THE UNIVERSITY OF CHICAGO

THRIFTY QUERY PROCESSING

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY

DIXIN TANG

CHICAGO, ILLINOIS
DECEMBER 2020

Dedicated to my parents and my wife

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Pursuing a Ph.D. degree is known to be a long and harsh journey, which is no exception to me. Despite this, I have had so much fun in the past five years plus two months because I have met so many fantastic people who help me grow, support me, and enrich my life. It has been a great honor and pleasure knowing them and working with them.

First of all, I want to thank my fabulous advisor Prof. Aaron J. Elmore. He has extraordinary patience in helping me grow step by step and provides me with an enjoyable and motivating environment for doing research. We spent more than half a year reading and discussing papers before we landed on the topic of this dissertation. I am so grateful for his patience and valuable guidance in helping me find the right topic. He did not just teach me a lot in doing research, but also showed me a way of being an inspirational leader. He is very positive and always encourages me to achieve more. It is so fortunate to work with him. I am very proud of the research I did under his advising.

I also appreciate the opportunities of working with two fantastic professors Prof. Michael J. Franklin and Prof. Sanjay Krishnan. Mike can use one sentence to create a dissertation topic. I still remember the day in this office when he told me that striking the middle ground between batch execution and stream computing could be an awesome topic, which is essentially what this dissertation is about. He asks many tough, but inspiring questions, which is very helpful in my research. Sanjay always provides interesting and insightful perspectives about my research, and can quickly identify the key problems during the discussion. I sincerely thank both Mike and Sanjay for their efforts in my research and their valuable advice for improving myself. I want to thank Prof. Raul Castro Fernandez for every fruitful chat with him and his helpful feedback on my talks and research.

Next, I would like to thank the postdoc researcher Zechao Shang. He encourages me to consider a more ambitious and impactful research topic. I appreciate his candid and sharp critiques on my research ideas, writing, and presentation, which have driven me to improve myself significantly. I am grateful for working with him in my Ph.D.

I appreciate Justin Levandoski offering me the opportunity of interning at Microsoft Research.

I am fortunate to have Umar Farooq Minhas as my mentor. I thank Umar for his patience and valuable advice in guiding me through the research project, especially when we met obstacles. I thank Cristian Diaconu and Donald Kossmann for their helpful feedback on my project and everyone I met at Microsoft Research.

I want to thank Prof. Aditya Parameswaran for offering me this valuable postdoc opportunity. I am so excited to explore the new research topics and look forward to working with him. I would like to thank my friend Prof. Bolong Zheng. The discussion with him inspires me to decide to advance my research career as a postdoc researcher.

I want to thank the administration staff and tech staff of the CS Department of UChicago for helping me have a smooth Ph.D. life. Special thanks to Nita Yack, Margaret Jaffey, Sandra Wallace, Sandy Quarles, Donna Brooms, Bob Bartlett, and Patricia Baclawski.

My Ph.D. life would be much less delightful without my cool friends and labmates. Many thanks to Yi Ding, Guangpu Li, Mingzhe Hao, Huaicheng Li, Renyu Zhang, Xuefeng Liu, Hao Jiang, Adam Dziedzic, Chunwei Liu, Suhail Reman, Rui Liu, William Brackenbury, Xi Liang, John Paparrizos, and everyone I met in the CS Department. Chatting with them gives me so much fun. We cheer for each other's achievement and show support in the frustrating moment. It is my great pleasure being friends with them. Thanks for the help and the happy memories they brought to me.

I also want to express my gratitude to my family and my friends I met before my Ph.D. Let me start with my first friend that I can recall.

I probably knew Yunwei Yang when he was born. I use "probably" because we are already best friends since I can remember. The most fun I had in my childhood is with him. I am so grateful for having him in my life, and thankfully we are still close today. More luckily, Yunwei and I met a group of interesting people: Wanyi Zhou, Ruoyun Li, Maoying Li, Kui Luo, Jiadi Zou, and Hanbing Chen. 2020 is the nineteenth year since we first got together. Nineteen years of friendship let us not just know each other's parents but also many of their cousins, uncles, and aunts. I appreciate the happiness and support they brought to me. Kui Luo, Huayu Liu, and I have

been close friends since middle school. Our friendship started initially due to our common passion for basketball and is further developed as we keep encouraging and inspiring each other to be a better self. It is so fortunate to be friends with them.

I also want to thank my friends Jian Wang, Lesi Zhao, and my undergraduate roommates Jian Peng, Min Hu, and Yazhou Chen. When I did my master study in ICT, CAS, I am fortunate to work with my advisor Wei Li, collaborators Taoying Liu and Rubao Li, and many great friends Yang Yang, Chen Feng, Fan Liang, Liang Li, Ruijian Wang, Jingjie Liu, Jian Lin, Xiaoyi Lu, and Hong Liu. Thanks for the happiness they brought to me.

As a single child of my parents, I am fortunate to have several great cousins: Yutian Lei, Qi Ni, Li Zhu. The most fun of family gathering is to play around with them. I am sincerely thankful for the good time I have with them.

I am so grateful for my last ex-girlfriend Lingna Yang for agreeing to be my wife. My wife and I met half a year before I started my Ph.D. at UChicago. However, more than ten thousand kilometers of distance did not separate us. We commit to each other and offer unconditional support for each other. There are countless times in my Ph.D. when I was frustrated but cheered up by her. I am so blessed to have her as my wife.

Finally, I would like to thank my parents and my wife's parents. My wife's parents, Shaohui Yang and Qiufang Hu are very supportive of the long-distance relationship between us. They trust their daughter's choice and never put any pressure on us. My parents, Jingsong Tang and Jianhua Lei always give me the best of their love. They always support my decisions and encourage me to live a fulfilling life. I am so grateful for their love and so proud of being their child.

ABSTRACT

Database systems have long been designed to take one of the two major approaches to process a dataset under changes (e.g. a data stream). Eager query processing methods, such as continuous query processing or immediate incremental view maintenance (IVM), are optimized to reduce query latency. They eagerly maintain standing queries by consuming all available resources to immediately process new data, which can be a major source of wasting CPU cycles and memory resources. On the other hand, lazy query processing methods, such as batch processing or deferred IVM, defer the query execution to a future point to reduce resource consumption but suffer high query latencies. We find that existing eager and lazy query execution approaches are optimized for the applications on the two ends of the resource-latency trade-off, but the middle ground between the two is rarely exploited.

This dissertation proposes a new query processing paradigm Thrifty Query Processing (TQP), for the middle-ground applications where users do not need to see the up-to-date query result right after the data is ready and allow a slackness of time before the result is returned. TQP exploits this time slackness to reduce resource consumption and allows users to tune this slackness to adjust query latencies and resource consumption.

Implementing TQP involves the redesigns of several core database components. First, we have a new user model that allows users to not just submit a SQL query, but also specify the time slackness information. Specifically, users can specify a performance goal that represents the maximally allowed time to return the result after the data is complete. After, we design a new query execution engine to leverage this performance goal information to reduce CPU cycles. This execution engine includes optimizations for both a single query and multiple queries. For a single query, we consider selectively delaying parts of a query to reduce the resource consumption while meeting the performance goals. For multiple queries, we find that shared execution may not decrease the resource consumption because sharing queries with different performance goals requires the whole plan to execute eagerly to meet the highest performance goal (i.e. the lowest query latency). Therefore, we consider selectively sharing queries to avoid the overhead of eager query execution but also

exploit the benefit of eliminating redundant work across queries. Finally, we design a memory management component to release occupied memory resources when the query is not active. We find that in many cases the data arrival rate is low (e.g. late data), where the query may have a long idle time. Therefore, we selectively release memory resources (e.g. intermediate states) that are least useful for processing the new data. We implement TQP in CrocodileDB, a resource-efficient database, and perform extensive experiments to evaluate each component of CrocodileDB. We show that CrocodileDB can significantly reduce CPU and memory consumption while providing similar query latencies compared to existing approaches.

CHAPTER 1

INTRODUCTION

Several on-going trends pose resource-efficiency as a crucial challenge to the designs of modern database systems. First, the unprecedented growth of data outpaces the expansion of memory and computing resources, yet at the same time data analytics applications are becoming more complex and resource-intensive. Second, the wide adoption of pay-per-use models pushes cloud databases to maximize the gains from the resources users paid for. Finally, environmental concerns demand databases to reduce resource consumption while not sacrificing query performance.

Unfortunately, many existing database designs for querying a dataset under changes (e.g. a data stream) are optimized to improve the raw query performance (e.g. reducing query latency), but not for resource-efficient query execution. These approaches, such as continuous query processing [12, 21, 80], stream computing [3, 17, 20], and immediate incremental view maintenance (IVM) [28, 5], start the query early and eagerly incorporate new tuples into prior query results via incremental execution to provide low-latency results. However, such eager query execution could significantly waste both memory resources and CPU cycles. This is because 1) the system may excessively maintain intermediate states (e.g. a hash table for a hash join) that are barely useful for processing new data and 2) the system can eagerly generate intermediate tuples that will be removed by later query executions and do not contribute to computing the query results. On the other hand, lazy query execution approaches, such as batch processing or deferred view maintenance [28], significantly defer the query execution (e.g. batch processing starts a query execution when all data for this query is ready) to reduce resource consumption but suffer high query latencies.

We find that existing eager and lazy query execution approaches are optimized for the two ends of the resource-latency trade-off, but the middle ground between the two is rarely exploited. Specifically, we consider the middle-ground applications where users do not need to see the query result immediately after the data is complete and allow a slackness of time before the result is returned (e.g. 5s after a window of data arrives). This *time slackness* provides new opportunities for reducing resource waste, which are not fully exploited by existing systems.

This dissertation proposes *Thrifty Query Processing* (TQP) that exploits the time slackness to reduce resource consumption of processing queries over a dataset under changes. The time slackness serves as a knob that allows users to make the trade-off between resource consumption and query latency, and also connects existing eager and lazy approaches at the two ends of the trade-off spectrum. For example, if users allow a high query latency, TQP can employ batch processing and defer the query execution to the point when all data is ready to save resources. On the other hand, if users prioritize low query latency, TQP can adopt a continuous query approach and eagerly maintain the query for every newly arrived tuple.

TQP can be used in an on-premise database to reduce resource consumption and support higher query throughput. More importantly, TQP can be integrated into the cloud database to provide service for stateful standing queries, which is not covered by today's cloud providers. We envision this service to allow users to register a data source, a query, and a sink for a query result, and specify the desired performance (i.e. performance goal). In addition, users are allowed to explore the trade-off between query performance and resource consumption. TQP exploits the information about the performance goal to reduce resource consumption by choosing the right system strategies (e.g. batch, continuous query, or the strategies proposed in the dissertation), and intelligently allocating CPU cycles for maintaining the standing queries and selectively investing memory into keeping the query's intermediate states.

We implement TQP in CrocodileDB [87], a resource-efficient database that exploits time slackness to reduce resource consumption. I lead the system designs of CrocodileDB including the components of query execution engine and memory management. Figure 1.1 shows an overview of the system design of CrocodileDB. The user model [94] of CrocodileDB allows users to specify a maximally allowed time slackness (i.e. performance goal) to make a trade-off between the resource consumption and the query latency. Here, the query latency is defined as the time between when all data arrives for a query (e.g. daily loaded data) and when the corresponding query result is returned. For example, consider a tumbling window query over a data stream. If users set the maximally allowed time slackness to 5 seconds, it means that after all data arrives for a

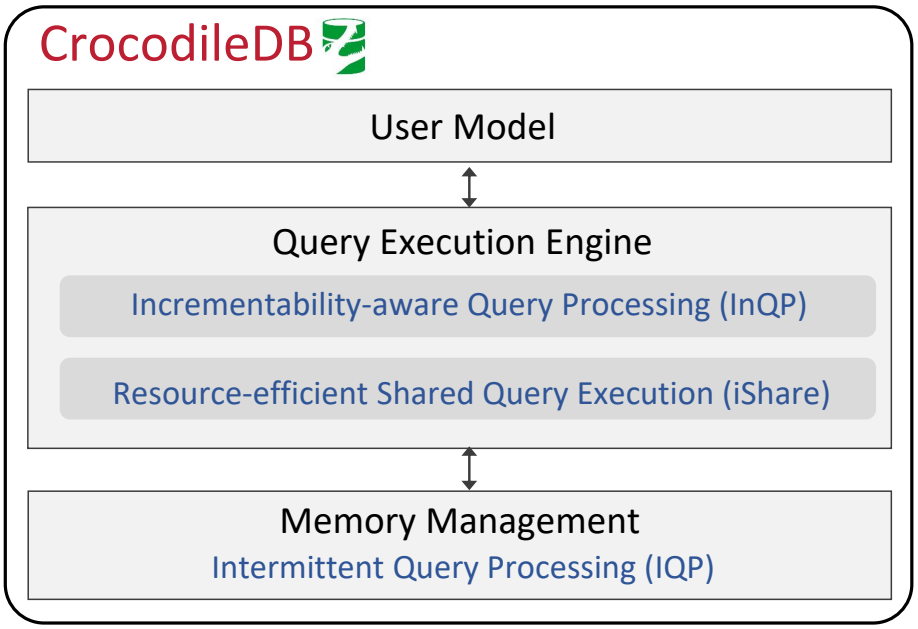


Figure 1.1: An overview of CrocodileDB

time window, users need to see the query result for that window of data within 5 seconds. This timing information is integrated into the underlying query execution engine such that the system can generate a query plan that minimizes CPU consumption and also meet the performance goal (i.e. InQP [95]). In addition, the query execution engine is extended to share the execution of the overlapping work across multiple queries to further reduce CPU consumption (i.e. iShare [96]). The memory management component (i.e. IQP [93]) monitors the data arrival rate and selectively discards some intermediate states to reduce memory consumption when the data arrival rate becomes low and there is an idle time when the query is not executed. We now give an overview of InQP, iShare, and IQP respectively:

Incrementability-aware Query Processing (InQP [95]) Many queries are scheduled before the data is ready (e.g. a window query over a stream of tuples). How early to start the query execution and how eagerly to maintain this query impacts the trade-off between the query latency and CPU consumption. InQP considers reducing the total query work (i.e. reducing CPU cycles) with respect to a performance goal. One major approach to reducing a query latency and meeting the performance goal is using incremental execution, where new data is incrementally incorporated

into prior results. However, incremental execution can increase total query work and waste CPU cycles because for some queries, tuples output in prior executions are removed by later executions. The observation in InQP is that eager incremental executions do not increase the total work for all parts of a query. Some parts of a query are amenable to incremental executions and executing them eagerly (i.e. start one execution for every small amount of data) does not increase the total query work. We define a metric, incrementability, to quantify the cost-effectiveness of incremental executions. In InQP, the higher incrementability a part of a query has, the more eagerly it is executed. InQP integrates this metric into the query optimizer to generate an optimized query plan that meets the performance goal and also reduces the total work. In addition, given this efficient execution engine, InQP supports non-positive query semantics, which is often lacking in continuous query processing systems [12, 21, 80]

Resource-efficient shared query execution (iShare [96]) iShare studies how to share queries with different performance goals when they process the same data (i.e. daily loaded data). Shared execution eliminates redundant computation to save CPU cycles. However, naively sharing the execution across different queries with different performance goals runs the whole shared plan eagerly to meet the highest goal (i.e. lowest latency constraint). This eager execution forces many participating queries with lower performance goals (i.e. higher latency constraints) to run more eagerly than they should. As shown in InQP, eager incremental executions increase the total work. Therefore, the overhead introduced by eager execution may offset the benefit of shared execution.

iShare does not execute a shared plan as a whole with a single frequency, but selectively untangles the execution of a shared plan in two aspects to reduce the overhead of eager incremental execution: 1) executing different subplans in different frequencies with respect to the performance goals; 2) breaking the shared subplans into separate ones (i.e. unshare) and run them at different frequencies based on the performance goals. The key challenge here is that the query optimization process is time-consuming due to the complex search space in finding the execution frequency for each subplan and the possible ways of decomposing a shared subplan. Therefore, we design a new search algorithm and a heuristic metric to quickly find a query plan that exploits the benefit of

shared query execution and avoids the overhead of eager execution.

Intermittent Query Processing (IQP [93]) When new data arrives intermittently or at a low rate and users choose to maintain the query lazily, there is an idle time when the query has no data to process and is inactive. IQP considers releasing some memory resources when the query is inactive. Specifically, IQP sets a memory budget for the query when it is inactive. IQP exploits this budget by selectively keeping a subset of intermediate states (e.g. hash tables for hash join) or building new states such that we can reuse these saved states to reduce the query latency of processing new data.

An efficient plan about which intermediate states should be materialized or built depends on information about the new data, such as the estimated size and distribution of the relations having new data. Since this information can be provided or predicted by upstream data systems (e.g. data collection and preparation), IQP leverages the information about new data to find an efficient query plan, where it chooses to materialize a subset of intermediate states or build new states that are most useful for reducing the latency of processing the new data. Therefore, for this intermittent and predictable data arrival pattern, IQP achieves low query latency with a memory limit.

In this dissertation, we first present the user model of CrocodileDB in Chapter 2. Then, we discuss InQP, iShare, and IQP in Chapter 3, Chapter 4, and Chapter 5 respectively. After, we discuss related work in Chapter 6 and conclude this dissertation in Chapter 7.

CHAPTER 2

CROCODILEDB USER MODEL

In CrocodileDB, we allow users to specify a *performance goal* that represents the maximally allowed time to return the result after the data is complete. Figure 2.1 shows an example of querying a window of data. Here, the performance goal is the maximally allowed time between when the last tuple arrives for this window and the query result is returned. Our query optimizer internally leverages the information about users' performance goals along with information about the query structure and data arrival patterns (i.e. which relations having new data and the corresponding data arrival rates) to generate a query plan that can reduce CPU consumption (i.e. InQP and iShare) and memory usage (i.e. IQP) while meeting the performance goal.

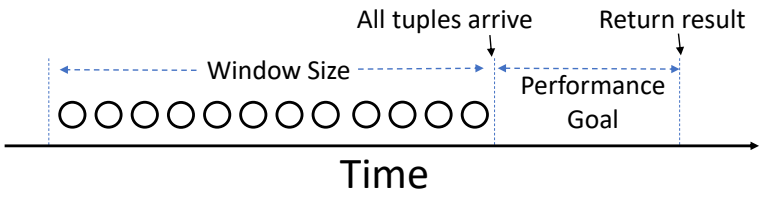


Figure 2.1: An example of a performance goal in CrocodileDB

2.1 Performance goal & Resources

CrocodileDB maintains a standing window query over a stream of tuples. While CrocodileDB currently supports tumbling windows, we can support other window semantics, such as sliding windows. We later briefly discuss how to support performance goals in more general cases.

Performance goals for tumbling window: In CrocodileDB, users can explicitly express a performance goal, which is the maximally allowed time slackness between when all tuples for a window arrive and the actual result is returned to users.

The performance goal is a knob that users can tune to make trade-offs between resource consumption and query latency. With different performance goals, the system will generate corresponding plans to minimize resource consumption. Consider an example of a windowed query

with a window of 10 minutes. If users allow a large slackness (e.g. a performance goal of 2 mins), CrocodileDB can selectively maintain some parts of the query lazily to reduce CPU consumption [95] or selectively discard some intermediate states of incremental executions to reduce memory consumption [93]. If the slackness is large enough (e.g. 10 mins), CrocodileDB can start the query after all tuples arrive (i.e. batch processing) and avoid the CPU or memory resources waste introduced by incremental executions. On the other hand, if users prioritize query performance (e.g. return the result within 1 sec for every 10 mins of data), CrocodileDB will execute this query more eagerly with higher resource consumption.

With the performance goal specified by users, CrocodileDB unlocks many optimization opportunities [87] that are impossible in existing systems[6, 13, 28]. Existing systems let users decide when to execute the query, instead of allowing users to specify when to expect a query result in CrocodileDB. For example, users need to set a time trigger of maintaining the whole query periodically (e.g. every 1 min) to achieve the desired performance. This query plan executes the whole query in a single pace and ignores that some parts of a query are less amenable to incremental executions. By contrast, CrocodileDB can exploit this performance goal to selectively delay parts of the queries to reduce CPU consumption but still meet the performance goal (i.e. InQP).

Extensions of performance goals to more general cases: The performance goal of CrocodileDB can be extended to sliding windows. Semantically, a sliding window can be regarded as a list of independent windows. We can apply the performance goal to each of them. We note that the underlying system optimizations should consider the overlaps between sliding windows to reduce redundant work, which is in the future work of CrocodileDB.

The performance goal can also be applied to general incremental view maintenance. Consider an example of maintaining a view over a stream of tuples. Users can specify the condition of computing an up-to-date result (e.g. updating the result for every 10 mins of data) and additionally submit a performance goal to decide when they can see an up-to-date result. For example, if the performance goal is 10 secs, for every 10 mins of data, we will incorporate them into the query result within 10 secs after the data is ready.

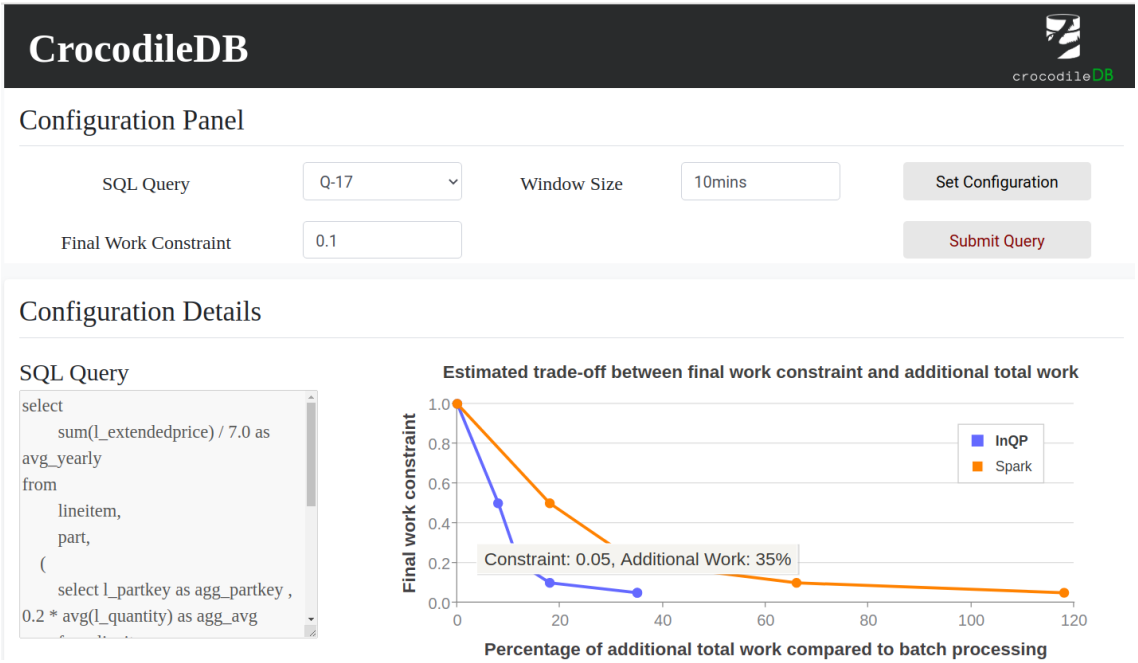


Figure 2.2: CrocodileDB configuration component

2.2 User model demonstration

We implement CrocodileDB in Spark and develop a framework to show how users interact with CrocodileDB. In this section, we show how users interact with the optimization of InQP, which can reduce CPU consumption compared to Spark with the same performance goal. This framework contains an interactive configuration interface and a real-time performance monitoring component.

The configuration interface allows users to 1) submit a window query and specify a final work constraint as a proxy for the performance goal; and 2) tune the final work constraint to make the trade-off between CPU consumption and query latency (i.e. the time of returning the result after all tuples for a window arrive). The final work constraint is based on a cost model and is used to quantify the remaining number of units of work the query needs to do after all data arrives. We provide the final work constraint as a knob instead of the performance goal to users because the actual query latency is hard to predict (e.g. query latency depends on the hardware configuration). Specifically, the final work constraint is a ratio between the final work users want to achieve and the one of executing a query in one batch. For example, a constraint 0.2 means that users want to

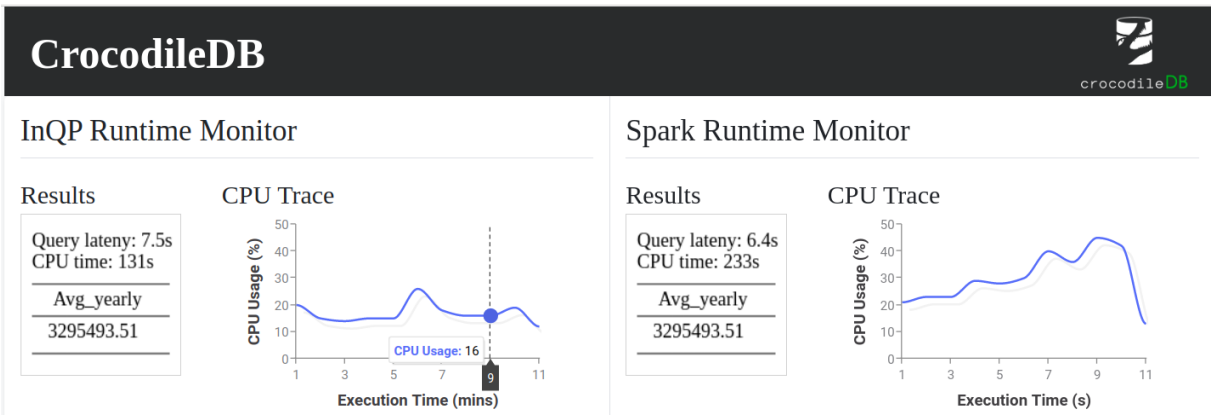


Figure 2.3: CrocodileDB monitoring component

reduce the query's final work to 20% of the one of batch execution.

Figure 2.2 shows the configuration interface of CrocodileDB. In the Configuration Panel, users first choose a query from all TPC-H queries and several hand-written queries based on the TPC-H schema. Then, users set the window size and the final work constraint. The Configuration Details shows the SQL query selected by users and the estimated trade-off between the additional work (with respect to the batch execution) invested into the query to meet the desired final work constraint. For example, Figure 2.2 shows that for a constraint 0.05, InQP needs to invest 35% work compared to the work of batch execution. Users can tune the final work constraint based on the estimated trade-off curve to achieve the desired query latency and control the CPU resources they are willing to pay for.

If users hit the Submit Query button, the configuration framework will submit the query to systems InQP and Spark. Users are able to observe the runtime statistics of both systems side-by-side in our monitoring component. The monitoring component is shown in Figure 2.3. It monitors the execution of the same query with the same final work constraint for InQP and Spark side-by-side. We show the returned result and the actual latency of returning this result. Users are expected to observe similar latencies for both systems since they use the same constraint. We also show CPU usages during the query execution and we see that InQP has lower overall CPU usages compared to Spark.

CHAPTER 3

INCREMENTABILITY-AWARE QUERY PROCESSING

Many 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 [93].

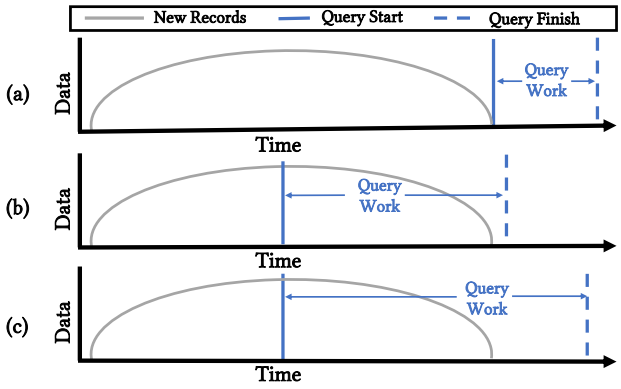


Figure 3.1: How incrementability can impact query latency and the amount of work done.

This project 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 3.1, where data is being progressively loaded into the database and the goal is to compute the result of a pre-defined query. In Figure 3.1a, 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 3.1b). 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 3.1c. 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 [49, 14, 42] 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 [27]. 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 CPU 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 method, *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 improve performance, InQP executes query paths at different paces (or frequencies) based on their respective incrementability. In InQP, users are allowed to specify a final work constraint for a query (i.e. a proxy for the performance goal), and the system finds an optimized query plan that minimizes the total work under the given final work 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.

The major contributions of InQP include

- we propose a new metric of incrementability to quantify the effectiveness of incremental execution;
- we define the incrementability and propose a cost model with improved cardinality estimation for computing incrementability;

- we decompose a query into query paths and design a novel algorithm based on incrementability to decide when to execute query paths;
- we integrate our ideas into a real system, Spark, and demonstrate the effectiveness of such an approach.

3.1 Background and Definitions

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

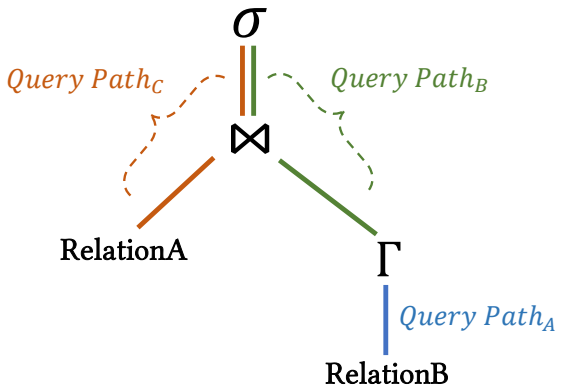


Figure 3.2: A query with multiple query paths.

3.1.1 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 [91]. With this knowledge, we can estimate when a query is triggered, and the final work and

the total work of a triggered query based on our cost model in Section 3.2.1. For simplicity, we assume a steady arrival rate for our cost model and we show our robustness for a bursty arrival rate in Section 3.4.4.

3.1.2 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 3.2 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.

3.1.3 Incrementability Definition

Incrementability: Incrementability describes how incrementable a pace configuration is. For a pace configuration P, we define $\mathcal{C}_F(P)$ as its final work and $\mathcal{C}_T(P)$ as its total work. Recall that the final work means the work the system does after data is complete and the 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:

$$\texttt{INC}(P_1, P_2) = \frac{\mathcal{C}_F(P_1) - \mathcal{C}_F(P_2)}{\mathcal{C}_T(P_2) - \mathcal{C}_T(P_1)} \tag{3.1}$$

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 3.3 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 reduces final work, compared to batch processing.

There are three levels of incrementability. If there is no additional total work for incremental

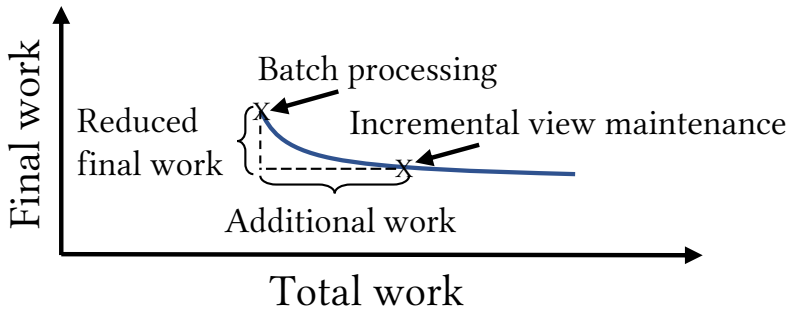


Figure 3.3: An example of the benefit (i.e. reduced final work) and cost (i.e. additional work) for an incremental execution plan.

executions (i.e. $\mathcal{C}_T(P)$ equals $\mathcal{C}_T(P_{\mathbb{1}})$), 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 query 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. The prior output unmatched left tuples

should be removed from the output. Therefore, outputting the unmatched tuples too eagerly wastes resources and is partially incrementable.

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.

3.2 Computing Incrementability

To calculate incrementability, we need to compute $\mathcal{C}_T(P)$ and $\mathcal{C}_F(P)$ given a pace configuration P. A critical challenge for this task is how to estimate the cost for each incremental execution given a pace configuration. We first discuss our modifications on existing cost models for computing $\mathcal{C}_T(P)$ and $\mathcal{C}_F(P)$, and then discuss how to compute the total work and the final work given a pace configuration in Section 3.2.2. We support insert, delete, and update operations, and support operators of select, project, join (i.e. inner, outer, anti, and semi-join), aggregate, distinct, sort, and limit.

3.2.1 Cardinality Estimation for Incrementability

As shown in prior work [103, 62, 86], 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 InQP, especially for non-positive queries. So we focus on the problem of cardinality estimation, and adopt the cost functions in conventional RDBMS cost model [62, 86, 65] for the second factor.

Problem and Intuition

We emphasize that the problem of cardinality estimation for InQP is different from the ones in existing work [103, 62] for either batch processing or incremental execution, because they mainly

consider the positive queries where new input data only produces new outputs but never removes 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 before become matched.

The core of our solution is to *distinguish the cardinalities of three categories of tuples: inserts, 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 project, but our approach does advance the state-of-the-art [103] 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.

Operators

We use a volcano-style query execution model [37] 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 [24], 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 $\mathbb{C}_{\texttt{IN}}$ and $\mathbb{C}_{\texttt{OUT}}$.

As with conventional cardinality estimation, we use statistical information to help estimate cardinalities. This includes *select selectivity* that models the probability that a tuple satisfies a certain predicate, *join selectivity factor* [103] 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 [103], we use statistical information from previous executions as the estimation for upcoming query executions. We also perform an experimental analysis in Section 3.4.4 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,

Input Operation

Output Operation: 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

Figure 3.4: An example selectivity matrix.

we use a selectivity matrix $S \in \mathbb{R}^{3\times3}$. Figure 3.4 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 $\mathbb{C}_{\texttt{OUT}} = S \times \mathbb{C}_{\texttt{IN}}$. 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 $K + \mathbb{C}^I_{\texttt{IN}} - \mathbb{C}^D_{\texttt{IN}}$. The output cardinality $(\mathbb{C}^I_{\texttt{OUT}}, \mathbb{C}^U_{\texttt{OUT}}, \mathbb{C}^D_{\texttt{OUT}}) = (K + \mathbb{C}^I_{\texttt{IN}} - \mathbb{C}^D_{\texttt{IN}}, 0, K)$.

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 [42]. 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 [26]. 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 $(\mathbb{C}^I_{\texttt{OUT}}, \mathbb{C}^U_{\texttt{OUT}}, \mathbb{C}^D_{\texttt{OUT}}) = (0, \mathbb{C}^I_{\texttt{IN}} + \mathbb{C}^U_{\texttt{IN}} + \mathbb{C}^D_{\texttt{IN}}, 0)$.

- 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 $\mathbb{C}^D_{\texttt{OUT}} = \min(\mathbb{C}^D_{\texttt{IN}}, N)$. Similarly, each insert tuple goes to an empty group, and emit a new aggregation tuple, so $\mathbb{C}^I_{\texttt{OUT}} = \min(\mathbb{C}^I_{\texttt{IN}}, M - N)$.

Update tuples update existing non-empty groups, so $\mathbb{C}^U_{\texttt{OUT}} = \min(\mathbb{C}^U_{\texttt{IN}}, N)$.

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 [106], 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 $\mathbb{C}_{L\texttt{IN}}$ and $\mathbb{C}_{R\texttt{IN}}$, 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 [103], 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 $\mathbb{C}_{L\texttt{OUT}} = S_L \times \mathbb{C}_{L\texttt{IN}} \times |R|$.

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 $\mathbb{C}^I_{L\texttt{IN}} \times (1-p_l)^{|R|}$. 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.

Algorithm 1: Computing $\mathcal{C}_T(P)$ and $\mathcal{C}_F(P)$.

$(K_1, K_2, \cdots, K_Q) \leftarrow (0, 0, \cdots, 0)$
for $m \leftarrow 1$ **to** M **do**
  $I_{Global} \leftarrow \frac{m}{M}$
  $PathSet \leftarrow \emptyset$
  for $i \leftarrow 1$ **to** Q **do**
    if $I_{Global} - \frac{K_i}{P_i} \geq \frac{1}{P_i}$ **then**
      Add path i to $PathSet$
  end
  for $i \in PathSet$ **do**
    $K_i \leftarrow K_i + 1$
  end
  $cost \leftarrow$ Estimated cost of a simulated execution
   that involves flushing buffers of paths in $PathSet$
  $\mathcal{C}_T(P) \leftarrow \mathcal{C}_T(P) + cost$
  if $I_{Global} = 100\%$ **then**
    $\mathcal{C}_F(P) \leftarrow cost$
end

3.2.2 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, $\texttt{INC}(P_1, P_2) = \frac{\mathcal{C}_F(P_1) - \mathcal{C}_F(P_2)}{\mathcal{C}_T(P_2) - \mathcal{C}_T(P_1)}$. So we focus on how to estimate $\mathcal{C}_F(P)$ and $\mathcal{C}_T(P)$ 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 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 $\mathcal{C}_T(P)$ and $\mathcal{C}_F(P)$ 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 $\mathcal{C}_T(P)$ and $\mathcal{C}_F(P)$. 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 $\mathcal{C}_T(P)$. When I_{Global} reaches 100%, all query paths flush the buffers and the cost is $\mathcal{C}_F(P)$.

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.

3.3 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 trade-off between the final work and the 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. 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 the following optimization problem: minimizing the total work given a final work constraint.

3.3.1 Problem Formalization

We define the 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:

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

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

3.3.2 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,

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

$P \leftarrow P_{\mathbb{1}}$
while *true* **do**
  $i \leftarrow \underset{i:P_i<P_j,\forall j\in\text{children}(i)}{\arg\max} \partial_i(P)$
  $P_{\text{new}} \leftarrow P_{[i\backslash P_i+1]}$
  if $\mathcal{C}_F(P_{new}) \leq L \times \mathcal{C}_F(P_{\mathbb{1}})$ **then**
    return P_{new}
  if $P = P_\infty$ *or* $\partial_i(P) < 0$ **then**
    return P
  $P \leftarrow P_{\text{new}}$
end

this approach has exponential complexity and is very time-consuming as shown in our experiments (Section 3.4.6). 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 $L \times \mathcal{C}_F(P_{\mathbb{1}})$, 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. $\mathcal{C}_F(P_1) - \mathcal{C}_F(P_2)$) and the increased total work (i.e. $\mathcal{C}_T(P_2) - \mathcal{C}_T(P_1)$) 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

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

where $P_{[i \setminus c]}$ 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_∞.

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 3.4.6.

Putting everything together: We use the example in Figure 3.2 to explain the optimization of InQP. Since this example has three query paths, we use a pace configuration $P = (P_A, P_B, P_C)$. The optimization starts with $P_{\mathbb{1}}$ and we consider increasing pace for one query path from 1 to 2. Specifically, we have three possible configurations $(1, 1, 2)$, $(1, 2, 1)$, and $(2, 1, 1)$. Recall that we require the pace of a parent query path is no larger than the pace of its child query path. Therefore, $(1, 2, 1)$ is not a valid pace configuration since query path B should not have larger pace than query path A. We only compare the incrementability of $(1, 1, 2)$ and $(2, 1, 1)$ with respect to $(1, 1, 1)$.

Recall that given two pace configurations P_1 and P_2, their incrementability is $\texttt{INC}(P_1, P_2) = \frac{\mathcal{C}_F(P_1)-\mathcal{C}_F(P_2)}{\mathcal{C}_T(P_2)-\mathcal{C}_T(P_1)}$. We use Algorithm 1 to compute $\mathcal{C}_T(P)$ and $\mathcal{C}_F(P)$. It simulates the process of executing the query using pace configuration P. Consider $P = (2, 1, 1)$ as an example. We trigger a simulated execution of query path A when 50% of data arrives because its pace is 2. The simulated execution uses our cost model to compute the cost of processing 50% of data for path A. When 100% of data arrives, we trigger another execution for all paths. After we compute the values of incrementability for all paths, we choose to increase the pace of the path with the highest incrementability. We repeat this process until we meet the final work constraint.

3.3.3 Applicability of InQP

InQP can be applied to systems that support incremental view maintenance, such as Spark [3], Flink [17], and a PostgreSQL modified for incremental executions [93]. For systems that only support insert tuples (e.g. Spark), operators need to be modified to support deletes and updates (Section 3.2.1). 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.

3.4 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 [48], how

much CPU consumption does InQP reduce given the similar query latency goal? (Section 3.4.3)

- How do the accuracy of our cost model and bursty workloads impact InQP's performance? (Section 3.4.4)
- What is the accuracy of our cardinality estimation compared to PostgreSQL? (Section 3.4.5)
- What are the overhead and benefits of the greedy algorithm of InQP? (Section 3.4.6)

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).

3.4.1 Prototype Implementation

We implement InQP in Spark 2.4.0 [3], and extend Structured Streaming to support deletes and updates based on existing IVM algorithms [24] 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. [102]. 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. 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 CPU time* to represent the CPU 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.

3.4.2 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
```

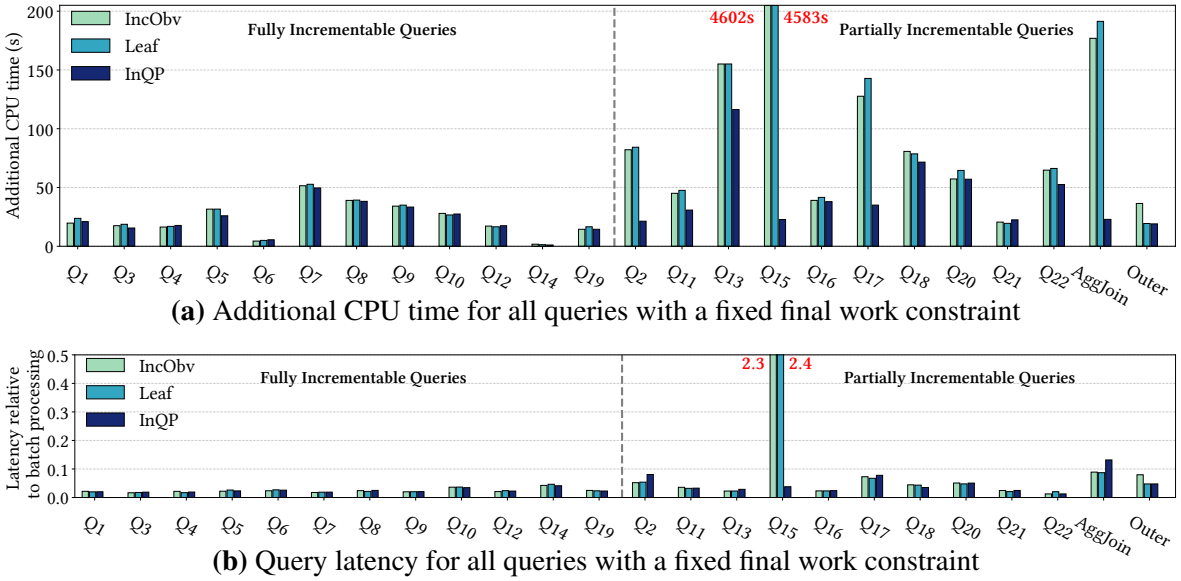
where Q_AggJoin joins an aggregate operator with a base table, and Q_Outer is a left-outer-join with two equal-joins. We preload the full dataset into Kafka, but let InQP pull data from Kafka at a data rate of 1GB/min. We generate datasets that are large enough to show the performance impact of partially-incrementable 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 information impacts InQP in Section 3.4.4. 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.

3.4.3 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 CPU 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 [48], 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. min-



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

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

Figure 3.5: Additional CPU 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).

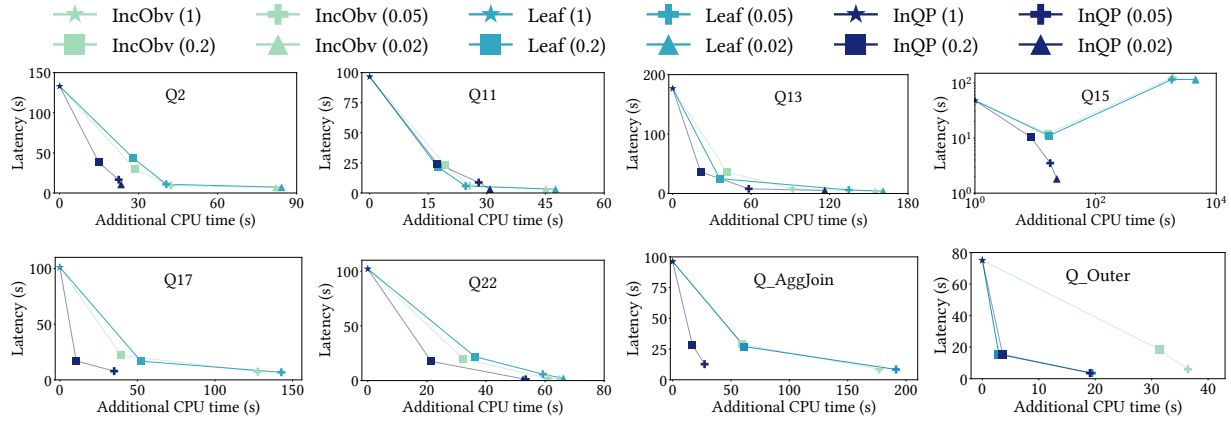


Figure 3.6: Trade-off between resource consumption and query latency under different final work constraints.

imizing 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 CPU 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 3.5 shows the results of all 24 queries. Figure 3.5a shows that InQP has much lower additional CPU 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 CPU 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 CPU time and query latency to IncObv and Leaf. We note that Leaf has similar additional CPU 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 have similar pace configurations for flushing input tuples of base relations and have similar additional CPU time.

We report additional CPU time and query latency with different final work constraints, which are in Figure 3.6. For each final work constraint, we compare their additional CPU 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 CPU 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 CPU 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.

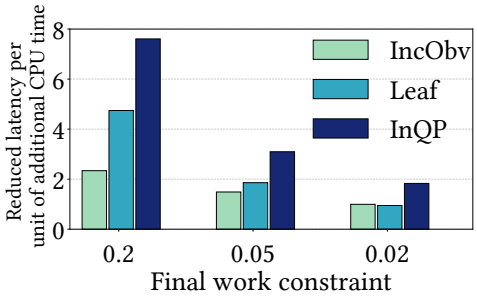


Figure 3.7: Reduced latency per unit of additional CPU time.

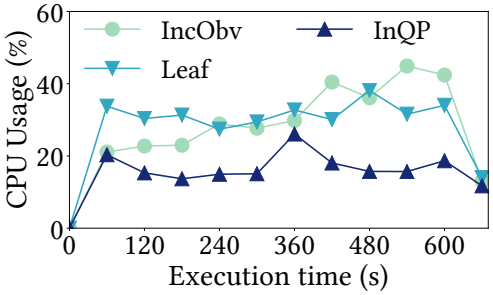


Figure 3.8: CPU usage trace (Q17, constraint = 0.05).

To highlight the cost-effectiveness of InQP, we report the ratio between the reduced query latency compared to batch processing and the additional CPU time. The higher the ratio is, the more latency we reduce per unit of additional CPU time we invest. Figure 3.7 shows the average ratio of all queries in Figure 3.6 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 3.8. Q17 uses the 10GB dataset and the whole data loading process takes 600 seconds (data rate of 1GB/min). Figure 3.8

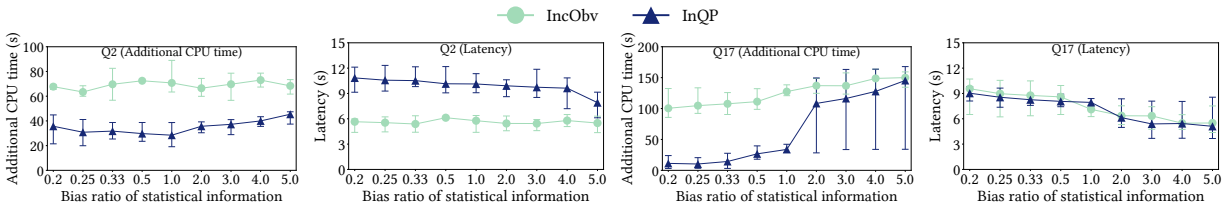


Figure 3.9: Performance impact of biased statistical information.

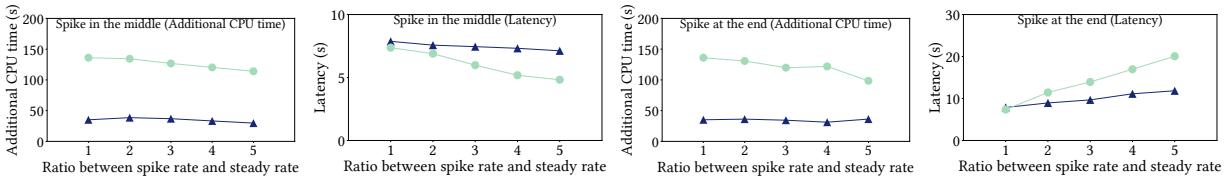


Figure 3.10: Performance impact of a bursty arrival rate (Q17).

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

3.4.4 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 CPU time and query latency.

We show the results of Q2 and Q17 in Figure 3.9. 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 CPU 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 CPU time and similar query latency compared to IncObv. For Q17, we have similar observation to Q2. When we overestimate, the additional CPU time increases and and the query latency decreases for both approaches. The average additional CPU 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 CPU time. These experiments show that biased statistical information could increase additional CPU 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 3.10 shows the test results. We see that the spike in the middle does not change additional CPU time of InQP, but slightly decreases its latency (i.e. the two leftmost figures in Figure 3.10). On the other hand, both additional CPU time and latency drop for IncObv. The rea-

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

son 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 CPU time drops for IncObv because with the spike rate increasing, more data are processed in one batch for the spike and thus IncObv does less incremental work. This reduces the cost of partially incrementable parts of IncObv. Nevertheless, InQP has a much lower additional CPU 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 3.10). 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.

3.4.5 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 [103] only supports select-project-join queries. In InQP, we need to support complex queries such as the query in Figure 3.2 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

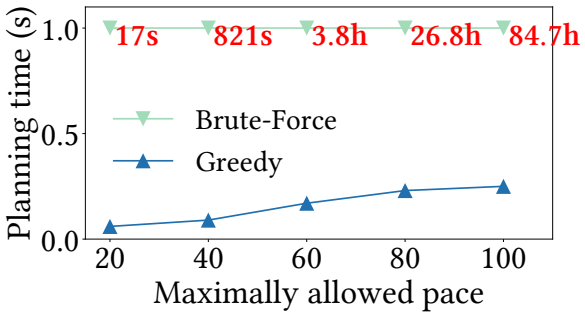


Figure 3.11: Planning time.

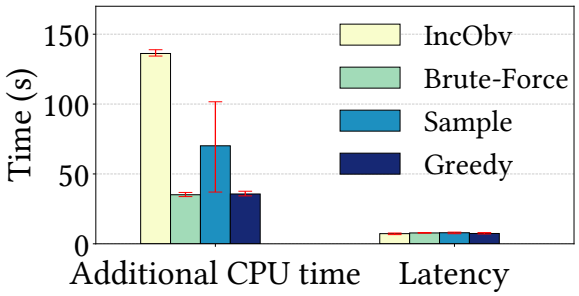


Figure 3.12: Q17 Performance

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 using the following formula: 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 3.1 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.

3.4.6 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 (`BruteForce`) 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 3.11 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 CPU 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 CPU time. Figure 3.12 shows that Greedy has similar performance to the optimal plan generated by Brute-Force, and that the additional CPU time of Sample varies. While Sample can find a plan that has similar performance to Brute-Force, it has much larger additional CPU 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.

3.5 Summary

We present InQP as a new query processing method that models how amenable a query is for incremental executions 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). The experiments show 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.

CHAPTER 4

RESOURCE-EFFICIENT SHARED QUERY EXECUTION

In addition to reducing CPU consumption for a single query, TQP considers optimizing multiple queries together and sharing their common work to further reduce CPU consumption. Prior studies in shared query execution [34, 47, 67] or multi-query optimization (MQO) [35, 85, 50] create a shared plan to reduce CPU consumption by eliminating the redundant work across queries when multiple queries intend to access the same data or perform the same job. However, sharing is not always beneficial in TQP.

As shown in InQP, there exists a trade-off between resource consumption and latency for querying a dataset under changes. Consider a regular query over a dynamic dataset, such as a regular ETL job or tumbling window over high ingest data, where the trigger condition that starts one or more queries is known and frequent. Waiting until all data is ready before starting the query (i.e., *batch execution*) offers low resource consumption, but high latency. If the user demands a lower latency, we can start processing data early before the trigger condition and incrementally maintain the query result. While this reduces the query latency (being the time between when the last record for the query arrrives and when the query result is returned), it may increase the total execution time and CPU consumption for certain queries [95, 28, 48, 20]. This is because tuples output in earlier executions are removed by later executions. These queries are partially incrementable queries as shown in InQP (as opposed to fully incrementable queries, which do not consume additional CPU cycles on eager incremental execution). Generally, if we increase the frequency or eagerness of incremental execution of a partially incrementable query, its query latency decreases but its resource consumption increases. Prior work demonstrates methods for tuning the eagerness to meet latency performance goals [95, 48].

In this context, when multiple queries have different latency goals, the shared execution may not be optimal. The main reason is that the shared plan needs to honor the tightest (i.e. lowest) latency goal and execute more frequently, potentially consuming more resources. The extra resource consumption may even offset the benefit of sharing. While recent research [59, 54, 85] judiciously

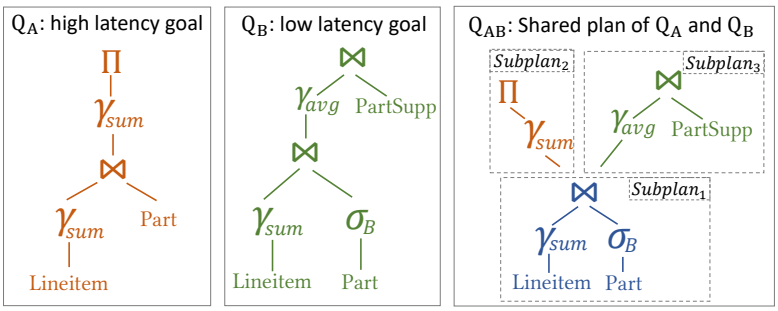
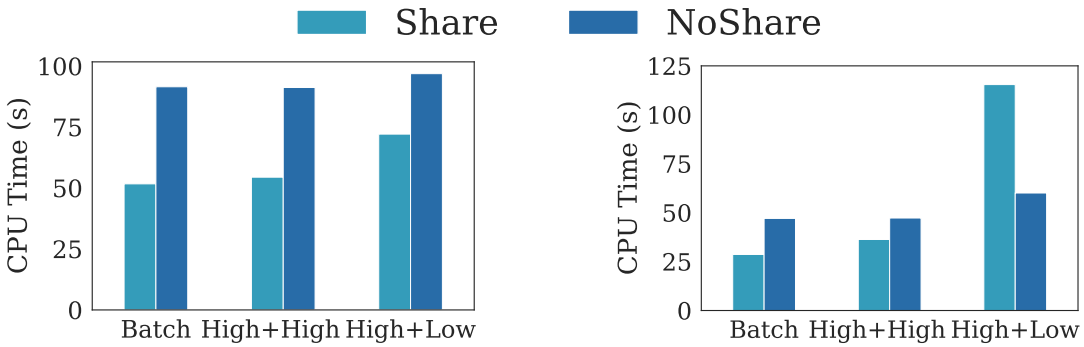


Figure 4.1: Example query plans w/(o) MQO

decides the parts of queries to be shared, an essential goal of MQO is minimizing overall resource consumption. No existing MQO approach considers the overhead of heterogeneous latency goals and the consequential eager execution on shared parts.

We illustrate the problem with an example. Consider query Q_A and Q_B in Figure 4.1. An MQO optimizer generates a plan Q_{AB} that shares two almost identical joins (with the difference of σ_B). Note that σ_B only marks tuples that belong to Q_B, and does not drop any tuple (which are all needed by Q_A). Consider the case that Q_A has high latency goal and Q_B has a low or tight latency goal. The shared plan Q_{AB} needs to meet the tighter goal (i.e. Q_B's). There are two cases of overhead compared with the scenario of not sharing. First, $Subplan_1$ needs to execute more frequently to meet Q_B's latency goal. Assume the selectivity of σ_B is 1%. Without sharing, all data can be executed lazily for Q_A (e.g. using one batch) and only 1% data is executed eagerly for Q_B. In the shared plan, all data is executed eagerly. If we eagerly maintain $Subplan_1$, the aggregate operator γ_{sum} needs to repeatedly remove prior output tuples when the aggregated values of corresponding groups change. Therefore, eagerly processing all data for $Subplan_1$ consumes more computing resources. Second, $Subplan_2$ has overly eager execution. This is because Q_{AB} is executed eagerly to meet the goal of Q_B, but Q_A has a high latency goal, which allows $Subplan_2$ to execute lazily.

We illustrate the complexity of the decision space by a micro-benchmark. The first workload includes two (almost) fully incrementable queries Q_5 and Q_8 from TPC-H. The second includes two partially incrementable queries Q_A and Q_B in Figure 4.1. For each workload, we evaluate them individually (denoted as NoShare) or in a shared plan by a state-of-the-art MQO optimizer [35]



(a) Tests of Q_5 and Q_8

(b) Tests of Q_A and Q_B

Figure 4.2: CPU seconds of executing two queries separately or in a shared plan w/(o) performance goals

(denoted as Share). We consider three scenarios: i) no latency constraint (Batch evaluation); ii) low latency goal for Q_5/Q_B and high latency goal for Q_8/Q_A; iii) high latency goal for both queries. A cost model chooses execution frequency (c.f. Section 4.2.2), and more experimental details are in Section 4.4.6. We measure the total CPU seconds in Figure 4.2. We see that for the first workload, Share uses less CPU in all cases. This is due to incremental execution consuming little additional resources. However, for a partially incrementable workload with latency performance goals (i.e. High+High), the benefit of sharing decreases. With heterogeneous latency goals (i.e. High+Low) Share actually has even higher resource consumption. Therefore, an optimizer should holistically consider the benefit of shared query execution and the overhead of incremental executions by leveraging the information about the diverse latency goals and how amenable the queries are to incremental executions.

We propose *iShare* for sharing queries with different latency goals. Instead of evaluating a shared plan with a single frequency, iShare selectively untangles a shared (sub)plan execution in two ways: 1) executing different subplans in different frequencies with respect to the latency goals; 2) breaking the shared subplans into separate ones based on latency goals (i.e. unshare) and run them at different frequencies.

However, such optimization is time-consuming due to the complex search space in finding the execution frequency for each subplan [95] and the possible ways to decompose the shared plan.

We propose several techniques to address this challenge and make the following contributions:

- First, we observe that some plans are more incrementable than others (as in Figure 4.2). Therefore, we evaluate more incrementable plans with higher frequencies. Specifically, we extend the metric of incrementability to quantify the cost-effectiveness of incremental execution for a subplan, and design an optimized algorithm to quickly find the execution frequencies.
- Second, we propose a heuristic metric, *sharing benefit*, which can estimate whether it is worthwhile to share a subplan for two sets of queries, and a greedy algorithm to decompose a shared subplan based on *sharing benefit*.
- Third, we design a algorithm to quickly compute the execution frequencies for the decomposed subplan without computing the execution frequencies for the whole plan from scratch.
- Finally, we perform extensive experiments to show that iShare has low optimization overhead and can significantly reduce CPU consumption compared to executing share plans (from the state-of-the-art MQO optimizer) in a single frequency and two approaches that execute queries separately.

4.1 Problem Statement and Overview

In this section, we discuss the context and definition of our optimization problem, present the definitions used in iShare, and the underlying shared query execution engine.

4.1.1 Problem context and definition

We consider a scenario where a stream of tuples is being loaded into the database, and users want to analyze this data stream via scheduled queries. The queries are scheduled based on pre-defined events (e.g., time/count-based). We name this pre-defined event trigger condition. We focus on optimizing the scheduled queries with the same trigger conditions (e.g., daily loaded data). We assume knowledge of the data arrival rate (i.e., number of new tuples per hour for each base relation). Historical statistics [92] can estimate this information. We use this information to

estimate the cost of query execution and query latency. For simplicity, we assume a fixed data arrival rate. As in prior work [95], we use the **total work** to represent the CPU consumption of all queries and the **final work** as a proxy for the latency of each query. The total work represents the total units of work done by all queries throughout the shared query execution. The final work is the remaining units of work to be done for each query after the trigger condition is met. Take the query Q_{AB} in Figure 4.1 as an example. The final work of Q_A includes the remaining units of work of $Subplan_1$ plus $Subplan_2$, and the final work of Q_B includes the remaining work of $Subplan_1$ plus $Subplan_3$. Both total work and final work are quantified based on the cost metric in a database optimizer. It could be a unified cost of estimated CPU cycles and I/O operations, or the number of tuples processed by all operators.

In iShare, users additionally submit a final work constraint for each query. This constraint allows users to make a trade-off between CPU consumption and query latency. Therefore, our optimization problem is given a set of scheduled queries with the same trigger conditions, how to find a query plan to minimize the total work of all queries while meeting each query's final work constraint.

4.1.2 Definitions and optimization overview

We observe that directly using the shared plan from existing MQO optimizers has high total work because the whole plan is executed in a single frequency. Therefore, our key idea is to break the shared plan into subplans, where each subplan can be executed via a separate frequency. For simplicity, our optimization assumes a shared plan generated by an existing MQO optimizer [35].

Subplan A subplan in iShare represents a subtree of operators that are shared by the same set of queries. We break the shared plan into subplans at the operators that have more than one parent operator. Consider the shared plan in Figure 4.1. iShare breaks it into three subplans, where all shared operators belong to a subplan (i.e., $SubPlan_1$) and the unique plans for Q_A and Q_B are two separate subplans. When the root operator of one subplan has two or more parent operators, it materializes its output into a buffer such that the parent subplans can consume the intermediate

results at individual frequencies [85]. Similarly, we treat all base relations or delta logs as buffers as well. Therefore, each incremental execution of one subplan processes all new data from the buffers of its child subplans or base tables. Then, it materializes the result tuples into this subplan's buffer or outputs them as the query results. We note that there are multiple parent subplans consuming the same child subplan's buffer. Therefore, each parent subplan will track the offsets of the tuples it has processed. We assume a shared query execution engine that requires the query set of a subplan subsume the query set of its parent subplans (e.g. the query set $\{Q_A, Q_B\}$ of $Subplan_1$ in Figure 4.1 contains the query set $\{Q_A\}$ of $Subplan_2$) We note that it is possible to break the shared plan in a more fine-grained way [95], which comes with a higher optimization cost. We use subplans as the granularity of control as it significantly reduces the optimization time, which we show in Section 4.2.2.

Pace The execution frequency of a subplan can be defined as time-based (every 5s), count-based (every 1000 tuples), or heuristic-based. We use the definition of *pace* in InQP to represent the execution frequency of a subplan and include it here for completeness. A pace k means that the subplan starts one execution whenever the system has received $\frac{1}{k}$ of the total estimated tuples for a trigger condition (e.g. daily loaded data). The higher the pace is, the more eagerly we execute the subplan. A *pace configuration* represents the set of paces $P = (p_1, p_2, \ldots, p_M)$ for all M subplans. The pace configuration $P_{\mathbb{1}} = (1, 1, \ldots, 1)$ represents the batch execution for all subplans.

Optimization overview In iShare, we take a shared plan generated by existing MQO optimizers and adopt the following two techniques to reduce its total work. First, we use a greedy algorithm to find a pace configuration to minimize the total work and also meet the final work constraints, which is discussed in Section 4.2. Based on the shared plan annotated with paces, we consider decomposing each shared subplan into multiple separate subplans. This way, we can execute different subplans at different paces to further reduce the total work. We discuss the subplan decomposition in Section 4.3.

4.1.3 Query execution

iShare combines the ideas of SharedDB [34] and prior work in incremental view maintenance [24] to support shared incremental execution of scan, select, project, aggregate, and inner join operators with respect to insert, delete, and update operations.

Our design considers two subplans to be sharable if they are exactly the same or are different only in their select and project operators. Merging two different project operators unions their projection expressions to generate a new project operator. If two select operators are different, they are not merged but directly copied from the original subplans. The key idea for enabling the shared execution of the subplan is to annotate each intermediate tuple with a bitvector $B = (b_1, b_2, \ldots, b_n)$, where one bit indicates whether this tuple is valid for a query [34], and each operator is also associated with a bitvector where one bit is set if a query shares this operator. To support delete operations, each intermediate tuple is additionally associated with a bit that indicates the insertion or deletion [24]. An update operation is implemented as a delete plus an insert. We now discuss the shared incremental execution of our supported operators.

Scan We support two types of scan operators: scanning a base table or scanning the materialized output tuples of a subplan. For each new tuple scanned from a base table, we create a bitvector for it and set a bit if this bit's corresponding query shares this scan. If the tuple is from a subplan, it has a bitvector that indicates which queries this tuple is valid for. We unset a bit of this bitvector if the corresponding query does not share the scan. Finally, a tuple is output if at least one bit is set for its bitvector.

Select and project A select operator does not directly discard a tuple if the evaluation of its predicates returns false. Instead, it checks the queries sharing on this select operator and unsets their corresponding bits for this tuple's bitvector. Same as scan, it only outputs a tuple when at least one bit is set. A project operator does not change an input tuple's bitvector.

Aggregate An aggregate operator is implemented using a hash table that maps the group-by attributes to the aggregated values. We currently support SUM, AVG, COUNT, MAX, and MIN

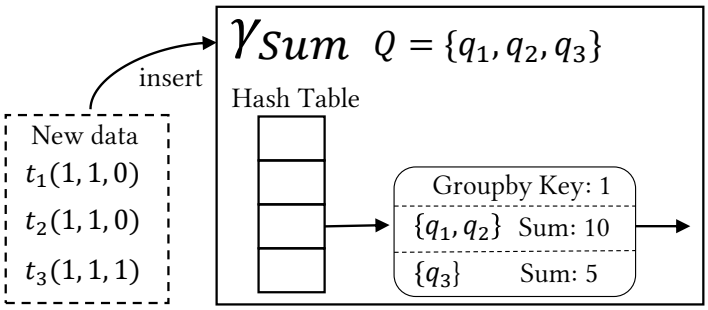


Figure 4.3: An example of a shared aggregate operator

aggregate operations. In our design, two aggregate operators are sharable if they have exactly the same group-by keys and the same aggregate expressions. If two aggregate operators have the same descendant operators with only the projection different, they have the same set of input tuples. Therefore, they can share the same aggregated value for an aggregate operation (e.g. SUM, COUNT). Consider the aggregate operator (γ_{sum}) in Figure 4.3 as an example. For each group-by key, q_1 and q_2 can share the sum value because they have the same sets of input tuples. We name those queries *a query cluster*. We use separate aggregated values for different query clusters (e.g. $\{q_1, q_2\}$ and $\{q_3\}$ are two query clusters). To process an input tuple, we identify the valid queries that this tuple belongs to based on its bitvector. Then, we use the valid queries to find the valid query clusters (e.g. t_1 in Figure 4.3 belongs to $\{q_1, q_2\}$). After, we find this tuple's group-by attributes and incorporate it into this group's aggregated values that belong to the valid query clusters. To support deletes and updates, we include a counter for every query cluster in each group to indicate how many tuples are aggregated [42]. If one group is first created for a query cluster, we output an insert. When one aggregated value is changed and the counter is larger than zero, we output an update. If the counter is decreased to zero for a group's query cluster, we remove the aggregated values in the hash table and output a delete. An output tuple is annotated with the bitvector that represents its query cluster. We note that to support MAX and MIN with deletes and updates, we need to 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.

Join We implement equi-join using symmetric hash join [106], which is widely used in incremental

execution because it is a non-blocking operator [95, 93, 49, 45]. It maintains two hash tables for inner and outer child subtrees separately. For each hash table, we use the join key as its key and the input tuple as the value. We note that when we insert a tuple into the hash table, we also keep its bitvector. A new tuple from one side updates the corresponding hash table and probes the hash table on the other side. For each matched tuple, we generate one joined tuple that adopts the intersected bitvector between the input and its matched tuple. Joined tuples with all bits unset are discarded. The implementation of non-equi-join is similar to equi-join with the difference that we maintain two arrays that materialize input tuples from the inner and outer children.

4.2 Finding the Pace configuration

iShare allows each subplan to have a different pace to reduce the total work with respect to the final work constraints of participating queries. Here, the system is allowed to lazily execute subplans in the queries that have higher final work constraints. We extend InQP to reduce the total work by considering the different final work constraints and the structure of shared query plans.

Specifically, we redefine the metric *incrementability*. Incrementability quantifies the cost-effectiveness of incremental executions and is a key metric for efficient incremental execution for a single query. In this section, we redefine this metric for shared query execution, and propose an optimized algorithm with a low running time to find a nonuniform pace configuration using incrementability.

4.2.1 Incrementability definition in iShare

If we execute a query more eagerly, we have a lower query latency but a higher resource consumption. The intuition of incrementability is to quantify how much query latency we can reduce given the same additional resources we invest. InQP defines incrementability as the ratio between the reduced final work and the increased total work.

For iShare's incrementability, we define the benefit of decreased final work differently by con-

sidering the final work constraints of different queries. Intuitively, if a pace configuration has already met the final work constraints of some queries, further increasing the paces for those queries' subplans does not yield benefit for them any more. Therefore, the benefit of a query here should be the reduced missed final work with respect to its final work constraint rather than the absolute reduction. With this observation, we now define the benefit for N queries $Q = (q_1, q_2, \ldots, q_N)$ between two pace configurations P_A and P_B, where P_A should be eagerer than P_B. This means that any pace in P_A is no smaller than the corresponding pace in P_B and there is at least one subplan's pace in P_A larger than the one in P_B. The formula of the benefit between the two is:

$$\texttt{Benefit}(P_A, P_B) = \sum_{\forall q_i \in Q} max(0, \mathcal{C}_F(P_B, q_i) - \mathcal{C}'_F(P_A, q_i)) \tag{4.1}$$

where $\mathcal{C}'_F(P, q_i) = max(L(q_i), \mathcal{C}_F(P, q_i))$.

Here, $L(q_i)$ represents a query's final work constraint and $\mathcal{C}_F(P, q_i)$ means the final work of a query given a pace configuration P. Therefore, $\mathcal{C}'_F(P, q_i)$ represents the bounded final work that is no lower than the constraint and $max(0, \mathcal{C}_F(P_B, q_i) - \mathcal{C}'_F(P_A, q_i))$ means the benefit of reducing the missed final work with respect to the query q_i's constraint. Finally, Equation 4.1 sums the per-query benefit to compute the overall benefit.

The overhead of the eager execution from P_B to P_A is $\mathcal{C}_T(P_A) - \mathcal{C}_T(P_B)$, where $\mathcal{C}_T(\cdot)$ represents the total work of a pace configuration. The incrementability definition for iShare is:

$$\texttt{InC}(P_A, P_B) = \frac{\texttt{Benefit}(P_A, P_B)}{\mathcal{C}_T(P_A) - \mathcal{C}_T(P_B)} \tag{4.2}$$

4.2.2 Pace configuration via incrementability

We now discuss the algorithm of estimating incrementability and leveraging incrementability to find the pace configuration. Finding the pace configuration essentially uses incrementability to prune the search space of different pace configurations. Therefore, the cost of estimating incrementability is the bottleneck of finding the pace configuration. As the definition shows, computing

incrementability requires estimating the final work $\mathcal{C}_F(P, \cdot)$ for each query and total work $\mathcal{C}_T(P)$ given a pace configuration P. We find that as the number and complexity of subplans grow, directly adapting the algorithm from InQP to compute the total work and the final work of a pace configuration is time-consuming. Our experiments in Section 4.4.5 shows that the original algorithm cannot finish within 30 mins for the full TPC-H query set when the max pace of each subplan is larger than 50. This is because the original algorithm computes the cost of a pace configuration by simulating its the execution from scratch. Therefore, we propose a memoization-based algorithm to quickly compute the cost of a pace configuration.

Memoization algorithm Recall that by definition, a pace k means that the subplan starts one incremental execution to process its new data whenever the system receives $\frac{1}{k}$ of the total estimated tuples. To estimate the cost of a pace configuration, the original algorithm simulates the execution of each subplan with respect to the progress of how much new data is loaded into the system. To enable more reuse opportunities, our algorithm does not strictly simulate the process of how a subplan should be executed based on the above pace definition. Instead, our memoization algorithm estimates the cost of a pace configuration by redefining the pace of a subplan to be dependent on the subplan's input data rather than the system's input data. Specifically, to estimate the cost of a subplan with pace k, we take the estimated total input data of this subplan, evenly divides it into k parts, and starts k incremental executions where each process $\frac{1}{k}$ of its total input data.

We now use an example to explain the algorithm of estimating the total work and the final work of a pace configuration. Consider a pace configuration $(3, 2, 1)$ for Q_{AB} in Figure 4.1. Here, the paces of $Subplan_1$, $Subplan_2$, and $Subplan_3$ are 3, 2, and 1 respectively. The algorithm estimates the cost of a pace configuration from the bottom to top. It first simulates 3 incremental executions for $Subplan_1$ to process its input data. We note that for each incremental execution, it updates the statistics of intermediate states (e.g. hash table size for symmetric hash join) and estimate the output cardinality of each execution. Here, each incremental execution consumes $\frac{1}{3}$ of its input data. We name the total cost of processing the input data of a subplan *private total work*. We additionally define the *private final work* of a subplan as the cost of the final execution (i.e. the 3rd

Algorithm 3: Estimate $\mathcal{C}_T(P)$ and $\mathcal{C}_F(P, \cdot)$

```
G_sorted ← Sorting subplans topologically from child
           to parent subplans
for g_i ∈ G_sorted do
    key ← Find the private pace configuration for g_i in P
    if memo_i.contains(key) then
        (pT, pF) ← memo_i(key)
    else
        (pT, pF, outCard) ← Estimating the cost and
             output cardinality of p_i simulated executions
        Add (key -¿ (pT, pF, outCard)) to memo_i
    end
    for q_i ∈ Q do
        if q_i includes g_i then
            Add pF to C_F(P, q_i)
    end
    Add pT to costT(P)
end
```

execution for $Subplan_1$). Now, we have the output cardinality of $Subplan_1$ for the 3 executions. This output cardinality will be the input data of $Subplan_2$ and $Subplan_3$. $Subplan_2$ and $Subplan_3$ take this output cardinality, and simulate 2 and 1 incremental executions respectively. Therefore, total work is estimated as the summation of the private total work of all subplans. The final work of a query includes the private final work of this query's subplans. For example, the final work of Q_A in Figure 4.1 is the sum of the private final work of $Subplan_1$ and $Subplan_2$.

Now we discuss how to reuse the prior estimated results to quickly estimate the cost of a pace configuration. The estimated results we want to reuse include output cardinality, private total work, and private final work of each subplan. We note that these estimated results depend on the paces of the subplan and its descendant subplans. We name these paces as *private pace configuration* for a subplan. To reuse prior estimated results, each subplan maintains a key-value memo table. The key is a private pace configuration and the value includes output cardinality, private total work, and private final work.

Algorithm 3 shows our memoization algorithm. We estimate the private total work and private

final work of each subplan (i.e. pT and pF in Algorithm 3) from the bottom to top. For each subplan, we first probe its memo table to look for prior estimated results. If not found, we start one simulation to estimate the private total work, the private final work, and its output cardinality (i.e. $outCard$). This information is then stored in the memo table. Finally, we add pT to the total work and add pF to the final work of the queries that include this subplan.

Algorithm of finding pace configuration Now we have an optimized algorithm for computing incrementability. We then adapt the algorithm from InQP to find the pace configuration in the shared setting. It starts at $P_{\mathbb{1}}$, which is the case of batch execution, and repeatedly increase the pace of one subplan having the highest incrementability. The loop includes two steps:

- Check whether all queries have met the final work constraints (i.e. $\forall q_i \in Q : \mathcal{C}_F(P, q_i) <= L(q_i)$) and whether all paces have reached the max pace J (i.e. $\forall p_i \in P : p_i >= J$). If either is true, the optimization stops.
- For each $subplan_i$, its incrementability is $Inc(P, P_{[p_i \backslash p_i+1]})$, where $P_{[p_i \backslash p_i+1]}$ means that we increase $subplan_i$'s pace p_i by one. Assuming $subplan_{i^*}$ has the highest incrementability, the pace configuration P is updated to $P_{[p_{i^*} \backslash p_{i^*}+1]}$.

We note that the pace of a parent subplan should be no larger than its child subplan. The step two of the above the algorithm will filter out a candidate pace configuration $P_{[p_i \backslash p_i+1]}$ if it violates this requirement.

4.3 Decomposing A Shared Subplan

iShare exploits the time slackness in the diverse final work constraints by selectively executing parts of the shared plan lazily to reduce the total work. After finding nonuniform paces for different subplans, iShare considers "unsharing" or decomposing each shared subplan for lazier execution. One "unsharing" method, *full decomposition*, is decomposing $Subplan_1$ in Figure 4.1 into two separate subplans such that Q_A and Q_B can be executed with different paces. Another "unsharing" method, *partial decomposition*, only decompose parts of $Subplan_1$, such as the join operator ($\bowtie$).

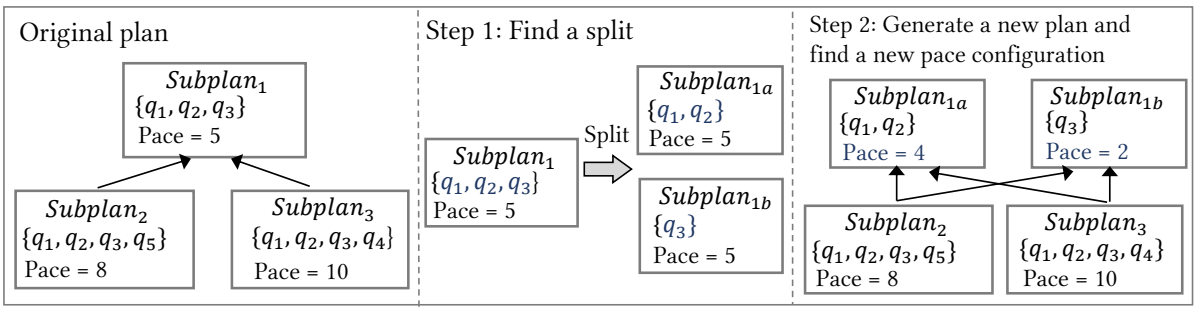


Figure 4.4: An overview of decomposing a shared subplan

In this case, the join operator is merged into $Subplan_2$ and $Subplan_3$ respectively and its two child subtrees form two separate subplans. For simplicity, our following discussion is focused on full decomposition and discuss partial decomposition in Section 4.3.3.

We find that decomposing a shared plan loses some opportunities for shared execution. Therefore, we need to systematically consider the shared opportunities and the benefit of lazy incremental executions. Figure 4.4 shows an overview of our decomposition algorithm. The input of this algorithm is the shared query plan annotated with a pace configuration. We consider decomposing $Subplan_1$ in Figure 4.4 with the two following steps:

- First, we need to split the queries that share this subplan. For example, Figure 4.4 shows that we split its query set $\{q_1, q_2, q_3\}$ into two subsets $\{q_1, q_2\}$ and $\{q_3\}$. Each subset of queries shares a single subplan (i.e. $Subplan_{1a}$ and $Subplan_{1b}$). The challenge here is that there is an exponential number of possible ways of splitting the queries. Therefore, we propose a clustering algorithm to heuristically split the queries. This algorithm uses a metric *sharing benefit* that can quickly decide whether it is worthwhile to share two sets of queries. We discuss splitting a subplan in Section 4.3.1.

- Second, we generate a new query plan for this decomposed subplan and find a new pace configuration. For example, Figure 4.4 shows that we replace $Subplan_1$ with $Subplan_{1a}$ and $Subplan_{1b}$ and we find a new pace for each subplan. We compute the new plan's total work, compare it with the total work of the original plan, and choose the one with the lowest total work. We discuss this step in Section 4.3.2.

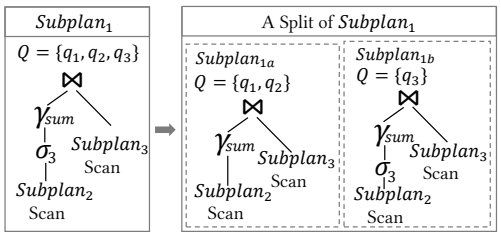


Figure 4.5: One split of $Subplan_1$

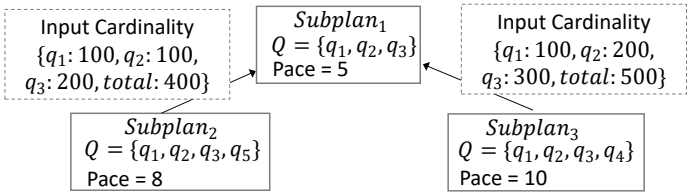


Figure 4.6: Input cardinalities of $Subplan_1$ using a pace configuration

After that, we talk about partial decomposition in Section 4.3.3 and applying our decomposition algorithm to the full plan in Section 4.3.4.

4.3.1 Finding a split for a shared plan

We define a `split` as a partitioning of the queries that share this subplan. Consider the example in Figure 4.5. The $Subplan_1$ is shared by three queries $Q = \{q_1, q_2, q_3\}$, which is split into two query sets $\{q_1, q_2\}$ and $\{q_3\}$. The new plan for one query set (e.g. $Subplan_{1a}$) copies all operators from the original subplan, except the select operators that do not belong to this query set (e.g. σ_3), and derives the same parent and child operators from the original subplan. Not shown in the figure are the project operators, which are copied from the original subplan and modified to include all attributes required by its ancestor operators.

There is an exponential number of ways of splitting a query set. Computing the total work from scratch for all splits can be time-consuming. Therefore, we define a local optimization problem of finding the split that best reduces the work of the subplan itself. The intuition here is that if a split can reduce the work of the subplan, it should also be able to reduce the total work of the whole plan. To solve this problem, we use a clustering algorithm to heuristically find the split that reduces the work of this subplan. This algorithm is based on a metric `sharing benefit` that decides whether it is worthwhile to share two query sets. Since this local optimization problem only relies on the statistics of this subplan itself, we can quickly compute this metric. In this subsection, we first talk about the definition of this local optimization problem and then discuss the clustering algorithm and sharing benefit.

Defining the local optimization problem

Before we formally define this problem, we use an example to conceptually explain it.

Explaining the optimization problem with an example Consider the query plan in Figure 4.6 as an example. Here, we have the nonuniform pace configuration for this shared plan and are considering if we should decompose $Subplan_1$. The input cardinality of $Subplan_1$ represents the estimated total number of tuples that $Subplan_1$ needs to process given the nonuniform pace configuration. For example, the input cardinality from $Subplan_3$ is 500, where 100, 200, and 300 tuples are valid for q_1, q_2, and q_3, respectively. The local optimization problem studies how to find a split that can reduce the work of the subplan to process the input data. If there is no split and $Subplan_1$ uses pace 5, we simulate 5 incremental executions for every one-fifth of the input data. The total work of this subplan is the sum of the five executions' work. We define the total work of this subplan as **local total work**.

If we split $Subplan_1$ as shown in Figure 4.5, we would have two partitions $\{q_1, q_2\}$ and $\{q_3\}$. Assuming that they use paces 2 and 4 respectively, the local total work of $Subplan_1$ is computed as follows. We first simulate 2 incremental executions for the partition $\{q_1, q_2\}$ to process the input data of $Subplan_1$. The total work of this partition is the sum of the two executions' work. We define the total work of a partition as **partial local total work**. Similarly, we can simulate another 4 incremental executions for the partition $\{q_3\}$ to compute its partial local total work. The local total work for this split is the summation of each partition's partial local total work. Therefore, we can compare the local total work of different splits and find the one with the lowest local total work.

After we define the optimization objective (i.e. local total work), we now explain the constraints for this optimization problem. We first define **local final work** of each query for this optimization problem. Recall that each partition is associated with a pace and, based on this pace, we need to simulate several incremental executions for this partition. Consider the queries q_1 and q_2 in Figure 4.5, where we simulate 2 incremental executions. The local final work of q_1 and q_2 is the work of the final execution for their partition.

Our optimization problem is constrained on the local final work of each query. We compute the **local final work constraints** as follows. Recall that we already have a final work constraint for each query. The idea is to proportionally scale each query's constraint to its local constraint for each subplan. Consider $Subplan_1$ in Figure 4.5. The query q_1 has two operators in this subplan (i.e. the join and the aggregate operators). We note that the two scan operators are not included because they are generated by the shared plan rather than by q_1. Assume that the two operators occupy 20% of the work for executing q_1 separately in one batch. Therefore, the local final work constraint for the two operators is also 20% of the constraint on q_1. We pre-compute the local final work constraints for each subplan before we start the decomposition phase.

We define the paces for all partitions in a split as the **local pace configuration**. We estimate the input data for each subplan's local optimization problem (i.e. the input cardinality in Figure 4.6) by simulating the execution of the nonuniform pace configuration found in Section 4.2.2. Therefore, the optimization problem is *finding a split along with its local pace configuration to minimize the local total work and meet local final work constraints.*

Formal definition We use $\mathcal{W}_T(O, R)$ to denote the local total work given a split $O = (O_1, \cdots, O_D)$ and a local pace configuration $R = (R_1, \cdots, R_D)$. D represents the number of partitions and R_i represents the pace of partition O_i. The partial local total work of a partition O_i and a pace R_i is denoted as $\mathcal{W}_{PT}(O_i, R_i)$. Therefore, we have

$$\mathcal{W}_T(O, R) = \sum_{i=1}^{D} \mathcal{W}_{PT}(O_i, R_i) \tag{4.3}$$

In addition, the local final work for partition O_i, and each of its queries, with pace R_i is $\mathcal{W}_F(O_i, R_i)$.

Assuming that we have H queries sharing a subplan with local final work constraints $S = (S_1, \cdots, S_H)$, the local optimization problem is formally defined as

$$\begin{aligned} \underset{(O,R)}{\text{minimize}} \quad & \mathcal{W}_T(O, R) \\ \text{subject to} \quad & \mathcal{W}_F(O_i, R_i) \leq \min_{j \in O_i} S_j \\ & \forall i \in [1, D] \end{aligned}$$

Here, the local final work of each partition $\mathcal{W}_F(O_i, R_i)$ needs to meet the lowest local final work constraint among the partition's queries (i.e. $\min_{j \in O_i} S_j$)

Finding the best split

Solving the above optimization problem requires consideration for both the split and its local pace configuration. Simply searching the space is time-consuming due to its exponential complexity. Thus, to prune this search space, we make the following key observation.

Observation Consider two partitions O_1 and O_2 in a split. We define the optimal pace, R_i^*, of a partition, O_i, as the smallest pace that allows O_i to meet its local final work constraints (i.e. $\mathcal{W}_F(O_i, R_i^*) \leq \min_{j \in O_i} S_j$). The optimal pace represents the laziest possible execution that reduces the most local total work. The observation here is that if we merge O_1 and O_2 into a single partition (i.e. O_{12}), the optimal pace R_{12}^* of O_{12} should be no smaller than that of O_1 and O_2. The reason is that the work O_{12} needs to do is the union of the work of O_1 and O_2, and this union of work will be no smaller than an individual partition. At the same time, O_{12} needs to meet the lowest final work constraint in the two partitions. Therefore, O_{12} will be no lazier than O_1 and O_2 respectively. To that end, we propose clustering the queries from the bottom up (i.e. keep merging partitions) and monotonically increasing the pace for the merged partitions (i.e. the optimal pace monotonically increases as we merge more partitions).

Sharing benefit As we are clustering the queries from the bottom up, we need to choose which partitions to merge. To do so, we develop a metric, sharing benefit, to quantify the reduced total work if we merge the two partitions. Consider merging two partitions O_i and O_j into a new partition O_{ij} The benefit is:

$$\begin{aligned} \texttt{Sharingbenefit}(Q_i, Q_j) = &\mathcal{W}_{PT}(O_i, R_i^*) + \mathcal{W}_{PT}(O_j, R_j^*) \\ &- \mathcal{W}_{PT}(O_{ij}, R_{ij}^*) \end{aligned} \tag{4.4}$$

Here, $\mathcal{W}_{PT}(O_i, R_i^*)$ represents the lowest partial total work of partition O_i given its optimal pace R_i^*.

The clustering algorithm The clustering algorithm keeps merging two partitions with the highest sharing benefit until there is no positive benefit or all queries are merged into a single partition. It starts with a split where each query is in a separate partition. The pace configuration is initialized to $P_{\mathbb{1}}$. Before we merge partitions, we increase the pace of each partition to find the optimal pace that meets the partition's local final work constraints. Afterwards, we compute the sharing benefit of each pair of partitions and merge the pair that has the highest benefit. As the observation indicates, the newly merged partition adopts the larger pace of the two old partitions and increases this pace to find the optimal one. Thus, the search for the optimal pace of the merged partition does not start from 1, but monotonically from the optimal paces of old partitions.

The split found by the clustering algorithm is outputed as the decomposed subplan. If there does not exist a decomposed plan (i.e. all queries merged in a single partition), we skip the next step of generating a new plan and finding a new pace configuration.

4.3.2 Generating a new plan & pace configuration

The first step proposes a decomposed subplan and then we check whether this new subplan can reduce the total work. We note that we cannot directly use the local pace configuration of the decomposed plan found in the previous step. This is because that the local pace configuration is based on the local optimization for this decomposed subplan and directly using it can violate the following requirement: the pace of a parent subplan should be no larger than the pace of its child subplans. Therefore, we generate a new query plan that includes the decomposed subplan and find the new pace configuration for the new plan based on the pace configuration of the original plan. Finally, if the total work of the new plan is lower than the original one, we use the new plan.

Generating a new plan Recall that we assume an execution engine that requires the query set of a subplan subsume the query sets of its parent subplans. However, the new decomposed subplan may not meet this requirement. Consider the subplan in Figure 4.7 as an example. We see that the query set $\{q_3\}$ of $Subplan_{1b}$ does not subsume the query set of its parent $Subplan_4$ (i.e. $\{q_1, q_3\}$). In this case, we split the parent subplans to align them with their child subplans (e.g. the middle

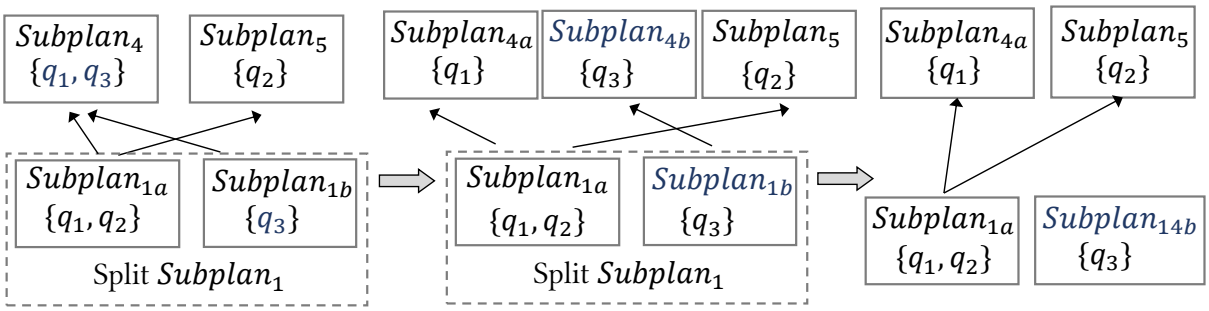


Figure 4.7: Generating a new plan using the decomposed $Subplan_1$

graph in Figure 4.7). We recursively do this for the parent subplans until we meet the requirement.

After that, we consider merging the newly generated subplans when a new subplan has only one parent subplan. Consider the $Subplan_{1b}$ and $Subplan_{4b}$ of the newly generated plan in Figure 4.7. Since $Subplan_{1b}$'s the only parent subplan is $Subplan_{4b}$, they should be merged (i.e. $Subplan_{14b}$ in the right graph of Figure 4.7).

Finding a new pace configuration Recall that the motivation of the decomposition is that the decomposed subplan enables us to execute the whole shared plan lazily by leveraging the diverse final work constraints. Therefore, we need to find a lazier pace configuration (i.e. smaller pace). The key idea is to initialize the newly generated plan with a pace configuration that is more eager than or equal to the original one (i.e. equal or larger pace) and incrementally decrease the paces. Therefore, we use two steps to generate the initial pace configuration:

- Step 1. For each newly generated subplan, we use the pace of the original subplan that the new subplan is derived from. For example, since $Subplan_{1a}$ and $Subplan_{1b}$ in Figure 4.7 are derived from $Subplan_1$, the two new plans then adopt the pace of $Subplan_1$.
- Step 2. If a newly generated subplan should be merged with another subplan (e.g. $Subplan_{1b}$ and $Subplan_{4b}$ in Figure 4.7) and the two subplans have different paces, we choose the larger pace for the merged subplan.

Starting from this pace configuration, we use a modified algorithm in Section 4.2.2 to incrementally find a new nonuniform pace configuration. The difference is that at each step instead of increasing the pace of the subplan with the highest incrementability, we decrease the pace of the subplan that

has the lowest incrementability. That is, we choose the subplan that can best lower the total work for the same final work increase.

4.3.3 Partial decomposition

Partially decomposing a subplan selects a subtree that shares the root of the subplan and then splits the subtree. For example, consider $Subplan_1$ in Figure 4.5. We can choose to split the join operator (i.e. $\bowtie$) and leave its child operators unchanged. The key idea is that we first break the subplan into three subplans: the join operator itself, and the left/right child subtree of the join operator. Afterwards, we split the join operator using the clustering algorithm.

We note that there is an exponential number of subtrees sharing the root of a subplan. Therefore, our partial decomposition considers a subset of them. Specifically, we generate the subtree candidates by starting with the root operator and gradually expanding the subtree to include its child operators using a breath-first like search. Each new subtree includes one additional child operator that is the closest to the root operator. Therefore, the number of subtree candidates is no larger than the number of operators in a subplan, which greatly reduces the optimization time while keeping the opportunities of decomposing the subplan with a fine-granularity.

4.3.4 Applying decomposition to the full plan

We now discuss applying the decomposition algorithm of a shared subplan to the full plan. After we find the nonuniform pace configuration for the full plan, we also collect statistics information required by the decomposition algorithm. We simulate the execution of the nonuniform pace configuration to generate the input cardinalities for each subplan and run each query separately in one batch to collect the local final work constraints. Then, we sort all subplans topologically from the parent to the child and apply the decomposition algorithm for each subplan in this order to generate a new plan with a smaller total work.

4.4 Experiments

Our experiments address the following questions:

- Compared to a state-of-the-art shared plan [35] that uses a single pace and two other approaches that execute queries separately, how much does iShare reduce CPU consumption given the same final work constraints? (Sec. 4.4.3)
- How much more efficient is our decomposition algorithm in reducing computing resources in iShare? (Sec. 4.4.4)
- What is the optimization overhead of iShare, and what is the overhead reduction of our memoization and clustering algorithms reduce compared to other algorithms? (Sec. 4.4.5)
- How does different levels of incrementability and varied final work constraints impact the resource consumption of baseline approaches and iShare? (Sec. 4.4.6)

All experiments are run on a server that has 196 GB of main memory and two Intel Xeon Silver 4116 processors, with 24 total physical cores. We use 20 cores for all experiments and leave the rest for the OS (Ubuntu 18.04) and other supporting processes (e.g. HDFS).

4.4.1 Prototype Implementation

iShare is implemented in Spark 2.4.0 [3]. We extend Spark SQL to support shared query execution based on SharedDB [34] and incremental execution of deletes and updates based on existing IVM algorithms [24]. We adopt techniques to reduce the cost of starting Spark jobs for incremental execution [102] . We use a Kafka [1] cluster as the data source that streams new data into Spark queries. The Kafka cluster is also used to materialize intermediate output tuples from a subplan that has two or more parent subplans. Each parent subplan pulls the new data of this subplan from the Kafka cluster. Additionally, the cluster is run on a different machine with the same hardware configuration. The two machines are placed on the same rack and have 10 Gbps Ethernet

connection to each other. We use Kafka because it provides better support for parallel data loading, out-of-memory data storage, and offset management (e.g. finding the offsets of new data for different subplans).

Users submit a set of SQL queries along with *relative final work constraints* to iShare. The relative final work constraint of a query is the ratio between the final work users want to achieve and the final work of separately executing the query in one batch. For example, a relative constraint of 0.1 means that users want to reduce the final work to 10% of the final work had the query executed in one batch. Furthermore, users can tune the relative constraints to explore the trade-off between resource consumption and query latency. We also define the *absolute final work constraint* for a query as the query's relative constraint multiplied by the work done when executing the query in one batch.

iShare uses a state-of-the-art MQO optimizer [35] to generate a shared query plan from the submitted queries. Note that we extend this optimizer to account for the materialization cost of intermediate tuples as suggested by an existing MQO optimizer [85]. iShare takes this shared plan and performs two optimizations: it finds the pace configuration and decomposes the shared plan into subplans in order to reduce its CPU consumption. iShare executes each subplan in the shared plan based on the pace configuration. Each incremental execution of a subplan uses all 20 CPU cores. When an execution is finished, we execute the next subplan based on the pace configuration. If multiple subplans start their incremental executions at the same time (e.g. they have the same pace), the child subplans are executed earlier than their parent subplans.

We use *CPU time* to represent the resource consumption of the shared query execution. It is defined as the summation of the execution time of all incremental executions of this shared plan. The query *latency* is defined as the summation of the final execution time of all subplans in this query. We report missed latency with respect to the *latency goal* in the experiment. Here, we compute the latency goal of a query by multiplying the query's relative final work constraint by the latency of its batch execution. We note that for different queries they may have different latency goals even for the same relative final work constraints. This is because the latency of

their respective batch execution is different. We report two types of missed latency, *absolute missed latency* and *relative missed latency*. The *absolute missed latency* represents difference between the tested latency and the latency goal. We calculate the absolute missed latency with $max(0, \text{tested latency} - \text{latency goal})$. The *relative missed latency* represents the percentage of the absolute missed latency compared to the latency goal. We calculate the relative missed latency with $\frac{\text{absolute missed latency}}{\text{latency goal}}$. Our experiments will show that iShare has a significantly lower CPU consumption and has a lower missed latencies relative to the baselines.

4.4.2 Experiment setup

Benchmark We use the TPC-H benchmark in our experiments and our prototype supports all 22 TPC-H queries. Furthermore, we test the two example queries Q_A and Q_B from Figure 4.1:

```
Q_A: SELECT SUM(agg_l.sum_quantity) as total_sum_quantity
     From part p,
          (SELECT SUM(l_quantity) as sum_quantity
           FROM Lineitem
           GROUP BY l_partkey) agg_l
     WHERE p_partkey == l_partkey

Q_B: SELECT ps_partkey
          FROM partsupp ps,
               (SELECT AVG(agg_l.sum_quantity) as avg_quantity
                From part p,
                     (SELECT SUM(l_quantity) as sum_quantity
                      FROM Lineitem
                      GROUP BY l_partkey) agg_l
                WHERE p_partkey == l_partkey
                AND   p_brand" == "Brand#23" AND p_size == 15)
          WHERE ps.ps_availqty < avg_quantity
```

We preload the full dataset into Kafka and let iShare pull data from Kafka at a rate of 100MB/min. This data pull rate allows the system to process all the data even if we run all 22 TPC-H queries

concurrently. We use a dataset with a scale factor of 5 to make sure that Spark does not run out of memory even for all TPC-H queries. The max pace is capped at 100. In our experiments, we run each test three times and report the average unless otherwise specified.

Baselines We compare against three baselines: *Share-Uniform*, a shared query plan with a uniform pace; *NoShare-Uniform*, independent query plans each with a uniform pace; and *NoShare-Nonuniform*, independent query plans with nonuniform pace. **Share-Uniform**, the shared query plan approach using a MQO optimizer [35], may include several separate plans, each shared by a set of queries. Separate plans are not shared because they have no sharable sub-expressions or the MQO optimizer finds the shareing cost too high (e.g. due to the high materialization cost). For each separate plan, we find the lowest absolute final work constraint among the shared queries. Then, we use the algorithm from Section 4.2.2 to find a single pace that minimizes the total work and makes the plan meet the lowest absolute final work constraint.

NoShare-Uniform executes each query separately with a single pace for each query. The pace is set to meet each query's absolute final work constraint using the algorithm from Section 4.2.2. Finally, **NoShare-NonUniform**, uses nonuniform paces to reduce CPU consumption for a single query as in InQP. Each query is broken into smaller parts and executed at different paces. We implement this idea by breaking a query into subplans at blocking operators (e.g. aggregate). The root of a subplan is either a blocking operator or the root of the query. To generate a subplan, we expand the subplan's root to gradually include its descendant operators until another blocking operator or a base relation. The pace configuration for each query with respect to the query's absolute final work constraint is found with the algorithm from Section 4.2.2

4.4.3 Low CPU consumption with the same final work constraints

In this subsection, we examine how much iShare reduces CPU consumption with similar or lower absolute and relative missed latencies compared to baseline approaches. We test 22 TPC-H queries and with two types of relative final work constraints. First, we generate a set of relative final work constraints for all queries by randomly picking relative constraints from $(1.0, 0.5, 0.2, 0.1)$

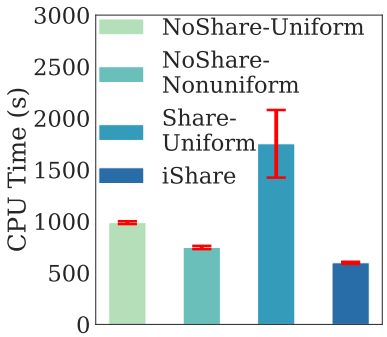


Figure 4.8: Tests of random relative constraints

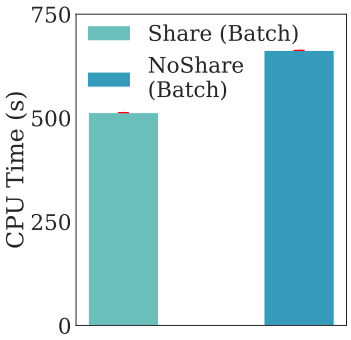


Figure 4.9: Batch execution (22 queries)

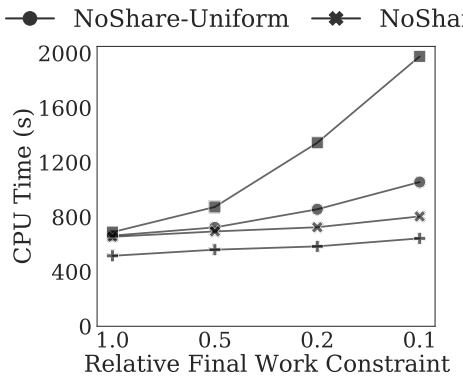


Figure 4.10: Tests of uniform relative constraints (22 queries)

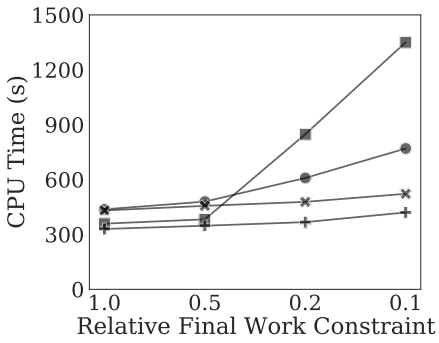


Figure 4.11: Tests of uniform relative constraints (10 queries)

for each query. Second, we use a uniform relative final work constraint from $(1.0, 0.5, 0.2, 0.1)$ for all queries.

Tests of random relative constraints For the first experiment, we test three sets of randomly generated relative constraints and report the mean, minimum, and maximum CPU time and missed latencies for all approaches. Figure 4.8 shows that iShare consumes 60.5%, 80.1%, and 34.1% of the CPU seconds compared to NoShare-Uniform, NoShare-NonUniform, and Share-Uniform. This is because iShare reduces redundant work from overlapping sub-expressions, compared to NoShare approaches and iShare lazily executes part of the shared plan to avoid the overly eager execution, relative to Share-Uniform. In addition, Share-Uniform has a larger variance in CPU consumption due to its need to meet the lowest absolute constraints, which is both highly variable from the random selection of relative constraints and the limiting factor in controlling performance.

For completeness, we show the CPU time reduction of executing the shared plan of Share-Uniform in one batch relative to executing all the queries independently in one batch, in Figure 4.9. Thus, the overhead of overly eager execution is what makes Share-Uniform have higher CPU consumption than other approaches.

	Random				Uniform			
	Mean %	Mean Sec.	Max %	Max Sec.	Mean %	Mean Sec.	Max %	Max Sec.
NoShare-Uniform	37.24	1.37	854.69	30.48	21.36	0.98	884.47	31.54
NoShare-Nonuniform	6.37	0.42	123.41	9.06	5.66	0.38	192.59	7.60
Share-Uniform	44.24	1.72	895.19	31.92	29.48	2.02	802.19	33.53
iShare	6.39	0.22	153.66	4.68	7.17	0.34	207.24	7.14

Table 4.1: Missed latencies of random and uniform relative constraints.

The `Random` column in Table 4.1 shows `Mean Sec.` and `Max Sec.` representing absolute missed latencies, and `Mean %` and `Max %` representing relative missed latencies. The minimum and median missed latencies, not shown, for all approaches are zero. We see that iShare has less absolute and relative missed latencies compared to the baselines. However, the main reason for missed latency is the inaccuracy of the cost model. Additionally, the maximum absolute and relative missed latencies for NoShare-Uniform and Share-Uniform are large because parts of some queries are not incrementable. Thus, using a single pace to eagerly execute these queries does not reduce the query latency, resulting in a high missed latency. One such example is Q_{15}. It maintains two aggregate operators, where one aggregate operator, `max` is parent of another aggregate operator, `sum`. When the aggregated values in the the `sum` operator are changed, this operator will output a delete and an insert operation to the parent `max` operator. If a max value is deleted, the `max` operator needs to rescan all arrived values to find the new max one. Eagerly maintaining the whole query does not reduce the cost of finding a new max value, which is why NoShare-Uniform and Share-Uniform have high query latencies. However, NoShare-Nonuniform and iShare use different paces for different parts of a shared plan and can maintain the `max` operator lazily to avoid deleting the max value and, thus, the cost of finding a new max value.

Tests of uniform relative constraints Our second test uses uniform relative final work constraints for all queries. Specifically, we use relative constraints of 1.0, 0.5, 0.2, and 0.1, and report the CPU time and missed latencies. Figure 4.10 shows that iShare lowers CPU consumption compared

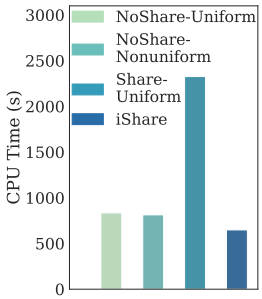


Figure 4.12: Manually tuned pace

	Mean %	Mean Sec.	Max %	Max Sec.
NoShare-Uniform	4.40	0.16	96.85	3.45
NoShare-Nonuniform	0	0	0	0
Share-Uniform	32.34	1.15	711.57	25.37
iShare	0	0	0	0

Figure 4.13: Missed latencies for manually tuned pace

to the baselines for all relative constraints. We also observe that the Share-Uniform has similar CPU consumption compared to NoShare approaches when the relative constraint is 1.0. A relative constraint of 1.0 means that the tested latency of a query should be no larger than the latency of executing the query independently in one batch. Therefore, even when the relative constraint is 1.0 for all queries, the absolute constraint for each query is different. To meet the lowest absolute constraint, Share-Uniform has overly eager executions, which offsets the benefit of shared query execution. To show the benefit of Share-Uniform, we perform an additional test of 10 TPC-H queries, Q_4, Q_5, Q_7, Q_8, Q_9, Q_{15}, Q_{17}, Q_{18}, Q_{20}, and Q_{21}, which have significant amounts of overlapping work and, for the same relative constraint, they have similar absolute final work constraints. We see, in Figure 4.11, that Share-Uniform has lower CPU consumption compared to NoShare approaches, because the absolute constraints are less diverse and Share-Uniform has smaller overhead of overly eager execution. This leads to better shared performance. For all constraints, iShare has lower CPU consumption compared to all other approaches.

The `Uniform` column in Table 4.1 shows the mean and maximum missed latencies for all queries tested in Figure 4.10 and Figure 4.11. Again, the minimum and median missed latencies are not shown because they are zero for all approaches. We have the same observation as the test of random relative constraints. iShare has less absolute and relative missed latencies compared to other approaches and NoShare approaches have higher maximum missed latencies because one tested query (i.e. Q_{15}) is not incrementable.

Tests for manually tuned pace configuration In this test, we manually tune the pace configuration

to make all approaches meet the latency goals (a relative constraint of 0.1). If there are queries that cannot meet the latency goal for some approaches, we make sure that these queries have the smallest missed latencies.

For NoShare-Uniform, we test all paces for each query; for Share-Uniform, we tune the pace for the whole plan; and for NoShare-NonUniform and iShare, we tune the pace configurations by setting smaller relative final work constraints for queries that otherwise have missed latencies.

Figure 4.12 shows that iShare uses uses 77.7%, 80.0%, and 27.9% of the CPU seconds compared to NoShare-Uniform, NoShare-Nonuniform, and Share-Uniform, respectively, for manually tuned pace configurations. Table 4.13 shows the results of the mean and maximum missed latencies, with the minimum and median excluded as they are all 0. We see that both NoShare-Uniform and Share-Uniform still have missed latencies because the query Q_{15} is not incrementable and increasing a single pace for the query plan could not achieve the desired query latency.

4.4.4 Performance impact of decomposition

We demonstrate how much our decomposition algorithm reduces CPU consumption compared to using the nonuniform pace configuration only. Here, we use `iShare (w/o unshare)` to represent the iShare variant that does not use the decomposition algorithm. We denote the variant with all optimizations as `iShare (w/ unshare)`. We create a query set that has much overlapping work and show the benefit of the decomposition algorithm for this "sharing-friendly" query set. Specifically, we take the 10 TPC-H queries in Figure 4.11, modify their predicates to generate new 10 TPC-H queries, and combine the original and new queries to create a new query set with 20 queries. For each query, we modify the two types of predicates: equality predicate (e.g. $name = "Tom"$) and range-based predicates (e.g. $A > 10$ and $A < 20$). For 50% of the equality predicates, we use a different value (e.g. $name = "Jerry"$), and for a range-based predicate, we generate a new predicate that with an overlap up to 50% (e.g. $A > 15$ and $A < 25$). We test uniform relative final work constraints of 1.0, 0.5, 0.2, and 0.1 for all 20 queries.

Figure 4.14 shows iShare (w/o unshare) has similar CPU consumption to iShare (w/ unshare)

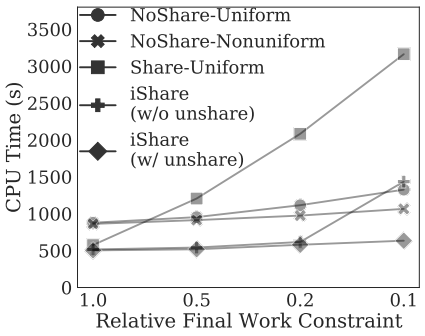


Figure 4.14: Tests for decomposition algorithm

	Mean %	Mean Sec.	Max %	Max Sec.
NoShare-Uniform	16.62	0.79	910.19	32.45
NoShare-Nonuniform	2.71	0.29	104.49	7.79
Share-Uniform	28.76	1.79	670.8	26.08
iShare (w/o unshare)	27.01	0.98	996.32	35.53
iShare (w/ unshare)	0.39	0.01	32.82	0.96

Figure 4.15: Missed latencies for the test of decomposition

for the first three relative constraints. However, when we use the relative constraint 0.1, we find that iShare (w/o unshare) uses more CPU seconds than the NoShare approaches, because there is significant overhead of overly eager execution introduced by shared subplans. For example, consider Q_{15} and its variant Q'_{15}. iShare (w/o unshare) shares the subplan of maintaining the max aggregate operator for the two queries. Since Q_{15} and Q'_{15} have different (but overlapping) predicates, the max aggregate of the shared plan needs to do more work than the individual max aggregate in each query. Therefore, a lower relative final work constraint (e.g. 0.1) pushes the shared plan to execute more eagerly. Eagerly maintaining this max operator is expensive. iShare (w/ unshare) avoids its cost by decomposing the shared subplan between Q_{15} and Q'_{15} and executing each lazily. iShare (w/ unshare) uses 52.3%, 58.6%, 31.8%, and 71.8% of the CPU seconds compared to NoShare-Uniform, NoShare-Nonuniform, Share-Uniform, and iShare (w/o unshare), respectively.

4.4.5 Optimization overhead

We test the optimization time of iShare, baselines, and an iShare variant that computes incrementability using a simulation algorithm of InQP rather than the memoization algorithm in Section 4.2.2. We denote this approach as `iShare (w/o memo)` and with the memoization as `iShare (w/ memo)`. We use all TPC-H queries, vary the value of the max pace from 10 to 100, and set a low relative final work constraint for all queries (i.e. 0.01) such that the optimization only finishes when we reach the max pace.

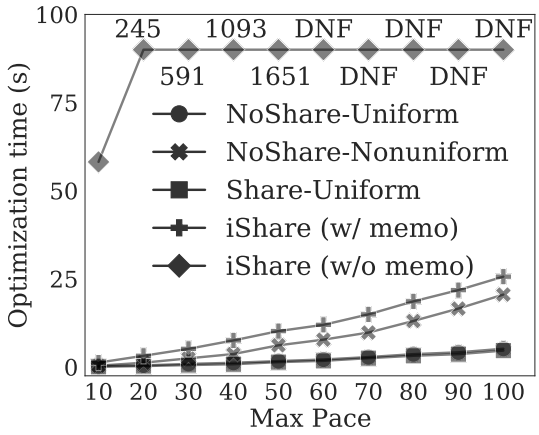


Figure 4.16: Overhead of end-to-end optimization

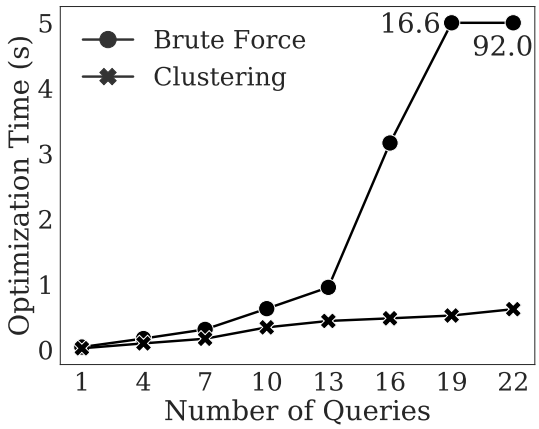


Figure 4.17: Optimization overhead of clustering algorithm

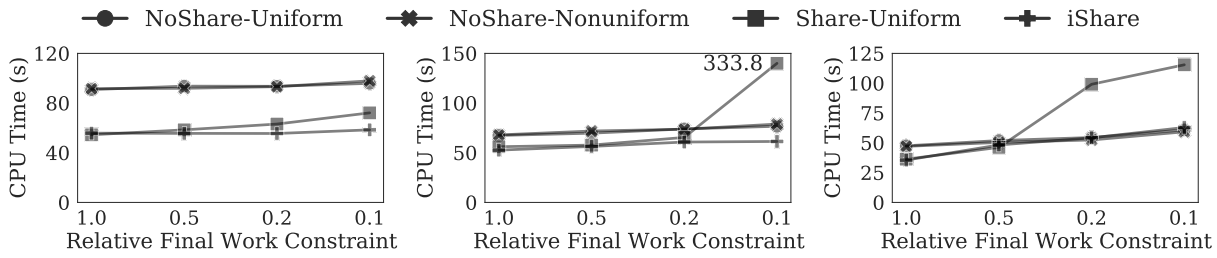
Figure 4.16 shows the optimization time, where iShare (w/ memo) has a much lower running time compared to iShare (w/o memo). We mark a test case as `DNF` if it does not finish within 30 minutes, where iShare (w/o memo) fails when the max pace is larger than 50. In the worst case, iShare (w/ memo) uses 25.6s to finish the optimization. While it is higher than the baselines, we believe the significant reduction in CPU consumption justifies the cost.

We, also, compare our clustering algorithm for decomposing a subplan to searching all possible ways of splitting the queries of a subplan (denoted as `Brute-force`). We use a max pace 100 and vary the number of queries we need to optimize. Figure 4.17 shows that the running time of our clustering algorithm is significantly smaller than that of the Brute-force method, which increases exponentially as we increase the number of queries to optimize.

4.4.6 Impact of incrementability and final work

In this subsection, we test how incrementability and relative final work constraints impact CPU consumption with three pairs of queries: 1) PairA: Q_5 and Q_8; 2) PairB: Q_7 and Q_{15}; and 3) PairC: Q_A and Q_B. PairA consists of two queries that are amenable to incremental executions, which does not have significant increases in CPU consumption with eager execution. PairB includes an incrementable query (Q_7) and a query that is not amenable to incremental executions (Q_{15}).

Finally, PairC has two queries that are less incrementable. For each pair, we fix one query's relative constraint to 1.0 (i.e. Q_5, Q_{15}, and Q_A) and change the relative constraint of the other query.



(a) CPU consumption of Q_5 and Q_8

(b) CPU consumption of Q_7 and Q_{15}

(c) CPU consumption of Q_A and Q_B

Figure 4.18: Micro benchmarks for queries with varied levels of incrementability and relative final work constraints

	PairA: Q_5 and Q_8				PairB: Q_7 and Q_{15}				PairC: Q_A and Q_B			
	Mean %	Mean S.	Max %	Max S.	Mean %	Mean S.	Max %	Max S.	Mean %	Mean S.	Max %	Max S.
NoShare-Uniform	0.07	0.01	1.71	0.15	2.77	0.15	21.01	1.40	0.20	0	4.74	0.11
NoShare-Nonuniform	0	0	0	0	5.26	0.56	37.85	2.30	1.53	0.07	14.28	0.69
Share-Uniform	0.58	0.04	7.26	0.57	7.43	0.28	107.97	3.60	0	0	0	0
iShare	0	0	0	0	1.44	0.17	17.09	1.14	0	0	0	0

Table 4.2: Missed latencies of micro benchmarks.

Figure 4.18a shows that the overhead of overly eager execution for Share-Uniform is small, since Q_5 and Q_8 are amenable to incremental executions. Thus, Share-Uniform has lower CPU consumption than NoShare approaches, and a slightly lower CPU consumption than Share-Uniform due to its nonuniform pace configuration. When we mix a less incrementable query (i.e. Q_{15}) with an incrementable query Q_7 and eagerly execute the incrementable query, Q_7, Share-Uniform is no longer better than the approaches of not sharing. Figure 4.18b shows that, when the relative constraint is 0.1, Share-Uniform consumes more CPU seconds than NoShare approaches. iShare can bring the benefit of shared query execution and lazy execution together and, thus, lead to a lower CPU consumption than the baselines. Finally, mixing two less incrementable queries Q_A and Q_B in Figure 4.18c, we also see that Share-Uniform becomes sub-optimal when either query's relative final work constraint is decreased. Here, iShare first shares Q_A and Q_B for the relative constraints 1.0 and 0.5. When the relative constraint is 0.2 and 0.1, it decomposes the shared plan and executes Q_A and Q_B separately, and has similar performance to NoShare-Uniform and

NoShare-Nonuniform in these cases. Table 4.2 shows the missed latencies of the three pairs. We see that all approaches have small missed latencies except the Share-Uniform for PairB because Share-Uniform executes the non-incrementable query Q_{15} eagerly.

4.5 Summary

We present iShare as a new optimization framework that exploits heterogeneous latency goals to judiciously decide what parts of a query to share and how eagerly or lazily to execute different parts of the shared plan. To address the challenge of a complex optimization space, we propose a memoization-based algorithm to quickly find the nonuniform pace configuration and a key metric sharing benefit to decide which parts of a query to share at a low optimization time. Our experiments demonstrate that iShare can significantly reduce CPU consumption for queries with diverse latency goals compared to the shared query execution using a single pace and two approaches that execute queries separately.

CHAPTER 5

INTERMITTENT QUERY PROCESSING

InQP and iShare assume that the new data arrives in a steady rate. However, we find that in many applications the new data can arrive intermittently or at a low rate. In these applications, the query is not necessarily active all the time (e.g. using deferred refresh), and can release some resources (i.e. memory) during inactivity. Interestingly, we find that the knowledge about the new data is predictable. Leveraging the predictable knowledge of the estimated size of the new data and distribution of the relations having new data, we can selectively keep a subset of resources that can best accelerate the query processing for the new data, and release the others to reduce memory consumption. Therefore, we propose intermittent query processing (IQP) to exploit the knowledge of incoming data to accelerate updating the result of a standing query for new new data with limited memory consumption. We find a few applications exhibiting intermittent and predictable workloads [64, 107, 83, 29, 92] 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 [29]. 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 [25] and cleaning [83]. Given a dirty dataset, the data cleaning system splits it into the clean partition, which includes most of the data [83], 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 5.4.3 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.

IQP 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 IQP) 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. We propose *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 IQP 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 DISS 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.

Chapter 5.1 provides an overview of DISS. We present our intermediate states selection algorithm, and its extensions and system optimizations in Chapter 5.2 and Chapter 5.3 respectively. After, we discuss the prototype implementation and its evaluation in Chapter 5.4.

5.1 DISS Overview

In this section we discuss major components of DISS and a query life cycle, as shown in Figure 5.1. 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).

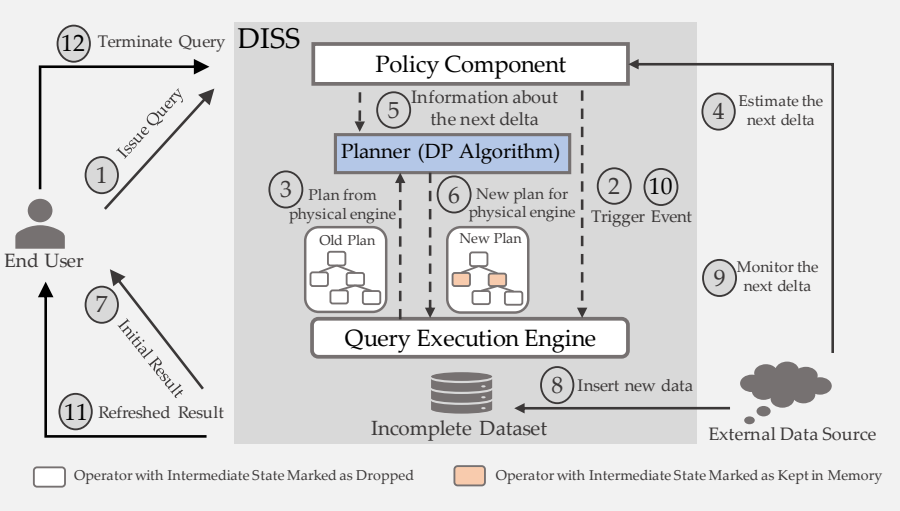
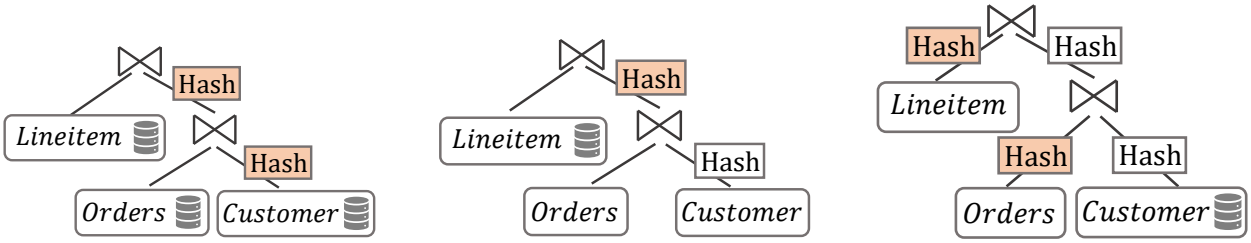


Figure 5.1: IQP Prototype Overview

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 5.1 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 [14], 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.



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

Figure 5.2: Examples of Intermediate State Selection

5.2 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 Chapter 5.3. 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 Chapter 5.3. We begin with a motivation in Chapter 5.2.1, present an overview of DISS in Chapter 5.2.2, and elaborate on the algorithm in Chapter 5.2.3.

5.2.1 Motivation

We propose an IQP system *DISS (Delta-oriented Intermediate State Selection)* that considers using a limited memory budget to 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 5.2a: $Lineitem \bowtie (Orders \bowtie Customer)$ 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 5.2b), keeping the top hash table (colored in Figure 5.2b) 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 5.2c shows a possible solution using symmetric hash joins [101] (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 5.2c) and keep the other two (colored in Figure 5.2c) 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.

5.2.2 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 pre-

cise; 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.

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.

5.2.3 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

Table 5.1: Notation Table

Notations	Meaning
$\|R\|$	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. Ch. 5.2.3)
$\mathcal{F}^{op}(m)$	min. cost for $op\Downarrow$ to emit $R+$ (c.f. Ch. 5.2.3)
$C_{op}(d)$, $C(d)$	cost for processing data d ($d = D$, ΔD, or $D+$) (estimated by RDBMS' cost estimator)

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 5.1.

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.

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 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 $R_{op}(D+) = op(D \cup \Delta D)$).

Algorithm 4: 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 $\mathcal{F}^{op}(m)$

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 $R_{op}(D+)$.

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 [86] 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 4, 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.

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

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 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 6: Keep Right Hash Table Only

Operator: **Hash Join**

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

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

Cost:

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

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)`. 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 sub-tree 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.

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 [106]). 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,

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.

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.

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, 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 $\mathbb{O}(NM^2)$.

5.3 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 [24].

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 Chapter 9 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 $\mathbb{O}(K2^K NM^2)$. 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.

5.4 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 5.4.3)
- What is the impact of delta prediction quality on DISS performance? (Section 5.4.4)
- How does DISS's dynamic programming algorithm gracefully trade memory consumption

for efficient delta processing compared to greedy algorithms? (Section 5.4.5)

- 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 5.4.6)
- How much does DISS lower delta processing time in workloads with deletes? (Section 5.4.7)

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.

5.4.1 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 5.4.3. 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 [41, 24]). For this prototype, each query monitors the number of delta tuples and when

it exceeds a threshold or when an pre-defined 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 IVM system, DBToaster [5], 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 $Lineitem \bowtie Orders$. 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 [2]. We denote this as ReBatch in our tests.

5.4.2 Benchmark Setup

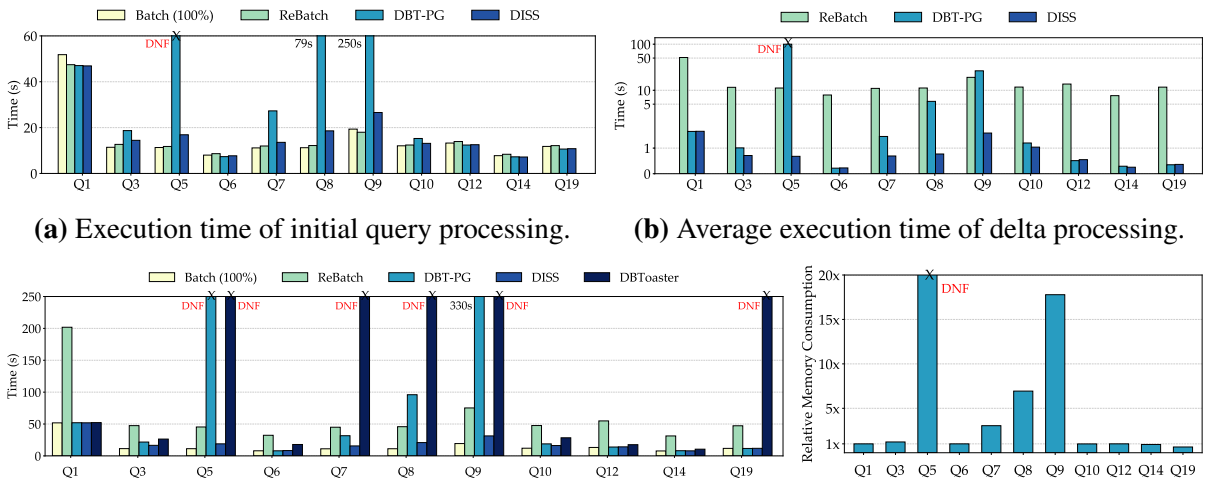
Our experiments use the TPC-H benchmark, a decision support benchmark that analyzes the activity of a wholesale supplier, and join ordering benchmark (JOB) [62] that is built on IMDB datasets to test queries with many joins (i.e. up to 16-way join). Our current prototype supports flat select-project-join-aggregate (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.

5.4.3 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.

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.



(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 5.3: DISS with late data processing on TPC-H scale factor 5.

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 5.3, where we report the initial query processing time (Figure 5.3a), average delta processing time (Figure 5.3b), total query processing time, which is the sum of initial query processing time and all delta processing time (Figure 5.3c), and relative memory usage of DBT-PG compared with DISS (Figure 5.3d). In Figure 5.3a, 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 5.3b, 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 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 5.3c and Figure 5.3d 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 5.3c 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 SF 0.1 and find that while all queries are finished by DBToaster, DBT-PG is faster in most cases.

Table 5.2: Aggregated results of join ordering benchmark

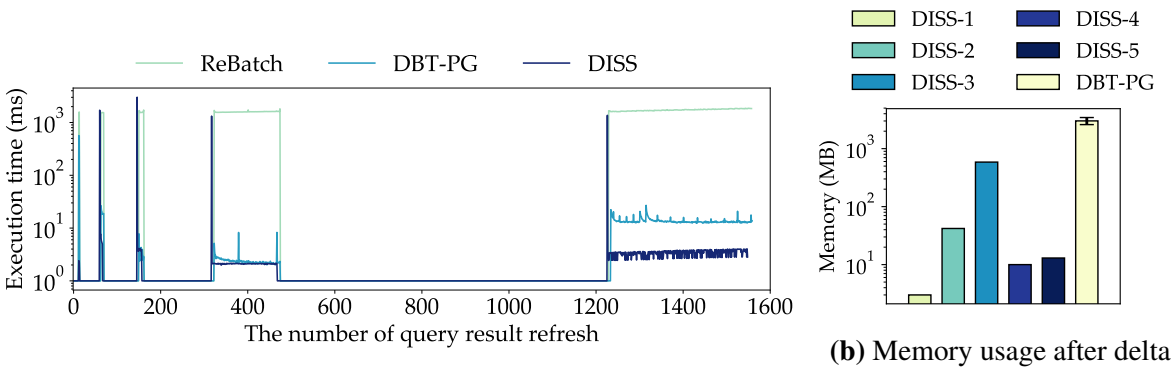
		ReBatch	DBT-PG	DISS
Number of Query Finished		33	28	33
Total Query Processing Time of One Query (s)	Avg	15.9	77.5	10.6
	Max	112.6	430	72.1
	Min	2.7	3.0	2.2
Memory Consumption (GB)	Avg	0	14.3	1.5
	Max	0	86.7	12.1
	Min	0	0.35	0.2

We also test the performance of DISS for JOB [62]. 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 5.2 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.

DISS with HoloClean

Our second IQP use scenario is a data cleaning system HoloClean [83]. 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 gets the information about the next delta from HoloClean regarding which relations it is cleaning and the tuples/sec of cleaning for each relation.



(a) Execution time of delta processing

(b) Memory usage after delta phase

Figure 5.4: DISS with HoloClean (Q8)

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 5.4a. We also show memory consumption of DISS when the query is inactive for each relation cleaning in Figure 5.4b. For example, DISS-1 in Figure 5.4b represents the memory consumption when HoloClean is cleaning the first relation SUPPLIER. Figure 5.4b 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 5.4b.

In Figure 5.4a, 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 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.

5.4.4 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.

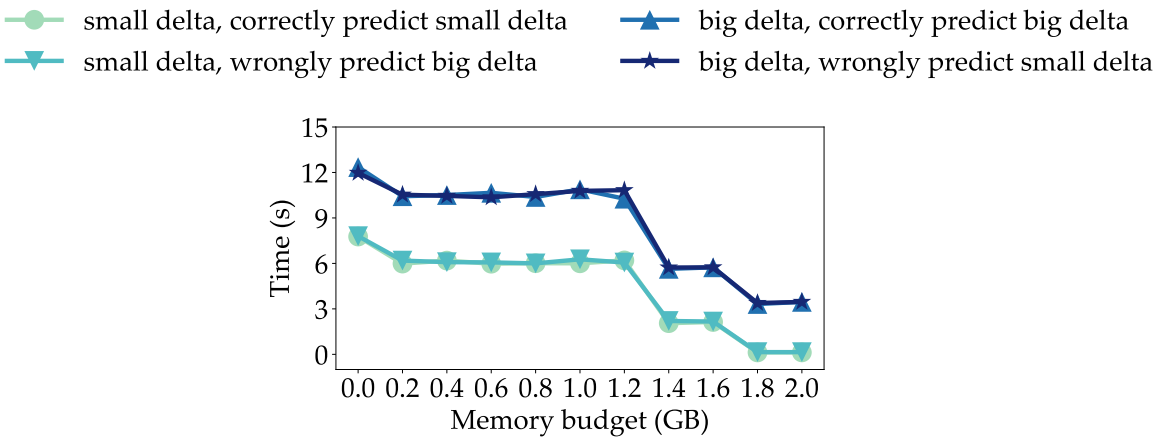
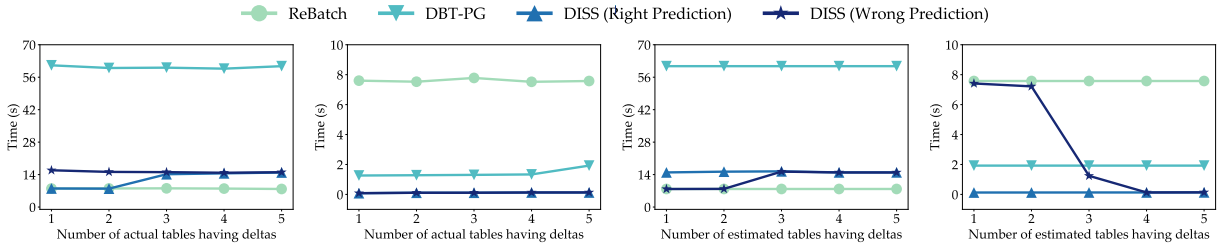


Figure 5.5: Quality of cardinality prediction (Q8)

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 5.5. 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.



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

Figure 5.6: 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).

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 5.6a and Figure 5.6b), 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 5.6c and Figure 5.6d), 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 5.6 shows that in the overestimation case, the batch processing time of a wrong prediction for DISS (i.e. DISS (Wrong Prediction) in Figure 5.6a) 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

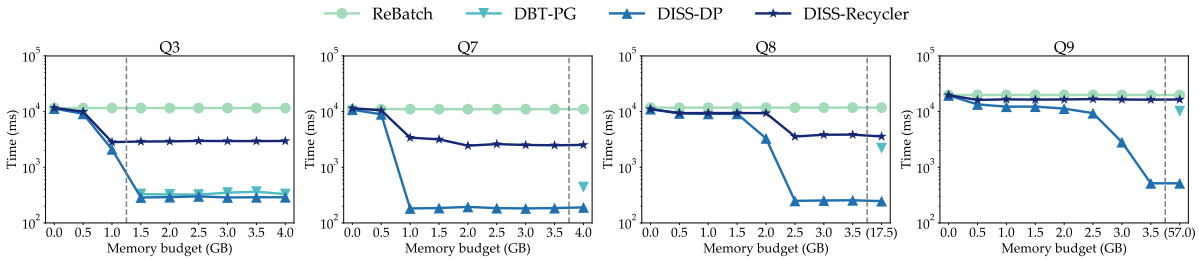


Figure 5.7: Delta processing time under different memory budgets (all relations have a single 1% delta): DISS and ReBatch can work for all memory budgets, but DBT-PG only works when the memory budget is larger than the vertical dashed line. (Y-axis is log-scale)

can correctly predict at least 3 incomplete relations out of all 5 (shown in Figure 5.6d). The above two experiments show that DISS can outperform ReBatch and DBT-PG even when the prediction is not perfect.

5.4.5 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 [72] (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 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 5.7. 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.

5.4.6 Impact of Additional Operators

In IQP, 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.

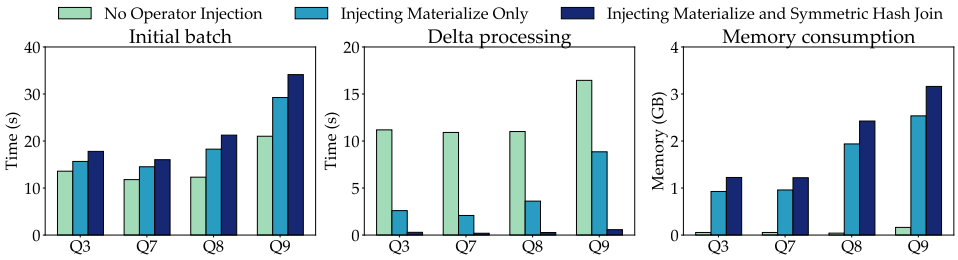


Figure 5.8: Impact of injecting operators in DISS

Figure 5.8 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.

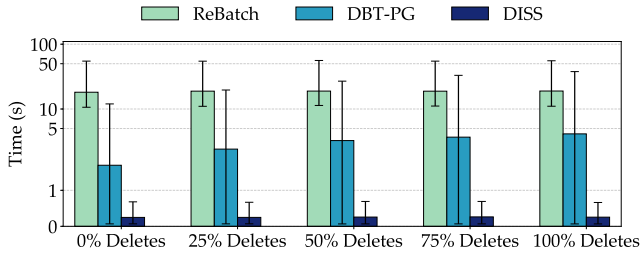


Figure 5.9: Average, min, and max delta processing time by varying percentage of deletes (1% delta)

5.4.7 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 5.9 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.

5.5 Summary

We introduce IQP as a new query processing method for standing queries that balances query processing latency and controlled memory 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.

CHAPTER 6

RELATED WORK

We discuss the related work on incremental view maintenance algorithms (IVM), view maintenance policies, view and intermediate states reuse, continuous query processing and stream computing, query pause and resume, shared query execution, and cardinality estimation.

Incremental View Maintenance Algorithms Materialized views are cached or pre-computed query results that are derived from base tables. 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. [14] 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 [42, 40], supporting recursive views and nested subqueries [42, 108, 74], and optimizing incremental executions for semi-join, outer join, and acyclic joins [39, 61, 51]. New IVM algorithms are also designed to optimize scenarios such as base tables and intermediate results having IDs [57], matrix calculations [75, 76], and deep learning [73]. Due to space constraints we point the reader to a comprehensive survey on materialized views [24].

We believe that these algorithms are orthogonal to TQP because TQP assumes existing IVM algorithms and considers the system strategies of how to using these algorithms, such as which intermediate states to keep (as in IQP) and at what execution frequencies different parts of a query should be executed (as in InQP and iShare).

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 [28]. *Immediate view maintenance* updates the view whenever base tables are updated or new tuples are inserted [5, 21]. This approach lowers query latency with higher cost of view maintenance. On the other hand, a *deferred view* [27] 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 [111]. *Snapshot view* [28] maintains a view that is consistent with a snapshot of base tables, but allows a stale result (i.e. not consistent with up-to-date base tables). It makes a better trade-off between the query latency and view maintenance cost than the previous two approaches, does not always return up-to-date results to users.

These works are mostly related to InQP. InQP is different from them in that it decomposes a query into multiple query paths and assign each query path a different pace based on the incrementability, while existing IVM approaches use a uniform execution pace for maintaining the whole query.

He et al. [48] 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.

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 [4, 43, 44, 58, 84]. Dynamic materialized views [113, 36] 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 [8] and selectively materializing sub-expressions can minimize query response time at "data center" scale [53]. Chaudhuri et al. incorporate materialized views into query optimization [22] and Mistry et al. share materialized views for multi-query optimization [71].

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) [52, 72]. Dursun et al. consider reusing intermediate data structures from join algorithms in main-memory databases [30]. ReCache studies the same problem for heterogeneous data sources [10]. Intermediate results can also accelerate approximate query pro-

cessing [32] and feature selection workloads [109].

These research projects are related to IQP, but the difference is that IQP considers how to efficiently incorporate delta into an existing query result, rather than storing materialized views or intermediate states for future queries.

Continuous query processing and stream systems Many continuous query processing and stream systems adopt IVM as its query execution engine to provide low query latency [21, 3, 17, 20]. 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 [3, 17, 20]. Several projects focus on finding query plans or execution plans to optimize different performance metrics, such as maximizing output rate [104], minimizing per-tuple processing latency [18], lowering memory consumption [11, 18], producing fast early results [98], or a mix of these metrics [9, 88].

These research projects are related to InQP. However, they 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 them 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.

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. [19] design lightweight asynchronous check-pointing 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 [38, 7]. Query suspend and resume is related to IQP, but has the difference in that a suspended query in the aforementioned approaches does not necessarily finish processing the desired partial workload and cannot present the corresponding incomplete query result to end users. In addition, they do not consider the knowledge about the new data arrival pattern as in IQP.

Multi-query optimization and shared query execution Many MQO and shared query execution projects focus on ad-hoc queries. Some work [67, 34, 82, 112, 85, 35] considers batching several queries together to exploit their common sub-expressions and builds a single query plan to maximally share the work of batched queries. Other projects consider specific operators, such as sharing scans [81, 79, 90] and joins [66, 16]. In addition, some projects also consider reusing intermediate results of running queries to process newly submitted queries on-the-fly [47, 78]. This idea of shared query execution is widely used in continuous query processing or stream computing [59, 105, 46, 99, 97, 110, 50]. For example, shared arrangements [70] considers sharing the intermediate states across standing queries and supports different indexed views over the same states. Other projects [55, 56] consider sharing the execution of standing and ad-hoc queries. Prior research works also studied whether queries should be shared by considering the overhead introduced by parallel execution [54] or materializing intermediate results [85].

iShare is different from these projects in that it considers heterogeneous latency goals and judiciously shares query execution by considering the overhead of eager execution.

Cardinality Estimation Conventional databases [86, 62] use statistical information (e.g. selectivity or number of distinct values) collected from base tables, to estimate cardinalities. Several techniques, such as data sketching [31, 15], index sampling [63], sampled executions [100], and leveraging runtime execution information [23], 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 [77, 68, 60, 89, 69]. 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 [103, 33]. Viglas et al. [103] 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 because 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.

CHAPTER 7

CONCLUSION AND FUTURE WORK

This dissertation presents a new query processing paradigm, Thrifty Query Processing (TQP), to strike the middle ground between eager and lazy query processing methods (e.g. batch vs. CQ) for querying a dataset under changes. It addresses the resource-efficiency challenge by exploiting the time slackness information to reduce CPU and memory consumption while providing similar query latencies compared to existing approaches. TQP includes three pieces of work. We design InQP to reduce the CPU consumption for a single query by selectively deferring the execution of some parts of the query that significantly increase CPU consumption if they are executed eagerly. We further present iShare to judiciously share queries with different performance goals to exploit the benefit of shared query execution and avoid the overhead of overly eager execution. Finally, we design IQP to reduce memory consumption when the data arrival rate becomes low and the query has a long time of inactivity.

We believe TQP has wide applications in both on-premise and cloud databases. For the database that runs in a resource constrained scenario (i.e. on-premise database), users are allowed to adjust the performance goals to spare resources for other queries (e.g. ad-hoc queries), which improves the overall query throughput. On the other hand, TQP can be integrated in the cloud databases to provide the stateful standing query service. Users are allowed to explore the trade-off between query performance and resource consumption and the underlying system can intelligently choose the right system strategies to save resources while meeting the performance goal. We envision that since TQP exploits the full spectrum between eager and lazy query execution, it can also be used to address dynamic workloads by adaptively adjusting when to maintain the query and how many intermediate states to keep.

Looking into the future, our research vision is to build a resource-efficient and general data analysis pipeline system with end-to-end optimizations. We plan to explore this topic in three directions:

User-driven optimization Today's interactive data analysis systems have advanced to predict users' access patterns including what queries users will issue and what time they expect to see the query results. This valuable information can be integrated into the database to speculatively start the query early even when not all data is ready or opportunistically materialize useful intermediate states. More interestingly, this information can be pushed down into the early stages of data pipelines to prioritize preparing data and computing intermediate results that are most useful and critical to users.

Cross-stage optimization Data pipelines are becoming more and more complex with multiple stages involved including data collection, preparation, analysis, and visualization. Different stages are relatively isolated from each other in that they are deployed on separate systems or have different run-time environments. The problem is that much useful information siloed in each stage cannot be shared to optimize the full data pipelines as a whole. I advocate a holistic approach that shares meta information across different stages to enable cross-stage optimizations. As an example, databases can actively ask data collection systems to prioritize loading particular data and generating useful information.

General incremental execution Data pipelines are involving complex operators (e.g. UDFs and machine learning inference) beyond the relational ones. This research direction studies how to expand my incremental execution engine to more general and complex operators, and support holistic optimizations across relational and non-relational operators. For example, the key metric for efficient incremental execution is incrementability, which quantifies the cost-effectiveness of incremental executions. One research problem is how to measure the incrementability of UDFs, where the semantics might be unknown to the database.

REFERENCES

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

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

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

[4] Sanjay Agrawal, Surajit Chaudhuri, and Vivek 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.

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

[6] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam 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.

[7] Panagiotis Antonopoulos, Hanuma Kodavalla, Alex Tran, Nitish Upreti, Chaitali Shah, and Mirek Sztajno. Resumable online index rebuild in SQL server. *PVLDB*, 10(12):1742–1753, 2017.

[8] Michael Armbrust, Eric Liang, Tim Kraska, Armando Fox, Michael J. Franklin, and David 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.

[9] Ahmed Ayad and Jeffrey 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.

[10] Tahir Azim, Manos Karpathiotakis, and Anastasia Ailamaki. Recache: Reactive caching for fast analytics over heterogeneous data. *PVLDB*, 11(3):324–337, 2017.

[11] Brian Babcock, Shivnath Babu, Mayur Datar, and Rajeev 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.

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

[13] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. One SQL to rule them all - an efficient and syntactically idiomatic approach to management of streams and tables. In *SIGMOD 2019*, pages 1757–1772. ACM, 2019.

[14] José A. Blakeley, Per-Åke Larson, and Frank Wm. 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.

[15] Walter Cai, Magdalena Balazinska, and Dan 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.

[16] George Candea, Neoklis Polyzotis, and Radek Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *PVLDB*, 2(1):277–288, 2009.

[17] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[18] Donald Carney, Ugur Çetintemel, Alex Rasin, Stanley B. Zdonik, Mitch Cherniack, and Michael 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.

[19] Badrish Chandramouli, Christopher N. Bond, Shivnath Babu, and Jun 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.

[20] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, John C. Platt, James F. Terwilliger, and John Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proc. VLDB Endow.*, 8(4):401–412, 2014.

[21] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul 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.

[22] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok 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.

[23] Surajit Chaudhuri, Vivek R. Narasayya, and Ravishankar 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.

[24] Rada Chirkova and Jun Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[25] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. Discovering denial constraints. *PVLDB*, 6(13):1498–1509, 2013.

[26] Yeounoh Chung, Michael Lind Mortensen, Carsten Binnig, and Tim 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.

[27] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard 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.

[28] Latha S. Colby, Akira Kawaguchi, Daniel F. Lieuwen, Inderpal Singh Mumick, and Kenneth 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.

[29] Allen 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.

[30] Kayhan Dursun, Carsten Binnig, Ugur Çetintemel, and Tim 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.

[31] Cristian Estan and Jeffrey 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.

[32] Alex Galakatos, Andrew Crotty, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. Revisiting reuse for approximate query processing. *PVLDB*, 10(10):1142–1153, 2017.

[33] Like Gao, Min Wang, Xiaoyang Sean Wang, and Sriram 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.

[34] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. Shareddb: Killing one thousand queries with one stone. *Proc. VLDB Endow.*, 5(6):526–537, 2012.

[35] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, and Donald Kossmann. Shared workload optimization. *Proc. VLDB Endow.*, 7(6):429–440, 2014.

[36] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Tappan 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.

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

[38] Goetz Graefe, Wey Guy, and Harumi 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.

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

[40] Timothy Griffin and Leonid 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.

[41] Ashish Gupta and Iderpal Singh Mumick. *Materialized views: techniques, implementations, and applications*. MIT press, 1999.

[42] Ashish Gupta, Inderpal Singh 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.

[43] Himanshu Gupta and Inderpal Singh 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.

[44] Himanshu Gupta and Inderpal Singh Mumick. Selection of views to materialize in a data warehouse. *IEEE Trans. Knowl. Data Eng.*, 17(1):24–43, 2005.

[45] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 287–298. ACM Press, 1999.

[46] Moustafa A. Hammad, Michael J. Franklin, Walid G. Aref, and Ahmed K. Elmagarmid. Scheduling for shared window joins over data streams. In Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer, editors, *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, pages 297–308. Morgan Kaufmann, 2003.

[47] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 383–394, 2005.

[48] Hao He, Junyi Xie, Jun Yang, and Hai Yu. Asymmetric batch incremental view maintenance. In Karl Aberer, Michael J. Franklin, and Shojiro 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.

[49] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In Joan 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.

[50] Mingsheng Hong, Mirek Riedewald, Christoph Koch, Johannes Gehrke, and Alan J. Demers. Rule-based multi-query optimization. In Martin L. Kersten, Boris Novikov, Jens Teubner, Vladimir Polutin, and Stefan Manegold, editors, *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings*, volume 360 of *ACM International Conference Proceeding Series*, pages 120–131. ACM, 2009.

[51] Muhammad Idris, Martín Ugarte, and Stijn 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.

[52] Milena Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo 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.

[53] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. Selecting subexpressions to materialize at datacenter scale. *PVLDB*, 11(7):800–812, 2018.

[54] Ryan Johnson, Nikos Hardavellas, Ippokratis Pandis, Naju Mancheril, Stavros Harizopoulos, Kivanc Sabirli, Anastassia Ailamaki, and Babak Falsafi. To share or not to share? In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 351–362. ACM, 2007.

[55] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. Ajoin: Ad-hoc stream joins at scale. *Proc. VLDB Endow.*, 13(4):435–448, 2019.

[56] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. Astream: Ad-hoc shared stream processing. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 607–622. ACM, 2019.

[57] Yannis Katsis, Kian Win Ong, Yannis Papakonstantinou, and Kevin Keliang 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.

[58] Yannis Kotidis and Nick 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.

[59] Sailesh Krishnamurthy, Chung Wu, and Michael J. Franklin. On-the-fly sharing for streamed aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 623–634, 2006.

[60] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *CoRR*, abs/1808.03196, 2018.

[61] Per-Åke Larson and Jingren 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.

[62] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, 2015.

[63] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas 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.

[64] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey 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.

[65] Lothar F. Mackert and Guy 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.

[66] Darko Makreshanski, Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. Mqjoin: Efficient shared execution of main-memory joins. *PVLDB*, 9(6):480–491, 2016.

[67] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. Batchdb: Efficient isolated execution of hybrid OLTP+OLAP workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 37–50, 2017.

[68] Tanu Malik, Randal C. Burns, and Nitesh 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.

[69] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, 2019.

[70] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. Shared arrangements: practical inter-query sharing for streaming dataflows. *Proc. VLDB Endow.*, 13(10):1793–1806, 2020.

[71] Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi 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.

[72] Fabian Nagel, Peter A. Boncz, and Stratis 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.

[73] Supun Nakandala, Arun Kumar, and Yannis Papakonstantinou. Incremental and approximate inference for faster occlusion-based deep CNN explanations. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.*, pages 1589–1606, 2019.

[74] Milos Nikolic, Mohammad Dashti, and Christoph 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.

[75] Milos Nikolic, Mohammed Elseidy, and Christoph 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.

[76] Milos Nikolic and Dan 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.

[77] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. An empirical analysis of deep learning for cardinality estimation. *CoRR*, abs/1905.06425, 2019.

[78] Iraklis Psaroudakis, Manos Athanassoulis, and Anastasia Ailamaki. Sharing data and work across concurrent analytical queries. *Proc. VLDB Endow.*, 6(9):637–648, 2013.

[79] Lin Qiao, Vijayshankar Raman, Frederick Reiss, Peter J. Haas, and Guy M. Lohman. Main-memory scan sharing for multi-core cpus. *PVLDB*, 1(1):610–621, 2008.

[80] Vijayshankar Raman, Amol Deshpande, and Joseph 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.

[81] Vijayshankar Raman, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donald Kossmann, Inderpal Narang, and Richard Sidle. Constant-time query processing. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 60–69, 2008.

[82] Robin Rehrmann, Carsten Binnig, Alexander Böhm, Kihong Kim, Wolfgang Lehner, and Amr Rizk. Oltpshare: The case for sharing in OLTP workloads. *Proc. VLDB Endow.*, 11(12):1769–1780, 2018.

[83] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. Holoclean: Holistic data repairs with probabilistic inference. *PVLDB*, 10(11):1190–1201, 2017.

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

[85] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 249–260. ACM, 2000.

[86] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas 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.

[87] Zechao Shang, Xi Liang, Dixin Tang, Cong Ding, Aaron J. Elmore, Sanjay Krishnan, and Michael J. Franklin. Crocodiledb: Efficient database execution through intelligent deferment. In *CIDR 2020*. www.cidrdb.org, 2020.

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

[89] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar 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.

[90] Michal Switakowski, Peter A. Boncz, and Marcin Zukowski. From cooperative scans to predictive buffer management. *PVLDB*, 5(12):1759–1770, 2012.

[91] Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael Stonebraker, Ricardo Mayerhofer, and Francisco 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.

[92] Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael Stonebraker, Ricardo Mayerhofer, and Francisco Jose Andrade. P-store: An elastic database system with predictive provisioning. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 205–219. ACM, 2018.

[93] Dixin Tang, Zechao Shang, Aaron J. Elmore, Sanjay Krishnan, and Michael J. Franklin. Intermittent query processing. *Proc. VLDB Endow.*, 12(11):1427–1441, July 2019.

[94] Dixin Tang, Zechao Shang, Aaron J. Elmore, Sanjay Krishnan, and Michael J. Franklin. Crocodiledb in action: Resource-efficient query execution by exploiting time slackness. *Proc. VLDB Endow.*, 13(12):2937–2940, 2020.

[95] Dixin Tang, Zechao Shang, Aaron J. Elmore, Sanjay Krishnan, and Michael J. Franklin. Thrifty query execution via incrementability. In *SIGMOD 2020*, pages 1241–1256. ACM, 2020.

[96] Dixin Tang, Zechao Shang, William Ma, Aaron J. Elmore, and Sanjay Krishnan. Resource-efficient Shared Query Execution via Exploiting Time Slackness. *Under submission*, 2020.

[97] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. General incremental sliding-window aggregation. *Proc. VLDB Endow.*, 8(7):702–713, 2015.

[98] Yufei Tao, Man Lung Yiu, Dimitris Papadias, Marios Hadjieleftheriou, and Nikos 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.

[99] Jonas Traub, Philipp M. Grulich, Alejandro Rodriguez Cuellar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. Efficient window aggregation with general stream slicing. In Melanie Herschel, Helena Galhardas, Berthold Reinwald, Irini Fundulaki, Carsten Binnig, and Zoi Kaoudi, editors, *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, pages 97–108. OpenProceedings.org, 2019.

[100] Immanuel 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.

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

[102] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion 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.

[103] Stratis Viglas and Jeffrey 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.

[104] Stratis Viglas, Jeffrey F. Naughton, and Josef 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.

[105] Song Wang, Elke A. Rundensteiner, Samrat Ganguly, and Sudeept Bhatnagar. State-slice: New paradigm of multi-query optimization of window-based stream queries. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 619–630, 2006.

[106] A.N. Wilschut and Peter M.G. 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.

[107] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy H. Katz, and Ion 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.

[108] Kai Zeng, Sameer Agarwal, and Ion 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.

[109] Ce Zhang, Arun Kumar, and Christopher 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.

[110] Rui Zhang, Nick Koudas, Beng Chin Ooi, and Divesh Srivastava. Multiple aggregations over data streams. In Fatma Özcan, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 299–310. ACM, 2005.

[111] Jingren Zhou, Per-Åke Larson, and Hicham 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.

[112] Jingren Zhou, Per-Åke Larson, Johann Christoph Freytag, and Wolfgang Lehner. Efficient exploitation of similar subexpressions for query processing. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 533–544. ACM, 2007.

[113] Jingren Zhou, Per-Åke Larson, Jonathan Goldstein, and Luping 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.