Tempura: A General Cost-Based Optimizer Framework for Incremental Data Processing

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]
[1]University of California, Irvine [2]Alibaba Group
[1] Irvine, United States [2] Hangzhou, China
[1]{zuozhiw, chenli}@ics.uci.edu, [2]{zengkai.zk, botong.huang, wickeychen.cw, xiaozong.cxz, yanyu.wb, niki.lj,
liya.fly, dachuan.qdc, zhenyuhou.hzy, tony.guan, jingren.zhou}@alibaba-inc.com

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.

PVLDB Reference Format:
Zuozhi Wang, Kai Zeng, Botong Huang, Wei Chen, Xiaozong Cui, Bo Wang, Ji Liu, Liya Fan, Dachuan Qu, Zhenyu Hou, Tao Guan, Chen Li, Jingren Zhou. Tempura: A General Cost-Based Optimizer Framework for Incremental Data Processing. PVLDB, 14(1): 14-27, 2021.
doi:10.14778/3421424.3421427

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, 18, 24, 30] and stream processing [1, 8, 19, 34, 42]. It has been adopted in various application domains such as active databases [4], resumable query execution [15], approximate query processing [16, 26, 50], etc. New advancements in big data systems make data ingestion more real-time and analysis increasingly time sensitive, which further boost the adoption of the incremental processing model. Here are a few examples of emerging applications.

Progressive Data Warehouse [46]. Enterprise data warehouses usually have a large amount of automated routine analysis jobs, which have a stringent schedule and deadline 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 [40]. 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, 18, 30], or algorithms to determine which intemediate states to materialize [36, 40, 51]. 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)) AS gross
  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

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:10.14778/3421424.3421427

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 [18, 23, 24, 50]. 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. (2) A typical stream-computing method (denoted as **IM-2**) avoids such retraction [25, 31, 33, 41]. It holds back sales orders that do not join with any returns orders until all data is available. 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 §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 [22, 39] (see §2.2): (1) Incremental query planning requires tradeoff analysis on more dimensions than traditional query planning, such as different incremental 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, and defines a formal algebra for TVRs (§3). In the TIP model, we also provide a rewrite-rule framework to describe different incremental computation techniques, and unify them to explore in a single search space for an optimal incremental plan (§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 (§5) and search for an optimal incremental plan in Tempura (§6).
- We conduct a thorough experimental evaluation of the Tempura optimizer in various application scenarios. The results show the effectiveness and efficiency of Tempura (§8).

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.
- $\vec{D} = [D_1, \cdots, D_k]$ 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 $\emptyset$.
- $\tilde{c}$ 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) $\forall 1 \leq i \leq k$, P_i can deliver the results defined by Q_i, and (2) the cost $\tilde{c}(\mathbb{P})$ is minimized. Next we use a few example 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 $\tilde{c}$ 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 §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 ΔD 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, $\vec{Q} = [\emptyset, \cdots, \emptyset, Q]$. We set $\tilde{c}$ as a weighted sum of the costs of all plans in $\mathbb{P}$ (see $\tilde{c}_w(O)$ in §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 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. 1(a). 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. 1(a), and $\tilde{c}$ 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, 8, 18, 24]) can be used here. Consider two commonly used methods IM-1 and IM-2.

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)

Figure 1: (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-1 at t_1 and t_2, and (d) incremental results of sales_status produced by IM-2 at t_1, t_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 $\{o_5, o_6, o_7\}$. Fig. 1(c) depicts sales_status's incremental outputs at t_1 and t_2, respectively, where # = +/−1 denote insertion or deletion respectively. Note that a returns record (e.g., 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. 1(c).

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 will be held back at t_1. Essentially the outer join is computed as an inner join at t_1. The incremental outputs of sales_status are shown in Fig. 1(d).

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 recomputation. 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, different 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.

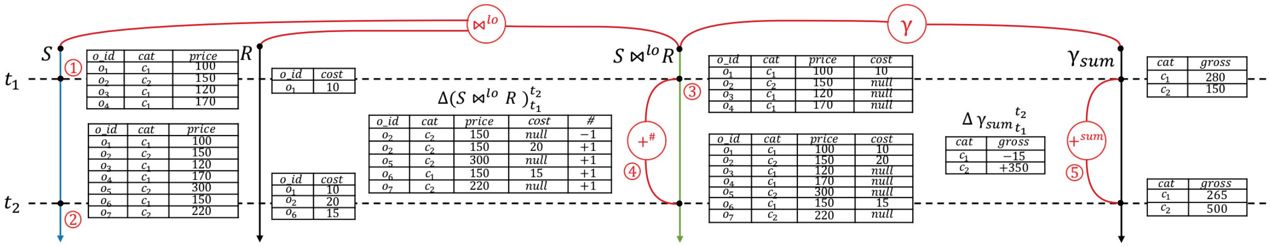
3 THE TIP MODEL

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) [8, 11, 37], 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 model is 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 §4.

3.1 Time-Varying Relations

Definition 2. *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. 2. On the



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

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

o_id	cost
o_1	10

o_id	cost
o_1	10
o_2	20
o_6	15

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

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

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

cat	gross
c_1	−15
c_2	+350

cat	gross
c_1	280
c_2	150

cat	gross
c_1	265
c_2	500

Figure 2: Example TVRs and their relationships.

line, tables ① and 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 3 (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. 2, 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 γ. 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 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 4 (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. 2 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 $\Delta\gamma_{sum}{}_{t_1}^{t_2}$ can be defined differently as Table ⑤ in Fig. 2. 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 $\gamma_{sum}{}_{t_1}$ and $\Delta\gamma_{sum}{}_{t_1}^{t_2}$ yields the value in $\gamma_{sum}{}_{t_2}$, 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 γ_{sum} as $+^{sum}$. This observation shows that the way to define TVR deltas and the merge operator + is not unique. In general, as studied in previous research [29, 50], 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., $+^{sum}$) 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 [49]: `Initialize`, `Iterate`, `Merge`, and `Final`. 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 $-^{sum}$ for $+^{sum}$ 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 incremental 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.

$$Q(R_{t'}) = Q(R_t + \Delta R_t^{t'}) = Q(R_t) + \mathfrak{d}Q(R_t, \Delta R_t^{t'}). \qquad (1)$$

[1] Note that `Final` also needs to filter out empty groups with zero contributing tuples. We omit this detail due to the limited space.

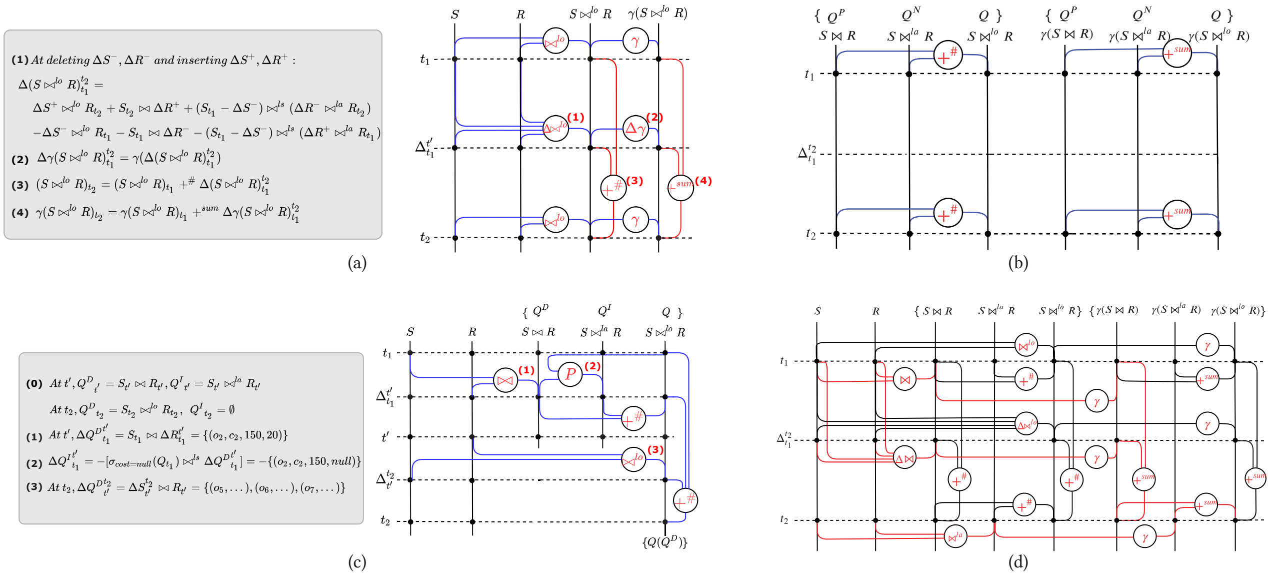

(1) At deleting $\Delta S^-, \Delta R^-$ and inserting $\Delta S^+, \Delta R^+$:
$\Delta(S \bowtie^{lo} R)_{t_1}^{t_2} =$
$\Delta S^+ \bowtie^{lo} R_{t_2} + S_{t_2} \bowtie \Delta R^+ + (S_{t_1} - \Delta S^-) \bowtie^{ls} (\Delta R^- \bowtie^{la} R_{t_2})$
$-\Delta S^- \bowtie^{lo} R_{t_1} - S_{t_1} \bowtie \Delta R^- - (S_{t_1} - \Delta S^-) \bowtie^{ls} (\Delta R^+ \bowtie^{la} R_{t_1})$
(2) $\Delta\gamma(S \bowtie^{lo} R)_{t_1}^{t_2} = \gamma(\Delta(S \bowtie^{lo} R)_{t_1}^{t_2})$
(3) $(S \bowtie^{lo} R)_{t_2} = (S \bowtie^{lo} R)_{t_1} +^{\#} \Delta(S \bowtie^{lo} R)_{t_1}^{t_2}$
(4) $\gamma(S \bowtie^{lo} R)_{t_2} = \gamma(S \bowtie^{lo} R)_{t_1} +^{sum} \Delta\gamma(S \bowtie^{lo} R)_{t_1}^{t_2}$
S
R
$S \bowtie^{lo} R$
$\gamma(S \bowtie^{lo} R)$
t_1
$\Delta_{t_1}^{t'}$
t_2
(1)
(2)
(3)
(4)
(a)
$\{ Q^P$
$S \bowtie R$
Q^N
$S \bowtie^{la} R$
$Q \}$
$S \bowtie^{lo} R$
$\{ Q^P$
$\gamma(S \bowtie R)$
Q^N
$\gamma(S \bowtie^{la} R)$
$Q \}$
$\gamma(S \bowtie^{lo} R)$
t_1
$\Delta_{t_1}^{t_2}$
t_2
(b)
(0) At t', $Q^D{}_{t'} = S_{t'} \bowtie R_{t'}$, $Q^I{}_{t'} = S_{t'} \bowtie^{la} R_{t'}$
At t_2, $Q^D{}_{t_2} = S_{t_2} \bowtie^{lo} R_{t_2}$, $Q^I{}_{t_2} = \emptyset$
(1) At t', $\Delta {Q^D}_{t_1}^{t'} = S_{t_1} \bowtie \Delta R_{t_1}^{t'} = \{(o_2, c_2, 150, 20)\}$
(2) $\Delta {Q^I}_{t_1}^{t'} = -[\sigma_{cost=null}(Q_{t_1}) \bowtie^{ls} \Delta {Q^D}_{t_1}^{t'}] = -\{(o_2, c_2, 150, null)\}$
(3) At t_2, $\Delta {Q^D}_{t'}^{t_2} = \Delta S_{t'}^{t_2} \bowtie R_{t'} = \{(o_5, \dots), (o_6, \dots), (o_7, \dots)\}$
S
R
$\{ Q^D$
$S \bowtie R$
Q^I
$S \bowtie^{la} R$
$Q \}$
$S \bowtie^{lo} R$
t_1
$\Delta_{t_1}^{t'}$
t'
$\Delta_{t'}^{t_2}$
t_2
P
(1)
(2)
(3)
$\{Q(Q^D)\}$
(c)
S
R
$\{ S \bowtie R$
$S \bowtie^{la} R$
$S \bowtie^{lo} R\}$
$\{\gamma(S \bowtie R)$
$\gamma(S \bowtie^{la} R)$
$\gamma(S \bowtie^{lo} R)\}$
t_1
$\Delta_{t_1}^{t_2}$
t_2
(d)

Figure 3: (a) Examples of TVR-generating and intra-TVR rules, (b) examples of inter-TVR equivalence rules in IM-2, (c) examples of inter-TVR equivalence rules in outer-join view maintenance, and (d) the incremental plan space of Example 1.

The idea is intuitive: 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 $\eth Q$. Essentially, Eq. 1 contains two key parts—the delta query $\eth Q$ and the merge operator +, which correspond to the first two types of TVR rewrite rules, namely *TVR-generating rules* and *intra-TVR rules*, respectively.

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 $\eth Q$ for each relational operator Q such that $\Delta Q_t^{t'} = \eth Q(R_t, \Delta R_t^{t'})$. Many previous studies on deriving delta queries under different semantics [13, 14, 18, 23, 24] fall into this category. As an example, Fig. 3(a) shows the TVR-generating rules used by IM-1 in Example 1. The rules for left outer-join (Rule (1))[2] and aggregate (Rule (2)) are from [23] and [24], respectively. For simplicity, we separate the inserted/deleted part in a TVR delta, and denote them by superscripting Δ with +/−. The blue lines in Fig. 3(a) 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 snapshots/deltas if the merge operator + has an inverse operator −, e.g., $R_{t'} - R_t = \Delta R_t^{t'}$. The red lines in Fig. 3(a) demonstrate the intra-TVR rules used by IM-1 in Example 1. Note that when merging the snapshot/delta of $S \bowtie^{lo} R$, we use $+^{\#}$ (Rule (3)), whereas when merging the snapshot/delta of $\gamma(S \bowtie^{lo} R)$ (query summary), we use $+^{sum}$ ((Rule (4)).

[2] For brevity, padding nulls to match outer join's schema is omitted in Fig. 3(a) and Fig. 3(c). This padding can simply be implemented using a project operator.

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 $\{Q'_1, \cdots, Q'_k\}$, and then apply another query P on their results to get Q, formally described as Eq. 2. The intuition is that $\{Q'_1, \cdots, Q'_k\}$ may be more amenable to incremental computation:

$$Q(R) = P(Q'_1(R), \cdots, Q'_k(R)) \tag{2}$$

Eq. 2 describe 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-2 using the terminology of inter-TVR rules. Formally, $Q = S \bowtie^{lo} R$ is decomposed into Q^P and Q^N:

$$Q^P{}_t = S_t \bowtie R_t, \quad Q^N{}_t = S_t \bowtie^{la} R_t, \quad Q_t = Q^P{}_t +^{\#} Q^N{}_t \tag{3}$$

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. 3 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 γ in *summary* too, just with a different merge operator $+^{sum}$. Fig. 3(b) 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 §4.1, whereas Q^N cannot be easily incrementalized, and is not computed until the completion time.

(2) Outer-join view maintenance: [30] proposed a method to incrementally maintain outer-join views. Its main idea can be summarized using two types of inter-TVR rules:

$$Q_t = {Q^D}_t +^{\#} {Q^I}_t +^{\#} {Q^U}_t, \qquad \Delta {Q^I}_t^{t'} = P(\Delta {Q^D}_t^{t'}, Q_t) \qquad (4)$$

The left rule 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, formally defined using the join-disjunctive normal form of Q. Due to space limitation we refer the readers to [30] for formal details. Intuitively, an insertion (deletion) into the input table will cause insertions (deletions) to Q^D and deletions (insertions) to Q^I, but leave Q^U unaffected. Eq. 4 right describes the second type of rules that directly compute the deltas of Q^I from the delta of Q^D and the previous snapshot of Q. At updates, one can use the TVR-generating rules to compute the delta of Q^D, and the inter-TVR rules in Eq. 4 right to get delta of Q^I; these two deltas can be merged to incrementally compute Q.

Take query sales_status as an example. Fig. 3(c) shows the corresponding inter-TVR rules. As the algorithm in [30] considers updating one input table at a time, we insert a virtual time point t' between t_1 and t_2, assuming R and S are updated separately at t' and t_2. Rule (0) shows the decomposition of sales_status at t' and t_2 following the inter-TVR rule in Eq. 4 left. By applying the TVR-generating rules, Q^D can be incrementally computed as rules (1) and (3); whereas Q^I can be incrementally computed following the inter-TVR rule in Eq. 4 right, as shown in rule (2). Combining them yields the delta of Q as in Table ④ in Fig. 2.

In our technical report [43], we demonstrate more examples of using inter-TVR rules to express complex incremental methods such as the higher-order view maintenance algorithm [3, 35].

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. Fig. 3(d) shows an example plan space by overlaying Fig. 3(a) and 3(b). Any tree rooted at $\gamma(S \bowtie^{lo} R)_{t_2}$ 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 (§5) and selecting an optimal incremental plan (§6).

5 PLAN-SPACE EXPLORATION

In this section we study how Tempura explores 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 model into a Cascades-style optimizer [20, 22], 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. We refer interested readers to our technical report [43] for more details.

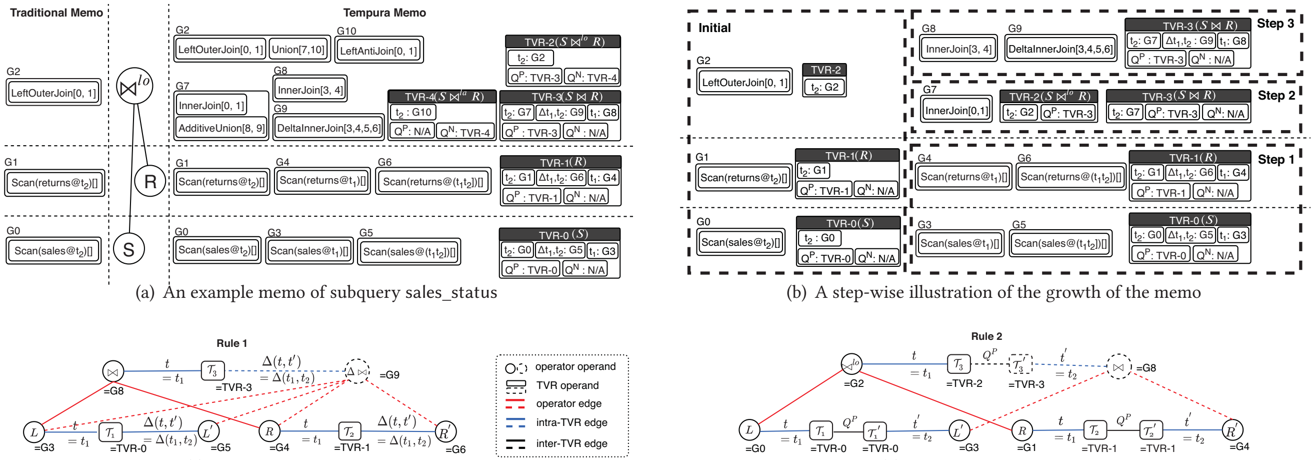
5.1 Memo: Capturing TVR Relationships

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. 4(a) 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 "Tempura Memo" part in Fig. 4(a): (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 full content of S, i.e., S_{t_2}, represented by G0, while the Tempura memo models two more scans: scanning the partial content of S available at t_1 (S_{t_1}), and scanning the delta input of S newly available at t_2 ($\Delta S_{t_1}^{t_2}$), 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 §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 Tempura strictly subsumes a traditional Cascades memo, traditional rewrite rules can be adopted and work without modifications. Besides, the rule engine of Tempura supports TVR rewrite rules. Tempura allows optimizer developers to define TVR



(a) An example memo of subquery sales_status

(b) A step-wise illustration of the growth of the memo

(c) A TVR-generating rule pattern

(d) An inter-TVR rule pattern

Figure 4: Examples of the memo and TVR rewrite-rule patterns in Tempura.

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.

Fig. 4(c)-4(d) depict 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. 4(a). **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*.

Fig. 4(b) 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, rule 2 fires on the tree rooted at *LeftOuterJoin[0,1]* in G2 as in Fig. 4(d). In step 3, rule 1 fires on the tree rooted at *InnerJoin[0,1]* in G7 as in Fig. 4(c). By applying other TVR rewrite rules, we eventually get the memo in Fig. 4(a).

To facilitate fast rule triggering, Tempura indexes the rule patterns by their operands and edges. Whenever changes in the memo are detected, Tempura only fires patterns with operands and edges that potentially match the changes. Tempura does not distinguish TVR rules from traditional rules in terms of rule firing. All rule matches are stored in the same queue, and the firing order is determined by the customizable scoring function. As TVR rules are transformations on logical plans, we adjusted the scoring function to fire TVR rules before physical implementation rules.

5.3 Speeding Up Exploration Process

In this section, we discuss optimizations to speed up the exploration process, which is needed since IQP explores a much bigger plan space than traditional query planning.

Translational symmetry of TVRs. The structures in the Tempura memo usually have translation symmetry along the timeline, because the same rule generates similar patterns when applied on snapshots or deltas of the same set of TVRs. For instance, in Fig. 4(d), if we let $t' = t_1$ instead, L' (R') will match G0 (G1) instead of G3 (G4), and we will 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 respectively.

By leveraging this symmetry, instead of repeatedly firing TVR rewrite rules on snapshots/deltas of the same set of TVRs, we can apply the rules on just one snapshot/delta, and copy the structures to other snapshots/deltas. This helps eliminate the expensive repetitive matching process of the same rule patterns on the memo. The improved process is as follows:

(1) We seed the TVRs of the leaf operators (usually *Scan*) with only one snapshot plus a consecutive delta, and fire all the rewrite rules to populate the memo.
(2) We seed the TVRs leaf operators with all snapshots and deltas, and copy the memo from step 1 by substituting its leaf operators with their snapshot/delta siblings in the corresponding TVRs.
(3) We continue optimizing the copied memo, as we can further apply time-specific optimization, e.g., pruning empty relations if a delta at a specific time is empty.

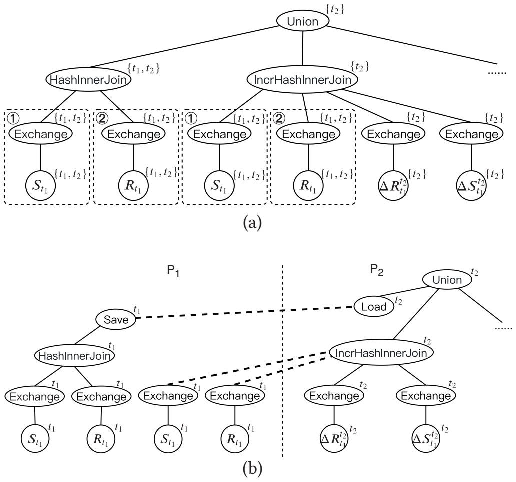


Figure 5: Examples of (a) the temporal plan space, and (b) a temporal assignment for subquery sales_status's plan.

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 [28], 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.

6 SELECTING AN OPTIMAL PLAN

In this section we discuss how Tempura selects an optimal plan in the explored space. The problem differs from existing query optimizers in two ways: (1) costing and searching the plan space need to consider the temporal execution of a plan; and (2) the optimal plan needs to decide which states to materialize to maximize the sharing between different time points within a query.

6.1 Time-Point Annotations of Operators

Costing the plan alternatives is not trivial because the temporal dimension is involved. Fig. 5(a) depicts one physical plan rooted at $(S \bowtie^{lo} R)_{t_2}$, as shown in red in Fig. 3(d). 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. 5(a), 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 times 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 5 (Valid Temporal Assignment). *An assignment of execution times to a physical plan is valid if and only if for each operator O, its execution time $\tau(O)$ satisfies $\tau(O) \in$ t-dom(O) and $\tau(O) \geq \tau(O')$ for all operators O' in the subtree rooted at O.*

Fig. 5(b) demonstrates a valid temporal assignment of the physical plan in Fig. 5(a). 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 times, then the output of I needs to be saved first at $\tau(I)$, and later loaded and fed into O at $\tau(O)$, e.g., *Union* at t_2 and *HashInnerJoin* at t_1. The cost of *Save* and *Load* needs to be properly included in the plan cost. It is worth noting that some operators save and load the output as a by-product, for which we can spare *Save* and *Load*, e.g., *Exchange* of S_{t_1}, R_{t_1} at t_1 for *IncrHashInnerJoin*.

6.2 Time-Point-Based Cost Functions

The cost of an incremental plan is defined under a specific assignment of execution times. 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 §6.3.

As an incremental plan can span across multiple time points, the cost function $\tilde{c}$ in an IQP problem (as in §2.1) is extended to a function taking into consideration of costs at different times. 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 $\tilde{c}$. 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 §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 times. $\tilde{c}_v$ 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 $i, j < i \leq T$.

Consider the plan in Fig. 5(a) 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. 5(b), 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 [22, 32, 38] 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 times, i.e., $\{O\} \times \text{t-dom}(\{O\})$. 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 τ. During optimization, the state-transition function is as Eq. 5. That is, the best cost of O if executed at τ is the best cost of all possible plans of computing O with all possible valid temporal assignments compatible with τ.

$$\tilde{\mathfrak{c}}[O, \tau] = min_{\forall \text{ valid } \tau_j}\Big(\sum_j \tilde{\mathfrak{c}}[I_j, \tau_j] + c_\tau(O)\Big) \qquad (5)$$

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

Theorem 6. *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.*

6.3 Deciding States to Materialize

The problem of deciding the states to materialize can be modeled as finding the sharing opportunities in the plan space. In other words, a shared sub-plan between P_i and P_j in an incremental plan is essentially an intermediate state that can be saved by P_i and reused by P_j. For example, in Fig. 5(a), since both *HashInnerJoin* and *IncrHashInnerJoin* require shuffling S_{t_1} and R_{t_1}, the two relations can be shuffled only once and reused for both joins. The parts ① and② circled in dashed lines depict the sharable sub-plans.

Finding the optimal common sub-plans to share is a multi-query optimization (MQO) problem. In this paper, we extend the MQO algorithm in [27]. The general idea is to greedily enumerate all possible shareable sub-plans, and for each case optimize the best plan given the enumerated sub-plans materialized. The optimum under all cases is the solution to the MQO problem. We refer the readers to our technical report [43] for the details of the algorithm.

7 TEMPURA IN ACTION

In this section, we discuss a few important considerations when applying Tempura in 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. Tempura can be adapted to a dynamic setting using re-optimization. Generally, an incremental plan $\mathbb{P} = [P_1, \cdots, P_{i-1}, P_i, \cdots, P_k]$ for $\vec{T} = [t_1, \cdots, t_{i-1}, t_i, \cdots, t_k]$ is only executed up to t_{i-1}, after which $\vec{T}$ and $\vec{D}$ change to $\vec{T}' = [t_{i'}, \cdots, t_{k'}]$ and $\vec{D}' = [D_{i'}, \cdots, D_{k'}]$. Tempura can adapt to this change by re-optimizing the plan under $\vec{T}'$ and $\vec{D}'$. We want to remark that during re-optimization, Tempura can incorporate the materialized states generated by $P_1, \cdots, P_{i-1}$ as materialized views. In this way Tempura can choose to reuse the materialized states instead of blindly recomputing 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 §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 [48] 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 [10, 21, 47] can be adopted to IQP too. These are out of the scope of this paper.

8 EXPERIMENTS

In this section, we study the effectiveness and efficiency of Tempura. We used the query optimizer of Alibaba Cloud MaxCompute [5], which was built on Apache Calcite 1.17.0 [6], as a traditional optimizer baseline. We implemented Tempura on the optimizer of MaxCompute. We integrated four commonly used incremental methods into Tempura using TVR-rewrite rules: (1) **IM-1** in §2.2, (2) **IM-2** in §2.2 and §4.2, (3) **OJV** the outer-join view maintenance algorithm in §4.2, (4) **HOV** the higher-order view maintenance algorithm in §4.2. By default, Tempura jointly considered all four methods in planning. In the experiments, we used Tempura to 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 §2.1, to demonstrate Tempura. PDW-PD uses the cost function $\tilde{c}_w(O)$ (in §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 [44] ($1TB$) to study the effectiveness (§8.1) and performance (§8.3) of Tempura. To further demonstrate the effectiveness of the plans, in §8.2 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. W-A and W-B have 274 and 554 queries each, among which over 60% have more than 1 join, and over 26% have more than 2 joins.

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.

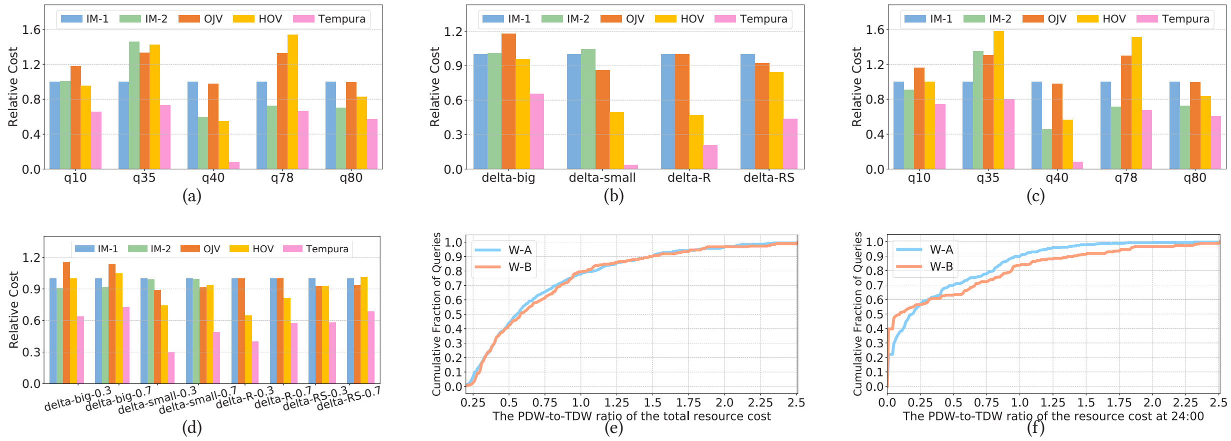


Figure 6: (a)(b) The optimal estimated costs of incremental plans in IVM-PD for different queries and data-arrival patterns. (c)(d) The optimal estimated costs of incremental plans in PDW-PD for different queries, data-arrival patterns and cost weights. (e)(f) The PDW-to-TDW ratio of the real total CPU cost and CPU cost at 24:00 for the data warehouse workloads respectively.

8.1 Effectiveness of IQP

We first evaluated the effectiveness of IQP by comparing Tempura with 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 $r = |D_1|/|D_2|$, and retractions in the input data. We chose four data-arrival patterns: delta-big ($r = 1$), delta-small ($r = 4$), delta-R ($r = 2$ with retractions in the sales table), and delta-RS ($r = 2$ with retractions in both sales and returns 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. The trend of real costs (reported in [43]) 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. 6(a). As shown, different queries prefer different incremental methods. For example, IM-1 outperformed both OJV and HOV for complex queries such as q35. This is because OJV computed Q^I by left-semi joining the delta of Q^D with the previous snapshot (§4.2), and a bigger delta incurred a higher cost of computing Q^I. Whereas for simpler queries such as q80, OJV degenerated to a similar plan as IM-1, and thus had similar costs. Note that HOV costs much less than both OJV and 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 OJV and 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. 6(b). Again, the data-arrival patterns affected the preference of incremental methods. For example, IM-2 could not handle input data with retractions. Compared to delta-big, HOV and OJV started to outperform IM-1 by 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, Tempura consistently delivered the best plans. For q40 in Fig. 6(a) and the delta-small case in Fig. 6(b), Tempura delivered a plan 5-10X better than others. Tempura combines all three of HOV, IM-2 and IM-1 to generate a mixed optimal plan, and thus leveraged all their advantages. E.g., in q40 Tempura used a similar incremental plan to HOV, but Tempura used the IM-2 approach to join the higher-order views M and ΔR, and applied IM-1 to 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 Figures 6(c) and 6(d). We make two remarks. (1) Since PDW-PD did not require any outputs at earlier runs, Tempura automatically avoided unnecessary computation, e.g., IM-2 avoided 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. 6(d), q10 preferred HOV to OJV when $w_1 = 0.3$, 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. Tempura automatically exploited this fact and adjusted the computation in each run, and moved some early computation from the first incremental run to the second.

Conclusion. The optimal incremental plan is affected by many factors and does need to be searched in a cost-based way. Tempura can consistently find better plans than incremental methods alone.

8.2 Case Study: Progressive Data Warehouse

To validate the effectiveness of Tempura in a real application, we conducted a case study of the PDW-PD scenario using two real-world

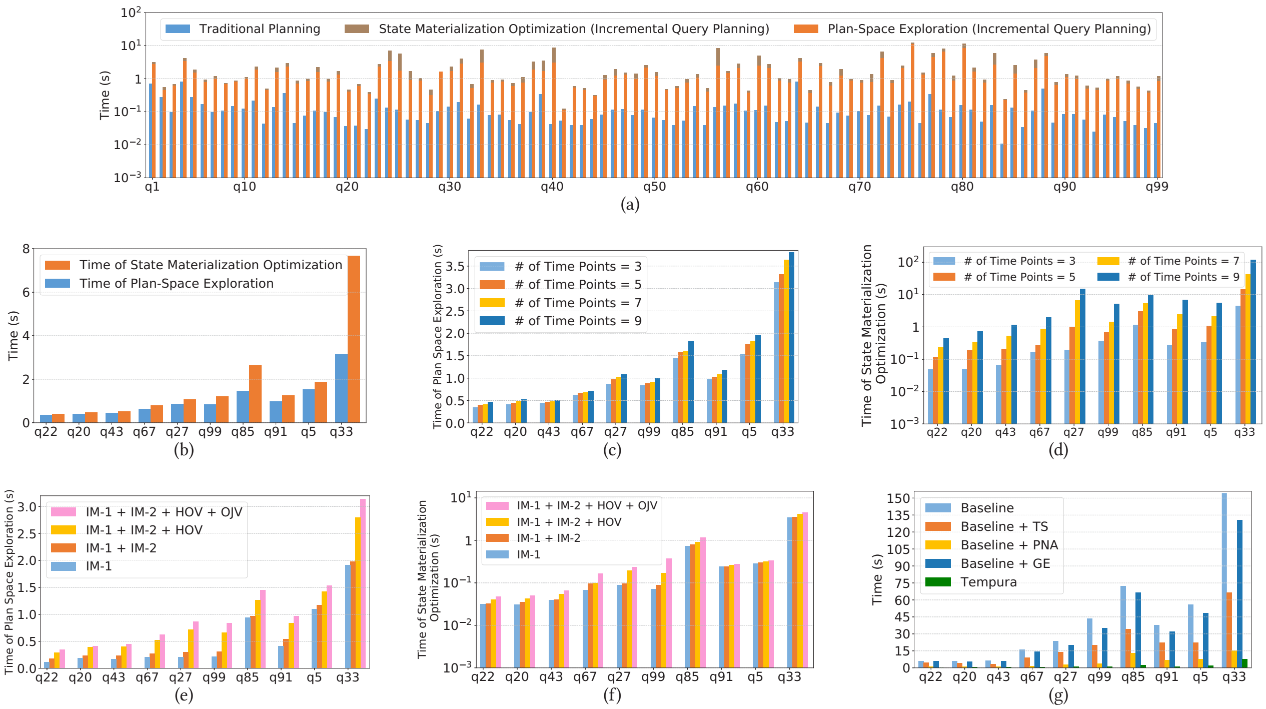


Figure 7: (a) Overall planning performance on the TPC-DS between traditional and incremental query planning. (b) Impact of the query complexity, (c) (d) the size of IQP, and (e)(f) the number of incremental methods on the planning performance. (g) Effectiveness of the speed-up optimization techniques. Note that the selected queries are ordered by their query complexity.

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.

Fig. 6(e) 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 Tempura searched 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. 6(f), 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 refer readers to our technical report [43] for detailed comparison results.

8.3 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 Tempura on the TPC-DS queries. Besides the overall performance study, we also present a detailed study on four aspects: (1) **Query complexity**: How does Tempura perform when queries become increasingly complex, e.g., with more joins or subqueries? (2) **Size of IQP**: How does Tempura perform when $|\vec{T}|$ changes? (3) **Number of incremental methods**: How does Tempura perform when users integrate more incremental methods into it? (4) **Optimization breakdown**: How effective are the speed-up optimizations in §5.3? To study the above four aspects, we selected ten representative TPC-DS queries (Q22, Q20, Q43, Q67, Q27, Q99, Q85, Q91, Q5, Q33) with increasing numbers of joins, aggregates, and subqueries.

Overall Planning Performance. We first studied the overall planning performance by comparing Tempura with traditional planning. Fig. 7(a) shows the end-to-end planning time on all TPC-DS queries. As shown, although planned a much bigger plan space, Tempura still 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. For most queries, the CPU time on planning was 2-3 orders of magnitude smaller than the CPU cost saved by PDW compared to TDW. We report the detailed numbers in [43].

Query Complexity. To study the impact of query complexity, we reported the planning time break-down on the selected TPC-DS queriesin Fig. 7(b). 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. 7(c) and 7(d). As depicted, the time on PSE stayed almost constant as the size of IQP changed. E.g., 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 §5.3. In comparison, the SMO time increased super-linearly with increasing number of incremental runs, due to the time complexity of the MQO algorithm we chose [27].

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`. Fig. 7(f) and 7(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. E.g., 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.

Optimization Breakdown. In the end, we evaluated the effectiveness of the speed-up optimizations discussed in §5.3, i.e., translational symmetry (TS), pruning non-promising alternatives (PNA), and guided exploration (GE). Fig. 7(g) 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.

9 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 [13, 14] and bag [18, 24] semantics, for queries with outer joins [23, 30], and using higher-order maintenance methods [3]. Previous studies mainly focused on delta propagation rules for relational operators. Stream computing [1, 19, 34, 42] adopts incremental processing and sublinear-space algorithms to process updates and deltas. Many approximate query answering studies [2, 9, 17] focused on constructing optimal samples to improve query accuracy. Proactive or trigger-based incremental computation techniques [16, 50] 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. [45] proposed a rate-based cost model for stream processing. The cost model is orthogonal to `Tempura` and 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. [40] 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.

Flink [7] uses Calcite [12] 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, `Tempura` provides 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[8] exploited the relational model to provide strong query semantics for stream processing. Sax et al. [37] introduced the Dual Streaming Model to reason about ordering in stream processing. The key idea behind [8, 37] 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 [11] 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 model for 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`.

10 CONCLUSION

In this paper, we proposed a theory called TIP model to formally model incremental processing in its most general form, and based on it developed a novel principled cost-based optimizer framework `Tempura` for incremental data processing. `Tempura` not 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 `Tempura` in various incremental-query scenarios to show its effectiveness and efficiency.

REFERENCES

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

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

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

[4] Alexander Aiken, Joseph M Hellerstein, and Jennifer Widom. 1995. Static analysis techniques for predicting the behavior of active database rules. *ACM Transactions on Database Systems (TODS)* 20, 1 (1995), 3–41.

[5] Alibaba Cloud MaxCompute [n.d.]. https://www.alibabacloud.com/product/maxcompute.

[6] Apache Calcite [n.d.]. https://calcite.apache.org.

[7] Apache Flink [n.d.]. https://flink.apache.org.

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

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

[10] Shivnath Babu, Pedro Bizarro, and David DeWitt. 2005. Proactive re-optimization. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 107–118.

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

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

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

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

[15] Badrish Chandramouli, Christopher N Bond, Shivnath Babu, and Jun Yang. 2007. Query suspend and resume. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 557–568.

[16] Badrish Chandramouli, Jonathan Goldstein, and Abdul Quamar. 2013. Scalable Progressive Analytics on Big Data in the Cloud. *Proc. VLDB Endow.* 6, 14 (Sept. 2013), 1726–1737. https://doi.org/10.14778/2556549.2556557

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

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

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

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

[21] Goetz Graefe, Wey Guy, Harumi Anne Kuno, and Glenn Paullley. 2012. Robust query processing (dagstuhl seminar 12321). In *Dagstuhl Reports*, Vol. 2. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

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

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

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

[25] Introducing Stream-Stream Joins in Apache Spark 2.3 [n.d.]. https://databricks.com/blog/2018/03/13/introducing-stream-stream-joins-in-apache-spark-2-3.html.

[26] Jianfeng Jia, Chen Li, and Michael J Carey. 2017. Drum: A rhythmic approach to interactive analytics on large data. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 636–645.

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

[28] Christoph Koch. 2010. Incremental query evaluation in a ring of databases. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 87–98.

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

[30] Per-Åke Larson and Jingren Zhou. 2007. Efficient Maintenance of Materialized Outer-Join Views. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*. 56–65. https://doi.org/10.1109/ICDE.2007.367851

[31] Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. 2004. 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* (Toronto, Canada) *(VLDB '04)*. VLDB Endowment, 492–503.

[32] Mavis K Lee. 1988. Implementing an Interpreter for Functional Rules in a Query Optimizer.

[33] David Maier, Jin Li, Peter Tucker, Kristin Tufte, and Vassilis Papadimos. 2005. Semantics of Data Streams and Operators. In *Database Theory - ICDT 2005*, Thomas Eiter and Leonid Libkin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 37–52.

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

[35] Milos Nikolic, Mohammad Dashti, and Christoph Koch. 2016. How to Win a Hot Dog Eating Contest: Distributed Incremental View Maintenance with Batch Updates. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, New York, NY, USA, 511–526. https://doi.org/10.1145/2882903.2915246

[36] Vijayshankar Raman and Joseph M Hellerstein. 2002. Partial results for online query processing. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 275–286.

[37] Matthias J. Sax, Guozhang Wang, Matthias Weidlich, and Johann-Christoph Freytag. 2018. Streams and Tables: Two Sides of the Same Coin. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics* (Rio de Janeiro, Brazil) *(BIRTE '18)*. Association for Computing Machinery, New York, NY, USA, Article 1, 10 pages. https://doi.org/10.1145/3242153.3242155

[38] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. 23–34.

[39] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. 2014. Orca: A Modular Query Optimizer Architecture for Big Data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. ACM, New York, NY, USA, 337–348. https://doi.org/10.1145/2588555.2595637

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

[41] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. 1992. Continuous Queries over Append-Only Databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data* (San Diego, California, USA) *(SIGMOD '92)*. Association for Computing Machinery, New York, NY, USA, 321–330. https://doi.org/10.1145/130283.130333

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

[43] The Tempura Technical Report [n.d.]. http://texera.ics.uci.edu/tempura/tempura-tech-report.pdf.

[44] The TPC-DS Benchmark [n.d.]. http://www.tpc.org/tpcds/.

[45] Stratis D Viglas and Jeffrey F Naughton. 2002. Rate-based query optimization for streaming information sources. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 37–48.

[46] Zuozhi Wang, Kai Zeng, Botong Huang, Wei Chen, Xiaozong Cui, Bo Wang, Ji Liu, Liya Fan, Dachuan Qu, Zhenyu Ho, Tao Guan, Chen Li, and Jingren Zhou. 2020. Grosbeak: A Data Warehouse Supporting Resource-Aware Incremental Computing. In *Proceedings of the 2020 ACM SIGMOD International Conference*

on Management of Data (Portland, Oregon, USA) *(SIGMOD '20)*. ACM, Portland, Oregon, USA.

[47] Florian Wolf, Norman May, Paul R. Willems, and Kai-Uwe Sattler. 2018. On the Calculation of Optimality Ranges for Relational Query Execution Plans. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 663–675. https://doi.org/10.1145/3183713.3183742

[48] Shaoyi Yin, Abdelkader Hameurlain, and Franck Morvan. 2015. Robust query optimization methods with respect to estimation errors: A survey. *ACM Sigmod Record* 44, 3 (2015), 25–36.

[49] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. 2009. Distributed aggregation for data-parallel computing: interfaces and implementations. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 247–260.

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

[51] Yang Zhang, Bret Hull, Hari Balakrishnan, and Samuel Madden. 2007. ICEDB: Intermittently-connected continuous query processing. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 166–175.