand IM-1, due to the fact that the maintained higher-order views avoid many repeated joins (e.g., catalog_sales inner joining warehouse, item, and date_dim) as in OJVand IM-1.

Next we chose q10 as a complex query with multiple left-outer joins, and varied the data-arrival patterns. The results are plotted in Fig. 14b. Again, the data-arrival patterns affected the preference of incremental methods. For example, IM-2could not handle input data with retractions. Compared to delta-big, HOVand OJVstarted to outperform IM-1by a large margin in delta-small, as both of them could use different join orders when applying updates to different input relations, and joining a smaller delta earlier could significantly reduce the join cost.

For both experiments, Tempuraconsistently delivered the best plans. For q40 in Fig. 14a and the delta-small case in Fig. 14b, Tempuradelivered a plan 5-10X better than others. Tempuracombines all three of HOV, IM-2and IM-1to generate a mixed optimal plan, and thus leveraged all their advantages. For example, in q40 Tempuraused a similar incremental plan to HOV, but Tempuraused the IM-2approach to join the higher-order views $M$ and $\Delta - R$, and applied IM-1to incrementalize the $Q^N$ part in IM-2.

*PDW-PD*. For the PDW-PD scenario, we conducted the same experiments as in IVM-PD, and in addition tried different weights used in the cost functions ($w_1 = 0.3$ vs. $w_1 = 0.7$). We have similar conclusions as in IVM-PD, and the results are reported in Fig. 14c,d. We make two remarks. (1) Since PDW-PD did not require any outputs at earlier runs, Tempuraautomatically avoided unnecessary computation, e.g., IM-2avoided computing the $Q^N$ part, and thus performed better for q10, q35, q40 than in IVM-PD. (2) The cost function can also affect the choice of the optimizer. For instance, in Fig. 14d, q10 preferred HOVto OJVwhen $w_1 = 0.3$, but the other way when $w_1 = 0.7$. This was because with the cost of early execution increasing, it was less preferable to store many intermediate states as in HOV. Tempuraexploited this fact and adjusted the computation in each run, and moved some early computation from the first incremental run to the second.

*Real CPU Costs*. We reported the real CPU costs in Fig. 15a–d for the experiments in Fig. 14a–d. The CPU costs are in the unit of number_of_cores·time_of_each_core_in _minutes ("CPU·Min" in short). As an example, if a query runs with 2 cores for 3 min, the total CPU cost is 6 CPU·Min's. In the PDW-PD experiments (Fig. 15c, d), the CPU costs were weighted with the cost function in PWD-PD.

Note that Fig. 15a, c is plotted in log scale due to the huge differences in CPU costs for different queries. As we can see, the real CPU costs were agreed with the planner's estimation (Fig. 14a–d) pretty well. Some of the real costs were different from the estimated ones because of the inaccuracy of the cost model. But note that Tempuraconsistently delivered the best plans with the lowest CPU consumption across all experiments.

*Real Wall-clock Execution Time*. We reported the real wall-clock execution time (in seconds) in Fig. 16 in the IVM-PD scenario. Due to the huge differences in execution time for different queries, each query is plotted with a separate scale on the y-axis. We noticed that the initial runs of a few jobs failed and the jobs were restarted by the fault-tolerance mechanism of the platform. To make the comparison fair, we only counted the execution time of successful runs and excluded the additional time of the failed runs. The real wall-clock execution time was mostly similar to the real CPU costs (Figs. 15a–b). Tempurastill consistently delivered the lowest wall-clock execution time across all experiments. Some of the wall-clock execution times were different from the CPU costs because some plans were easier to be parallelized and the query optimizer assigned a higher degree of parallelism to such queries. In such a case, although the wall-clock execution time was lower, the total CPU costs could be similar as more cores were used.

*State Sizes*. In this set of experiments, we study the storage costs of materialized states between Tempuraand each individual incremental methods. We first fixed the data-arrival pattern to delta-big and tested different queries under IVM-PD settings, respectively. The results are reported in Fig. 15e. As shown, for most queries, the sizes of states materialized by Tempurawere smaller than or comparable to each individual incremental algorithms. This is due to the fact that Tempurais able to reuse the shuffled data as the states without incurring additional storage overheads (see Sect. 6.1). Thus, we further reported the sizes of the shuffled data reused by Tempurain the figures. Next we chose query q10 and varied the data-arrival patterns. The results are reported in Fig. 15f. Again, the storage costs of Tempurawere lower than or comparable to that of each individual incremental algorithms.

*Sensitivity to Inaccurate Estimates*. Next, we evaluated the sensitivity of Tempurato inaccurate cardinality estimation. We used q10 in the IVM-PD scenario. We gave Tempurathe estimation of delta-small when running q10 with input delta-big, and gave the estimation of delta-big when running q10 with input delta-small. Figure 15g reports the real CPU costs. For delta-big, Tempurawith the inaccurate estimation ran slower compared to Tempurawith accurate estimation. This is expected because Tempurachose a plan that is optimal to the inaccurate cost model. Nevertheless, Tempurawas still faster than IM-1, OJV, HOV, and comparable to IM-2. For delta-small, inaccurate estimation had a small impact on execution time, and Tempurawas still faster than each individual incremental method.

*Conclusion*. The optimal incremental plan is affected by many factors and does need to be searched in a cost-based

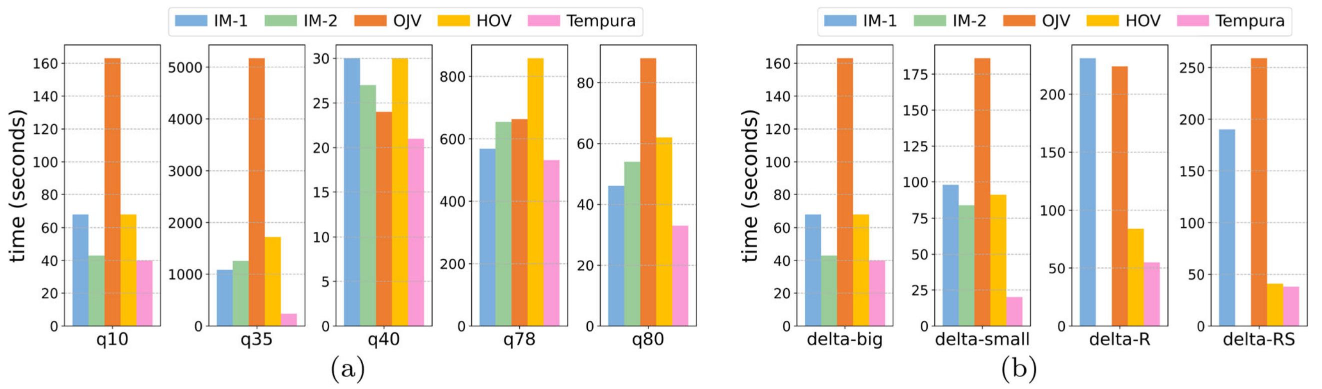


**Fig. 16** **a** The wall-clock execution time in `IVM-PD` for different queries (corresponding to Fig. 15a). **b** The wall-clock execution time in `IVM-PD` for different data-arrival patterns for TPC-DS q10 (corresponding to Fig. 15b)

way. Temperacan consistently find better plans than incremental methods alone.

## 10.2 Case study: progressive data warehouse

To validate the effectiveness of Tempurain a real application, we conducted a case study of the `PDW-PD` scenario using two real-world analysis workloads `W-A` and `W-B` at Alibaba. We compared the resource usage of these workloads in two ways: (1) **Traditional** (TDW), where we ran the workloads at 24:00 according to a schedule using the plans generated by a traditional optimizer; and (2) **Progressive** (PDW), where besides 24:00, we also early executed the workloads at 14:00 and 19:00 (chosen to simulate the observed cluster usage pattern) using the plans generated by Tempura.

Figure 14e shows the real CPU cost of executing the workloads (scored using the cost function in the `PDW-PD` setting), where we plotted the cumulative distribution of the ratio between the CPU cost in PDW versus that in TDW. We can see that PDW delivered better CPU cost for 80% of the queries. For about 60% of the queries, PDW was able to cut the CPU cost by more than 35%. Remarkably, PDW delivered a total cost reduction of 56.2% and 55.5% for `W-A` and `W-B`, respectively. Note that Tempurasearched plans based on the estimated costs which could be different from the real execution cost. As a consequence, for some of the queries (less than 10%) we see more than 50% cost increase. Accuracy of cost estimation is not within the scope of the paper. We further reported the PDW-to-TDW ratio of the CPU cost at 24:00 in Fig. 14f, as this ratio indicated the resource reduction during the "rush hours." As shown, for both workloads, PDW reduced the resource usage at peak hours for over 85% of the queries, and for over 70% of the queries we can see significant reduction of more than 25%.

We also reported the absolute values of CPU costs of `W-A` and `W-B`. However, as `W-A` and `W-B` have 274 and 554 queries each, it is not realistic to show all of them. Instead we reported the total CPU cost breakdowns for TDW and PDW in Fig. 15h. Specifically for PDW, we reported the absolute values of CPU costs at each time, and the total CPU costs weighted according to the cost function in `PDW-PD`. As we can see, Tempuraindeed picked better plans with less resource consumption: PDW saved 38.7% and 32.6% CPU costs compared to TDW for `W-A` and `W-B`, respectively. On the other hand, with incremental computation, PDW had relatively low overheads compared to TDW, 19.6% and 37.6% for `W-A` and `W-B`, respectively. The PDW overheads are computed by summing up the absolute values of CPU costs at each time, minus the CPU costs of TDW. We further randomly selected 15 queries from `W-A` and `W-B`, respectively, and reported their CPU costs in TDW and PDW in Fig. 15i. Again, for most queries PDW reduced the CPU costs by a significant amount.

## 10.3 Performance of IQP

Next, we evaluated the performance of Tempura. IQP has two salient characteristics: (1) In *Plan-Space Exploration* (PSE) phase, IQP explores a larger plan space. (2) IQP has a new *State Materialization Optimization* (SMO) phase to decide the intermediate states to share. We will present performance results on these two phases.

We used PDW-PD as the IQP problem definition. Unless otherwise specified, we set $|\vec{T}| = 3$. We tested Tempuraon the TPC-DS queries. Besides the overall performance study, we also present a detailed study on four aspects:

(1) *Query complexity*: How does Tempuraperform when queries become increasingly complex, e.g., with more joins or subqueries? (2) *Size of IQP*: How does Tempuraperform when $|\vec{T}|$ changes? (3) *Number of incremental methods*: How does Tempuraperform when users integrate more incremental methods into it? (4) *Optimization breakdown*: How effective are the speed-up optimizations in Section 8?

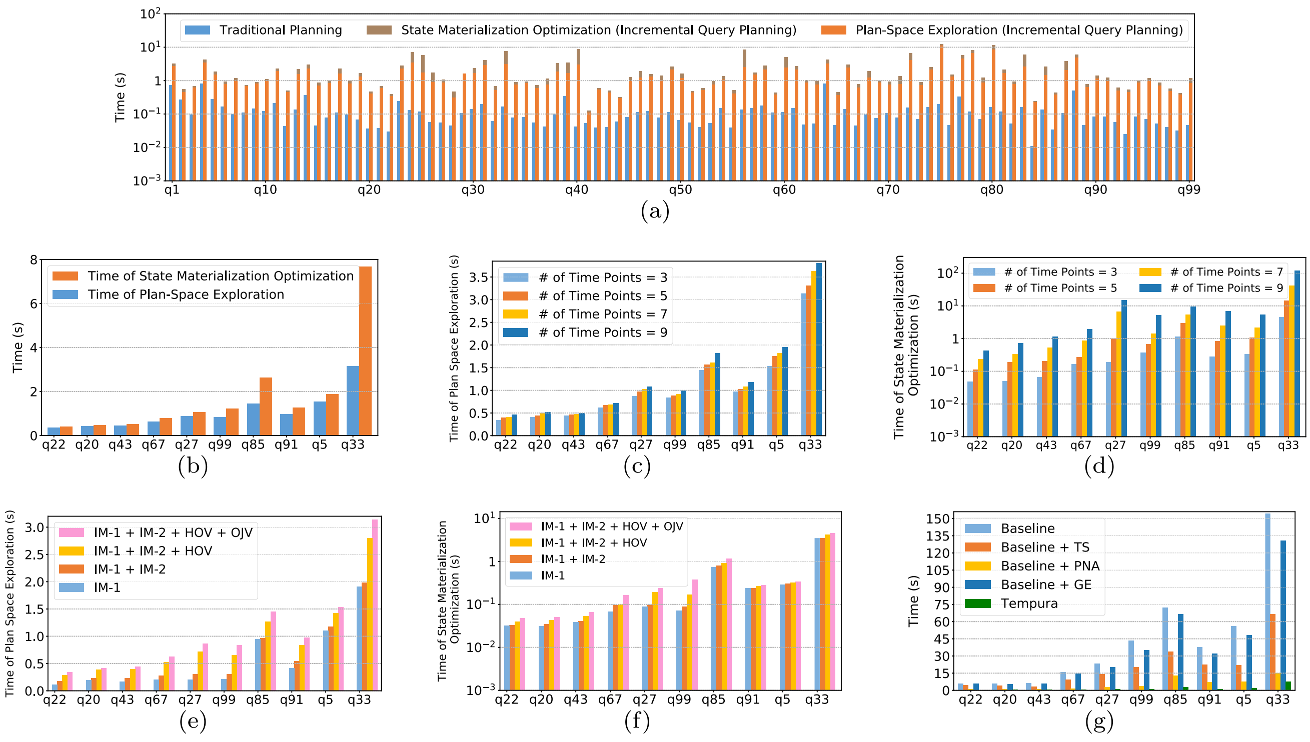


**Fig. 17** **a** Overall planning performance on TPC-DS between traditional and incremental query planning. **b** Impact of the query complexity, **c**, **d** the size of IQP, and **e**, **f** the number of incremental methods on the planning performance. **g** Effectiveness of the speed-up optimization techniques. Note that the selected queries are ordered by their query complexity(as listed in Table 2)

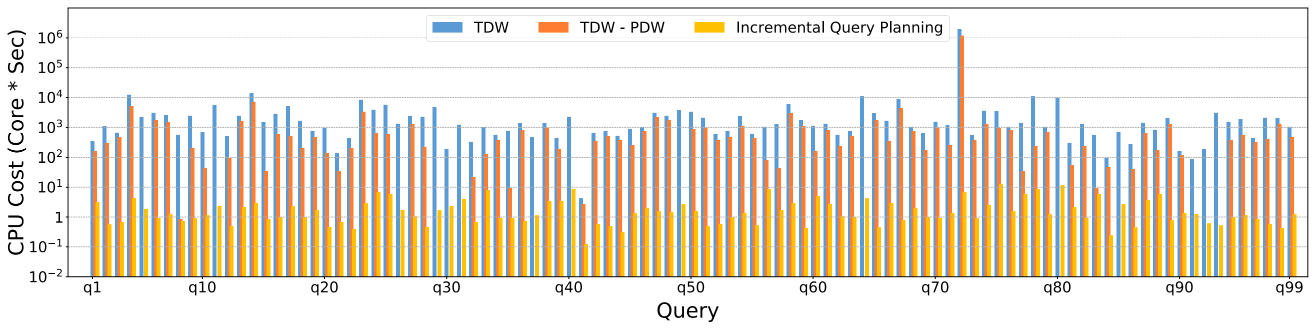


**Fig. 18** Real resource consumption of Tempura's plan as in Fig. 17a on 1T TPC-DS benchmark

To study the above four aspects, we selected ten representative TPC-DS queries with different numbers of joins, aggregates, and subqueries. The selected queries are shown in Table 2.

*Overall Planning Performance*. We first studied the overall planning performance by comparing Temprurawith traditional planning. Figure 17a shows the end-to-end planning time on all TPC-DS queries. As shown, although planned a much bigger plan space, Temprurastill delivered high planning performance: IQP finished within 3 seconds for 80% queries, and for all queries finished within 14 seconds. For over 80% queries, the IQP optimization time was less than 24X of the traditional planning time. Even though slower than traditional planning at optimization time, IQP generated much better incremental plans that brought significant benefit in resource usage and query latency. We can further reduce the planning time by adopting a parallel optimizer [43].

As a reference, we also reported the real CPU cost used by TDW, the CPU costs saved by PDW compared to TDW, and the planning time in Fig. 18. We can see that for most queries, the CPU time on planning was 2-3 orders of magnitude smaller than the saved CPU costs. This shows that the planning cost is negligible compared to the execution cost. Thus, the benefit of a better plan outweighs the extra time spent on planning.

**Table 2** 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 |

*Query Complexity.* To study the impact of query complexity, we reported the planning time breakdown on the selected TPC-DS queriesin Fig. 17b. As shown, the planning time increased when the query complexity increased, because the plan space grew larger for complex queries. The time spent on PSE was less than that spent on SMO in general, and also grew with a slower pace. This shows that query complexity has a smaller impact on PSE.

*Size of IQP*. To study the impact of the size of the planning problem, we gradually increased the number of incremental runs planned from 3 to 9, and reported the time on PSE and SMO in Fig. 17c, d. As depicted, the time on PSE stayed almost constant as the size of IQP changed. For example, when the number of incremental runs grew 3X, the time for q33 only slightly increased by 20%. This was mainly due to the effective speed-up optimization techniques introduced in Sect. 8. In comparison, the SMO time increased superlinearly with increase in number of incremental runs, due to the time complexity of the MQO algorithm we chose [30].

*Number of Incremental Methods*. To study the impact of more incremental methods, we gradually added methods `IM-1`, `IM-2`, `HOV`and `OJV`into Tempura. Figure 17f, g show the time on PSE and SMO, respectively. As illustrated, the time on both PSE and SMO increased with more incremental methods, due to the increased plan space. There are two interesting findings. (1) The PSE time did not grow linearly with the number of incremental methods, but rather the the plan space size that each method newly introduces. For example, the increase of PSE time at adding `HOV`was bigger than that at adding `OJV`. This was because both `HOV`and `OJV`update a single relation at a time, which are very different from `IM-1`and `IM-2`that update all relations each time. (2) The number of incremental methods had less impact than the size of the IQP problem, which can be observed on the SMO time. This is because the plan space explored by different incremental methods often have overlaps, whereas the plan spaces of different incremental runs do not.

*Exploration Optimization Breakdown*. We evaluated the effectiveness of the speed-up optimizations of exploring the plan space discussed in Sect. 8, i.e., translational symmetry (TS), pruning non-promising alternatives (PNA), and guided exploration (GE). Figure 17g reports the PSE times of different combinations of the speed-up optimizations. We compared the implementations with no optimization (Baseline), with each individual optimization (Baseline+TS, +PNA, +GE), and with all three optimizations (Tempura). The optimizations together brought up to 20X speed-up, among which the most effective ones were PNA and TS, bringing 5-12X and 1.5-2.5X improvements each.

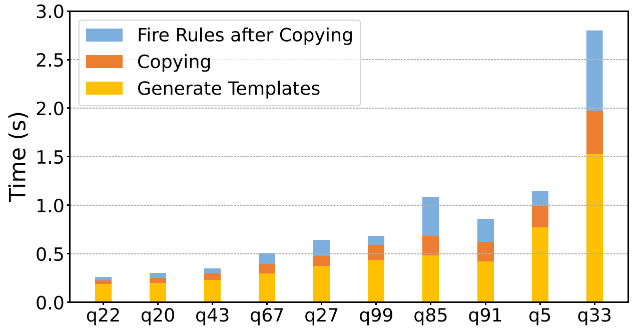


**Fig. 19** Time breakdown of three steps in the memo-copying process: template generation, template copying, and firing non-translational symmetric rules after copying

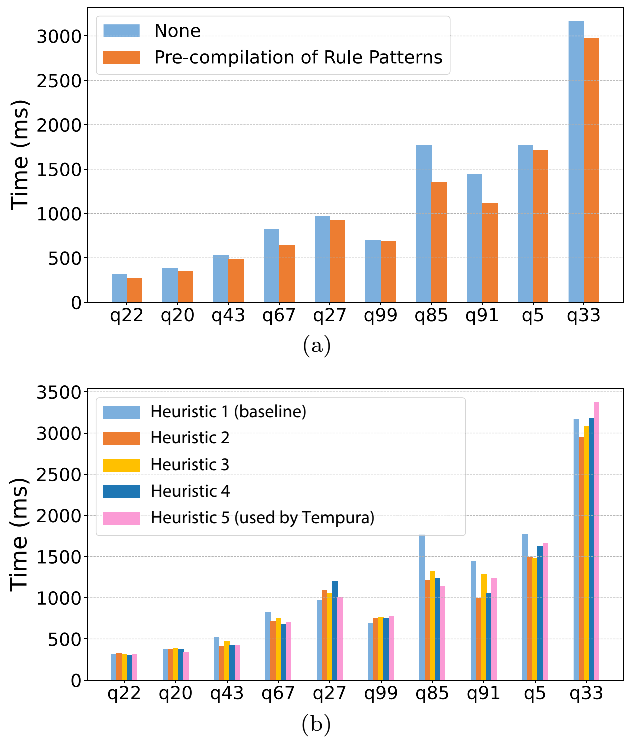


**Fig. 20** Effect of different rule engine optimization techniques on overall planning performance: **a** pre-compilation of rule patterns and **b** different match-order heuristics

*Effect of Exploiting TVR Translational Symmetry*. We evaluated the memo-copying process using the TVR translational symmetry in Sect. 8.1. Figure 19 shows the breakdown of planning time of the three steps used in the copying process, with two time points in the initial template-generation phase, and three additional time points in the template-copying phase. We have the following observations: (1) the time of the template-generation phase varied and it was determined by the complexity of each query; (2) the time spent on copying the template to three additional time points was much less

**Table 3** Configurations of different match-order heuristics used in Fig. 20b

| | Operator Vertex prioritize TVR/operator | TVR Vertex prioritize TVR/operator | Intra-TVR edge prioritize TVR/operator | Inter-TVR edge prioritize TVR w/ more/less edges |
|---|---|---|---|---|
| 1 | TVR | Operator | Operator | Less |
| 2 | Operator | Operator | Operator | Less |
| 3 | Operator | TVR | Operator | Less |
| 4 | Operator | TVR | TVR | Less |
| 5 | Operator | TVR | TVR | More |

than generating the template on two time points; and (3) the time taken by firing non-translational symmetric rules after the copying were usually small, but it took a long time in a few queries. This is because the new operators generated by non-translational symmetric rules further triggered many traditional rewrite rules such as enforcer rules.

*Effect of Rule Engine Optimizations*. We study the effect of the optimization techniques to speed up the rule-matching process of the rule engine in Sect. 8.3. Figure 20a shows the benefit of pre-compilation of rule patterns. We observed that the pre-compilation optimization introduced performance gains on almost all queries tested. Recall that pre-compilation can avoid re-computing the match order on each rule firing. Since the number of rule firings was very large, this optimization cumulatively saved a large amount of time. Figure 20b shows the effect of different match-order heuristics on the optimization speed. The specific configuration of each heuristic can be found in Table 3. We had the following observations. (1) The experiment showed that different match orders indeed had different impacts on the optimization speed. For example, in query 85, heuristic 1 was about 30% slower than heuristic 5. Therefore, it is important to choose a good match order to accelerate the optimization. (2) The best heuristic was query-dependent and there was no single heuristic match order that performs the best in all the cases. The heuristic we chose out-performs the baseline for most queries. Based on these observations, Tempuraallows a developer to tune the match-order heuristics based on the query and workload.

## 11 Related work

*Incremental Processing*. There are rich research works on incremental processing, ranging from incremental view maintenance, stream computing, to approximate query answering and so on. Incremental view maintenance has been studied under both the set [10,11] and bag [14,20] semantics, for queries with outer joins [21,33], and using higher-order maintenance methods [3]. Previous studies mainly focused on delta propagation rules for relational operators. Stream computing [1,16,37,47] adopts incremental processing and sublinear-space algorithms to process updates and deltas. Many approximate query answering studies [2,6,15] focused on constructing optimal samples to improve query accuracy. Proactive or trigger-based incremental computation techniques [13,54] were used to achieve low query latency. These studies proposed incremental techniques in isolation, and do not have a general cost-based optimization framework. In addition, they can be integrated into Tempura.

*Query Planning for Incremental Processing*. Previous work studied some optimization problems in incremental computation. Viglas et al. [48] proposed a rate-based cost model for stream processing. The cost model is orthogonal to Tempuraand can be integrated. DBToaster [3] discussed a cost-based approach to deciding the views to materialize under a higher-order view maintenance algorithm. Tang et al. [45] focused on selecting optimal states to materialize for scenarios with intermittent data arrival. They proposed a DP algorithm for selecting states to materialize given a fixed physical incremental plan and a memory budget, by considering future data-arrival patterns. These optimization techniques all focus on the optimal materialization problem for a specific incremental plan or incremental method, and thus are not general IQP solutions. Tang et al. [44] discussed the idea of eagerly (or lazily) executing parts of a query that is more (or less) amenable to incremental execution. Tempuracan also support this style of execution in the `PDW-PD` setting, where the final results are delivered only at the last run. At earlier runs, the optimizer can choose to incrementally execute only a sub-part of the query based on cost. In fact, we often observed this behavior in the `PDW-PD` setting in the experiments. [44] analyzes the cost of incremental execution based on the concept of incrementability. This can be adopted in Tempuraas a new cost function following the discussion in Sect. 6.2.

Flink [25] uses Calcite [9] as the optimizer to support stream queries, which only provides traditional optimizations on the logical plan generated by a fixed incremental method, but cannot combine multiple incremental methods, nor consider correlations between incremental runs. On the contrary, Tempuraprovides a general framework for users to integrate various incremental methods, and searches the plan space in a cost-based approach.

*Semantic Models for Incremental Processing.* CQL[5] exploi ted the relational model to provide strong query semantics for stream processing. Sax et al. [41] introduced the dual streaming model to reason about ordering in stream processing. The key idea behind [5,41] is the duality of relations and streams, i.e., time-varying relations can be modeled as a sequence of static relations, or a sequence of change logs. The recent work [8] proposed to integrate streaming into the SQL standard, and briefly mentioned that TVRs can serve as a unified basis of both relations and streams. However, their models do not include a formal algebra and rewrite rules on TVRs. To the best of our knowledge, our TIP modelfor the first time formally defines an algebra on TVRs, providing a principled way to model different types of snapshots/deltas and operators between them. The trichotomy of TVR rewrite rules subsumes many existing incremental methods, laying a theoretical foundation for Tempura.

## 12 Conclusion

In this paper, we proposed a theory called TIP modelto formally model incremental processing in its most general form, and based on it developed a novel principled cost-based optimizer framework Tempurafor incremental data processing. Tempuranot only unifies various existing techniques to generate an optimal incremental plan, but also allows the developer to add their rewrite rules. We conducted thorough experimental evaluation of Tempurain various incremental query scenarios to show its effectiveness and efficiency.

We implemented Tempurabased on Apache Calcite and it is open sourced at [26]. Temputais also being incorporated into Apache Calcite codebase as a configurable feature at [27].

## References

1. Abadi, D.J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., et al.: The design of the borealis stream processing engine. In: Cidr, vol. 5, pp. 277–289 (2005)
2. Acharya, S., Gibbons, P.B., Poosala, V., Ramaswamy, S.: The aqua approximate query answering system. In: ACM Sigmod Record, vol. 28, pp. 574–576. ACM (1999)
3. Ahmad, Y., Kennedy, O., Koch, C., Nikolic, M.: Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. PVLDB **5**(10), 968–979 (2012)
4. Aiken, A., Hellerstein, J.M., Widom, J.: Static analysis techniques for predicting the behavior of active database rules. ACM Trans. Database Syst. (TODS) **20**(1), 3–41 (1995)
5. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. VLDB J. **15**(2), 121–142 (2006)
6. Babcock, B., Chaudhuri, S., Das, G.: Dynamic sample selection for approximate query processing. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 539–550. ACM (2003)
7. Babu, S., Bizarro, P., DeWitt, D.: Proactive re-optimization. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, pp. 107–118 (2005)
8. Begoli, E., Akidau, T., Hueske, F., Hyde, J., Knight, K., Knowles, K.L.: One SQL to rule them all - an efficient and syntactically idiomatic approach to management of streams and tables. In: Boncz, P.A., Manegold, S., Ailamaki, A., Deshpande, A., Kraska, T. (eds.) Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30–July 5, 2019, pp. 1757–1772. ACM (2019). https://doi.org/10.1145/3299869.3314040
9. Begoli, E., Camacho-Rodríguez, J., Hyde, J., Mior, M.J., Lemire, D.: 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, pp. 221–230. ACM, New York, NY, USA (2018). https://doi.org/10.1145/3183713.3190662
10. Blakeley, J.A., Larson, P.A., Tompa, F.W.: Efficiently updating materialized views. In: Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, SIGMOD '86, pp. 61–71. ACM, New York, NY, USA (1986). https://doi.org/10.1145/16894.16861
11. Buneman, O.P., Clemons, E.K.: Efficiently monitoring relational databases. ACM Trans. Database Syst. **4**(3), 368–382 (1979). https://doi.org/10.1145/320083.320099
12. Chandramouli, B., Bond, C.N., Babu, S., Yang, J.: Query suspend and resume. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, pp. 557–568 (2007)
13. Chandramouli, B., Goldstein, J., Quamar, A.: Scalable progressive analytics on big data in the cloud. Proc. VLDB Endow. **6**(14), 1726–1737 (2013). https://doi.org/10.14778/2556549.2556557
14. Chaudhuri, S., Krishnamurthy, R., Potamianos, S., Shim, K.: Optimizing queries with materialized views. In: Proceedings of the Eleventh International Conference on Data Engineering, ICDE '95, pp. 190–200. IEEE Computer Society, Washington, DC, USA (1995). http://dl.acm.org/citation.cfm?id=645480.655434
15. Chaudhuri, S., Das, G., Narasayya, V.: Optimized stratified sampling for approximate query processing. ACM Trans. Database Syst. (TODS) **32**(2), 9 (2007)
16. Ghanem, T.M., Elmagarmid, A.K., Larson, P.Å., Aref, W.G.: Supporting views in data stream management systems. ACM Trans. Database Syst. (TODS) **35**(1), 1 (2010)
17. Graefe, G., Guy, W., Kuno, H.A., Paullley, G.: Robust query processing (dagstuhl seminar 12321). In: Dagstuhl Reports, vol. 2. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2012)
18. Graefe, G., McKenna, W.J.: The volcano optimizer generator: Extensibility and efficient search. In: Proceedings of IEEE 9th International Conference on Data Engineering, pp. 209–218. IEEE
19. Graefe, G.: The cascades framework for query optimization. Data Eng. Bull. **18**, 19–29 (1995)
20. Griffin, T., Libkin, L.: Incremental maintenance of views with duplicates. In: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, SIGMOD '95, pp. 328–339. ACM, New York, NY, USA (1995). https://doi.org/10.1145/223784.223849
21. Griffin, T., Kumar, B.: Algebraic change propagation for semijoin and outerjoin queries. SIGMOD Rec. **27**(3), 22–27 (1998). https://doi.org/10.1145/290593.290597
22. http://www.tpc.org/tpcds/
23. https://calcite.apache.org
24. https://databricks.com/blog/2018/03/13/introducing-stream-stream-joins-in-apache-spark-2-3.html
25. https://flink.apache.org
26. https://github.com/alibaba/cost-based-incremental-optimizer

27. https://issues.apache.org/jira/browse/CALCITE-4568
28. https://www.alibabacloud.com/product/maxcompute
29. Jia, J., Li, C., Carey, M.J.: Drum: a rhythmic approach to interactive analytics on large data. In: 2017 IEEE International Conference on Big Data (Big Data), pp. 636–645. IEEE (2017)
30. Kathuria, T., Sudarshan, S.: Efficient and provable multi-query optimization. In: Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS '17, pp. 53–67. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3034786.3034792
31. Koch, C.: Incremental query evaluation in a ring of databases. In: Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 87–98 (2010)
32. Lang, W., Nehme, R.V., Robinson, E., Naughton, J.F.: Partial results in database systems. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14, pp. 1275–1286. ACM, New York, NY, USA (2014). https://doi.org/10.1145/2588555.2612176
33. Larson, P., Zhou, J.: 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, pp. 56–65 (2007). https://doi.org/10.1109/ICDE.2007.367851
34. Law, Y.N., Wang, H., Zaniolo, C.: Query languages and data models for database sequences and data streams. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30, VLDB '04, p. 492-503. VLDB Endowment (2004)
35. Lee, M.K.: Implementing an interpreter for functional rules in a query optimizer (1988)
36. Maier, D., Li, J., Tucker, P., Tufte, K., Papadimos, V.: Semantics of data streams and operators. In: Eiter, T., Libkin, L. (eds.) Database Theory - ICDT 2005, pp. 37–52. Springer, Berlin, Heidelberg (2005)
37. Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., Varma, R.: Query processing, resource management, and approximation in a data stream management system. In: CIDR (2003)
38. Nikolic, M., Dashti, M., Koch, C.: 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, pp. 511–526. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2882903.2915246
39. Raman, V., Hellerstein, J.M.: Partial results for online query processing. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, pp. 275–286 (2002)
40. Roy, P., Seshadri, S., Sudarshan, S., Bhobe, S.: Efficient and extensible algorithms for multi query optimization. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, SIGMOD '00, pp. 249–260. ACM, New York, NY, USA (2000). https://doi.org/10.1145/342009.335419
41. Sax, M.J., Wang, G., Weidlich, M., Freytag, J.C.: Streams and tables: Two sides of the same coin. In: Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics, BIRTE '18. Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3242153.3242155
42. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, pp. 23–34 (1979)
43. Soliman, M.A., Antova, L., Raghavan, V., El-Helw, A., Gu, Z., Shen, E., Caragea, G.C., Garcia-Alvarado, C., Rahman, F., Petropoulos, M., Waas, F., Narayanan, S., Krikellas, K., Baldwin, R.: Orca: A modular query optimizer architecture for big data. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14, pp. 337–348. ACM, New York, NY, USA (2014). https://doi.org/10.1145/2588555.2595637
44. Tang, D., Shang, Z., Elmore, A.J., Krishnan, S., Franklin, M.J.: Thrifty query execution via incrementability. In: Maier, D., Pottinger, R., Doan, A., Tan, W., Alawini, A., Ngo, H.Q. (eds.) Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14–19, 2020, pp. 1241–1256. ACM (2020). https://doi.org/10.1145/3318464.3389756
45. Tang, D., Shang, Z., Elmore, A.J., Krishnan, S., Franklin, M.J.: Intermittent query processing. Proc. VLDB Endow. **12**(11), 1427–1441 (2019). https://doi.org/10.14778/3342263.3342278
46. Terry, D., Goldberg, D., Nichols, D., Oki, B.: Continuous queries over append-only databases. In: Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, SIGMOD '92, pP. 321–330. Association for Computing Machinery, New York, NY, USA (1992). https://doi.org/10.1145/130283.130333
47. Thakkar, H., Laptev, N., Mousavi, H., Mozafari, B., Russo, V., Zaniolo, C.: Smm: A data stream management system for knowledge discovery. In: 2011 IEEE 27th International Conference on Data Engineering, pp. 757–768. IEEE (2011)
48. Viglas, S.D., Naughton, J.F.: Rate-based query optimization for streaming information sources. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, pp. 37–48 (2002)
49. Wang, Z., Zeng, K., Huang, B., Chen, W., Cui, X., Wang, B., Liu, J., Fan, L., Qu, D., Ho, Z., Guan, T., Li, C., Zhou, J.: Grosbeak: A data warehouse supporting resource-aware incremental computing. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20. ACM, Portland, Oregon, USA (2020)
50. Wang, Z., Zeng, K., Huang, B., Chen, W., Cui, X., Wang, B., Liu, J., Fan, L., Qu, D., Hou, Z., Guan, T., Li, C., Zhou, J.: Tempura: a general cost-based optimizer framework for incremental data processing. Proc. VLDB Endow. **14**(1), 14–27 (2020). https://doi.org/10.14778/3421424.3421427
51. Wolf, F., May, N., Willems, P.R., Sattler, K.U.: On the calculation of optimality ranges for relational query execution plans. In: Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18, p. 663-675. Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3183713.3183742
52. Yin, S., Hameurlain, A., Morvan, F.: Robust query optimization methods with respect to estimation errors: a survey. ACM Sigmod Record **44**(3), 25–36 (2015)
53. Yu, Y., Gunda, P.K., Isard, M.: Distributed aggregation for data-parallel computing: interfaces and implementations. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, pp. 247–260 (2009)
54. Zeng, K., Agarwal, S., Stoica, I.: iolap: Managing uncertainty for efficient incremental olap. In: Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, pp. 1347–1361. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2882903.2915240

55. Zhang, Y., Hull, B., Balakrishnan, H., Madden, S.: Icedb: Intermittently-connected continuous query processing. In: 2007 IEEE 23rd International Conference on Data Engineering, pp. 166–175. IEEE (2007)
56. Zhou, J., Larson, P.A., Larson, P.A., Freytag, J.C., Lehner, W.: Efficient exploitation of similar subexpressions for query processing. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD '07, pp. 533–544. ACM, New York, NY, USA (2007). https://doi.org/10.1145/1247480.1247540