Thrifty Query Execution via Incrementability

Dixin Tang, Zechao Shang, Aaron J. Elmore, Sanjay Krishnan, Michael J. Franklin
University of Chicago
{totemtang@,zcshang@cs.,aelmore@cs.,skr@cs.,mkfranklin@}uchicago.edu

ABSTRACT

Many applications schedule queries before all data is ready. To return fast query results, database systems can eagerly process existing data and incrementally incorporate new data into prior intermediate results, which often relies on incremental view maintenance (IVM) techniques. However, incrementally maintaining a query result can increase the total amount of work mainly as some early work is not useful for computing the final query result. In this paper, we propose a new metric incrementability to quantify the cost-effectiveness of IVM to decide how eagerly or lazily databases should incrementally execute a query. We further observe that different parts of a query have different levels of incrementability and the query execution should have a decomposed control flow based on the difference. Therefore, to address these needs, we propose a new query processing method Incrementability-aware Query Processing (InQP). We build a prototype InQP system based on Spark and show that InQP significantly reduces resource consumption with a similar latency compared with incrementability-oblivious approaches.

CCS CONCEPTS

• **Information systems → Database query processing**; **Query optimization**; **Query planning**.

KEYWORDS

incremental view maintenance; non-positive query; resource efficiency; incrementability; query service; cloud database

ACM Reference Format:
Dixin Tang, Zechao Shang, Aaron J. Elmore, Sanjay Krishnan, Michael J. Franklin. 2020. Thrifty Query Execution via Incrementability. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3389756

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'20, June 14–19, 2020, Portland, OR, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389756

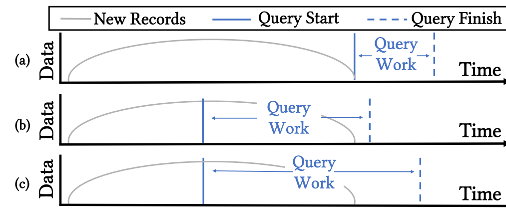


Figure 1: How incrementability can impact query latency and the amount of work done.

1 INTRODUCTION

Almost all open-source and commercial database systems support triggers, which are stored procedures executed when an event occurs. Examples of triggering events include a time frequency (e.g. every hour), a progress condition (e.g. data completely loaded), or a constraint violation (e.g. duplicate user ids added to a database). As is often the case, the stored procedure is itself a query, and there is an interesting question of how to process this pending query. One could simply wait until the trigger to begin processing in a way similar to traditional batch query execution. Or, one could treat the query like a standing query in a streaming system by continuously updating the results in anticipation of a future trigger. In general, there is a trade-off space between the resource-hungry but low-latency streaming approach and a resource-efficient but higher-latency batch evaluation [40].

This paper studies how a user can effectively exploit such a middle-ground for scheduled or triggered queries. For example, suppose she would like to reduce her latency by 50%, how much more resources would she have to use? In the context of triggered queries, an important question towards this goal is when to start processing a query. Consider the motivating example in Figure 1, where data is being progressively loaded into the database and the goal is to compute the result of a pre-defined query. In Figure 1a, a traditional batch query does not begin until all new data arrives. No resources are held or used while data are arriving. If a system wanted to provide the result earlier, it would need to start processing existing data earlier by investing additional resources and incrementally incorporating new data into prior results (Figure 1b). Exactly how much benefit there is for eager processing depends on the structure of the pending query; for example, the latency could see less improvement

as in Figure 1c. Some queries are amenable to incremental computation while others can incur steep overheads that may not be worth the additional resources.

Not surprisingly, our study is related to algorithms for Incremental View Maintenance (IVM). Prior work [6, 23, 25] shows IVM is efficient for select-project-join-aggregate (SPJA) queries, but less so for more complex queries, such as those involving nested queries or outer/anti-joins [16]. One major reason is that many complex queries are non-monotonic: newly arriving data can force these queries to delete previously produced output tuples. For example, consider a SQL query that finds all tuples with an above-average attribute value. To incrementally maintain this result, on each new tuple, the maintenance algorithm has to not only update the running average, but also re-scan all the previous tuples to update query result if the average changes. In other words, some amount of the incremental work in such a query removes old results instead of simply making forward progress; making it less beneficial to maintain frequently.

However, we noticed that many such queries, while expensive to incrementally maintain, have substructures that are amenable to incremental computation. For the example query above, a better strategy is to eagerly maintain the average values, and less frequently re-scan to find the tuples that are above the average. State-of-the-art IVM systems lack the ability to tune maintenance frequencies for individual dataflow paths to optimize overall system performance. Making these tuning decisions requires a metric of "incrementability" to indicate how amenable a particular operator or pipeline of operators is for incremental execution.

One of our contributions is to propose such a metric aptly called *incrementability*. A query with a high incrementability reduces its final work without much increase to its total work. We define *total work* as all work done by the system for the query to compute the final query result (which can be a viewed as a proxy for resource consumption) and *final work* as the work spent after data is complete and the trigger starts the query (which can be viewed as a proxy for a query's latency). We quantify the final work and total work based on the cost metric in a RDBMS optimizer, which could be a unified cost of estimated CPU time and I/O operations, or number of tuples processed by all operators.

Ideally a system would more eagerly schedule query parts with higher incrementability than those with a lower one. We leverage this definition to propose a new query processing paradigm, *Incrementability-aware Query Processing (InQP)*, that leverages incrementability to efficiently improve query performance. We decompose a query into query paths of tuples' data flow between buffered operations. We propose a new cost model that computes incrementability for each query path from a decomposed query. To intelligently

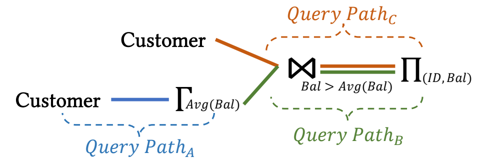


Figure 2: A query with multiple query paths.

improve performance, InQP executes query paths at different paces (or frequencies) based on their respective incrementability. For our initial prototype, users specify a final work constraint, and InQP automatically minimizes the total work under the given constraint.

We address two challenges of InQP. First, computing incrementability requires estimation of total work and final work, but conventional cost models are designed for one-batch processing instead of incremental executions. We address this with a cardinality estimation method that works better for incremental executions. Specifically, we separately estimate cardinalities of tuples that are new, updated, or deleted. Second, we need to assign different paces for different query paths. We propose a greedy algorithm to decide the paces to minimize a query's total work and meet a final work goal.

Our major contributions include: 1) defining and integrating incrementability for resource efficient incremental query execution; 2) a new cost model that accounts for inserts, updates, and deletes for incremental execution plans; and 3) an implementation in a popular data processing system, Spark.

2 BACKGROUND AND DEFINITIONS

In this section, we introduce the problem context and assumptions, InQP's system model, formally define incrementability that captures the ratio of reduced final work to increased total work, and analyze the key factors for incrementability.

2.1 Problem Context and Assumptions

We consider an application scenario where data is being loaded into a database and users want to query the loading data based on trigger conditions, such as time-based (e.g. daily loaded data) or count-based (e.g. for every 100M tuples) conditions. Each triggered query returns an exact result over its conditioned data (e.g. daily loaded data). We emphasize that our approach also applies to general incremental query evaluation and view maintenance, including stream query processing. We assume knowledge of the data arrival rate, which can be predicted based on historical statistics [39]. With this knowledge, we can estimate when a query is triggered, and the final work and total work of a triggered query based on our cost model in Section 3.1. For simplicity, we assume a steady arrival rate for our cost model and we show our robustness for a bursty arrival rate in Section 5.4.

2.2 System Model

Unlike conventional IVM systems, InQP decomposes a query into different query paths.

Query paths: A query path is a dataflow segment in the query operator tree delineated by blocking operators, inputs, or outputs. We note that an operator may belong to multiple query paths. Figure 2 illustrates a sample query that finds the IDs and balance of customers with a balance larger than the average balance (i.e. $Bal > Avg(Bal)$). This query has three query paths: (1) the first query path A takes balance from `Customer` to compute the average balance (i.e. $\Gamma_{Avg(Bal)}$), (2) the second query path B takes $\Gamma_{Avg(Bal)}$, joins it with the all tuples from `Customer` and outputs customer IDs and balance, and (3) query path C takes tuples from `Customer` and joins them with the average value.

Intuitively, query paths represent a stream of tuples between buffers in a pipelined query execution engine. All blocking operators including aggregate, sort, and distinct have output buffers. Similarly, all base relations or delta logs can be treated as buffers as well, and so can the output of the whole query. On the other hand, simple operators like a filter or a join can yield outputs in a streaming fashion.

Query paths naturally decompose the query operator tree, and the individual dataflow paths are the ideal unit for fine-grained resource or latency management. Buffers for blocking and scan operators can be flushed with a varying frequency (called the pace) depending on the incrementability.

Pace configuration: Generally, the buffers could be flushed in different ways, such as a count-based flush (i.e., after 1000 tuples in buffer), a time-based flush (i.e., every 10 seconds), or a heuristic-based flush. For simplicity in our prototype, we use mini-batch execution and consider a flushing with respect to the percentage of the total number of tuples arrived for the system. Each query path with a **pace** k flushes its input buffer whenever the system has received new $\frac{1}{k}$ of all the estimated tuples. A *pace configuration* can be represented as a vector $P = (K_1, K_2, \ldots, K_Q)$ for Q query paths. A special pace configuration $P_{\mathbb{1}} = (1, 1, \ldots, 1)$ represents batch processing where all tuples are processed by a single final step.

2.3 Incrementability Definition

Incrementability: Incrementability describes how incrementable a pace configuration is. For a pace configuration P, we define $C_F(P)$ as its final work and $C_T(P)$ as its total work. Recall that final work means the work the system does after data is complete and total work is all work done by the system to compute the result. Consider two pace configurations $P_2 > P_1$, such that each query path's pace in P_2 is no smaller than the pace in P_1, and there is at least one query path in P_2 whose pace is larger than the pace in P_1. Here, P_2 has a larger total work than P_1 and the incrementability of P_2 over P_1 (e.g. the "benefit" of extra total work) is defined as:

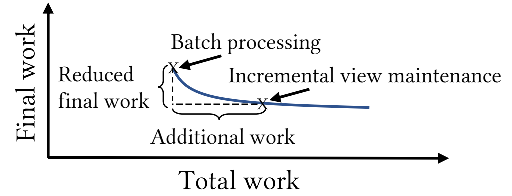


Figure 3: An example of the benefit (i.e. reduced final work) and cost (i.e. additional work) for an incremental execution plan.

$$\texttt{INC}(P_1, P_2) = \frac{C_F(P_1) - C_F(P_2)}{C_T(P_2) - C_T(P_1)} \tag{1}$$

This is defined on two pace configurations that evaluate the same query plan. A similar relationship could also be extended to pairs of different query plans, or more generally, to pairs of two broadly defined "mechanisms" that answer the query (e.g., one count-based trigger and one time-based trigger), but we leave this for future work. Figure 3 shows an example of the benefit and cost of more incremental executions. This curve presents the trade-off between total work and final work. It starts at the point of batch processing (i.e. $P = P_{\mathbb{1}}$). When we invest more resources into incremental executions (i.e. by increasing pace in P), the final work drops and the total work increases. A special incrementability that is relative to the batch execution may be of particular interest. Specifically, $\texttt{INC}(P, P_{\mathbb{1}})$ models the effectiveness of how extra total work saves final work, compared to batch processing.

There are three levels of incrementability. If there is no additional total work for incremental executions (i.e. $C_T(P)$ equals $C_T(P_{\mathbb{1}})$), the incremental executions are *fully incrementable*. Here the incrementability is ∞. If Incrementability is less than ∞, but larger than 0, it means incremental executions are *partially incrementable*, that is, we need to pay some additional cost for total work to reduce the final work. If Incrementability is no larger than zero, more total work is not helpful in reducing the final work, or it even prolongs the overall final work. Here, the query is *non-incrementable* and thus should not be executed until a result is triggered. We summarize the three cases in the following:

- $Incrementability = \infty$: Fully incrementable
- $0 < Incrementability < \infty$: Partially incrementable
- $Incrementability \leq 0$: Non-incrementable

We note the levels of incrementability depend on both input data and query semantics. We now use examples to illustrate this.

Fully incrementable: Positive queries (e.g. SPJ queries) with insert data are fully incrementable because prior output tuples are not removed by new insert tuples and early work of outputing tuples is not wasted.

Partially incrementable: When we have non-positive queries or the data involving deletes or updates, later executions will remove some of prior output tuples, which makes queries partially incrementable. One such example is left-outer-join. In addition to joined tuples, it outputs tuples from the left side that do not match right side tuples. It is possible that new tuples from the right side successfully join with a previous unmatched left tuple, which shall be removed from the output. Therefore, outputting the unmatched tuples too eagerly wastes resources and is less incremental.

Non-incrementable: This is an extreme case of partially incrementable queries. For example, if all data we have processed are deleted later, we should not start incremental executions. Therefore, this case is non-incrementable.

3 COMPUTING INCREMENTABILITY

To calculate incrementability, we need to compute $C_T(P)$ and $C_F(P)$ given a pace configuration P. A critical challenge for this task is how to estimate the *incremental* computation cost for each incremental execution given a pace configuration. We first discuss our modifications on existing cost models for computing $C_T(P)$ and $C_F(P)$. and then discuss how to compute total work and final work given a pace configuration in Section 3.2. In this paper, we support queries with negations (i.e. deletes and updates). Specifically, we support operators for select, project, join (i.e. inner, outer, anti, and semi-join), aggregate, distinct, sort, and limit.

3.1 Cardinality Estimation for Incrementability

As shown in prior work [29, 36, 44], cost modeling involves two key pieces: output cardinality estimation (i.e. the number of tuples each operator output) and relating cardinalities to a unit of work such as I/O cost or CPU time. We find that existing cardinality estimation approaches are ill-suited for the problems studied in this paper, especially for non-positive queries. So we focus on the problem of cardinality estimation, and adopt the cost functions in conventional RDBMS cost model [29, 31, 36] for the second factor.

3.1.1 Problem and Intuition. We emphasize that the problem cardinality estimation for InQP is different as existing solutions [29, 44] for either batch processing or incremental execution, mainly consider the positive queries where extra inputs only produce new outputs but never remove previous outputs. However, an important source of non-incrementable execution are operators that output tuples which are later removed. Therefore, existing cardinality estimation solutions cannot fully consider the effects of non-incrementable parts and fail to compute an accurate incrementability. `Anti-join`, for example, is not a positive operator. `R Anti-join S` outputs tuples in R that do not match any tuples in S. However, extra input tuples of S could delete prior joined tuples because tuples in R that were unmatched become matched.

The core of our solution is to *distinguish the cardinalities of three categories of tuples: inputs, updates, and deletes.* Specifically, updates are those tuples who change previously emitted tuples. Note that we do not regard this as the primary contribution of this paper, but our approach does advance the state-of-the-art [44] in this area. We distinguish these three types of tuples' cardinalites for three reasons.

First, three types of tuples usually have different maintenance costs. For example, if tuples are materialized in a log structure (i.e., unsorted append-only array), inserts are much more efficient to perform than deletes and updates. Distinguishing the cardinalities gives us a better cost estimation. Conventional cost models do not distinguish types, as they typically focus on inserts.

Second, different types of inputs could have different probabilities to produce outputs. Consider natural join as an example. If the insert tuples' join keys are randomly distributed and delete tuples are those who have been previously joined, then the expected cardinalities of their output are different.

Third, operators in incremental executions are stateful, and the cardinality estimator is supposed to take the statistics of states (e.g., the size of the hash tables in a hash join operator) into consideration, and maintain these statistics during/after the estimation so following estimations have accurate information. Distinguishing three types of tuples helps us maintain the statistics of an operator's state. For example, being able to tell whether the input tuples are inserts or updates gives us a better estimation of the hash table size.

3.1.2 Operators. We use a volcano-style query execution model [20] and assume operators are pipelined such that output tuples of operators are not materialized as intermediate results, but directly sent to their parent operators. We support inserts, deletes, and updates for all operators in InQP including scan. A delete is a tuple that has the same content (e.g. attributes and values) as their insert counterparts with an additional tombstone bit indicating the delete. We represent an update as a delete plus an insert tuple. We additionally include a bit in the delete tuple to show that it is the leading tuple of an update.

For each operator, we first discuss its physical design that is borrowed from prior work [14], and then present the cardinality estimation based on the physical choice. Note that we include the physical designs of supported operators for completeness and do not perceive them as our contribution. We represent insert, update, and delete cardinalities as a vector $\mathbb{C} = (\mathbb{C}^I, \mathbb{C}^U, \mathbb{C}^D)$. We denote input's and output's cardinality vectors as $\mathbb{C}_{\mathtt{IN}}$ and $\mathbb{C}_{\mathtt{OUT}}$.

As with conventional cardinality estimation, we use statistical information to help estimate cardinalities. This includes

Selectivity Matrix	Insert	Delete	Update
Insert	0.01	0.0	0.01
Delete	0.0	0.02	0.01
Update	0.0	0.0	0.02

(Columns: Input Operation; Rows: Output Operation)

Figure 4: An example selectivity matrix.

select selectivity that models the probability that a tuple satisfies a certain predicate, *join selectivity factor* [44] that models the probability that any two tuples from two relations successfully join, and *number of groups* for aggregate operators. As in prior work on estimating cardinalities for incremental execution [44], we use statistical information from previous executions as the estimation for upcoming query executions. We also perform an experimental analysis in Section 5.4 to show how biased statistical information impacts the performance of InQP.

Select and project: As select and project operators are stateless, the incremental approach effectively has no difference from a batch execution. For a select operator insert and delete tuples only produce insert and delete outputs correspondingly. However, an update tuples could emit delete and/or insert tuples as a changed tuple may no longer satisfy the predicate (or vice-versa). For its cardinality estimation, different types of tuples could have different selectivity values, which highly depend on the application that generates the input data. Thus, instead of a single selectivity, we use a selectivity matrix $S \in \mathbb{R}^{3\times3}$. Figure 4 shows an example, where columns represent the input operation and rows represents the output operation. So a cell at $S[Delete, Update]$ represents the probability of an update tuple generating a tuple of delete operation, which is 0.01 in our example. We estimate the cardinality as $\mathbb{C}_{\mathtt{OUT}} = S \times \mathbb{C}_{\mathtt{IN}}$. Project operators do not change the cardinality.

Sort and limit: Sort operators maintain a sorted array for all processed tuples emitted. When new tuples arrive, we buffer them into a temporary array. If the sort operator needs to output an updated sorted array, we sort the temporary array and merge it with the original sorted array. During the merge, a delete tuple will remove the corresponding tuple in the original array, which also applies for deletes generated by an update tuple. Before we output the new array as insert tuples, we output the original array with all tuples as deletes to invalidate the prior output. Assuming that the size of the original array is K, the size of new array is $K + \mathbb{C}^I_{\mathtt{IN}} - \mathbb{C}^D_{\mathtt{IN}}$. The output cardinality $(\mathbb{C}^I_{\mathtt{OUT}}, \mathbb{C}^U_{\mathtt{OUT}}, \mathbb{C}^D_{\mathtt{OUT}}) = (K + \mathbb{C}^I_{\mathtt{IN}} - \mathbb{C}^D_{\mathtt{IN}}, 0, K)$.

We only consider limit operators that have a sort as its child. A limit operator takes a parameter N and outputs the first N tuples with respect to the order they arrive from its child sort operator. Recall that the incremental execution of a sort operator first removes all prior output tuples and then inserts newly sorted tuples. For an incremental execution of a limit operator, it outputs the first N delete tuples arrived from its sort child to remove the prior output tuples. For the newly inserted tuples, it outputs the first N.

Aggregate and distinct: We implement the aggregate operator using a hash-based aggregation and support SUM, AVG, COUNT, MAX, and MIN aggregate operations. Since we regard an update as a delete and an insert, we only discuss the case of processing insert/delete tuples. For each input tuple, a hash aggregate operator identifies its group-by attributes and incorporates that tuple into that group's aggregated value. To maintain aggregate operators with deletes and updates, we include a counter for each group to indicate how many tuples are aggregated [23]. We output an insert for a group when that group is first created. If one group's value is changed and its counter is larger than zero, we output an update for this group. When the counter reaches zero, we remove this group from the hash table and output an delete tuple. To support MAX and MIN with deletes and updates, we materialize all prior input tuples for each group. When the tuple for the current aggregate max/min value is deleted, we find the new max/min value in the materialized tuples.

Cardinality estimation on aggregate operators is based on our observation that the operator has different behaviors when all groups are covered by at least one tuple or not. Specifically, when the number of tuples is big enough that all groups have at least one tuple (i.e., "saturated"), new insert tuples only produce *update* outputs. Otherwise it can output *insert*, *delete*, and *update* tuples.

Based on this intuition, we leverage statistics that estimate the total number of groups. This can come from previous executions or statistical approaches [15]. We denote this number as M. When we estimate cardinalities for each incremental execution, we also track the total number of tuples of "net" input tuples as its state information. It represents the sum of input inserts minus input deletes in all previous incremental estimations, which is denoted as N. The estimation of output cardinality is divided into two cases:

- If $N \geq M$, we consider each group has at least one tuple. So each input tuple, regardless of its type, updates a group and thus emits an update tuple. So $(\mathbb{C}^I_{\mathtt{OUT}}, \mathbb{C}^U_{\mathtt{OUT}}, \mathbb{C}^D_{\mathtt{OUT}}) = (0, \mathbb{C}^I_{\mathtt{IN}} + \mathbb{C}^U_{\mathtt{IN}} + \mathbb{C}^D_{\mathtt{IN}}, 0)$.
- If $N < M$, each group has less than one tuple "on average". We adopt a simple model that each of N groups has one tuple, and the remaining groups contain no tuple at all. Thus, each delete input tuple removes one group. So $\mathbb{C}^D_{\mathtt{OUT}} = \min(\mathbb{C}^D_{\mathtt{IN}}, N)$. Similarly, each insert tuple goes to an empty group, and emit a new aggregation tuple, so $\mathbb{C}^I_{\mathtt{OUT}} = \min(\mathbb{C}^I_{\mathtt{IN}}, M - N)$. Update tuples update existing non-empty groups, so $\mathbb{C}^U_{\mathtt{OUT}} = \min(\mathbb{C}^U_{\mathtt{IN}}, N)$.

We implement distinct operators using a hash table which uses the whole tuple as key and the number of duplicated tuples as value. We estimate its cardinalities in a similar way to aggregate operators.

Physical design of join operators: For equi-join we use a symmetric hash join [46], which maintains two hash tables for input tuples from the left and right children. For each hash table, we use the join key as the key and the input tuples as the value. A new tuple from one side updates the corresponding hash table and probes the other one to produce output tuples. For non-equi-join, we maintain two arrays that materialize input tuples from the left and right children. For one new tuple from a child sub-tree, we update its corresponding array, join the new tuple with all tuples in the other array, and produce output tuples. The types of the output tuples (e.g. insert, delete, or update) depend on the types of input tuples and the semantics or join operators (e.g. inner or outer), which we discuss next.

Inner-join, semi-join: We denote two left and right sub-relation cardinalities as $\mathbb{C}_{L\mathtt{IN}}$ and $\mathbb{C}_{R\mathtt{IN}}$, and assume that the sizes of "net" input tuples from previous incremental estimations for left and right sub-relation are $|L|$ and $|R|$ respectively (e.g. number of tuples in a hash table or a materialized array). $|L|$ and $|R|$ are state information and should be updated for each incremental estimation. We first discuss *inner-join*, which emits all pairs of input tuples from left and right sub-relation that meet the join condition. Without loss of generality, we discuss the scenario that input comes from left. In contrast to prior approach that uses a single join selectivity factor to estimate select-project-join queries [44], we use a matrix of join selectivity factors $S_L \in \mathbb{R}^3$. A selectivity factor in S_L represents the probability of an input tuple with a specific operation (e.g. update) successfully joining one tuple from R and producing a tuple with a specific operation (e.g. insert). Given that we have $|R|$ tuples for right sub-relation, we estimate the cardinality as $\mathbb{C}_{L\mathtt{OUT}} = S_L \times \mathbb{C}_{L\mathtt{IN}} \times |R|$.

Semi-join is different from inner-join in the way that it only outputs tuples from the left sub-relation that match with at least a tuple from the right sub-relation. The cardinality of this operator is estimated similarly as inner-join.

Outer-join and anti-join: Estimating cardinality of *outer-join* and *anti-join* output is more challenging. One fundamental difference between outer/anti-join from inner/semi-join is they output tuples that do not meet the join condition. We use left outer-join as an example, and right/full outer-join or anti-join can be handled similarly. Left outer-join, besides the matched tuples, also output tuples from the left sub-relation that are not matched. We denote them as *unmatched tuples*. Estimating the cardinality of matched tuples is similar to inner/semi-join, and here we focus on the cardinality of unmatched tuples. We discuss how to estimate cardinality when inputs come from the left and right sub-relation:

- If input comes from the left side, we need to estimate the probability of one input tuple not matching all tuples of the right sub-relation. Assume that the probability that an insert is matched with one right sub-relation tuple is p_l. Then the probability of an inserted left tuple not matching with any tuples in right sub-relation is $(1-p_l)^{|R|}$ where $|R|$ is the size of the right sub-relation. Thus, the cardinality of unmatched inserts is $\mathbb{C}^I_{L\mathtt{IN}} \times (1-p_l)^{|R|}$. The cardinality of unmatched deletes is the same, and an update can be treated as an insert plus a delete.
- If input comes from the right side, it could turn a left tuple from *matched* to *unmatched* or vice versa. Assume a right tuple is an insert, it changes a left tuple from unmatched to matched if the left one has no match so far, and the two tuples match together. Assume the probability of a right insert matching with one left tuple is p_r and there are $|L|$ tuples for left sub-relation. So the number of tuples in L that match with this insert tuple is $|L| \times p_r$. Among these matched tuples in L, we further consider whether they do not have matches before (and thus the current insert tuple is their first match). Recall that the probability of one tuple on the left side not having any matches for R is $(1-p_l)^{|R|}$. So among $|L| \times p_r$, the number of tuples that do no have matches before and we need to delete is $|L| \times p_r \times (1-p_l)^{|R|}$. For the deletes from right, they may flip left tuples from matched to unmatched status, and thus emit insert outputs for these unmatched tuples. The cardinality of such inserts can be estimated similarly, and an update can be treated as an insert plus a delete.

3.2 Computing Incrementability with a Cost Model

We now discuss how to utilize the cost model to compute incrementability. Recall that given two pace configurations P_1 and P_2, $\mathtt{INC}(P_1, P_2) = \frac{C_F(P_1)-C_F(P_2)}{C_T(P_2)-C_T(P_1)}$. So we focus on how to estimate $C_F(P)$ and $C_T(P)$ for a given pace configuration P. Cost estimation for a pace configuration is challenging for two primary reasons. First, the paces of a parent query path and its child query path may be different. Here, the parent query path needs to know the correct input cardinality from child query path to estimate the cost of its incremental executions. Second, a join operator may have two input query paths with different paces. So the join operator will interleave the incremental executions of different input query paths. One incremental execution of one query path impacts the state information for the other query path. The challenge here is how to estimate the cost for interleaved incremental

Algorithm 1: Computing $C_T(P)$ and $C_F(P)$.

```
1  (K_1, K_2, ··· , K_Q) ← (0, 0, ··· , 0)
2  for m ← 1 to M do
3      I_Global ← m/M
4      PathSet ← ∅
5      for i ← 1 to Q do
6          if I_Global − K_i/P_i ≥ 1/P_i then
7              Add path i to PathSet
8      end
9      for i ∈ PathSet do
10         K_i ← K_i + 1
11     end
12     cost ← Estimated cost of a simulated execution
13       that involves flushing buffers of paths in PathSet
14     C_T(P) ← C_T(P) + cost
15     if I_Global = 100% then
16         C_F(P) ← cost
17 end
```

executions of input query paths. We approach the two challenges by simulating the process of incremental executions based on a pace configuration.

We first discuss how to estimate the cost for an incremental execution of a query path. Recall that an incremental execution of a query path takes all input buffered tuples and flush them all the way to the end of this query path. Here, we use our cardinality estimation methods to recursively compute the cardinalities of each operator in this query path for an incremental execution and use cost functions to convert the cardinalities into cost. After, we also update the state information for each operator based on their input cardinality (e.g. updating the number of input tuples for a join operator).

Given that we know how to estimate the cost for a single query path, we now discuss computing $C_T(P)$ and $C_F(P)$ for the pace configuration P. Since we assume base relations have steady arrival rates, we use a global indicator $I_{Global} \in [0, 1]$ to represent the data arrival progress of all input data. For example, $I_{Global} = 50\%$ means 50% of total data has arrived. Assuming that P has Q query paths, we additionally include an array $K = (K_1, K_2, \cdots, K_Q)$ to record how many times each query path has simulated flushing its input buffer. Algorithm 1 shows the algorithm of computing $C_T(P)$ and $C_F(P)$. We simulate the continuous data arrival process in a discrete way, which includes M steps, where each step represents $\frac{1}{M}$ of total data. Here, M is the maximally allowed pace. After m steps, the current progress I_{Global} is $\frac{m}{M}$. For each simulation step, we need to find query paths that should be triggered to flush their input buffers. Given a query path with pace P_i, it flushes its buffer if at least another $\frac{1}{P_i}$ of new data arrives since its last flush (i.e. $\frac{K_i}{P_i}$), that is, $I_{Global} - \frac{K_i}{P_i} \geq \frac{1}{P_i}$. After we find the set of paths (i.e. $PathSet$), we estimate the cost of a simulated execution that involves flushing buffers for query paths in $PathSet$. We add this cost to $C_T(P)$. When I_{Global} reaches 100%, all query paths flush the buffers and the cost is $C_F(P)$.

Complexity analysis: We note that each operator has a cost estimation function. We use the number of cost functions being invoked to quantify the complexity. The worst case of our simulation is all paces for a pace configuration are M, the maximally allowed pace. This means for each simulation step, we need to invoke cost functions for all query paths and thus all operators. Assuming the number of operators in a query is N. Note that the number of simulation steps is M. So in the worst case, the simulation algorithm needs to run $O(N \times M)$ numbers of cost estimation functions.

4 INCREMENTABILITY-AWARE QUERY PROCESSING

In this section, we discuss how to utilize incrementability to find a pace configuration for a query to make a better tradeoff between final work and total work than the approach of assigning a uniform pace for the whole query. Specifically, given the same target final work InQP uses less total work, or given the same target total work, InQP uses less final work. The basic idea is to execute query paths with higher incrementability more eagerly (i.e., higher pace) and query paths with lower incrementability more lazily (i.e., smaller pace). We consider two optimization tasks: we choose a pace configuration to 1) minimize total work given a maximum final work constraint, or 2) minimize final work given a maximum total work constraint. We focus on the first problem and discuss the second later.

4.1 Problem Formalization

We define final work constraint L as the ratio between the final work users want to achieve and the final work of executing the query in one batch, where $L \in [0, 1]$. Consider an example of a final work constraint $L = 0.3$. The pace configuration for batch processing is $P_{\mathbb{1}}$. If we increase pace configuration of $P_{\mathbb{1}}$, we decrease final work. When we reach 30% of the final work of $P_{\mathbb{1}}$, we meet the constraint $L = 0.3$. The problem is formally stated as:

$$\begin{aligned} \underset{P}{\text{minimize}} \quad & C_T(P) \\ \text{subject to} \quad & C_F(P) \leq L \times C_F(P_{\mathbb{1}}) \\ & P_i \leq P_j, \forall j \in \text{children}(i) \end{aligned}$$

The constant L is specified by the user, which indicates the maximally allowed final work. Query path j is the direct child of query path i: its output tuples are query path i's

Algorithm 2: Greedy algorithm of selecting pace configuration for query paths by minimizing total work with a final work constraint L.

```
1  P ← P_𝟙
2  while true do
3      i ← argmax_{i:P_i < P_j, ∀j∈children(i)} ∂_i(P)
4      P_new ← P_[i\P_i+1]
5      if C_F(P_new) ≤ L × C_F(P_𝟙) then
6          return P_new
7      if P = P_∞ or ∂_i(P) < 0 then
8          return P
9      P ← P_new
10 end
```

input. We specifically require $P_i \le P_j$ so query path i always has the necessary input data to process.

4.2 Greedy Algorithm

We can solve the optimization problem by enumerating all possible pace configurations and find the pace configuration that satisfies the final work constraint and has the lowest total work. However, this approach has exponential complexity and is very time-consuming as shown in our experiments (Section 5.6). Instead, we design a greedy algorithm that leverages incrementability to reduce the search space and still generates a query plan that has low total work.

The greedy algorithm starts with a pace configuration $P_{\mathbb{1}}$, where total work is the smallest. When we increase the pace configuration P, we increase total work, but decrease final work. The algorithm stops when we first meet the final work constraint. The intuition of our algorithm is that given we need to meet the final work constraint $L \times C_F(P_{\mathbb{1}})$, we want to increase pace for the query path that decreases the most final work per unit of total work increased, so that we can best utilize the additional total work.

We find that when we increase a pace configuration for a query from P_1 to P_2, the ratio between the decreased final work (i.e. $C_F(P_1) - C_F(P_2)$) and the increased total work (i.e. $C_T(P_2) - C_T(P_1)$) is the definition of incrementability. Intuitively we should always increase the pace for the query path with the highest incrementability, so query paths with higher incrementability are executed more eagerly, while query paths with lower incrementability are executed more lazily. We notice that as we increase the pace for a query path, its incrementability changes. Thus, we increase at the minimum granularity and recompute the incrementability after each step. We formalize this approach as follows. For a pace configuration P, we denote its *marginal incrementability at query path i* as

$$\partial_i(P) = \mathtt{INC}(P_{[i\backslash P_i+1]}, P)$$

where $P_{[i\backslash c]}$ represents another pace configuration by replacing the i-th query path's pace by c. In short, $\partial_i(P)$ represents the incrementability of increasing i-th query path's pace by 1. We note that if we increase a pace configuration from P_1 to P_2 and both final work and total work can increase (i.e. non-incrementable case) we should never increase the pace no matter whether we currently meet the final work constraint. This mainly happens when a pace configuration has very large paces. We also set a maximum pace configuration $P_\infty = (M, M, M, \ldots, M)$, where M is the maximally allowed pace for each query path.

Our algorithm is illustrated in Algorithm 2. We start with the initial pace configuration $P_{\mathbb{1}}$. We search for the query path i that gives the highest marginal incrementability and is also feasible to increase (i.e. strictly less than all children paces). At each search step, we increase it by 1 and terminate when one of the three conditions is met: 1) the increment first meets the constraint; 2) the incrementability is less than 0 (i.e. non-incrementable); 3) we reach P_∞.

Optimizing final work: Previously we discuss how to optimize total work given a final work constraint. Now we consider how to minimize final work given a total work constraint. The algorithm is similar to the previous one. We start with pace configuration $P_{\mathbb{1}}$. We increase pace for query paths unless decrements violate the total work constraint. Each time we choose the query path i with the *maximal* incrementability. We terminate the algorithm if one of the three conditions is met: 1) a decrement violates the constraint; 2) we reach the maximum allowed pace configuration P_∞; 3) the incrementability is less than 0.

Complexity analysis: Assuming that we have Q query paths, for each step we need to compute the incrementability for all of them. Combined with the complexity of computing incrementability, the complexity for each step is $O(Q \times N \times M)$. This greedy algorithm runs at most $Q \times M$ steps, so in total the greedy algorithm needs to run $O(Q^2 \times N \times M^2)$ number of cost estimation functions. We test its overhead in Section 5.6.

4.3 Applicability of InQP

InQP can be applied to systems that support incremental view maintenance, such as Spark [2], Flink [8], and a PostgreSQL modified for incremental executions [40]. For systems that only support insert tuples (e.g. Spark), operators need to be modified to support deletes and updates (Section 3.1.2). In addition, to control the execution frequencies of different query paths the system needs a mechanism to pause and start the execution of a query path. Recall that we break a query plan tree into query paths at the blocking

operators. Therefore, a query path either starts at a scan or a blocking operator. If we choose to pause a query path, a scan operator buffers input tuples, and a blocking operator processes the input tuples, but delays pushing changes to the query path. To start the execution of a query path, we include a variable into the query path's starting operator (i.e. a scan or blocking operator), which indicates whether this query path is executable or not. InQP is responsible for setting this variable to control which query paths to execute or delay.

5 EXPERIMENTS

Our experimental study addresses the following questions:

- Compared to an incrementability-oblivious approach, which uses a uniform pace for a single query, and an approach that processes input tuples for leaf nodes at different paces [24], how much computing resources does InQP save given the similar query latency goal? (Section 5.3)
- How do the accuracy of our cost model and bursty workloads impact InQP's performance? (Section 5.4)
- What is the accuracy of our cardinality estimation compared to PostgreSQL? (Section 5.5)
- What are the overhead and benefits of the greedy algorithm of InQP? (Section 5.6)

We evaluate InQP in one server with 196 GB of main memory and two Intel Xeon Silver 4116 processors, each with 12 physical cores. For all experiments, we use 20 physical cores and the rest for the OS (Ubuntu 18.04).

5.1 Prototype Implementation

We implement InQP in Spark 2.4.0 [2], and extend Structured Streaming to support deletes and updates based on existing IVM algorithms [14] and support incremental execution based on a pace configuration. We reduce the cost of starting Spark jobs for each incremental execution based on techniques in Venkataraman et al. [43]. We use a Kafka [1] cluster on a different machine with the same hardware configuration as the data source of Spark queries.

Users submit a SQL query to Spark and InQP maintains the query results with a stream of data loaded from Kafka. Our evaluation focuses on the optimization of minimizing total (or extra) work, but nothing in our solution prohibits the dual optimization. For our system, users specify a final work constraint that indicates the percentage of final work to reduce to compared to the final work of executing the query in one batch, and InQP finds a pace configuration to minimize total work. For example, a constraint of 0.02 means users want to reduce the final work to 2% of batch processing's final work. This optimization explores the trade-off between resource consumption and query latency. In this experiment, we use *additional time* to represent the resource consumption invested into incremental executions. It is defined as the total query processing time for all incremental executions minus the time of executing the query in one batch. A query's *latency* is defined as the time of the final incremental execution or the processing time if the query is executed in one batch.

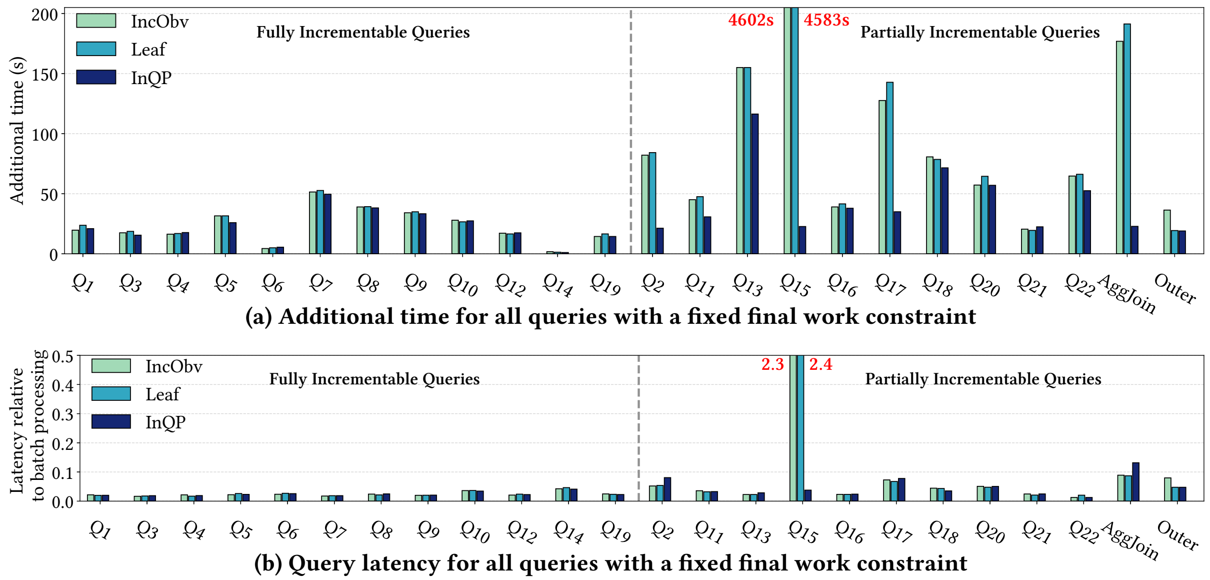
We use the Spark SQL optimizer to generate a physical query plan for the submitted query and decompose the query plan into query paths. After, InQP determines the pace configuration of this query plan for computing the query result with respect to the performance constraint.

5.2 Experiment Setup

We use the TPC-H benchmark in our experiments, and our prototype supports all 22 TPC-H queries, where 10 of them are not fully incrementable. We additionally write 2 queries based on the TPC-H schema to test partially incrementable parts caused by individual operators including aggregate operators and outer-join operators. The 2 queries are shown in the following:

```
Q_AggJoin: SELECT AVG(avg_price)
           FROM customer c,
                (SELECT o_custkey,
                        AVG(o_totalprice) avg_price
                 FROM orders GROUP BY o_custkey) agg_o
           WHERE c.c_custkey = agg_o.o_custkey
Q_Outer: SELECT COUNT(*) FROM part
         LEFT JOIN partsupp on p_partkey = ps_partkey
         JOIN lineitem on p_partkey = l_partkey
         JOIN orders on l_orderkey = o_orderkey
```

where Q_AggJoin joins an aggregate operator with a base table, and Q_Outer is a left-outer-join with two equal-joins. We preload the full dataset into Kafka, but let InQP pull data from Kafka at a data rate of 1GB/min. We generate datasets that are large enough to show the performance impact of partially-incrementable parts. Using a single large scale factor results in some queries running out of memory on the test machine. The reason is that the table `Lineitem` in TPC-H occupies more than 70% of the data. Queries that involve `Lineitem` have significantly larger data to process than queries that do not involve `Lineitem`. To make sure that every query has enough data and does not run out of memory, we generate two datasets: scale factor 100 and 10, where the former is used for queries that do not access `Lineitem` (i.e. Q2, Q11, Q13, Q16, Q22, Q_AggJoin), and the latter is used for the rest queries. While we only show insert-only workloads, operators within a query plan can generate deletes and updates. We also evaluated a workload with mixed inserts and deletes and find similar performance to the insert-only workload: InQP has a much lower resource consumption and similar latency compared to the baselines. To simulate prior executions, we calibrate our cost model statistics with several warm up runs. We show how the quality of statistical



(a) Additional time for all queries with a fixed final work constraint

(b) Query latency for all queries with a fixed final work constraint

Figure 5: Additional time and query latency for a final work constraint. It is set to 0.02 for a query if the cost model finds the query can meet the constraint, otherwise we use constraint 0.05 (i.e. Q17, Q_AggJoin, and Q_Outer).

information impacts InQP in Section 5.4. We set the max pace for a query path to 100. In our experiments, we run each test three times and report the average.

We compare InQP with an *incrementability-oblivious* baseline (**IncObv**), which is a mini-batch approach in Spark that uses a single pace value for all query paths of a query. For this approach, we search over uniform paces using InQP's cost model to find a pace configuration we estimate to meet the performance constraint. We also evaluated against a strategy with a hand-tuned single pace, where the pace is the inverse of the final work constraint (e.g. for constraint 0.02, we use pace 50). We test 5 constraints (0.5, 0.2, 0.1, 0.05, and 0.02) for TPC-H queries and find that the hand-tuned approach meets the target query latency constraint in only 8% of the time, compared to 64% for InQP and 38% for IncObv. Additionally, the trade-off between latency and resource consumption between the hand-tuned approach and IncObv is the similar as both use the a uniform pace configuration. Therefore, in our experiments we only include the results of IncObv. We note that it is possible the cost model estimates that some queries cannot meet the final work constraint 0.02. In this case, we do not report the results of final work constraint 0.02 and use constraint 0.05 instead. These queries include Q17, Q_AggJoin, and Q_Outer.

5.3 Low Resource Consumption with Similar Latency

In this subsection, we examine how much InQP lowers resource consumption compared with similar query latencies for IncObv. We use final work constraints (1.0, 0.2, 0.05, 0.02), and minimize the additional time. Recall that final work constraint is the percentage of final work we want to reduce to compared to the final work of executing the query in one batch. Inspired by prior work [24], we consider an alternative approach, **Leaf**, where query paths are made from the leaf nodes (scans) to the root node. As the original paper considers a different optimization (i.e. minimizing the work to refresh a stale view), Leaf uses InQP's cost model and greedy algorithm to find a pace configuration to minimize total work and satisfy a final work constraint. We discuss the difference between InQP and this work in the related work.

We test all 24 queries, and report additional time and the ratio of query latency to the latency of executing a query in a batch for a fixed final work constraint 0.02. If the cost models find it is impossible to meet this constraint, we use the constraint of 0.05, which occurs for Q17, Q_AggJoin and Q_Outer. Figure 5 shows the results of all 24 queries. Figure 5a shows that InQP has much lower additional time and similar query latency compared to IncObv and Leaf for not-fully incrementable queries (right of the vertical dashed line). Specifically, InQP uses as low as 1.5% of additional time compared to IncObv and Leaf for the same final work constraint (i.e. Q15). For fully-incrementable queries (left of the vertical dashed line), InQP has a similar additional time and query latency to IncObv and Leaf. We note that Leaf has similar additional time to InQP when we test Q_Outer. The partially incrementable parts for Q_Outer come from its left-outer-join operator. To reduce the cost of partially incrementable parts, both InQP and Leaf consider flushing tuples for base relations at different paces. Therefore, they

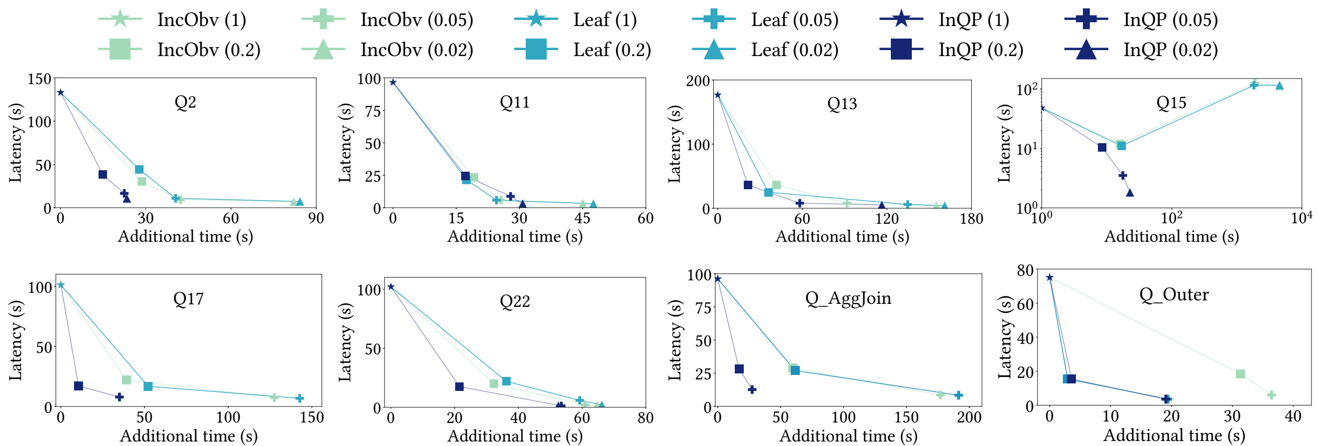


Figure 6: Trade-off between resource consumption and query latency under different final work constraints.

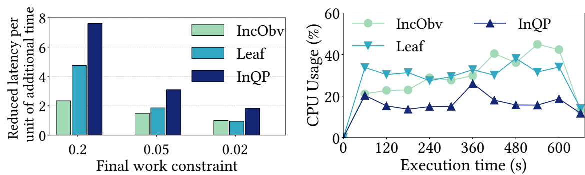


Figure 7: Reduced latency per unit of additional time.

Figure 8: CPU usage trace (Q17, constraint = 0.05).

have similar pace configurations for flushing input tuples of base relations and have similar additional time.

We report additional time and query latency with different final work constraints, which are in Figure 6. For each final work constraint, we compare their additional time and query latency. For the same final work constraint we use the same point shape for all approaches, which is highlighted in the legend. For the same shape note that InQP has similar latency but with much lower additional time. If a query cannot meet the final query constraint based on the cost model, we do not show its results. Here, we see that InQP uses much less resource consumption with a similar query latency compared to IncObv and Leaf, especially when the constraint is low (e.g. 0.05 or 0.02). InQP makes a better trade-off for these queries because we selectively increase the pace of query path with higher incrementability. Consider Q17 as an example, it includes an aggregate operator that joins with two relations (i.e. `Lineitem` and `Part`). When aggregated values change, it needs to output the new values and delete the old ones. The tuples inserted, but deleted later, need to join with `Lineitem` and `Part`, but do not contribute to the final query result. InQP delays outputting the updated values for the aggregate operator to reduce additional time, and eagerly executes other operators to meet the final work constraint.

In addition, we find in Q15 IncObv and Leaf have a higher query latency when we reduce the final work constraint. Q15 has two aggregate operators, where the parent aggregate operator is a `max` without group-by and the child aggregate operator is a `sum` with a group-by statement. So the child aggregate operator `sum` will update the `sum` value per group and the max value in the parent `max` operator. In this case, we need to sort all input values for the `max` operator to find the new max value. When we set a lower final work constraint, the cost model tends to increase the pace (i.e. higher number of incremental executions), which increases the chance of updating the max value. So IncObv has higher query latency when the constraint is low. While Leaf is able to tune the frequencies of flushing tuples for base relations, it cannot delay the incremental executions for aggregate operators, which makes it has similar performance to IncObv. Specifically, the case of updating the max value of the `max` operator happens when we set the constraint to 0.05 and 0.02.

To highlight the cost-effectiveness of InQP, we report the ratio between the reduced query latency compared to batch processing and the additional time. The higher the ratio is, the more latency we reduce per unit of additional time we invest. Figure 7 shows the average ratio of all queries in Figure 6 by constraint. This figure shows InQP is more thrifty at utilizing computing resources to reduce query latency, especially when the final work constraint is larger. An interesting question to explore is how systems could expose such information to users, so they can make decisions about the trade-off—especially in pay-per-use environments.

We also report the trace of CPU usage during query processing to show how InQP reduces computing resources with similar query latency to IncObv and Leaf. We report the average CPU usage every 60s for Q17 with the final work constraint as 0.05 in Figure 8. Q17 uses the 10GB dataset and the whole data loading process takes 600 seconds (data rate of 1GB/min). Figure 8 shows that InQP has lower CPU consumption than IncObv and Leaf, which reduces total time of query processing. We also trace, but do not show, the I/O

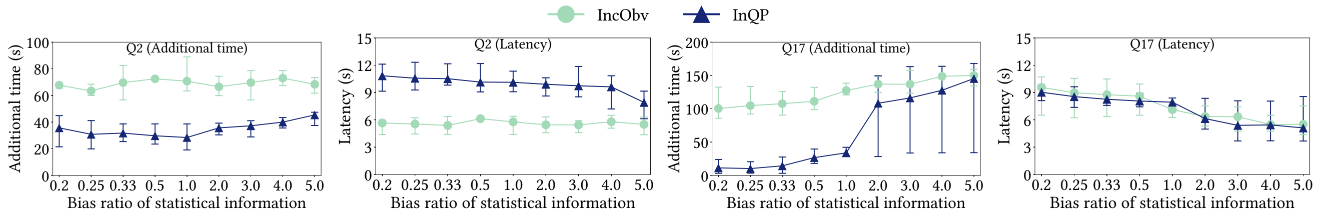


Figure 9: Performance impact of biased statistical information.

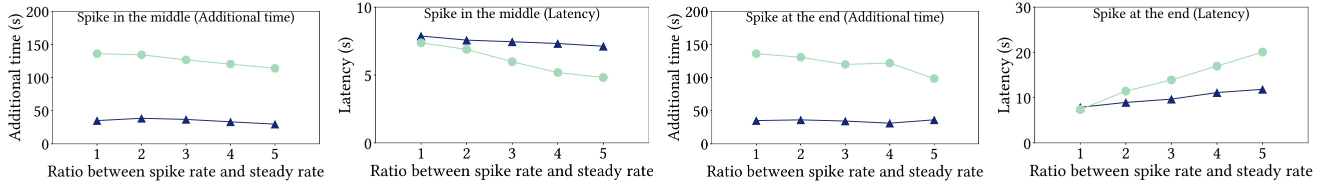


Figure 10: Performance impact of a bursty arrival rate (Q17).

operations for Q17 and find that InQP has 47% and 53% less I/O operations compared to IncObv and Leaf respectively.

5.4 Performance Impact of the Accuracy of Cost Model and Bursty Workloads

InQP utilizes statistical information of previous executions to build a cost model and uses the cost model to compute incrementability to decide the pace configuration. Examples of collected statistical information include selectivity for joins and select operators, and number of groups for aggregate operators. In this subsection, we first test the performance impact of biased statistical information on InQP and IncObv. Note that the performance of IncObv is affected by the statistical information because we use the cost model to determine its pace configuration. After, we test how bursty workloads impact the performance of InQP. We use Q2 with the final work constraint 0.02, and Q17 with the constraint 0.05.

For the first experiment, we apply a bias ratio to the statistical information we collect. The bias ratio represents the ratio between the biased statistical information and the one we collect. For example, if the collected selectivity is 0.1 and the bias ratio is 0.2, the biased selectivity is 0.02. We consider two cases: overestimation and underestimation. For the first one, we vary the bias ratio from 2.0 to 5.0 with an interval 1.0. For a given ratio R, we allow each operator chooses a random ratio between [1.0, R]. For the underestimation case, we use the ratio {0.5, 0.33, 0.25, 0.2} to represent that we underestimate by a factor of 2, 3, 4, 5 respectively. For each given bias ratio, we test 10 times and report the average, minimum, and maximum additional time and query latency.

We show the results of Q2 and Q17 in Figure 9. We see that for Q2, with the value of bias ratio increasing InQP has higher resource consumption, but lower query latency. The reason is that high bias ratio makes the cost model schedule incremental executions more eagerly. For InQP, it needs to schedule the non-incrementable parts more frequently to meet the final work constraint, which increases the additional time of executing the query and decreases the query latency. However, in an extreme case (i.e. bias ratio = 5.0) InQP has lower additional time and similar query latency compared to IncObv. For Q17, we have similar observation to Q2. When we overestimate, the additional time increases and and the query latency decreases for both approaches. The average additional time of InQP is lower than IncObv when bias ratio is no larger than 4.0. When the bias ratio reaches 5.0, both approaches have similar additional time. These experiments show that biased statistical information could increase additional time of InQP, and makes the performance of InQP similar to the performance of IncObv.

We now report the performance impact of bursty workloads. We decide the paces for InQP and IncObv assuming a steady rate of 1GB/min. We generate bursty workloads by introducing a spike in the data arrival. We vary the ratio between the spike rate and steady rate from 1 to 5. Note that we load the same amount of data, so when we increase the spike rate, data rates of other periods drop. The whole data loading process takes 10 mins. We use Q17 and set the time span of the spike rate to 1 min. We test two cases where the spike is in the middle or at the end (i.e. the last min) of the data loading processing.

Figure 10 shows the test results. We see that the spike in the middle does not change additional time of InQP, but slightly decreases its latency (i.e. the two leftmost figures in Figure 10). On the other hand, both additional time and latency drop for IncObv. The reason for both approaches having lower latency is that when the spike rate increases in the middle, the data rate at the end drops. Compared to IncObv, InQP has higher latency than IncObv because it delays some partially incrementable work to the end. Additional time drops for IncObv because with the spike rate increasing, more data are processed in one batch for the spike and thus

	Q2	Q11	Q13	Q15	Q16	Q17	Q18	Q20	Q21	Q22
InQP	**-15.5%**	-28.6%	**41.7%**	**-9.1%**	**-36.8%**	**7.4%**	**-0.1%**	**-3.4%**	**-2.0%**	**2.9%**
PostgreSQL	-53.2%	**-13.9%**	-89.2%	-85.0%	-71.6%	-45.3%	-7.5%	-99.7%	-45.1%	-85.1%

Table 1: Accuracy of cardinality estimation of InQP and PostgreSQL for incremental executions.

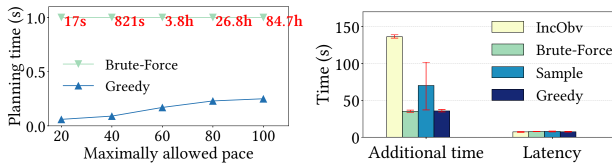


Figure 11: Planning time. **Figure 12: Q17 Performance.**

IncObv does less incremental work. This reduces the cost of partially incrementable parts of IncObv. Nevertheless, InQP has a much lower additional time than IncObv. If the spike is at the end, the query latency for both InQP and IncObv increases because they do not expect a high arrival rate in the last minute (i.e. the two rightmost figures in Figure 10). We see that InQP has a lower latency compared to IncObv since InQP executes its incrementable parts eagerly and has lower work for the last mini-batch, which makes it less sensitive to the higher rate.

5.5 Cardinality Estimation Accuracy Compared to PostgreSQL

We evaluate the accuracy of our cardinality estimation for incremental executions and compare it with PostgreSQL's estimation. Note that we choose PostgreSQL because it supports a wide range of complex queries and existing cardinality estimation for incremental executions [44] only supports select-project-join queries. In InQP, we need to support complex queries such as the query in Figure 2 that involves an aggregate operator in the query plan tree. For PostgreSQL, we obtain the estimated cardinality for each incremental execution in three steps: 1) we first use its batch-based cost model to estimate the cardinality for existing data; 2) we then insert input data for the incremental execution into the base relations and obtain the cardinality for new data; 3) finally, we use the difference of two estimated cardinalities as the cardinality for this incremental execution. We use a pace of 100 incremental executions and test the partially incrementable queries in TPC-H. We use our best effort to adjust the Spark SQL query plan to make sure that a query running on Spark and PostgreSQL has exactly the same physical plan for this experiment. Only Q20 has a different plan on the two systems because PostgreSQL enforces the index scan for Q20, but Spark only uses sequential scan. We collect the ground truth by running queries on Spark.

We compute the accuracy of cardinality estimation as $\text{Accuracy} = \frac{Estimated - Ground\ Truth}{Ground\ Truth}$. If the value is 0, it means the estimation is the same as the ground truth. If the value is positive (i.e. estimated cardinality > ground truth), it represents the case of overestimation. On the other hand, a negative value represents the case of underestimation. We report the average accuracy of all 100 incremental executions. Table 1 shows that InQP has a more accurate estimation compared to PostgreSQL in all queries except Q11, and in some queries (e.g. Q21 and Q22) utilizing cardinality estimation of PostgreSQL could be very inaccurate.

5.6 Overhead and Benefits of InQP's Greedy Algorithm

We evaluate the planning time of InQP's greedy algorithm (`Greedy`) and compare it with a brute-force method (`Brute-Force`) that enumerates all possible pace configurations to find a plan that has the lowest total work while meeting the final work constraint. We vary the maximally allowed pace from 20 to 100 and report the planning time. We use final work constraint 0.01 to force Greedy to take the maximum number of search steps, as Greedy takes less planning time with a larger constraint. Figure 11 shows Greedy has much lower planning time than Brute-Force for Q17 as Greedy leverages the key metric incrementability to largely prune the search space. We test all queries and find the maximum planning time is 640ms and the 80th percentile is 340ms.

We test query latency and additional time of the generated query plan for the above methods, and include a sampling method (`Sample`) that randomly samples pace configurations and selects the one with the lowest total work while meeting the final work constraint. We allow Sample to run the same planning time as Greedy. We test Q17 with final work constraint 0.05 for 10 times, and report the mean, min, and max query latency and additional time. Figure 12 shows that Greedy has similar performance to the optimal plan generated by Brute-Force, and that the additional time of Sample varies. While Sample can find a plan that has similar performance to Brute-Force, it has much larger additional time in the average and worst case compared to Greedy and Brute-Force. The two experiments show that our greedy algorithm can find a good plan with limited planning time.

6 RELATED WORK

Incremental View Maintenance Algorithms Materialized views are cached or pre-computed query results that are derived from base tables [14]. When base tables are updated,

incremental view maintenance (IVM) algorithms incrementally incorporate new data into the prior view without recomputing the view from scratch. Larson et al. [6] introduces IVM algorithms for select-project-join (SPJ) views. Later work proposes new IVM algorithms for more complex queries such as maintaining views with negation and aggregate operators [22, 23], supporting recursive views and nested subqueries [23, 34, 47], and optimizing incremental executions for semi-join, outer join, and acyclic joins [21, 26, 28]. We believe that these algorithms are orthogonal to InQP. InQP does not propose new IVM algorithms, but execute different parts of a query at different frequencies based on incrementability.

View Materialization Policies There are several different policies for maintaining a materialized view to makes different trade-offs between view maintenance cost and query latency [17]. *Immediate view maintenance* updates the view whenever base tables are updated [3, 11, 13]. This approach lowers query latency with higher cost of view maintenance. On the other hand, a *deferred view* [16] does not update the view immediately, but defers view maintenance to some future point such as when the view is queried or when the system has free cycles for view maintenance [48]. InQP is different from these approaches in that it decomposes a query into multiple query paths and assign each query path a different pace based on their incrementability, while existing IVM approaches use a uniform execution pace for maintaining the whole query.

He et al. [24] observes the asymmetric maintenance cost for different access methods (e.g. index scan or sequential scan). Therefore, they propose to process modifications of different base relations at different batch sizes. This work focuses on SPJA queries, but InQP considers more complex queries, such as outer-joins. In addition, InQP decomposes the query plan into query paths that offer more fine-grained control flow compared to this work, which only considers paths from a leaf to the root.

Continuous query and stream systems Many continuous query processing and stream systems adopt IVM as its query execution engine to provide low query latency [2, 8, 10, 11, 13]. These systems often provide a trade-off between query latency and computing resource consumption by allowing users to adjust the amount of tuples to be processed for each incremental execution [2, 8, 10]. Several projects focus on finding query plans or execution plans to optimize different performance metrics, such as maximizing output rate [45], minimizing per-tuple processing latency [9], lowering memory consumption [5, 9], producing fast early results [41], or a mix of these metrics [4, 37].

However, all these works are limited in SQL support and only allow select-project-join-aggregate queries. For complex queries, they do not consider the semantics (e.g. outer join) that make the query not fully incrementable. InQP is different from the aforementioned works in that it supports complex queries and exploits the knowledge of diverse levels of incrementability within a query to execute different parts of a query at different paces.

Cardinality Estimation Conventional databases [29, 36] use statistical information (e.g. selectivity or number of distinct values) collected from base tables, to estimate cardinalities. Several techniques, such as data sketching [7, 18], index sampling [30], sampled executions [42], and leveraging runtime execution information [12], are proposed to improve or bound the accuracy of cardinality estimation. Different from statistics-based cardinality estimation, some recent works consider leveraging machine learning techniques to more accurately estimate cardinality [27, 32, 33, 35, 38]. However, all these research works are focused on the context of batch processing and are limited in the estimation for incremental executions. We also find that several works focus on estimating cardinality or statistics for incremental executions [19, 44]. Viglas et al. [44] introduces rate-based cardinality estimation to estimate output data rate of each operator in a continuous query. But this work only considers select-project-join operators and does not address the problem of estimating cardinalities for deletes or updates. Cardinality estimation in InQP is different from these works as it uses different estimation methods based on operation semantics (i.e. insert, delete, and update), which can more accurately compute the cardinalities for incremental executions.

7 CONCLUSION

We present InQP as a new query processing paradigm that models how amenable a query is for incremental execution and uses fine-grained control flow to schedule more incrementable parts (e.g. dataflow paths) eagerly for efficient query execution. We propose a metric, incrementability, to quantify the cost-effectiveness of incremental executions, a cost model for computing incrementability, and a greedy solution for minimizing additional work for incremental execution subject to a final work goal (i.e. latency). We implement an InQP prototype in Spark and demonstrate that compared to a baseline using coarse-grained scheduling (via mini-batch size), InQP reduces final work up to 3.3x per unit of additional work.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their valuable feedback. This work was supported by NSF Award CCF-1139158, a Google DAPA Research Award, the CERES Center for Unstoppable Computing, a US Army TATRC Research Award, and Intel.

REFERENCES

[1] Apache kafka. https://kafka.apache.org/.

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

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

[4] A. Ayad and J. F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 419–430, 2004.

[5] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain : Operator scheduling for memory minimization in data stream systems. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 253–264, 2003.

[6] 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.

[7] W. Cai, M. Balazinska, and D. Suciu. Pessimistic cardinality estimation: Tighter upper bounds for intermediate join cardinalities. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 18–35, 2019.

[8] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[9] D. Carney, U. Çetintemel, A. Rasin, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, pages 838–849, 2003.

[10] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proc. VLDB Endow.*, 8(4):401–412, 2014.

[11] 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.

[12] S. Chaudhuri, V. R. Narasayya, and R. Ramamurthy. Estimating progress of long running SQL queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 803–814, 2004.

[13] 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.

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

[15] Y. Chung, M. L. Mortensen, C. Binnig, and T. Kraska. Estimating the impact of unknown unknowns on aggregate query results. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 861–876, 2016.

[16] 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.

[17] 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.

[18] C. Estan and J. F. Naughton. End-biased samples for join cardinality estimation. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 20, 2006.

[19] L. Gao, M. Wang, X. S. Wang, and S. Padmanabhan. A learning-based approach to estimate statistics of operators in continuous queries: a case study. In *Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery, DMKD 2003, San Diego, California, USA, June 13, 2003*, pages 66–72, 2003.

[20] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)*, 25(2):73–169, 1993.

[21] T. Griffin and B. Kumar. Algebraic change propagation for semijoin and outerjoin queries. *SIGMOD Record*, 27(3):22–27, 1998.

[22] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, pages 328–339, 1995.

[23] 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.

[24] H. He, J. Xie, J. Yang, and H. Yu. Asymmetric batch incremental view maintenance. In K. Aberer, M. J. Franklin, and S. Nishio, editors, *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, pages 106–117. IEEE Computer Society, 2005.

[25] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In J. Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 171–182. ACM Press, 1997.

[26] 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.

[27] S. Krishnan, Z. Yang, K. Goldberg, J. M. Hellerstein, and I. Stoica. Learning to optimize join queries with deep reinforcement learning. *CoRR*, abs/1808.03196, 2018.

[28] P. Larson and J. Zhou. Efficient maintenance of materialized outer-join views. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 56–65, 2007.

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

[30] V. Leis, B. Radke, A. Gubichev, A. Kemper, and T. Neumann. Cardinality estimation done right: Index-based join sampling. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[31] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for local queries. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 28-30, 1986*, pages 84–95, 1986.

[32] T. Malik, R. C. Burns, and N. V. Chawla. A black-box approach to query cardinality estimation. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 56–67, 2007.

[33] R. C. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A learned query optimizer.

Proc. VLDB Endow., 12(11):1705–1718, 2019.

[34] 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.

[35] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. An empirical analysis of deep learning for cardinality estimation. *CoRR*, abs/1905.06425, 2019.

[36] 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.

[37] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Algorithms and metrics for processing multiple heterogeneous continuous queries. *ACM Trans. Database Syst.*, 33(1):5:1–5:44, 2008.

[38] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - db2's learning optimizer. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 19–28, 2001.

[39] R. Taft, N. El-Sayed, M. Serafini, Y. Lu, A. Aboulnaga, M. Stonebraker, R. Mayerhofer, and F. Andrade. P-store: An elastic database system with predictive provisioning. In *Proceedings of the 2018 International Conference on Management of Data*, pages 205–219. ACM, 2018.

[40] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin. Intermittent query processing. *Proc. VLDB Endow.*, 12(11):1427–1441, July 2019.

[41] Y. Tao, M. L. Yiu, D. Papadias, M. Hadjieleftheriou, and N. Mamoulis. RPJ: producing fast join results on streams through rate-based optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 371–382, 2005.

[42] I. Trummer. Exact cardinality query optimization with bounded execution cost. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 2–17, 2019.

[43] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 374–389, 2017.

[44] S. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, pages 37–48, 2002.

[45] S. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, pages 285–296, 2003.

[46] 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.

[47] 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.

[48] 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.