Intermittent Query Processing

Dixin Tang
University of Chicago
totemtang@uchicago.edu

Zechao Shang
University of Chicago
zcshang@cs.uchicago.edu

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

Sanjay Krishnan
University of Chicago
skr@cs.uchicago.edu

Michael J. Franklin
University of Chicago
mjfranklin@uchicago.edu

ABSTRACT

Many applications ingest data in an intermittent, yet largely predictable, pattern. Existing systems tend to ignore how data arrives when making decisions about how to update (or refresh) an ongoing query. To address this shortcoming we propose a new query processing paradigm, Intermittent Query Processing (IQP), that bridges query execution and policies, to determine when to update results and how much resources to allocate for ensuring fast query updates. Here, for a query the system provides an initial result that is to be refreshed when policy dictates, such as after a defined number of new records arrive or a time interval elapses. In between intermittent data arrivals, IQP inactivates query execution by selectively releasing some resources occupied in normal execution that will be least helpful (for future refreshes) according to the arrival patterns for new records. We present an IQP prototype based on PostgreSQL that selectively persists the state associated with query operators to allow for fast query updates while constraining resource consumption. Our experiments show that for several application scenarios IQP greatly lowers query processing latency compared to batch systems, and largely reduces memory consumption with comparable latency compared to a state-of-the-art incremental view maintenance technique.

PVLDB Reference Format:
Dixin Tang, Zechao Shang, Aaron J. Elmore, Sanjay Krishnan, Michael J. Franklin. Intermittent Query Processing. *PVLDB*, 12(11): 1427-1441, 2019.
DOI: https://doi.org/10.14778/3342263.3342278

1. INTRODUCTION

Databases maintain standing queries or materialized views over growing datasets without knowledge of where and when new inserts will arrive – thus they make conservative decisions without exploiting knowledge about new data. In many modern applications, it is possible to predict or know the arrival rates and relations with inserts. Examples of these applications include sensor and IoT data, data cleaning systems, admission controlled systems, and extract, transform, and load processes. While classical solutions offer trade-offs between the latency of updating query results for inserts and resource consumption, none of these solutions exploits knowledge about data arrival patterns, which limits opportunities for new planning and execution strategies.

Classical architectures that favor update latency (e.g. continuous query databases [18, 10, 16, 49] or immediate incremental view maintenance (IVM) [23, 4]) tend to keep nearly all state required for processing the queries and update the query results immediately after new data arrives. Not leveraging knowledge about data arrival patterns can result in excess resource consumption from keeping all state in memory and from using CPU cycles for integrating every new record. On the other hand, architectures on the other end of the spectrum favor efficient resource consumption (e.g. batch execution or deferred IVM [23]), and subsequently better query generality, but suffer from high update latency as they tend to discard required state and entirely re-build them for updating the query result – even if the system expects new records in the near future.

This paper considers a system that exploits knowledge of incoming data to accelerate updating the result of a standing query for new data with limited resource consumption. More specifically, we study *querying an incomplete dataset with the remaining data arriving in an intermittent yet predictable way*. Since the missing data arrives discontinuously or at a low rate, the query is not necessarily active all the time (e.g. using deferred refresh), and can release some resources during inactivity (e.g. memory). Due to the predictable arrival patterns of missing data (i.e. the estimated size of new data and distribution of the relations having new data), the query execution engine can leverage this information to optimize query processing, such as injecting new operators or discarding states of existing operators, with limited resource consumption. We find a few applications exhibiting intermittent and predictable workloads [41, 61, 51, 24, 55] when the database is used either to analyze data from external sources or as a component in an analytical pipeline. Here, we describe two representative applications.

- **Late Data Processing:** A user wants to query a dataset that is newly collected from external sources (e.g. sensors). Most data generated for a time interval can be collected under a time threshold, but due to network disconnection or congestion some data arrives late. The remaining data will arrive intermittently at a low rate due to long-tail transfer times of Internet traffic [24]. To predict the arrival pattern of missing data, we build the cumulative distribution functions (CDF) of the arrival time based on historical statistics. With CDFs built, the system can tell the estimated number of data items to arrive for a time window.
- **Data Cleaning:** We consider an analytical pipeline between a data cleaning system and a database. A typical data cleaning process includes two steps: error detection [14, 20, 38] and cleaning [11, 21, 42, 60, 51]. Given a dirty dataset, the data

Proceedings of the VLDB Endowment, Vol. 12, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3342263.3342278

cleaning system splits it into the clean partition, which includes most of the data [51], and the dirty partition. Here, the clean partition can be loaded into a database and is ready for answering queries. Then, the time-consuming cleaning phase is started on the dirty partition, and inserts the cleaned tuples into the database at a low rate. In fact, our experiment in Section 6.3 shows that cleaning 1 GB data can take hours for a state-of-the-art data cleaning system. For this application, the arrival rate of cleaned tuples for each relation is predictable because the data cleaning system provides the database the information of the relations it is cleaning and the estimated cleaned tuple rate.

For applications with predictable and non-continuous data arrival, we put forth a new query processing paradigm, *Intermittent Query Processing* (IQP), to balance latency for maintaining standing queries and resource consumption by exploiting knowledge of data arrival patterns.

To address this challenge, an IQP system integrates three components: a policy component, a query execution engine, and a planner. After a user submits a long-term query and receives an initial query result, the policy component repeatedly schedules the intermittent execution to refresh the query result. Each intermittent execution is defined by a *trigger event* that determines when to update or refresh the query result, the estimated size of new data for each relation, and how many resources are available to prepare for future updates. An event policy can trigger intermittent execution in several ways, such as periodically or by a predefined number of new tuples. After the initial query processing or each intermittent execution, the planner component uses the knowledge of the next trigger event to build a new execution plan for the query execution engine that meets the resource usage constraint. With this new physical plan, the query execution engine makes the query inactive by releasing resources (i.e. memory) to explicitly control the amount of resources used during inactivity. When the query re-activates, the query execution engine uses IVM algorithms to incorporate new data (a *delta* in this paper) to refresh the result. Afterwards, the query execution will either terminate or inactivate if another delta is expected, with the process repeating until termination.

IQP introduces a novel planner that couples policies with query processing engines. The planner builds a query execution plan based on knowledge of trigger events. In this paper, we propose an IQP system, *DISS (Delta-oriented Intermediate State Selection)*, to prototype this planner. DISS generates a specification of a subset of *intermediate states* to persist by the query execution and reuse when processing and incorporating a delta into the prior result. Examples of such state include hash tables for joins and aggregations, as well as materialized relational operators. DISS addresses the key challenge of how to selectively keep the optimal subset of intermediate states according to intermittent delta prediction to minimize query refresh latency while meeting a memory budget.

The major contributions of this paper include:

- We propose intermittent query processing (IQP) to efficiently support querying an incomplete dataset with predictable and intermittent arrival patterns by exploiting information of trigger events.
- We design a prototype IQP system that can select intermediate states to keep in memory to minimize delta processing time with constrained memory consumption.
- We implement DISS on top of PostgreSQL 10 and perform extensive experiments to evaluate its efficiency. Compared with batch processing and an incremental view maintenance system, we have remarkable performance improvements and significantly lower memory usage.

The rest of this paper is organized as follows: Section 2 provides an overview of DISS. We discuss related work in Section 3. We present our intermediate states selection algorithm, and its extensions and system optimizations in Section 4 and Section 5 respectively. Finally, we discuss the prototype implementation and its evaluation in Section 6, and Section 7 concludes this paper.

2. DISS OVERVIEW

In this section we discuss major components of DISS and a query life cycle, as shown in Figure 1. The key component for DISS is the dynamic programming (DP) algorithm of the planner that runs between the policy component and query execution engine to select intermediate states for processing deltas with a memory budget. This algorithm has a linear running time with respect to the number of intermediate states and can inject new operators into the plan. Specifically, our algorithm considers three types of intermediate states: i) data structures along with intermediate tuples that are maintained by blocking operators, with state that is materialized during query processing; ii) intermediate tuples generated by each operator but not materialized (i.e. pipelining); iii) data structures that are not generated but may help upcoming delta processing (e.g. additional hash tables for symmetric hash join).

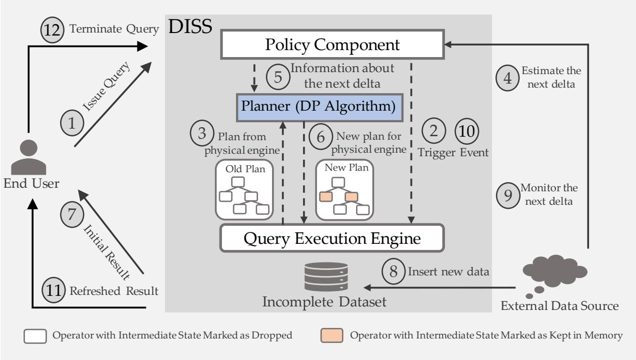


Figure 1: IQP Prototype Overview

DISS initially uses *batch processing* to execute a query over an incomplete dataset with majority of the expected data present, and uses *delta processing* to incorporate one or more data deltas into the prior query result. Figure 1 shows an overview of a query life cycle. A user first issues a query to DISS ①, where the policy component triggers one query execution over an incomplete dataset ②. The query is compiled and generated into a query plan as a tree of operators. Before executing the query, the planner uses our core DP algorithm to determine the intermediate states that should be kept subject to a memory budget. It first extracts the query plan from query execution engine ③, and then obtains the information from the policy component ⑤ according to the delta prediction model ④. The DP algorithm marks a subset of intermediate states of the query plan for the execution engine to keep ⑥. The query engine, based on canonical IVM algorithms [13], executes the plan and returns an initial query result to the end user ⑦. After that, we persist the intermediate states that are marked as kept in the query plan and drop the rest. When new data is added to the database ⑧, the policy component monitors the new data ⑨ and creates a trigger event based on a defined policy ⑩. If another delta is expected, DISS repeats the DP algorithm and generates a new plan; otherwise, we use the same plan. DISS then runs this plan to return a refreshed result to the user ⑪. This process repeats either the dataset is complete or the user terminates the query ⑫.

3. RELATED WORK

We discuss related research on view maintenance, intermediate state reuse, query suspension, and continuous queries. We emphasize that none of these projects consider leveraging intermittent and predictable delta arrival patterns to optimize query processing.

Incremental View Maintenance: The problem of updating a query result is akin to incremental view maintenance (IVM), which considers a materialized view that is derived from base realtions. When base relations are updated, IVM finds a query plan to compute the changes to the old view rather than recompute the view. We focus only on relational algebra and do not discuss other incremental solutions, such as matrix calculations [47, 48], or user defined functions (UDFs) [12]. We refer the reader to a comprehensive survey on materialized views [19]. Larson et al. introduce the idea of IVM and propose algorithms for select-project-join (SPJ) views [13]. Later work introduces methods to support materialized views with negation, aggregation, and recursive view definitions [30], efficient IVM for nested queries [62, 46], optimized incremental computation for acyclic joins [33], and how to exploit ids in base relations [36]. We believe these algorithms are applicable to IQP, as our work aims to bridge between the policy component and query execution engines instead of designing IVM algorithms.

In addition to IVM algorithms, prior work also considers building additional auxiliary views for a materialized view to accelerate view refreshing. Ross et al. selectively build additional views according to the workload [52]. Its selection algorithm enumerates all possible subsets of additional views, which has the complexity of being exponential to the number of additional views. DBToaster improves on this idea and supports fast immediate incremental view maintenance by building additional recursive higher-order views [4]. IQP differs from DBToaster in that it considers information of trigger events and selectively keeps or builds intermediate states according to the future deltas.

Multiple view maintenance policies, such as immediate or deferred maintenance, have been proposed based on different consistency requirements and performance objectives [22, 23]. Lazy view maintenance [65] takes a similar approach to a deferred view [22] by deferring the view maintenance work to the time when the system has free cycles or when it is queried such that the overall performance of maintaining and querying the view can be improved. These projects decouple the policy component from the underlying query execution engine, but do not consider leveraging information from a policy component to optimize query processing.

Materialized View Selection and Reuse: Building materialized views can accelerate query processing but with additional cost. Several efforts exploit this trade-off in data warehouses [3, 31, 32, 37, 53]. Dynamic materialized views [66, 27] maintain partial views according to hot/cold access patterns to answer parameterized queries and reduce maintenance cost. In distributed systems, pre-computation can achieve linear scalability [7] and selectively materializing sub-expressions can minimize query response time at "data center" scale [35]. Chaudhuri et al. incorporate materialized views into query optimization [17] and Mistry et al. share materialized views for multi-query optimization [43].

A related topic to materialized view selection is reusing intermediate states. Several projects explore caching intermediate states based on its reuse frequency, performance contribution, and its cost (i.e. memory size) [34, 45]. Dursun et al. consider reusing intermediate data structures from join algorithms in main-memory databases [25]. ReCache studies the same problem for heterogeneous data sources [8]. Intermediate results can also accelerate approximate query processing [26] and feature selection workloads [63]. IQP considers how to efficiently incorporate delta into an existing query result, rather than storing materialized views or intermediate states for future queries. In Section 6.5, we compare IQP with a view cache algorithm [45] with a memory constraint, and show that IQP's ability to include the delta arrival pattern in its decision making process improves query processing performance.

Query Suspend and Resume: Several previous projects study the problem of suspending query execution due to system failures or query scheduling, and then efficiently resuming the query later. Chandramouli et al. [15] design lightweight asynchronous checkpointing to store the states of operators during suspension phase, and resume the query by restoring the consistent states of operators. Later work studies the same problem in the context of index construction [28, 6]. Query suspend and resume is not viable for IQP, as a suspended query does not necessarily finish processing the desired partial workload and cannot present the corresponding incomplete query result to end users.

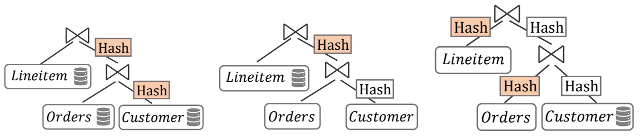
CQ Processing and Stream Computing: Many research projects explore continuous query processing and stream computing to address the problems of late data and bursty workloads. To efficiently process late data, some projects investigate buffering all intermediate states [50, 64] . Similar to these projects, modern dataflow systems [5, 2, 44] keep intermediate states in memory for late data processing, and further allow users to specify a time threshold of how late the data is expected to be and discard data later than the threshold [5]. Krishnamurthy et al. allow processing arbitrarily late data, but limited support for aggregation operations [39]. Several projects address bursty workloads by selectively discarding input data to maximize the query result utility for end users [57, 9, 56], but do not consider incorporating the discarded data into query results later. IQP can efficiently process late data under a constrained resource by intelligently persisting a subset of intermediate states.

4. DELTA-ORIENTED INTERMEDIATE STATE SELECTION

In this section, we introduce the intermediate states selection algorithm for DISS. Materializing intermediate states and auxiliary data structures speeds up delta processing, but comes with the cost of higher memory consumption and longer initial batch processing time. To strike a balance between batch processing and delta processing, we carefully persist a subset of intermediate states and build optional auxiliary data structures when necessary. This is enabled by our dynamic programming algorithm that considers the cost of batch processing and delta processing together based on the predicted information about the next delta. Our algorithm currently does not consider using a different join order from the one generated by the database query optimizer: for this work we assume that majority of the data for relations exists for the initial query and we use the plan that is optimized for the initial data. Thus, we leave adaptive query execution for future work. Since the applications we have discussed so far are insert-only workloads, our algorithm discussion in this section only considers insert-only deltas, and we discuss how to process deletes and updates in Section 5. In this section, we propose our dynamic programming-based optimization algorithm that handles one delta at a time, and discuss the case of processing multiple deltas in Section 5. We begin with a motivation in Section 4.1, present an overview of DISS in Section 4.2, and elaborate on the algorithm in Section 4.3.

4.1 Motivation

We propose an IQP system *DISS (Delta-oriented Intermediate State Selection)* that considers using a limited memory budget to



(a) Original query plan **(b)** Delta from Lineitem **(c)** Delta from Customer

Figure 2: Examples of Intermediate State Selection

store a subset of intermediate states for efficient delta processing. Intermediate states are critical to the performance of delta processing and a major source of memory consumption. Consider a simple example query shown in Figure 2a: $Lineitem \bowtie (Orders \bowtie Customer)$ implemented using hash joins. During the batch processing, the hash table is built for the right sub-tree, and the left sub-tree probes the hash table. If the delta only includes data for $Lineitem$ (i.e. Figure 2b), keeping the top hash table (colored in Figure 2b) is enough to process this delta efficiently without recomputing $Orders \bowtie Customer$, and we can discard the other hash table. However, if the delta only comes from $Customer$, these two hash tables cannot help delta processing as we need to re-scan $Lineitem$ and $Orders$. This motivates us to consider building new intermediate states for delta processing. Figure 2c shows a possible solution using symmetric hash joins [58] (if we know delta only includes data for $Customer$). During the batch processing, we build two new hash tables for $Lineitem$ and $Orders$. After that, we discard two hash tables (not colored in Figure 2c) and keep the other two (colored in Figure 2c) assuming the memory budget permits. Building new intermediate states comes with additional cost. DISS provides a holistic solution to choosing which intermediate states to keep, and if necessary where to build new states.

4.2 DISS Overview

In this subsection, we give an overview of DISS. It takes several inputs: a query plan (e.g. tree of relational operators) $\mathbb{T}$, meta-information of all intermediate states, a cardinality estimator, and an operator cost estimator (from the conventional RDBMS that executed batch processing), a memory budget $\mathbb{M}$, and prediction of the next delta (i.e. the numbers of new tuples for each base relation). Note that we currently use a static memory budget $\mathbb{M}$ set by a user, and leave dynamic memory budget allocation for future work. Similar to classical query optimization, we run the optimization algorithm as if the cardinality estimates are accurate and the predicted delta position/sizes are precise; we later evaluate the impact of prediction quality on IQP's performance in Section 6.4. With this information, DISS solves the problem of selecting a subset of intermediate states to persist, where the sum of their sizes is within the budget $\mathbb{M}$, such that the summation of delta processing time and the overhead of materializing new operators (based on the estimator) is minimized. The query is compiled into a tree of operators and each operator may include intermediate states. Using DISS to select the optimal set of intermediate states includes four steps.

In the first step, DISS obtains the prediction about the next delta, including which base relation(s) it belongs to and the number of new tuples. Then, DISS propagates the delta information from base relations to the top operator such that each operator knows the cardinalities of (delta) tuples from its child subtrees when the delta will be processed. DISS reuses the cardinality estimator from the underlying RDBMS (that handles batch processing).

After, for each operator DISS estimates the operator's query processing time using the RDBMS' *cost estimator*. We may apply one of several actions on each operator. For example, a join operator may only contain a hash table for the right child. Thus, there are at least four state configurations on the hash table: drop it; keep it; drop it and build a left one; keep it and build a left one. We need to choose exactly one action for each operator, and each action incurs a different time cost (for processing deltas) and memory cost.

Table 1: Notation Table

Notations	Meaning
$\lvert R\rvert$	size for relation R (also costs for read/write R)
$\mathbb{M}$	memory budget for IQP
$\mathbb{T}$	query plan tree
op	an operator (vertex) in $\mathbb{T}$
$op.c$	unary operator *op*'s (single) child
$op.l$, $op.r$	binary operator *op*'s two children
$op\Downarrow$	a subtree of plan $\mathbb{T}$: *op* and its descendants
D	the dataset (for batch processing)
ΔD	the new dataset (for delta processing)
$D+$	$D \cup \Delta D$
Q	a query (explicitly given or induced by $op\Downarrow$)
$R_Q(D)$, R	$Q(D)$
$R_Q(\Delta D)$, ΔR	$Q(D+) - Q(D)$ (**not** $Q(\Delta D)$)
$R_Q(D+)$, $R+$	$Q(D+)$
$\mathcal{D}^{op}(m)$	min. cost for $op\Downarrow$ to emit ΔR (c.f. Sec. 4.3.2)
$\mathcal{F}^{op}(m)$	min. cost for $op\Downarrow$ to emit $R+$ (c.f. Sec. 4.3.2)
$C_{op}(d)$, $C(d)$	cost for processing data d ($d = D$, ΔD, or $D+$) (estimated by RDBMS' cost estimator)

Finally, the cost information and the query operator tree are used by our core dynamic programming algorithm to decide which intermediate states to keep (if they will be built by the batch processing) or build (if batch processing does not build them). As in conventional RDBMS query optimization, we use a normalized processing time that combined both the main-memory processing time and I/O time into a unified metric.

4.3 DISS Algorithm

Here, we discuss how to choose a subset of intermediate states to keep, and build new intermediate states if necessary based on one predicted delta, a memory budget $\mathbb{M}$, and a query plan tree $\mathbb{T}$ generated by the query optimizer. Without loss of generality, each tree vertex is a relational operator *op* which has one child $op.c$ or two children $op.l$, $op.r$. For each operator *op*, we also consider it as a query, which is made of *op* and its descendants. For simplicity, we use $op(D)$ to denote the evaluation of *op* and its descendants on dataset D. We summarize our notations in Table 1.

4.3.1 Pre-processing

Before working on the problem of selecting the optimal subset of intermediate state to build or keep for a future delta, the system is ready to process the query on the existing set of data (batch processing). Thus, for each operator *op*, we know the (estimated) cardinality of its output. We also know the cardinality of each base relation's delta (from the delta predictor). For pre-processing, we propagate the delta cardinality information to each operator, so we know each operator's input(s)' sizes, which will be used in the next step. We delegate the delta cardinality estimation to the RDBMS' cardinality estimator.

4.3.2 Problem Definitions

Generally, for each query Q, dataset D, and a delta dataset ΔD, there are two ways to compute the query result on the union of D and ΔD: **re-computation** $Q(D+)$ where $D+ = D \cup \Delta D$, and **incremental computation**, which finds a query $Q_{incr}(D, \Delta D)$ such that $Q(D+) = Q(D) \oplus Q_{incr}(D, \Delta D)$. For performance, ideally

Algorithm 1: Memoization-based Dynamic Programming

```
Parameters : operator op, memory budget m
if $\mathcal{D}^{op}(m)$ and $\mathcal{F}^{op}(m)$ have been processed then
    return the result from memoization
for op's all available action act do
    $\mathcal{D}^{op}(m) = \mathcal{F}^{op}(m) = \infty$
    if act is applicable under current setting then
        $\mathcal{D}^{op}(m) = \min(\mathcal{D}^{op}(m), \text{cost according to } act)$
        $\mathcal{F}^{op}(m) = \min(\mathcal{F}^{op}(m), \text{cost according to } act)$
end
memoize $\mathcal{D}^{op}(m)$ and $\mathcal{F}^{op}(m)$
```

incremental computation is faster than re-computation. However, incremental mechanisms do not always accelerate computation due to two possible reasons.

First, in some cases fully supporting incremental computation requires persisting intermediate states. It comes with extra cost and can make the incremental approach less efficient compared to re-computation. Consider joining two sub-trees L and R using a simple hash join, which builds one hash table for one of its two sub-trees (assuming R) and uses the result pulled from the other sub-tree (i.e. L) to probe the hash table. To enable full incremental computation for deltas from L and R, the hash join operator needs to persist hash tables for both sub-trees and the output result of this hash join. The extra overhead of building one more hash table (i.e. for L) and materializing the output result might be larger than the cost of re-computation, which includes re-scanning from the sub-tree L and performing the join with the hash table of R.

Second, as discussed previously, since the deltas arrive in an intermittent way, the system may not have sufficient memory to keep all intermediate states for all concurrent standing queries that are waiting intermittent deltas. Thus, we have to drop some intermediate states, and incremental computation may be slower than the re-computation due to lack of necessary intermediate states.

Therefore, since incremental computation is not always the faster choice for all operators, each operator needs to choose what kind of input it needs from its child operators depending on that this operator chooses re-computation or not. Informally speaking, it may only need to see the "new" input (generated due to the arrival of delta) or ingest the "full" input (a combination of the "new" input and previous inputs). Thus, for each operator *op*, we consider the costs of two output requirements. One is how efficient can *op* output a delta output (defined as $R_{op}(\Delta D) = op(D+) - op(D)$); the other is how efficient can *op* output a full output (defined as $R_{op}(D+) = op(D \cup \Delta D)$). We emphasize that, depending on the actual costs, a full output can be computed incrementally and a delta output can be computed by re-execution as well.

Based on this observation, we define two cost functions. Assume the memory budget for operator $op\Downarrow$ (*op* and its descendants) is m ($0 \leq m \leq \mathbb{M}$). $\mathcal{D}^{op}(m)$ is the minimum cost for $op\Downarrow$ to emit delta output $R_{op}(\Delta D)$. Similarly, $\mathcal{F}^{op}(m)$ is the minimum cost for operator $op\Downarrow$ to emit full output $R_{op}(D+)$.

4.3.3 Recursive Dynamic Programming

We introduce a *memoization-based* top-down dynamic programming algorithm. Assume the operation $root$ is the root of query plan $\mathbb{T}$, the better (smaller) solution of $D^{root}(\mathbb{M})$ and $F^{root}(\mathbb{M})$ is the solution of the intermediate state selection problem. For any operator *op* and a budget m, to compute $\mathcal{D}^{op}(m)$ (or $\mathcal{F}^{op}(m)$), we need to recursively calculate $D^{op'}(m')$ and/or $F^{op'}(m')$ for *op*'s descendant op' and a budget $m' \leq m$. Throughout the recursive computation process, we *memoize* all results for $\mathcal{D}^{op}(\cdot)$ and $\mathcal{F}^{op}(\cdot)$, so we can reuse the existing results if we need them later. This top-down memoization process is equivalent to a bottom-up dynamic programming. We present the former for better clarity, and analyze the complexity of our algorithm later. We emphasize that our algorithm is different from classical query optimization dynamic programming algorithms [54] in that we consider the cost of batch processing and delta processing together with an memory constraint rather than just the batch processing time.

In our recursive algorithm, we focus on one operator at one time. Each operator has several **action templates** (*actions* for short). For an operator *op*, each action corresponds to one configuration of *op*'s intermediate states and/or auxiliary data structures. For example, for a *sort* operator one action is to keep the sorted result, and another is to drop the sorted result. Each action includes a constraint indicating when this action is applicable based on a memory budget and is associated with two formulas $\mathcal{D}^{op}(\cdot)$ and $\mathcal{F}^{op}(\cdot)$, which represent the cost of computing the delta output ΔD and full output $D+$ respectively. An example of action is illustrated in Action 1.

We assume a pipelined query execution engine, and consider injecting new operators (e.g. Materialize) or building new data structures (e.g. hash table) for fast delta processing if necessary. DISS currently supports the following operators:

- Scan including sequential scan, and index scan
- Materialize
- Sort
- Join[1] including hash, sort-merge, and nested loop join
- Aggregate including hash aggregate and sort aggregate

We briefly introduce these operators and the corresponding actions. It is straightforward to extend our DISS solution to support more operators or more actions. We illustrate the framework of our dynamic programming solution in Algorithm 1, and discuss each action as follows. For simplicity, the following discussion considers the first delta after the initial batch processing. In this case, the DP algorithm generates a specification of which intermediate states to materialize. According to this specification, the query plan in the batch phase is modified to materialize or build new intermediate states that do not exist in the original plan. After batch processing, a delta plan is generated by keeping and discarding corresponding intermediate states based on the specification. Processing successive deltas is similar.

Scan (including Projection and Selection): Scan is a leaf operator, it performs projection and predicate filtering for tuples scanned from base relations. Since we assume a pipelined execution engine, a scan operator does not maintain any intermediate state. To avoid re-scanning the base relations during delta processing, we can inject a materialize operator as its parent.

Materialize: A materialize operator *op* can be inserted as the parent of an operator to materialize their output tuples, which will be used for future delta processing. There are two actions: no operation (i.e. do not materialize, Action 1), and materialization (Action 2). If we do not materialize these tuples in the batch processing, the cost of evaluating *op* over either delta data ΔD or full data $D+$ (i.e. $\mathcal{D}^{op}(m)$ or $\mathcal{F}^{op}(m)$ in Action 1) includes the cost of pulling the corresponding result from its child (i.e. $\mathcal{D}^{op.c}(m)$ or $\mathcal{F}^{op.c}(m)$) and the cost of delivering them to its parent operator $C(\cdot)$. If the memory budget m is sufficient to keep R (the query result of *op*) in the batch processing, we can choose to keep it for

[1] Our current design only considers inner joins.

Action 1: No Materialization

Operator: **Materialize**
Applicable: Always
Cost: $\mathcal{D}^{op}(m) = C(\Delta D) + \mathcal{D}^{op.c}(m)$
$\mathcal{F}^{op}(m) = C(D+) + \mathcal{F}^{op.c}(m)$

Action 2: Keep Materialization

Operator: **Materialize**
Applicable: $|R| \leq m$
$\mathcal{D}^{op}(m) = C_{mat}(R) + C(\Delta D) + \mathcal{D}^{op.c}(m - |R|)$
Cost: $\mathcal{F}^{op}(m) = C_{mat}(R) + C(\Delta D) + \mathcal{D}^{op.c}(m - |R|)$
$+C_{scan}(R)$

more efficient delta processing. In this case, we need to pay an additional cost of materializing R (i.e. $C_{mat}(R)$ in Action 2). Here, if its parent operator asks for delta result ΔR, the time $\mathcal{D}^{op}(m)$ in Action 2 includes the materialization time $C_{mat}(R)$, the time of pulling delta result from its child $\mathcal{D}^{op.c}(m - |R|)$, and delta processing $C(\Delta D)$ time. If the upper layer operator asks for a full re-evaluation $R+$, the time $\mathcal{F}^{op}(m)$ in Action 2 for emitting full result $R+$ needs to additionally account for the time of scanning the materialized result $C_{scan}(R)$.

We note that a materialize operator only applies to child operators when a new delta can be merged with the previous output straightforwardly without additional effort, that is, the output only requires bag semantics. For child operators that require richer semantics, such as a sorted output, this materialize action does not apply, and a specialized materialize action is required (i.e. Action 3).

Action 3: Keep Sort

Operator: **Sort**
Applicable: $|R| \leq m$
$\mathcal{D}^{op}(m) = C(\Delta D) + \mathcal{D}^{op.c}(m - |R|)$
Cost: $\mathcal{F}^{op}(m) = C(\Delta D) + \mathcal{D}^{op.c}(m - |R|)$
$+C_{merge}(\Delta R, R)$

Sort: A sort operator outputs sorted tuples. The drop action of a sort operator is similar to Action 1, where the $C(\cdot)$ represents the time of sorting data pulled from its child operator and delivering them to its parent operator. We omit the drop action for space. If the memory budget m is sufficient to keep the sorted intermediate result R, we can apply keep action shown in Action 3. Here, keeping the intermediate result does not introduce additional time cost because the sort operator is part of the original batch processing. Therefore, the cost of emitting the delta result ΔR for a sort operator (i.e. $\mathcal{D}^{op}(m)$ in Action 3) includes the cost of pulling delta result from its child operator ($\mathcal{D}^{op.c}(m - |R|)$), and the cost of computing the delta result $C(\Delta D)$. If the operator is expected to output the full result $R+$, the keep action needs to account for the cost of merging of sorted result ΔR and the sorted (i.e. R $C_{merge}(\Delta R, R)$).

Action 4: Drop Aggregation

Operator: **Aggregate**
Applicable: Always
Cost: $\mathcal{D}^{op}(m) = \mathcal{F}^{op}(m) = C(D+) + \mathcal{F}^{op.c}(m)$

Action 5: Keep Aggregation

Operator: **Aggregate**
Applicable: $|R| \leq m$
Cost: $\mathcal{D}^{op}(m) = C(\Delta D) + \mathcal{D}^{op.c}(m - |R|)$
$\mathcal{F}^{op}(m) = C(\Delta D) + C_{scan}(R) + \mathcal{D}^{op.c}(m - |R|)$

Aggregate: Before introducing DISS for an aggregate operator, we note that the aggregate operator is not a *monotonic* operator, even if the aggregate function itself is mathematically monotonic. Informally speaking, if an operator is monotonic, one delta (i.e. new tuples) only generates zero or more extra output tuples. However, an aggregate operator may introduce extra tuples, and remove existing ones: assume the `SUM`-aggregated result contains a tuple `('Tom', 15)`, so a delta `('Tom', 3)` turns the previous tuple into `('Tom', 18)`. In this paper, we define the delta result ΔR contains these two tuples with appropriate annotations.

An aggregate operator can be implemented in hash-based or sort-based approaches. For the former, a hash table is built with group-by ID as the key and aggregated value as the value. For each tuple from an aggregate operator's child's output, a hash aggregate operator identifies its group-by ID and incorporates that tuple into the aggregated value. A sort aggregate operator assumes tuples from the child operator are already sorted by the group-by ID. It scans the tuples and aggregate numerical values that share the same group-by ID. We use hash-based aggregate as an example, while our two actions apply on both aggregate methods.

If we discard the intermediate state, regardless whether the operator is supposed to output a delta output ΔR or a whole output $R+$, the aggregate operation has to redo the whole aggregate process to generate the positive tuples (that are new due to deltas) and the negative tuples (that shall be removed due to deltas). Thus, the cost of computing the delta and full output (i.e. $\mathcal{D}^{op}(m)$ and $\mathcal{F}^{op}(m)$ in Action 4) equals the cost of pulling the full output from descendant operators $\mathcal{F}^{op.c}(m)$ and redoing the aggregate $C(D+)$.

If we keep the intermediate state (the hash table), for each new tuple, we use its key to look up in the hash table. Based on the existing aggregated tuple and the new tuple's value, we can calculate the new aggregated value. Thus, the cost of generating the delta output of the aggregate operator ($\mathcal{D}^{op}(m)$ in Action 5) includes the cost of processing the delta input $C(\Delta D)$, and the cost incurred by the descendant operators $\mathcal{D}^{op.c}(m - |R|)$. Similarly, if we aim at the whole output ($\mathcal{F}^{op}(m)$ in Action 5), we only need to merge new tuples into the hash table and scan the whole table (i.e. $C_{scan}(R)$).

All the above discussions about aggregate functions assume the aggregate function f is "incrementable": in order to compute $X = f(a_1, a_2, \ldots, a_n)$, we can find two functions g and h such that $X = h(g(a_1, a_2, \ldots, a_{n-1}), a_n)$. Most of SQL's standard aggregate functions have this nice property: when f is `MIN`, g and h is `MIN` as well; when f is `STDDEV`, g and h are not `STDDEV`, but some simple arithmetic functions (sum and sum of squares). However, if the aggregate function f is a user-defined function (UDF), it is not trivial, or even impossible to find the corresponding g and h, or g and h are not efficient. Therefore, to process the UDF-aggregate the default action is redo, unless the user hints otherwise.

Hash Join: DISS supports hash join[2], nested-loop join, and sort-merge join. We only discuss hash join here as nested-loop and sort-merge join operators do not persist intermediate states, but let their child operators do this job (i.e. sort operators for a sort-merge join and materialize operators for nested-loop join). In the following discussion, for a join operator *op* we use subscripts *l* and *r* to denote its left and right child operators, as well as other values associated with two sub-trees. For example, *op.l* is the *op*'s left child, D_l is the data associated with the left sub-relation, query result of left subtree R_l is *op.l*(D_l). For this discussion, $C_B(D)$ is the estimated cost of building hash table for dataset D, $C_I(\Delta D, D)$ represents the estimated cost of inserting the result of ΔD into the hash table for D, and $C_{HJ}(D_L, D_R)$ denotes the estimated cost of scanning tuples from D_L and probing them to the hash table for D_R.

[2] We currently only support simple hash-join as we only persist intermediate state in memory, but nothing in our approach limits supporting other hash join algorithms.

Action 6: Keep Right Hash Table Only

Operator: **Hash Join**

Applicable: $|R_r| \leq m$ ($|R_r|$ is right hash table's size)

Cost:

$$\mathcal{D}^{op}(m) = \min_{\substack{0 \leq m_l \leq m-|R_r| \\ m_l+m_r=m-|R_r|}} \underbrace{\mathcal{F}^l(m_l) + \mathcal{D}^r(m_r)}_{\text{pull from both sub-trees}} + \underbrace{C_B(\Delta D_r)}_{\text{hash table for } \Delta D_r} + \underbrace{C_{HJ}(\Delta D_l, D_r \cup \Delta D_r)}_{\text{left delta}} + \underbrace{C_{HJ}(D_l, \Delta D_r)}_{\text{right delta}}$$

$$\mathcal{F}^{op}(m) = \min_{\substack{0 \leq m_l \leq m-|R_r| \\ m_l+m_r=m-|R_r|}} \underbrace{\mathcal{F}^l(m_l) + \mathcal{D}^r(m_r)}_{\text{pull from both sub-trees}} + \underbrace{C_I(\Delta D_r, D_r)}_{\text{insert } \Delta D_r \text{ into right hash table}} + \underbrace{C_{HJ}(D_l \cup \Delta D_l, D_r \cup \Delta D_r)}_{\text{full hash join}}$$

For a hash join operation, a hash table is built on one of two joined sub-trees' keys. After the hash table is built, the hash join iterates through the tuples from the other sub-tree and probes the hash table based on join keys. Without loss of generality, we assume the hash table is always built for the **right** side. Here, it is easy to incrementally process deltas from the left side (given the right hash table is built), but processing deltas from the right side requires recomputing the full result from the left sub-tree. To address this issue, our algorithm additionally considers building left hash table if necessary (also known as symmetric hash join [59]). Therefore, we discuss three actions: keep the right hash table only, keeping the right hash table and building a left one, and drop both. Other possible actions, including building the left hash table and dropping the right one, can be handled in a similar approach. Throughout the discussion, we assume that both sides have deltas, and other cases (e.g. only left side has delta) can be easily derived from this one.

We begin with discussing the case of keeping the right hash table only. The cost of computing the delta result and full result is shown in Action 6. The cost of computing the delta result (i.e. $\mathcal{D}^{op}(m)$) includes four parts:

- Pulling full output from the left sub-tree and delta output from the right sub-tree. To find the minimum cost, we need to enumerate all possible memory allocation of the remaining memory budget $m - |R_r|$ into two sub-trees (i.e. m_l and m_r). The cost for this part is $\mathcal{F}^l(m_l) + \mathcal{D}^r(m_r)$.
- Afterwards, we build a hash table for the right delta ΔD_r, which is used to process the data pulled from the left sub-tree. The cost for building this hash table is $C_B(\Delta D_r)$.
- Next, we begin the join process for the left delta ΔD_l. It joins with $D_r \cup \Delta D_r$ using the right hash table we have kept and the newly built hash table in the last step. The cost here is $C_{HJ}(\Delta D_l, D_r \cup \Delta D_r)$.
- Finally, the right delta ΔD_r joins with D_l by scanning D_l and probes the hash table of ΔD_r. The cost of this part is $C_{HJ}(D_l, \Delta D_r)$.

The cost of computing the full result ($\mathcal{F}^{op}(m)$ in Action 6) is similar to $\mathcal{D}^{op}(m)$. We first insert the right delta into the right hash table (i.e. $C_I(\Delta D_r, D_r)$). Then we join the full result pulled from the left sub-tree with this hash table via the hash join.

Next, we discuss the case of building a left hash table and keeping the right table at the same time. Since the left hash table is not originally built in the batch processing, building it costs $C_B(D_l)$ for either computing the delta output or full output. If the operator needs to compute the delta output, the right delta ΔD_r is first inserted into the right hash table, and probes the left hash table. The left delta then joins with the right hash table as well. If the operator need to emit the full result, we also insert the right delta ΔD_r into the right hash table and probe it by scanning the left hash table and the left delta. We omit the action description for space limits.

The final action is to drop the right hash table. In this case, we do not keep any intermediate states, so we need to recompute the join. We pull full results from both sub-trees, build a hash table for the right sub-tree, and use the left full result to probe it. Its action description is similar to the previous two and we omit it here.

4.3.4 Computational Complexity

The time complexity depends on two factors: the number of operators and the complexity for applying actions for each operator. In our algorithm, each operator takes a memory budget m ($0 \leq m \leq \mathbb{M}$) as input and evaluates all associated actions. The number of different budgets depends on the granularity of budget: if $\mathbb{M}$ =1 GB, we could use Byte as the basic unit of memory, or round each intermediate state's size up to the nearest MB. Assuming there are M budget units and the query plan tree has N operators, the computational complexity of applying actions of all operators is $\mathbb{O}(NM)$. The computational complexity for each action depends on the action itself. For all the actions except join discussed in this paper, their computational complexities are $\mathbb{O}(1)$. The time complexity for join operators is $\mathbb{O}(M)$ because they require enumeration on the memory allocated to each sub-relation. Thus, the overall complexity is $\mathbb{O}(NM^2)$.

5. EXTENSIONS AND OPTIMIZATIONS

In this section we describe how to extend DISS to support updates and deletes, multiple subsequent deltas, and optimizations for our DP algorithm.

Processing Deletes and Updates: As one update can be modelled as a delete and an insert, we only discuss how to process deletes here. To extend our framework to support deletes, we require that the underlying IVM system can incrementally process deletes, and estimate the corresponding cost, cardinality, and selectivity. Here, cost formulas in each action should be modified to consider the cost of deletes. We modify the underlying IVM system to support deletes for the operators we have discussed so far. Due to space limits, we only discuss an IVM algorithm of processing deletes for symmetric hash join, and use it as an example of explaining how to support deletes in DISS. For other IVM algorithms for processing deletes, we refer the reader to a comprehensive survey on materialized views [19].

Each delete is represented as a new tuple with an additional flag field indicating the deletion. Processing a new tuple for symmetric hash join includes two basic steps: 1) maintaining the hash table that is built on the same side where the new tuple comes from, and 2) probing the hash table of the other side to generate new tuples. For the first step, one delete needs to delete the tuple of the hash table on the same side. It finds the corresponding bucket of the hash table and scans the list of tuples associated with that bucket to find the exact tuple to delete. Its cost could be higher than inserting a new tuple because for insert operation, once the right bucket is found, the inserted tuple is added to the list of tuples for that bucket without scanning it. Therefore, the corresponding cost formulas are modified to account for this cost. For example, for $\mathcal{F}^{op}(m)$ in Action 6, if the delta includes deletes, we need to split the cost of inserting a delta into the right hash table (i.e. $C_I(\Delta D_r, D_r)$) into two parts, where one represents the cost of inserts and the other represents the cost of deletes. For the second step, the cost of generating new tuples for a delete by probing the hash table of the other side is the same as an insert. Other operators discussed in Section 4.3.3 can be supported in a similar way.

Multiple Deltas: Until now we only consider one delta at one time, containing tuples for one or more relations. In practice, there will likely be multiple deltas. For this case, there are two possible solutions. If we are able to predict multiple deltas together in the future, we can extend our DP algorithm to minimize the running time of batch processing and multiple delta processing as a whole. However, this approach makes computational complexity of the DP algorithm too high. For one delta, each operator needs to find the minimal cost of computing delta output (i.e. $\mathcal{D}^{op}(m)$) and the minimal cost of computing full output (i.e. $\mathcal{F}^{op}(m)$). If we consider K deltas together, all possible output combinations for K deltas are $\mathbb{O}(2^K)$, and computing the cost for one possible combination is K. Combined with the complexity for one delta, the complexity for K deltas is $\mathbb{O}(K2^KNM^2)$. Therefore, we choose an alternative way of applying our DP algorithm for one delta at a time. Specifically, we choose to select a new subset of intermediate states to persist and build if the predicted next delta is different from the current delta (i.e. the sizes of new tuples for base relations). Otherwise, we use the same plan. We emphasize that to determine the intermediate states for the next delta, we run our algorithm *before* processing the current delta because we can only build intermediate states, if any, while we process the current delta (or initial data).

Accelerating DP Algorithm: Here we propose an optimization of our DISS algorithm. The optimization is based on an observation that intermediate states' sizes are usually sparse, so the optimal intermediate states usually stays the same when the memory budget does not change drastically. Although theoretically there are M possible values, in practice the number of distinct $\mathcal{D}^{op}(\cdot)$ and $\mathcal{F}^{op}(\cdot)$ is far less than M.

We exploit the sparsity of unique $\mathcal{D}^{op}(\cdot)$ and $\mathcal{F}^{op}(\cdot)$ values to optimize our algorithm. The key observation is that both $\mathcal{D}^{op}(\cdot)$ and $\mathcal{F}^{op}(\cdot)$ are non-increasing monotonic functions with respect to memory budgets. Therefore, instead of computing cost values from child operators for all possible memory budgets from 0 to M, we run a binary search of memory budgets. Specifically for each operator, we start with computing its cost values with the memory budget 0 and M respectively. If they have the same value, the costs with memory budgets between 0 and M are the same and we do not need to compute them from child operators; otherwise, we divide this range $[0, M]$ into two equal ones and repeat the aforementioned process to compute the two separate ranges until all cost values are computed for this operator.

6. EXPERIMENTS

Our experimental study addresses the following questions:

- How much does DISS lower delta processing latency and memory consumption compared with IVM and (re-)batch processing under IQP applications? (Section 6.3)
- What is the impact of delta prediction quality on DISS performance? (Section 6.4)
- How does DISS's dynamic programming algorithm gracefully trade memory consumption for efficient delta processing compared to greedy algorithms? (Section 6.5)
- What is the benefit and cost of injecting operators or building new states (i.e. MATERIALIZE and SYMMETRIC HASH JOIN) into the query plan? (Section 6.6)
- How much does DISS lower delta processing time in workloads with deletes? (Section 6.7)

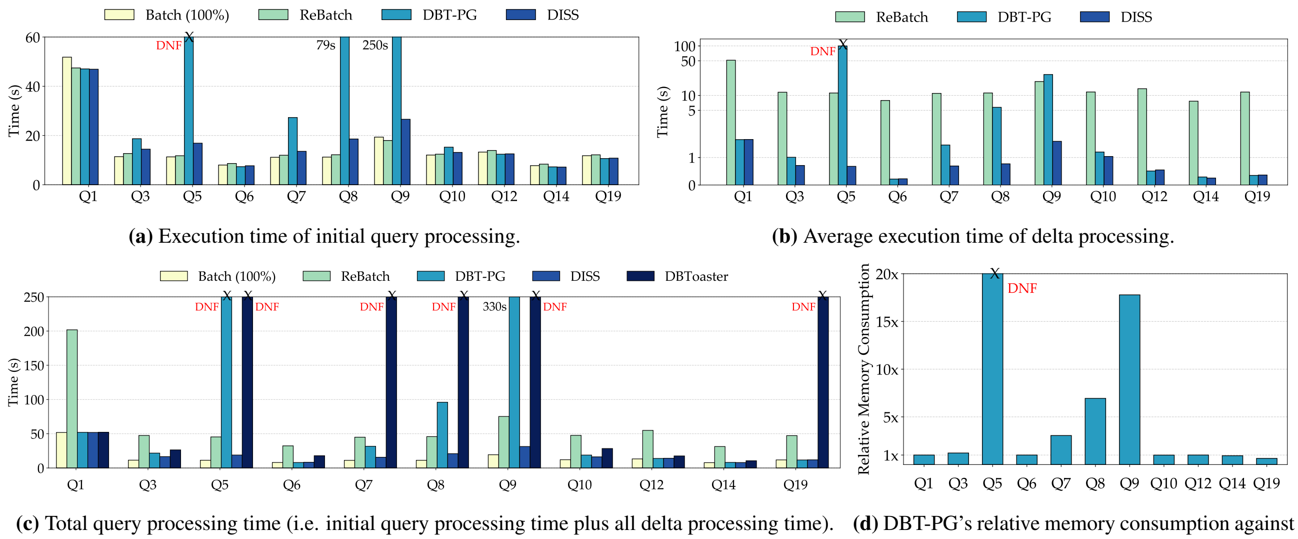
We evaluate the performance of DISS on a machine with two Intel Xeon Silver 4116 processors (i.e. 2.10GHz), 192 GB of RAM, and Ubuntu 16.04 operating system. For all experiments we report single threaded query execution with no concurrent requests.

6.1 Prototype Implementation

We implement the DISS prototype in PostgreSQL 10. When a query is issued to DISS, it uses the query optimizer of PostgreSQL to process this query and generate a query plan. DISS then obtains information about new data from a delta predictor without requiring any user specification. DISS periodically asks for information about the next delta that specifies how many new tuples are expected to arrive for each incomplete table and whether that table will be complete after the next delta. We discuss two scenarios of obtaining such information in Section 6.3. After, we use DISS to choose intermediate states to keep (and to rebuild). Intermediate states that are marked as kept will be materialized during the initial query processing. Specifically, if DISS chooses to materialize the output tuples of an operator it inserts a Materialize node, and if DISS chooses a symmetric hash join it adds a Hash node. DISS adopts the execution engine of PostgreSQL to run this modified query plan over the incomplete dataset and when it finishes, DISS discards unnecessary intermediate states and waits for a delta. We also modify PostgreSQL to keep the query alive after the initial query result is returned and the client is able to refresh the query result when the next delta is processed.

We generate delta tuples using INSERT SQL statements of PostgreSQL. We modify the insert operation such that it not only inserts tuples into the database, but also notifies the queries (e.g. a delta log [29, 19]). For this prototype, each query monitors the number of delta tuples and when it exceeds a threshold or when an predefined time elapses, delta processing is triggered. DISS repeats the aforementioned process to generate a modified query plan that specifies the intermediate states to persist, and delegate query processing to PostgreSQL. During delta processing, we use our modified operators (based on the implementation of PostgreSQL) to incrementally process delta tuples or re-generate full output from child operators. The query terminates when the delta predictor informs that there will be no additional deltas.

We compare DISS against a state-of-the-art incremental view maintenance system, DBToaster [4], that supports continuous query processing. Different from DISS, which selectively materializes intermediate states by considering intermittent and predictable arrival patterns, DBToaster recursively maintains all higher-order views (i.e. intermediate states with indexes) to support frequently refreshing query results in response to high-velocity data streams. To make a fair comparison of the query execution plan between DISS and DBToaster, we migrate DBToaster's query plans to PostgreSQL (denoted as DBT-PG). This includes which intermediate states to materialize and physical execution steps of maintaining those intermediate states for each new tuple. DBToaster uses hash join as its physical join operator implementation. We use TPC-H Q3 to explain the execution of DBToaster in PostgreSQL. Q3 joins three relations $Lineitem \bowtie Orders \bowtie Customer$. The recursive view maintenance algorithm of DBToaster not only builds hash tables for $Lineitem$, $Orders$, and $Customer$, but also builds hash tables for $Lineitem \bowtie Orders$ and $Orders \bowtie Customer$. To process a new tuple from $Lineitem$, DBToaster joins it with the hash table for $Orders \bowtie Customer$, and also inserts it to related hash tables such as the ones for $Lineitem$ and $Lineitem \bowtie$



(a) Execution time of initial query processing.

(b) Average execution time of delta processing.

(c) Total query processing time (i.e. initial query processing time plus all delta processing time).

(d) DBT-PG's relative memory consumption against DISS.

Figure 3: DISS with late data processing on TPC-H scale factor 5.

Orders. Processing tuples from *Orders* and *Customer* follows the similar steps. This approach has the benefit of reducing the number of joins for maintaining the final join results, but comes with the cost of maintaining additional materialized views.

Our experiments also include a conventional batch processing in PostgreSQL. After the initial query processing or each delta processing, it discards all intermediate states and re-computes on arrivals of deltas. Note that this is how PostgreSQL supports refreshing materialized views [1]. We denote this as ReBatch in our tests.

6.2 Benchmark Setup

Our experiments use the TPC-H benchmark, a decision support benchmark that analyzes the activity of a wholesale supplier, and join ordering benchmark (JOB) [40] that is built on IMDB datasets to test queries with many joins (i.e. up to 16-way join). Our current prototype supports flat select-project-join-aggregate (SPJA) queries, which covers 11 queries of TPC-H and all 33 queries of JOB. We generate an incomplete dataset by removing some portion of tuples from the complete dataset and then insert them back as deltas. We build a primary index for each relation in TPC-H and JOB. We assume small dimension relations including REGION and NATION are always complete for TPC-H and relations having less than 10,000 tuples are always complete for JOB. We use a dataset with scale factor 5 for TPC-H since DBT-PG exceeds memory limitation on larger scale factors on our test machine. We also test a large scale factor (SF=50) in a larger machine and find they result in similar observations, which we omit here due to space limits. JOB includes 21 IMDB tables with 4.3 GB of data in total. In our experiments, we run each test three times and take the average number. For experiments of DISS, DBT-PG, and ReBatch, we use hot start, which means all base tables are either in buffer pools or OS caches.

6.3 IQP Use Scenarios

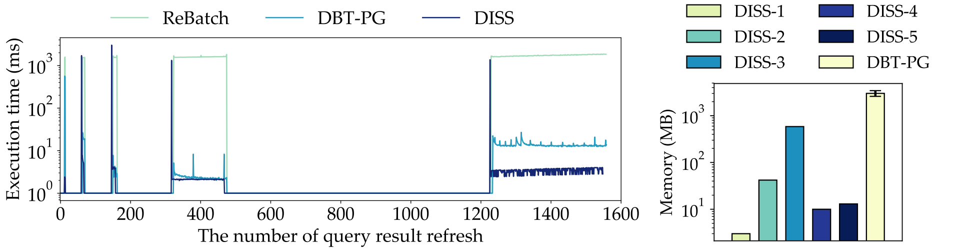
We verify the performance of DISS on two representative scenarios: late data processing and data cleaning. For each scenario, we explain how to predict delta information and discuss experiment setups and results.

6.3.1 DISS with Late Data Processing

We consider a scenario where a dataset is collected from external sources (e.g. sensors), and users demand the refreshed results periodically. While most data arrives on time, some data items can be delayed due to network conditions (i.e. long-tail network traffic). In this application, we can predict the arrival pattern of missing data using historical statistics (e.g. building cumulative distribution functions).

In this experiment, we model the long-tail of late data by a geometric distribution. Specifically, the arrival time of each data item is independent from each other. Each data item arrives within a time interval with a probability p, and if not, it has the same probability to arrive in the next time interval. We set p as 0.9 and the time interval as 60 seconds. We assume there are three deltas: 90% of the complete dataset are available initially, and the incoming three deltas are 9%, 0.9%, and 0.1% respectively. DISS refreshes query result every 60 seconds after the initial query processing is finished. We assume all relations (except REGION and NATION) have deltas. For reference we also include the result of batch processing on a complete dataset, denoted as Batch (100%). We also assume the memory budget is sufficient. If a query cannot finish within 500 seconds, we mark it as DNF (i.e. Did Not Finish).

The experiment results are shown in Figure 3, where we report the initial query processing time (Figure 3a), average delta processing time (Figure 3b), total query processing time, which is the sum of initial query processing time and all delta processing time (Figure 3c), and relative memory usage of DBT-PG compared with DISS (Figure 3d). In Figure 3a, DISS is slower than ReBatch in the initial query processing because it needs to build more intermediate states (e.g. hash table in symmetric hash join) to accelerate future delta processing. On the other hand, DISS is much faster than DBT-PG because it builds fewer views and fewer intermediate states. For the delta processing time shown in Figure 3b, we see that DISS performs better than both ReBatch and DBT-PG because it selectively keeps intermediate states that are useful for delta processing, without introducing the heavy cost of maintenance. Specifically for queries with 5-way join or more (i.e. Q5, Q7, Q8, and Q9), the delta processing of DISS is at least 2.1x faster than DBT-PG. The reason is that DBT-PG not only builds more intermediate states with more joins present (e.g. 21 materialized hash tables for Q9 with 6-way join), but also is unable to avoid large intermediate state. For example, DBT-PG needs to materialize the joined results of tables Customer and Supplier on nationkey for Q5. This means that on average each tuple in Supplier can successfully join 30000 tuples



(a) Execution time of delta processing

(b) Memory usage after delta phase

Figure 4: DISS with HoloClean (Q8)

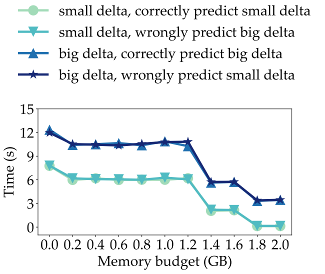


Figure 5: Quality of cardinality prediction (Q8)

Table 2: Aggregated results of join ordering benchmark

		ReBatch	DBT-PG	DISS
Number of Query Finished		33	28	33
Total Query Processing Time of One Query (s)	Avg	15.9	77.5	10.6
	Max	112.6	430	72.1
	Min	2.7	3.0	2.2
Memory Consumption (GB)	Avg	0	14.3	1.5
	Max	0	86.7	12.1
	Min	0	0.35	0.2

in Customer. Such joins with extremely high selectivity should be avoided. For DISS, this case can be avoided by leveraging the query optimizer of underlying databases. While DBT-PG runs faster than ReBatch in most queries, in some cases the cost of maintaining intermediate states dominates and makes DBT-PG slower than ReBatch (e.g. Q9). Figure 3c and Figure 3d show the total query processing time and relative memory consumption of DBT-PG to DISS. We see that DISS uses less overall query processing time than ReBatch and DBT-PG, and consumes less memory than DBT-PG. These figures show DISS strikes a good trade-off between resource consumption and delta processing efficiency. Specifically, DISS is up to 240x and 25x faster than ReBatch and DBT-PG respectively during delta processing, and only consumes at best 5.6% of the memory consumed by DBT-PG.

We also include the total query processing time of the native DBToaster system (the latest release of the C++ version) in Figure 3c for reference. We find that DBToaster cannot finish for 5 queries, and performs worse than DBT-PG for many queries. One reason we observed during testing is that DBToaster's generated code consumes enormous amounts of memory, which we believe is due to memory management issues. For example, we observed the execution of Q7 for 20 minutes, and found it consumes 70% memory of our test machine, which translates to 137 GB. By contrast, DBT-PG only consumes 3.6 GB. For Q19, it does not consume much memory, but is very slow when it performs string matching for predicate evaluation. We also test a smaller-scale (i.e. SF 0.1) dataset and find that while all queries are finished by DBToaster, DBT-PG is faster in most cases.

We also test the performance of DISS for JOB. Our test starts with 99% of data in the batch phase, and inserts a 1% delta. If a query cannot finish within 500s, we mark it as DNF. In this test, we assume the memory budget is sufficient for DISS. We report the results of variant A for 33 queries; other variants, which have different values on predicates, result in similar performance.

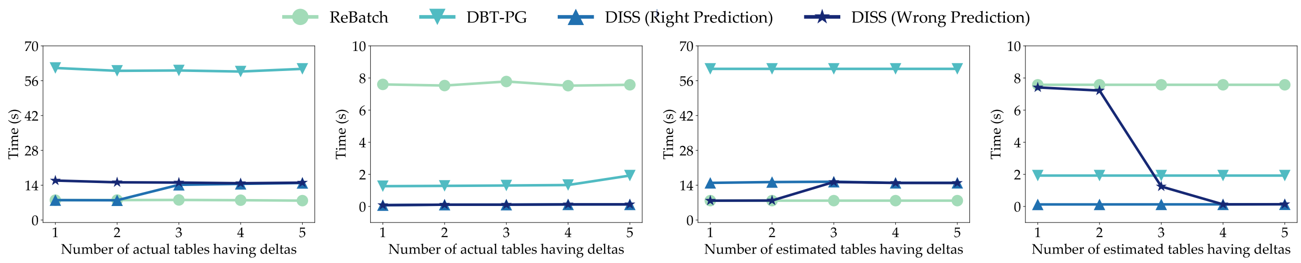
Table 2 shows the number of queries that finish within 500s, total query processing time of one query (batch and delta), and memory consumption after batch processing. We find the results are consistent with TPC-H. DISS can finish all queries, and is faster than both ReBatch and DBT-PG. Seven queries cannot finish for DBT-PG because it takes too much time to recursively materialize intermediate states. For example, Q29 involves a 16-way join, which leads DBT-PG to materialize more than 1000 intermediate states. In addition, DISS consumes much less memory than DBT-PG, which also validates the memory consumption results of TPC-H.

6.3.2 *DISS with HoloClean*

Our second IQP use scenario is a data cleaning system HoloClean [51]. Given a dataset with dirty tuples HoloClean detects dirty tuples based on pre-defined rules, and then executes a cleaning algorithm over the identified dirty tuples. The cleaning algorithm trains a statistical model based on clean tuples and uses the model to predict correct values for dirty tuples. We build a full pipeline between HoloClean and DISS, where DISS executes queries over the initial clean tuples (i.e. initial query processing), receives cleaned tuples (i.e. delta tuples) from HoloClean, and incorporates delta tuples into the query result. DISS obtains the information about the next delta from HoloClean regarding which relations it is cleaning and the pace (tuples/sec) of cleaning for each relation.

In this experiment, we use TPC-H dataset and assume 20% of the records are dirty. We set scale factor as 1 to allow HoloClean to finish within a reasonable amount of time. HoloClean cleans dirty relations one by one and delivers cleaned tuples to the data processing engines (e.g. DISS and ReBatch), which refreshes the query result every 5 seconds regardless whether new data appears. We report the results on Q8, which includes five dirty relations (SUPPLIER, CUSTOMER, PART, ORDERS, LINEITEM), and report the execution time of refreshing the query result each time in Figure 4a. We also show memory consumption of DISS when the query is inactive for each relation cleaning in Figure 4b. For example, DISS-1 in Figure 4b represents the memory consumption when HoloClean is cleaning the first relation SUPPLIER. Figure 4b also includes average memory consumption of DBT-PG along with its minimum and maximum cost shown as error bars. Note that we use log scale for y-axis in Figure 4b.

In Figure 4a, HoloClean repeats the process of training statistical models and cleaning tuples via the trained models for each relation. Query result refreshing is trivial when HoloClean is training and the data processing engine is inactive (i.e. refresh execution time is 0 ms). When HoloClean is cleaning (and delivering cleaned tuples continuously), we see that DISS refreshes the query result much faster than ReBatch and DBT-PG in most cases except when the cleaned tuples come from a different relation. This only happens when HoloClean completes one relation and moves on to the next (for example, the 310th refresh). In this case, DISS needs to build and keep new intermediate states for processing delta tuples from a different relation, and we find that it has comparable performance to ReBatch here. In other cases, DISS outperforms DBT-PG and ReBatch by up to 6x and 500x. Additionally, DISS only needs to keep no more than 15MB of data in most cases and 600MB in the



(a) Overestimation (Batch) **(b)** Overestimation (Delta) **(c)** Underestimation (Batch) **(d)** Underestimation (Delta)

Figure 6: Impact of individual relation's completeness prediction's quality (Q8): effect of overestimation and underestimation of the number of incomplete relations. For overestimation (i.e. the first two figures), DISS predicts all relations being incomplete, while the number of incomplete relation varies (in x-axis). For underestimation (i.e. the last two figures), all relations are incomplete while DISS foresees a subset of them (in x-axis).

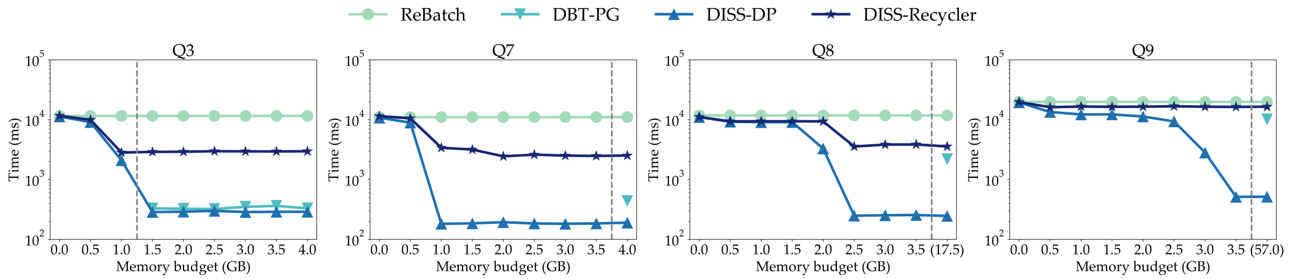


Figure 7: Delta processing time under different memory budgets (all relations have a single 1% delta): DISS and ReBatch can work for all memory budgets, but DBT-PG only works when the memory budget is larger than the vertical dashed line. (Y-axis is log-scale)

worst case (i.e. the third relation PART), whereas DBT-PG consumes about 3000MB of memory all the time. This experiment shows that in a real application, DISS can quickly process delta tuples and at the same time consume limited memory.

6.4 Impact of Prediction Quality

In previous experiments, we assume that the prediction of delta is always accurate. Here we inspect how imperfect prediction affects the performance of DISS. Our experiment investigate two situations: carnality discrepancy and categorical discrepancy. We report our results on TPC-H's Q8.

In the first experiment, we assume the predictor correctly predicts that all relations are incomplete, but the prediction of deltas' sizes could be wrong. A relation being incomplete means that it expects new data in the future. Such information is obtained from the delta predictor. For example, HoloClean can tell the predictor that a table is complete if it has completely cleaned that table. Here, we assume that the initial batch contains 70% data and consider four possible scenarios: the actual delta as big or small, and the predicted delta as big or small. A small delta contains 1% of the complete relation, and a big delta contains 30%. We vary the memory budget from 0 to 2 GB, and report the delta processing time in Figure 5. We find the performance of right and wrong delta prediction are close. This is because the cost of rebuilding the intermediate states dominates the cost of delta processing, so DISS chooses the correct intermediate states to keep even if the prediction is not perfect.

Next, we consider the impact of incorrect completeness prediction. We assume the delta size as 1% for the following experiments. We separate the overestimation and underestimation scenarios. For the overestimation case (Figure 6a and Figure 6b), the predictor asserts all 5 relations are incomplete and the actual number of incomplete relations varies from 1 to 5. We assume larger relations are more likely to be incomplete and are chosen as incomplete relations first (e.g. when there is only 1 incomplete relation, it is LINEITEM). For the underestimation case (Figure 6c and Figure 6d), all relations are actually incomplete, but the prediction only contains a subset of them. Here, we vary the number of predicted incomplete relations from 1 to 5. Figure 6 shows that in the overestimation case, the batch processing time of a wrong prediction for DISS (i.e. DISS (Wrong Prediction) in Figure 6a) is higher because it keeps more intermediate states for the future delta processing, but DISS's delta ingestion performance is stable regardless the prediction. Conversely, DISS has a longer delta processing time but a shorter batch processing time in the underestimation cases. Compared to ReBatch and DBT-PG, DISS has a similar performance of delta processing to ReBatch in its worst case and has better performance than DBT-PG when we can correctly predict at least 3 incomplete relations out of all 5 (shown in Figure 6d). The above two experiments show that DISS can outperform ReBatch and DBT-PG even when the prediction is not perfect.

6.5 Effectiveness of State Selection

Here, we test the effectiveness of intermediate state selection of our dynamic programming algorithm. We vary the memory budget, and measure the performance of delta processing based on our dynamic programming algorithm (DISS-DP) and an intermediate state cache algorithm [45] (DISS-Recycler). DISS-Recycler caches a subset of intermediate states for future queries with respect to a memory budget. The cache algorithm is based on a heuristic metric BENEFIT associated with each intermediate state. It represents the cost of recomputing it from other cached intermediate states or base relations, multiplied by the number of times it has been (or will be) used, and then divided by its memory usage. For a new intermediate state, DISS-Recycler chooses to cache it if there is enough memory or DISS-Recycler can find a set of cached intermediate states to evict with a lower average benefit such that these intermediate states can create enough memory to cache the new state. Note

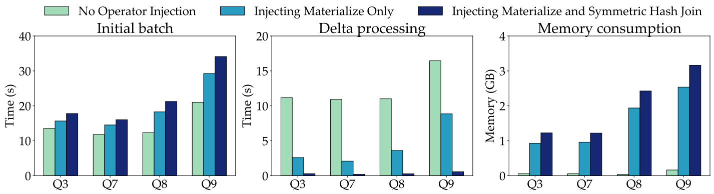


Figure 8: Impact of injecting operators in DISS

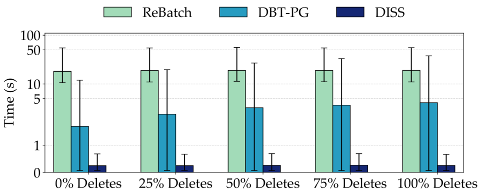


Figure 9: Average, min, and max delta processing time by varying percentage of deletes (1% delta)

that if one intermediate state is updated, DISS-Recycler regards it as a new state and repeats the aforementioned algorithm. For reference, we compare their performance with ReBatch and DBT-PG, which do not choose a subset of intermediate states to materialize.

We vary the memory budget from 0 to 4 GB with a step of 0.5 GB and report the delta processing time for Q3, Q7, Q8, and Q9 of TPC-H. We choose these queries because they have the most number of joins (and also intermediate states) and can be finished by DBT-PG. With more intermediates states in a query plan, we can better observe the behavior of our DP algorithm compared to other approaches. Here we test a single 1% delta that includes delta tuples for all relations (except REGION and NATION). The experimental results are shown in Figure 7. We see that the DP algorithm has lower delta processing time than DISS-Recycler, because DISS-Recycler does not consider information about a future delta. Specifically, DISS-DP is to up 30x faster than DISS-Recycler.

ReBatch fails to utilize the available memory budget to accelerate delta processing. DBT-PG, however, only works after we provide enough memory (i.e. after the vertical dashed line) and is not always the most time-efficient since it has to maintain the extra intermediate states. When there is no memory budget available, DISS-DP uses the approach of ReBatch by discarding all intermediate states after the initial query processing and recomputes from base relations for delta processing. Therefore, it has the same performance as ReBatch when the memory budget is 0. As the memory budget increases, DISS-DP keeps more intermediate states and becomes close to the performance of continuous query processing (i.e. DBT-PG) for delta processing. By materializing a subset of intermediate states, DISS-DP even outperforms DBT-PG with less memory consumption. Overall, these results show that DISS-DP improves the performance by selectively persisting intermediate states with limited memory consumption.

6.6 Impact of Additional Operators

In this paper, we assume a pipelined execution engine, but also consider injecting new operators (e.g. MATERIALIZE) to improve delta processing efficiency when necessary. We measure the benefit and cost of injecting operators for materializing pipelined operators and converting hash-joins to be symmetric (i.e. injecting HASH operator). Specifically, we consider DISS on four TPC-H queries. There are three possible scenarios: the original pipelined query plan without operator injection, the DISS-optimized plan which only allows extra MATERIALIZE operators, and the DISS-optimized plan which may MATERIALIZE and build extra intermediate states (i.e. HASH for symmetric hash join). We report the initial batch processing time, delta processing time, and the memory consumption for storing intermediate states. We assume all relations have a single 1% delta and the memory is sufficient.

Figure 8 shows when building intermediate states is permitted, DISS has a much lower delta processing time, but at a higher initial query processing time and higher memory consumption. This is because DISS can keep or build more states for delta processing, but has to pay the corresponding costs during initial query processing. Our DP algorithm can intelligently select the intermediate states to keep or to build, and thus minimizes the overall query processing time, especially in the presence of multiple deltas.

6.7 Performance Impact of Delete Workloads

We test 11 TPC-H queries using delta data with mixed inserts and deletes. We start with 99% data in the batch phase, and then processes a single 1% delta. We vary the percentage of deletes in the delta to be 0%, 25%, 50%, 75%, and 100%. We report the average delta processing time along with minimum and maximum time in Figure 9 and find that DISS always outperforms DBT-PG and ReBatch. An interesting observation is that with a higher percentage of deletes, delta processing time for DBT-PG increases too, while the processing time for ReBatch and DISS stays the same. The reason is DBT-PG cannot avoid materializing join operators with high selectivity (i.e. a tuple from one table can successfully join many tuples of the other one). One such example is that in Q9 DBT-PG needs to materialize the join results of tables Supplier and Lineitem joined on supplier key. Since there are no predicates on the two tables, each Supplier tuple joins 600 Lineitem tuples on average. With the hash table for Lineitem built on supplier key, deleting one tuple from Lineitem's hash table needs to find the right bucket and scan through the list of tuples associated with the bucket (i.e. at least 600 for each bucket) to find the right one to delete. In contrast, DISS uses PostgreSQL's query optimizer to join Lineitem with other tables having lower selectivities first, and then join Supplier, which greatly reduces the cost of finding the tuple to delete.

7. CONCLUSION

We introduce IQP as a new query processing paradigm for ongoing queries that balances query processing latency and controlled resource consumption by exploiting knowledge of data arrival patterns. We develop an IQP prototype, DISS, based on PostgreSQL that selects a subset of intermediate states from query execution to persist for efficient processing of future data arrivals; this state selection algorithm minimizes resource consumption for queries when not updating results, and lowers query refresh time by selecting a set of intermediate states within a budget constraint. Our experimental evaluation shows that DISS is able to achieve low latency and limited memory consumption simultaneously for many applications and offers significant performance improvements over state-of-the-art IVM systems that do not leverage knowledge about future data arrivals.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers, Goetz Graefe, Arnab Nandi, Aditya Parameswaran, Andy Pavlo, and Eugene Wu for their valuable feedback. This work was supported by a gift from Google and NSF grant CCF-1139158.

9. REFERENCES

[1] Refresh materialized view. https://www.postgresql.org/docs/10/static/sql-refreshmaterializedview.html.

[2] Spark structured streaming. https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html.

[3] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 496–505, 2000.

[4] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB*, 5(10):968–979, 2012.

[5] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 8(12):1792–1803, 2015.

[6] P. Antonopoulos, H. Kodavalla, A. Tran, N. Upreti, C. Shah, and M. Sztajno. Resumable online index rebuild in SQL server. *PVLDB*, 10(12):1742–1753, 2017.

[7] M. Armbrust, E. Liang, T. Kraska, A. Fox, M. J. Franklin, and D. A. Patterson. Generalized scale independence through incremental precomputation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 625–636, 2013.

[8] T. Azim, M. Karpathiotakis, and A. Ailamaki. Recache: Reactive caching for fast analytics over heterogeneous data. *PVLDB*, 11(3):324–337, 2017.

[9] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*, pages 350–361, 2004.

[10] S. Babu and J. Widom. Continuous queries over data streams. *ACM Sigmod Record*, 30(3):109–120, 2001.

[11] G. Beskales, I. F. Ilyas, L. Golab, and A. Galiullin. On the relative trust between inconsistent data and inaccurate constraints. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 541–552, 2013.

[12] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: Mapreduce for incremental computations. In *ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011*, page 7, 2011.

[13] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 28-30, 1986.*, pages 61–71, 1986.

[14] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 746–755, 2007.

[15] B. Chandramouli, C. N. Bond, S. Babu, and J. Yang. Query suspend and resume. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 557–568, 2007.

[16] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR 2003, First Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 5-8, 2003, Online Proceedings*, 2003.

[17] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 190–200, 1995.

[18] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 379–390, 2000.

[19] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[20] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *PVLDB*, 6(13):1498–1509, 2013.

[21] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 458–469, 2013.

[22] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.*, pages 469–480, 1996.

[23] L. S. Colby, A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, and K. A. Ross. Supporting multiple view maintenance policies. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA.*, pages 405–416, 1997.

[24] A. B. Downey. Evidence for long-tailed distributions in the internet. In *Proceedings of the 1st ACM SIGCOMM Internet Measurement Workshop, IMW 2001, San Francisco, California, USA, November 1-2, 2001*, pages 229–241, 2001.

[25] K. Dursun, C. Binnig, U. Çetintemel, and T. Kraska. Revisiting reuse in main memory database systems. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1275–1289, 2017.

[26] A. Galakatos, A. Crotty, E. Zgraggen, C. Binnig, and T. Kraska. Revisiting reuse for approximate query processing. *PVLDB*, 10(10):1142–1153, 2017.

[27] J. Gjengset, M. Schwarzkopf, J. Behrens, L. T. Araújo, M. Ek, E. Kohler, M. F. Kaashoek, and R. T. Morris. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.*, pages 213–231, 2018.

[28] G. Graefe, W. Guy, and H. A. Kuno. 'pause and resume' functionality for index operations. In *Workshops Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 28–33, 2011.

[29] A. Gupta and I. S. Mumick. *Materialized views: techniques, implementations, and applications*. MIT press, 1999.

[30] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993.*, pages 157–166, 1993.

[31] H. Gupta and I. S. Mumick. Selection of views to materialize under a maintenance cost constraint. In *Database Theory - ICDT '99, 7th International Conference, Jerusalem, Israel, January 10-12, 1999, Proceedings.*, pages 453–470, 1999.

[32] H. Gupta and I. S. Mumick. Selection of views to materialize in a data warehouse. *IEEE Trans. Knowl. Data Eng.*, 17(1):24–43, 2005.

[33] M. Idris, M. Ugarte, and S. Vansummeren. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1259–1274, 2017.

[34] M. Ivanova, M. L. Kersten, N. J. Nes, and R. Goncalves. An architecture for recycling intermediates in a column-store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 309–320, 2009.

[35] A. Jindal, K. Karanasos, S. Rao, and H. Patel. Selecting subexpressions to materialize at datacenter scale. *PVLDB*, 11(7):800–812, 2018.

[36] Y. Katsis, K. W. Ong, Y. Papakonstantinou, and K. K. Zhao. Utilizing ids to accelerate incremental view maintenance. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1985–2000, 2015.

[37] Y. Kotidis and N. Roussopoulos. Dynamat: A dynamic view management system for data warehouses. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA.*, pages 371–382, 1999.

[38] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 802–803, 2006.

[39] S. Krishnamurthy, M. J. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre. Continuous analytics over discontinuous streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 1081–1092, 2010.

[40] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[41] L. Ma, D. Van Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 631–645. ACM, 2018.

[42] C. Mayfield, J. Neville, and S. Prabhakar. ERACER: a database approach for statistical inference and data cleaning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 75–86, 2010.

[43] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 307–318, 2001.

[44] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 439–455, 2013.

[45] F. Nagel, P. A. Boncz, and S. Viglas. Recycling in pipelined query evaluation. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 338–349, 2013.

[46] M. Nikolic, M. Dashti, and C. Koch. How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 511–526, 2016.

[47] M. Nikolic, M. Elseidy, and C. Koch. LINVIEW: incremental view maintenance for complex analytical queries. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 253–264, 2014.

[48] M. Nikolic and D. Olteanu. Incremental view maintenance with triple lock factorization benefits. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 365–380, 2018.

[49] V. Raman, A. Deshpande, and J. M. Hellerstein. Using state modules for adaptive query processing. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 353–364, 2003.

[50] V. Raman and J. M. Hellerstein. Partial results for online query processing. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, pages 275–286, 2002.

[51] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. *PVLDB*, 10(11):1190–1201, 2017.

[52] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.*, pages 447–458, 1996.

[53] N. Roussopoulos. An incremental access method for viewcache: Concept, algorithms, and cost analysis. *ACM Trans. Database Syst.*, 16(3):535–563, 1991.

[54] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1.*, pages 23–34, 1979.

[55] R. Taft, N. El-Sayed, M. Serafini, Y. Lu, A. Aboulnaga, M. Stonebraker, R. Mayerhofer, and F. J. Andrade. P-store: An elastic database system with predictive provisioning. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston,*

TX, USA, June 10-15, 2018, pages 205–219, 2018.

[56] N. Tatbul, U. Çetintemel, and S. B. Zdonik. Staying FIT: efficient load shedding techniques for distributed stream processing. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 159–170, 2007.

[57] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*, pages 309–320, 2003.

[58] T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *Bulletin of the Technical Committee on Data Engineering*, page 27, 2000.

[59] A. Wilschut and P. Apers. Pipelining in query execution. In *Proceedings of the International Conference on Databases, Parallel Architectures and Their Applications (PARBASE 1990)*, pages 562–562, United States, 3 1990. IEEE Computer Society.

[60] M. Yakout, L. Berti-Équille, and A. K. Elmagarmid. Don't be scared: use scalable automatic repairing with maximal likelihood and bounded changes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 553–564, 2013.

[61] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 29–42, 2008.

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

[63] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 265–276, 2014.

[64] Y. Zhang, B. Hull, H. Balakrishnan, and S. Madden. ICEDB: intermittently-connected continuous query processing. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 166–175, 2007.

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

[66] J. Zhou, P. Larson, J. Goldstein, and L. Ding. Dynamic materialized views. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 526–535, 2007.