UC Irvine

UC Irvine Electronic Theses and Dissertations

Title

Texera: A System for Collaborative and Interactive Data Analytics Using Workflows

Permalink

https://escholarship.org/uc/item/48w77232

Author

Wang, Zuozhi

Publication Date

2023

Copyright Information

This work is made available under the terms of a Creative Commons Attribution License, availalbe at https://creativecommons.org/licenses/by/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

Texera: A System for Collaborative and Interactive Data Analytics Using Workflows

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Zuozhi Wang

Dissertation Committee:
Professor Chen Li, Chair
Professor Michael J. Carey
Professor Sharad Mehrotra

2023

Portions of Chapter 3 © 2023 VLDB Endowment
Portions of Chapter 4 © 2023 Springer
All other materials © 2023 Zuozhi Wang

DEDICATION

To my family.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

I would like to express my utmost gratitude to my advisor, Professor Chen Li, who has been a pivotal figure in my academic journey. Prof. Li introduced me to the fascinating world of research when I was still an undergraduate student. He has taught me invaluable lessons on research, critical thinking, system building, paper writing, and so much more. He has provided me with countless opportunities to learn, grow, and excel. His support and mentorship have shaped my professional growth and personal development. I am truly fortunate to have learned from such an inspiring role model and will forever be grateful.

I am grateful to Prof. Michael J. Carey and Prof. Sharad Mehrotra, who have served on my thesis committee. Their insightful feedback, constructive criticism, and encouragement have been invaluable in refining and improving my work.

I wish to express a huge thank-you to the members of the Texera team, including Shengquan Ni, Yicong Huang, Avinash Kumar, Sadeem Alsudais, Xiaozhen Liu, and Xinyuan Lin. We shared countless moments of joy in developing this amazing system. These memories hold a special place in my heart that I will always treasure. My thanks also extend to all contributors of Texera for their great efforts, and our users for their trust and active engagement.

I would like to thank my mentor, Kai Zeng, for our long term collaboration at Alibaba during my PhD study. Working alongside him has been an eye-opening experience. I am grateful for learning from him and gaining numerous insights during our collaboration. I also wish to thank my colleagues at Alibaba, including Botong Huang and Wei Chen.

I would like to thank my mentor, Bailu Ding, for hosting my internship at Microsoft Research, and thank Phil Bernstein for his valuable guidance throughout the experience. This was a unique opportunity to expand my knowledge and skills, and I truly enjoyed the experience.

I would like to thank my colleagues and friends at the ISG group, including Qiushi Bai, Jianfeng Jia, Taewoo Kim, Chen Luo, Yiming Lin, and Fangqi Liu for their help and support. I thank my collaborators Zhihui Yang, Yao Lu, and Yutong Han for our great collaborative efforts. I would like to give a lighthearted nod to the video games Zelda, Dota, and Genshin, which provided me with much-needed relaxation during challenging phases of research.

I would like to express my most sincere appreciation to my fiancée, Xuxi Pan, for her support, care, and love during my PhD journey. Her constant encouragement and understanding have been invaluable to my success and well-being. I am grateful for the precious moments we have shared and the strength she has given me by always being by my side.

Finally, I want to offer my deepest gratitude to my parents, my mother Li Liu and my father Hongting Wang, as well as my grandparents. Their unwavering support and belief in me have allowed me to fully immerse myself in research. Their constant care and understanding have been a source of comfort and strength. Words cannot adequately convey my appreciation for their selfless love and support.

The third chapter of this thesis is adapted from my research paper publication in VLDB 2023 titled "Fries: Fast and Consistent Runtime Reconfiguration in Dataflow Systems with Transactional Guarantees". The co-authors are Shengquan Ni, Avinash Kumar, and Chen Li. Chen Li directed and supervised research which forms the basis for the dissertation.

The fourth chapter of this thesis is adapted from my research paper publication in VLDB Journal titled "Tempura: a general cost-based optimizer framework for incremental data processing (Journal Version)". The co-authors listed in these papers are Kai Zeng, Botong Huang, Wei Chen, Xiaozong Cui, Bo Wang, Ji Liu, Liya Fan, Dachuan Qu, Zhenyu Hou, Tao Guan, Jingren Zhou, and Chen Li. Chen Li directed and supervised research which forms the basis for the dissertation.

This work has been supported in parts by NSF IIS-1745673 and IIS-2107150 awards and a grant from the Google Cloud Research Credits program. I have also been supported by cloud credits from the NSF Cloudbank program.

VITA

Zuozhi Wang

EDUCATION

Doctor of Philosophy in Computer Science **2023**
University of California, Irvine *Irvine, CA*

Bachelor of Science in Computer Science **2017**
University of California, Irvine *Irvine, CA*

RESEARCH EXPERIENCE

Graduate Research Assistant **2017–2023**
University of California, Irvine *Irvine, California*

Research Intern **2020**
Microsoft Research *Redmond, Washington*

Research Collaborator **2018–2020**
Alibaba *Hangzhou, China and Irvine, California*

PUBLICATIONS

1. Building a Collaborative Data Analytics System: Opportunities and Challenges (accepted tutorial) **2023**
Zuozhi Wang, Chen Li
Proceedings of the VLDB Endowment (PVLDB)

2. Tempura: a general cost-based optimizer framework for incremental data processing (Journal Version) **2023**
Zuozhi Wang, Kai Zeng, Botong Huang, Wei Chen, Xiaozong Cui, Bo Wang, Ji Liu, Liya Fan, Dachuan Qu, Zhenyu Hou, Tao Guan, Chen Li, Jingren Zhou
The International Journal on Very Large Data Bases (VLDB Journal)

3. Fries: Fast and Consistent Runtime Reconfiguration in Dataflow Systems with Transactional Guarantees **2023**
Zuozhi Wang, Shengquan Ni, Avinash Kumar, Chen Li
Proceedings of the VLDB Endowment (PVLDB)

4. Demonstration of Collaborative and Interactive Workflow-Based Data Analytics in Texera **2022**
Xiaozhen Liu, **Zuozhi Wang**, Shengquan Ni, Sadeem Alsudais, Yicong Huang, Avinash Kumar, Chen Li
Proceedings of the VLDB Endowment (PVLDB)

5. Optimizing Machine Learning Inference Queries with Correlative Proxy Models **2022**
Zhihui Yang, **Zuozhi Wang**, Yicong Huang, Yao Lu, Chen Li, X. Sean Wang
Proceedings of the VLDB Endowment (PVLDB)

6. Demonstration of Accelerating Machine Learning Inference Queries with Correlative Proxy Models **2022**
Zhihui Yang, Yicong Huang, **Zuozhi Wang**, Feng Gao, Yao Lu, Chen Li, X. Sean Wang
Proceedings of the VLDB Endowment (PVLDB)

7. Tempura: A General Cost-based Optimizer Framework for Incremental Data Processing **2021**
Zuozhi Wang, Kai Zeng, Botong Huang, Wei Chen, Xiaozong Cui, Bo Wang, Ji Liu, Liya Fan, Dachuan Qu, Zhenyu Hou, Tao Guan, Chen Li, Jingren Zhou
Proceedings of the VLDB Endowment (PVLDB)

8. Grosbeak: A Data Warehouse Supporting Resource-Aware Incremental Computing **2020**
Zuozhi Wang, Kai Zeng, Botong Huang, Wei Chen, Xiaozong Cui, Bo Wang, Ji Liu, Liya Fan, Dachuan Qu, Zhenyu Hou, Tao Guan, Chen Li, Jingren Zhou
Proceedings of the International Conference on Management of Data (SIGMOD)

9. Amber: A Debuggable Dataflow System Based on the Actor Model **2020**
Avinash Kumar, **Zuozhi Wang**, Shengquan Ni, Chen Li
Proceedings of the VLDB Endowment (PVLDB)

10. Demonstration of Interactive Runtime Debugging of Distributed Dataflows in Texera **2020**
Zuozhi Wang, Avinash Kumar, Shengquan Ni, Chen Li
Proceedings of the VLDB Endowment (PVLDB)

11. A Demonstration of TextDB: Declarative and Scalable Text Analytics on Large Data Sets (Best Demo Award) **2017**
Zuozhi Wang, Flavio Bayer, Seungjin Lee, Kishore Narendran, Xuxi Pan, Qing Tang, Jimmy Wang, Chen Li
Proceedings of the IEEE International Conference on Data Engineering (ICDE)

ABSTRACT OF THE DISSERTATION

Texera: A System for Collaborative and Interactive Data Analytics Using Workflows

By

Zuozhi Wang

Doctor of Philosophy in Computer Science

University of California, Irvine, 2023

Professor Chen Li, Chair

In the world of data analytics, domain experts, such as public health scientists and medical researchers, play a crucial role as their domain knowledge can unlock valuable insights from data. However, they face several challenges in the current landscape of data analytics tools. They often lack the technical skills necessary to analyze large datasets, requiring collaboration with technical experts who may not have relevant domain knowledge. Moreover, when processing large volumes of data, the execution times can be lengthy, and non-technical users are left in the dark without feedback.

Over the past six years, our team has been developing Texera, a workflow-based data analytics system specifically designed to enable non-technical users to perform data analytics tasks with ease by promoting seamless collaboration and responsive interactions. Texera enables multiple users to collaboratively construct workflows, offering an experience similar to that of Google Docs and Overleaf. Furthermore, Texera allows users to interact with the workflow execution, enabling them to pause/resume workflows, inspect execution states, and modify logic as needed.

In this thesis, we first present an overview of the Texera system in Chapter 2, discussing the design choices and the associated tradeoffs of several key components within Texera that enable these powerful features of real-time collaborations and user interactions. Following

this, in Chapter 3, we explore a specific use case of user interaction: modifying the logic of operators in a workflow, also referred to as reconfigurations. We develop an algorithm called Fries, which can schedule these reconfigurations with minimal delay while maintaining transactional guarantees, particularly when a reconfiguration involves multiple operators. In Chapter 4, we shift our focus to incremental data processing, as Texera uses progressive computation to deliver early results to users. We present Tempura, a cost-based optimization framework designed for incremental processing. As a general framework, Tempura can support various incremental computation requirements for many different applications and use cases even beyond Texera's scope. Tempura can select the best incremental computation plan based on the specific query and data involved. In Chapter 5, we conclude this thesis and discuss future work.

Chapter 1

Introduction

Real-time collaborative editing services, such as Google Docs and Overleaf, have gained significant popularity and brought immense value to society. These services enable individuals to easily collaborate and jointly contribute to various tasks, including document creation, spreadsheet management, presentations, and drawings. The benefits of these services have become even more appealing following the recent shift toward remote work. Although real-time collaborative editing is becoming increasingly prevalent in editing applications, it remains a rare feature in data-analytics applications. The need for collaboration features is arguably even more crucial in data-analytics applications, particularly with the growing involvement of domain scientists in the process.

Domain experts, such as public health scientists or medical researchers, are crucial in the context of data analytics because they possess valuable domain knowledge that can unlock the full potential of data-driven insights. However, they face several challenges in the current landscape of data analytics tools. First of all, they often lack the technical skills needed to analyze large datasets, such as proficiency in programming languages (e.g., Python, R), understanding of machine learning algorithms, and knowledge of data visualization techniques.

This necessitates the collaboration with technical experts who have the technical skills but might not possess the relevant domain knowledge, such as comprehension of public health policies or familiarity with medical terminologies. Additionally, data analytics jobs processing large datasets can be time-consuming, leaving users without feedback on results until the very end. This lack of real-time visibility can impede collaboration, as users cannot view results, identify workflow issues, or implement quick fixes and iterations to improve the data workflows.

Consequently, it is essential for a data analytics system to enable real-time collaboration and user interaction throughout its operation. In terms of collaboration, these systems should facilitate real-time cooperation during both the workflow editing phase and execution phase. Regarding interactivity, the systems must support progressive computation and provide real-time updates on execution status and early results. Moreover, they should empower users to control the system during runtime by offering options to pause or resume execution, monitor progress, inspect intermediate states and outcomes, and modify logic of operators as necessary.

Current data analytics systems present several limitations. Several Python notebook-style platforms have recently incorporated real-time collaboration features, including DeepNote [3], Google Colab [5], and Databricks Notebooks [7]. These tools are useful for technical experts proficient in programming languages like Python and SQL. However, the necessity of understanding complex code can impede collaboration with domain experts.

Workflow-style systems are popular among domain scientists due to their easy-to-understand graphical user interface. Example systems include Alteryx [2], KNIME [8], and RapidMiner [9]. Despite their popularity, these systems also have shortcomings. Firstly, they are often built with a "pre-cloud" architecture and require users to install desktop software or clients. This restricts their ability to support collaborative features. Furthermore, they often run on a single machine, limiting their capacity to scale when working with large data sets.

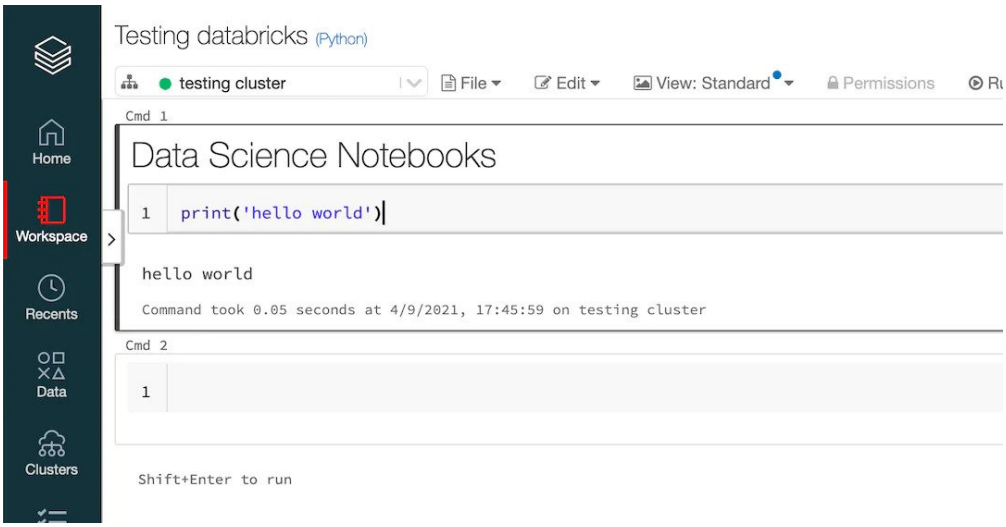


Figure 1.1: A screenshot of Databricks Notebooks, which supports real-time collaboration in executing Python and SQL code.

On the other hand, big data processing systems such as Spark [115], Flink [35], and Dask [87], excel in terms of scalability and efficiency, making them ideal for handling large volumes of data. However, these systems are less accessible to non-technical users who lack programming skills. Moreover, they do not prioritize interactivity and responsiveness, leaving users in the dark during processing, which hampers collaboration and communication with data scientists responsible for running the analytics processes.

We outline the essential requirements of a system that effectively addresses the previously mentioned challenges:

1. Interactive and responsive: The system needs to allow users to interact with it during execution, allowing them to pause/resume the execution, examine the execution status, inspect intermediate states and results, and even modify the workflow. Moreover, the system should be responsive to user interaction requests, providing near-instantaneous response times, ideally within a second .

2. Collaborative interface: A user-friendly web-based GUI is necessary to allow multiple users to concurrently edit, run, monitor, and interact with workflows.

3. Parallel and fault-tolerant: The system should be capable of running on a cluster of machines and ensuring scalability to handle large data sets. Furthermore, the system must be fault-tolerant because machine failures are common in a cluster.

In response to these requirements, our team has spent the past six years developing Texera, a workflow-based data analytics system specifically designed to enable non-technical users to perform data analytics tasks with ease by promoting seamless collaboration and responsive interactions. As shown in Figure 1.2, Texera enables multiple users to collaboratively construct and control workflows, offering an experience similar to that of Google Docs and Overleaf. Texera supports progressive execution that allows the users to see early results during execution. Texera allows users to interact with the workflow execution, enabling them to pause/resume workflows, inspect execution states, and modify logic as needed. In Chapter 2 of the thesis, we explore the challenges, key design decisions, and trade-offs involved in enabling the collaboration and interaction features within the Texera system.

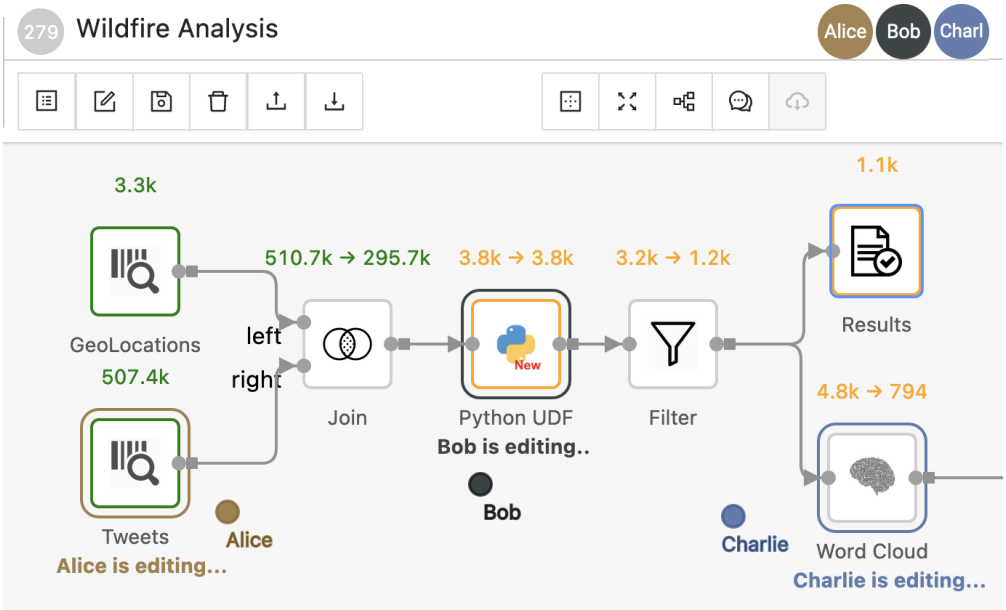


Figure 1.2: Illustration of collaborative data analytics in Texera. Multiple users simultaneously contribute to various components within an active workflow.

The Texera system has been utilized by a wide range of users from different fields of study,

# of users	# of workflows	# of versions	# of executions
80	>1,000	>100,000	>10,000

Table 1.1: Statistics of the Texera Service as of May 2023

demonstrating its versatility and effectiveness. Table 1.1 gives a simple summary of how Texera was used up to May 2023. It shows the number of users, workflows, versions, and times workflows are executed, which shows just how much people are using the system. These users have engaged with the system over an extended period and constructed numerous workflows tailored to their needs. Table1.2 provides a spotlight on several notable research projects that have used the Texera service.

In Chapter 3, we examine a specific use case of user interaction: modifying the logic of operators within a workflow, also known as reconfigurations. Computation jobs in modern big data systems that process large amounts of data can take a considerable amount of time to run, ranging from hours to days or even weeks to complete. In these applications, when a long-running job continuously processes ingested data, developers often need to modify the computing logic of the job without disrupting the execution. Reconfigurations are not only highly beneficial in the context of systems like Texera, which employ pipelined progressive execution, but also prove valuable in streaming systems such as Apache Flink, that process real-time streaming data. When users issue reconfiguration requests, the delay until the changes are applied in the system is critical, as users often want to apply the changes as soon as possible. A primary limitation of existing systems supporting runtime reconfigurations is that they may have long reconfiguration delays. These systems either have to stop and restart the execution, or wait for all in-flight tuples to be completely processed. We develop a novel reconfiguration scheduler called Fries, which uses fast control messages to schedule reconfigurations. This algorithm ensures the consistency of the reconfiguration, particularly when involving multiple operators and complex workflows. More importantly, this work addresses a crucial question regarding the meaning of consistency in the context of runtime

Project Lead	Research Topic	Workflow Operations	Duration
UCI Data Science Students	COVID signal detection	Data wrangling, model training	09/2022 - 12/2022
UCI Statistics Students	Reddit data analysis	Data crawling, wrangling, visualization, and model training	02/2023 - Present
Prof. Suellen Hopfer UCI Public Health	Extreme heat tweet analysis	Data wrangling, model training	01/2023 - Present
Prof. Natalia Komarova UCI Mathematics	HPV vaccine networks	Retweet network crawling, topic modeling, visualization	01/2021 - Present
Prof. Wei Wang UCLA Computer Science	ML task paradigm comparison	KGE inferencing, FSQA model training, multi-label classification	01/2021 - Present
Prof. David Timberlake UCI Public Health	Tobacco tweet analysis	SVM training/inference, topic modeling, visualizations	03/2021 - 10/2022
Prof. Gloria Mark UCI Informatics	COVID vaccine attitude analysis	Ideology score computing, visualizations	06/2021 - 5/2022
Prof. Gloria Mark UCI Informatics	COVID tweets linguistic analysis	Ideology score computing	5/2022 - 3/2023
Prof. Lina Rosengren-Hovee UNC Chapel Hill Public Health	Social media analysis	Text classification model training, sentiment analysis	06/2021 - Present
Prof. Kai Zheng UCI Informatics	COVID mask mandate analysis	Data crawling, Twitter search	01/2022 - 7/2022
Prof. Suellen Hopfer UCI Public Health	Wildfires and climate change	Data wrangling, BERT model training	01/2022 - Present
Profs. Sunny Jiang and Volodymyr Minin UCI Microbiology and Statistics	Wastewater management	Data extraction and cleaning	02/2023 - Present
Prof, Ju Fan Renmin University of China	Social media analysis	Data extraction, cleaning, semi-structured data analytics	09/2019 - 12/2019
Prof. Gloria Mark UCI Informatics	Immigration policies	Data extraction	01/2018 - 10/2018
Prof. Elvan Bayraktaroglu Istanbul Technical University	Police violence	Data extraction, cleaning, and wrangling	06/2017 - 10/2018
Dr. Zhihui Yang Zhijiang Lab, China	Machine learning acceleration	Machine learning model training and inference acceleration	07/2018 - 06/2022
Prof Irene Zhang, National Natural Science Foundation of China	Proposal report analysis	Data cleaning and wrangling	10/2018 - 05/2019
Prof Hui Zhang Henan Academy of Agricultural Sciences, China	Patent document analysis	Data cleaning and wrangling	01/2017 - 05/2017
Prof. Suellen Hopfer UCI Public Health	Hurricane Maria analysis	Data extraction and visualization	01/2018 - 08/2018
Prof. Sean Young UCLA School of Medicine	Hurricane Harvey analysis	Data extraction and cleaning	08/2017 - 10/2017
Prof. Jun Wu UCI Public Health	Zika virus analysis	Spatial and temporal distribution analysis and visualization	07/2016 - 12/2016

Table 1.2: Project highlights using Texera for collaborative data analytics.

reconfiguration by integrating concepts from database transactions.

In Chapter 4, we shift our focus to incremental data processing. Incremental computing plays a vital role in numerous applications. In Texera, it is utilized to progressively deliver early results to users. Beyond Texera, incremental computation is advantageous in several other situations, such as progressively maintaining the results of routine data analytics jobs as data gets ingested in a data warehouse, and in the context of intermittent late data processing, where results need to be updated when late data arrives. We develop a general cost-based optimization framework called Tempura, which differs from a traditional optimizer in that it is specifically designed to support various forms of incremental processing. We observe that there are many incremental computation algorithms, and the optimal choice depends on the application's requirements, the query, and the input data's characteristics. We propose a novel query planning model based on time-varying relations (TVR's), which can model incremental processing in its most general form. Furthermore, we provide a comprehensive specification of the Tempura optimizer framework, including rewriting rules specific to incremental computation, plan-space exploration, the selection of an optimal incremental plan, the integration of Tempura into a traditional Volcano-style optimizer, and various techniques to improve query planning speed.

In Chapter 5, we conclude this thesis and discuss future work.

Chapter 2

Texera: A System for Collaborative and Interactive Data Analytics Using Workflows

In this chapter, we first give an overview of the Texera system's architecture, explain a query's execution lifecycle, and discuss the major components of the system in Section 2.1. We then delve into one of Texera's most crucial features: pausing the execution of the workflow in Section 2.2. We examine various methods for implementing pausing, discuss their trade-offs, and present Texera's approach to ensuring responsiveness to pause requests while still supporting user interactions after pausing. Subsequently, we extend the discussion to address Texera's ability to support a wide range of user interactions in Section 2.3. We also discuss the complexities in fault tolerance introduced by supporting user interactions in Section 2.3.2, as these interactions must be recovered in the event of a failure. We present Texera's approach to fault tolerance in Section 2.3.3. Finally, we address the challenges Texera faces in facilitating collaborative workflow editing and intelligent auto completions, as well as the difficulties of sharing execution states among multiple users in Section 2.4.

2.1 System Overview

Figure 2.1 shows the overall architecture of Texera, which consists of three main layers: the web UI, the web service, and the dataflow execution engine called Amber [65]. Next, we outline the role of each component, following the order of a workflow's lifecycle, including constructing a workflow, submitting the workflow, and executing it using the engine.

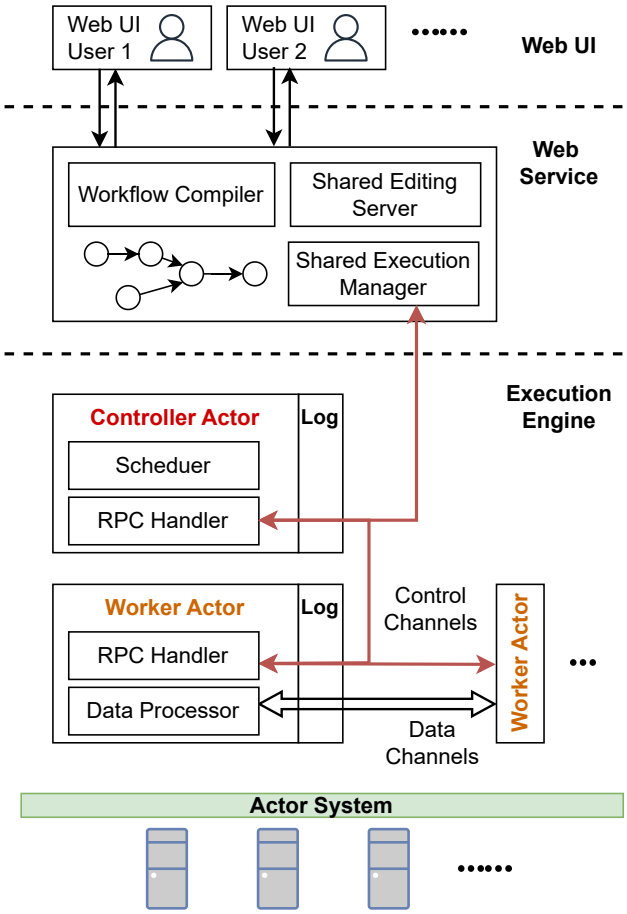


Figure 2.1: Texera's System Architecture

Workflow Construction and Shared Editing: At the web UI layer, Texera offers a graphic user interface for users to construct workflows through intuitive drag-and-drop operations. The system supports collaborative workflow construction by multiple users. The web UI layer and the shared editing server in the web service work together to resolve concurrent editing conflicts and propagate editing changes. Texera ensures that all users maintain a

consistent view of the workflow. Section 2.4.1 provides a detailed discussion on the design of the collaborative editing feature.

Submitting a Workflow to the Execution Engine: Once users have completed constructing a workflow, they submit it to the web service. The workflow compiler component first maps the workflow to a logical execution plan and verifies its validity. Subsequently, the compiler translates the logical plan to a parallel physical execution plan, and determines the parallelism and placement of each operator, as well as the data-transfer strategy between workers. The workflow compiler also rewrites the execution plans to perform various optimizations, such as reusing cached data and shuffle removal. After finalizing the physical execution plan, the web service launches the job on the actor system.

The actor model is a computing paradigm that provides concurrent units of computation called "actors." Each actor possesses a mailbox for receiving received messages. Upon receipt of a message, an actor can send messages to other actors (or itself), create new actors, and alter its state. Texera builds its execution engine on top of the actor model, leveraging several of its inherent benefits. First, it is fundamentally parallel, enabling efficient computation across clusters, which in turn facilitates the processing of large volumes of data. Second, its message-passing mechanism provides a clean way of supporting both data computation and control processing via distinct types of messages. Finally, several mature open-source implementations exist for the Actor Model, such as Akka [17] and Orleans [84], that provide robust frameworks for implementing this model. Texera's execution engine, constructed atop the Actor Model, is capable of running on more than 100 machines in a cluster and efficiently processing terabytes of data. Further details regarding the implementation of the execution engine on the actor model can be found in our VLDB 2020 paper [65].

Executing the Workflow on the Execution Engine: The first step in executing a workflow involves creating a controller actor responsible for its execution. This controller

actor has a scheduler component that instantiates the corresponding worker actor instances for each parallel worker of an operator. The underlying actor system ensures that these actors are created at the desired physical machine location. The controller actor establishes control channels with each worker actor to exchange control commands, such as operator lifecycle operations, statistic updates, and user interaction requests. These control commands are sent as remote procedure calls (RPC) messages and processed by the RPC handler.

Each worker actor includes an RPC handler component to process control messages and a data processor component to run the operator logic of the worker. Workers establish data channels with other workers to exchange input and output data according to the physical execution plan. They can also create control channels with other workers, apart from the control channel with the controller. Section 2.3.1 discusses the details of how the RPC handler and data processor work together within a worker. To achieve fault tolerance, each worker actor has a logging component that records input information from all incoming channels, which will be further elaborated in Section 2.3.2 and Section 2.3.3. The controller establishes a control channel with the shared execution manager within the web service layer, allowing the web service to support runtime interactions from users and relay execution-status updates back to them. Section 2.4.2 delves into the details of how the shared execution manager enables multiple users to monitor and manage the execution process, as well as how users can disconnect and subsequently reconnect (re-attach) to an ongoing execution at any time.

2.2 Designing an Interruptible Dataflow Engine

The ability to interrupt (pause) the execution of an operator is a critical aspect of facilitating user interactions with the Texera system. By allowing users to pause the execution of an operator, the system enables them to perform additional interactions, such as inspecting operator state or modifying operator logic. In this section, we focus on how to pause the execution of an operator. We present a few alternative designs and discuss their tradeoffs.

2.2.1 Forcibly Interrupting Workflow Execution

We first explore methods that rely on external entities to forcibly interrupt and halt the execution of a running workflow. We examine these methods at two levels: the operating system (OS) level through stopping a process, and the Java Virtual Machine (JVM) level through suspending a thread. Although these methods ensure a swift and timely pause, we show that they have significant limitations, particularly in terms of allowing users to interact with the program's state and ensuring meaningful user interactions.

OS-level Process Interruption One approach to interrupting the execution of an operator is to utilize the `SIGSTOP` signal available in Unix-based systems. This signal can be sent to a running process to halt the execution immediately. When the process receives the signal, the OS suspends the process by saving its current execution context, including the program counter, register, and memory states. The process remains in a stopped state until it receives a `SIGCONT` signal, which instructs the OS to resume the process by restoring its previously saved context. This mechanism enables a quick and efficient way to pause and resume the execution of a program without terminating it.

However, this approach has significant limitations, particularly in its inability to access and interact with the program's state, e.g., when the user wants to access the build table in a

hash join operator, or accumulated values of the processed tuples in an aggregation operator. After pausing a process at the OS level, accessing the program state requires reading the memory content of the paused process, for instance, by accessing the `/proc/<pid>/mem` file in Unix systems. However, even if the memory content is obtained, interpreting the binary content is very difficult, if not impossible, especially when the data processor is implemented using a high-level language such as Java. Given this limitation, this approach is not suitable for the Texera system.

JVM-level Thread Interruption Another approach for interruptible execution utilizes Java's built-in `Thread.suspend()` method to pause the execution of an operator. In this strategy, each operator employs two threads: a data-processing (DP) thread, which is responsible for conducting the computation of an operator, and a control-processing (CP) thread, which listens to user requests. As illustrated in Figure 2.2, upon receiving a message, the CP thread invokes the `Thread.suspend()` method to halt the DP thread's execution.

This method can address the primary limitation of the OS-level pausing method, namely the inability to interact with the program's state. As depicted in Figure 2.2, when the DP thread is suspended, the CP thread can access the operator's state through shared variables storing that state. Additionally, the CP thread is capable of executing additional control logic in the same runtime environment. For instance, the user can inspect the content of state variables or examine complex data structures. The operator's logic can also be modified by changing variable values, allowing actions such as updating the computation logic of an operator or adjusting the workflow topology by modifying the routing table.

However, this method has a significant drawback: the operator's processing logic can be interrupted at arbitrary points in its execution, leading to two main issues. First, this interruption may result in a state that is in the middle of an update, potentially leaving data structures in an inconsistent state. For example, consider a case where a join operator

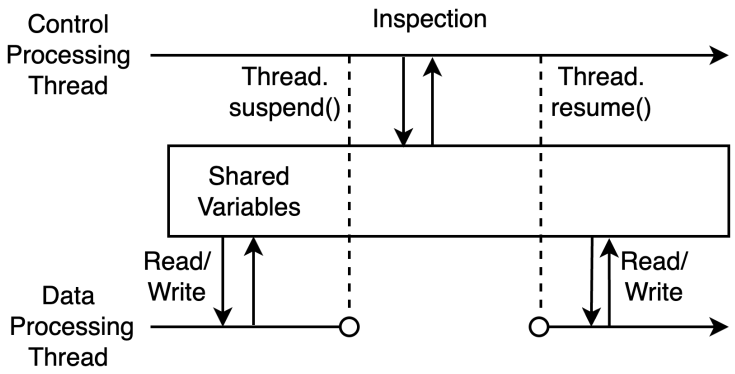


Figure 2.2: Using Java thread level interruption to pause

is updating its hash table. If a pause occurs during an operation of resizing the hash table, when the data of the smaller hash table are gradually transferred to a larger one, the resulting view of the hash table might be incomplete. Second, it is challenging to reason about the application's state because the stopping point might be incomprehensible to the user. For example, in the case of a filter operator with a complex user-defined function, the pause might happen within the function or even during the execution of an external library, rather than within the user's code. Note that these limitations also apply to the aforementioned OS-level process interruption method, as both methods forcibly interrupt an operator at an arbitrary point.

2.2.2 Texera's Design: Voluntarily Suspending Execution at Pre-determined Points

In light of the drawbacks of the previous methods, Texera employs a novel design that avoids forcibly interrupting the execution of an operator. We observe that data processing systems have natural stopping points, such as between the processing of two tuples, and many interactions only make sense at these stopping points. In this section, we present Texera's design of letting the data processor actively check for pause signals at pre-determined and predictable points and voluntarily suspend its execution. The following code snippet shows

a simple implementation to suspend execution between two tuples.

Pausing Execution Between Tuples. We first present a design that allows an operator to pause between the execution of two tuples, as shown in Listing 2.1 In this design, the operator continuously processes input tuples. For each input tuple, the `process_tuple` function is invoked to perform the necessary computation of an operator. Before processing the next tuple, the operator checks for the `paused` flag, which is set if the user has requested the system to pause. If the flag is set, the operator voluntarily suspends its own execution, yielding control to allow subsequent user interactions and inspections to take place. When the user resumes the execution, a signal is sent to the thread, waking it up and allowing it to continue processing the next tuple. This process is also visualized in Figure 2.3.

```
while (not finished):
  tuple = data_channel.get()
  process_tuple(tuple)
  if (paused flag is set):
    suspend execution until resumed
```

Listing 2.1: Suspending operator execution between two tuples.

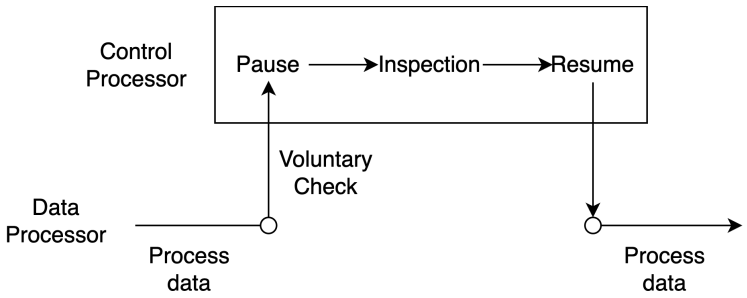


Figure 2.3: Texera DP thread mechanism

This method has a drawback: while the processing of a single tuple is typically very fast for relational operators, there could be expensive user-defined functions that take a long time to complete. For example, consider the sentiment analysis operator in Listing 2.2 that tokenizes the input text, annotates the tokens, and infers the sentiment. Completing these steps might take a significant amount of time. If the operator can only pause between two

tuples, then the user may need to wait for a long period before the system pauses, which reduces the responsiveness. Next, we discuss how to support pausing at a finer-granularity.

```
def process_tuple(tuple):
    // step 1: tokenize the text
    tokens = tokenize(tuple)
    // step 2: tag tuples with annotations
    annotations = tag(tokens)
    // step 3: infer sentiment
    sentiment = infer(annotations)
    return sentiment
```

Listing 2.2: A sentiment analysis operator that takes a long time for a single tuple.

Pausing Execution at a Finer Granularity. A key observation is that an operator's processing of even a single tuple can be further divided into mini-steps, with each step performing a portion of the operator's computation. For instance, in the sentiment analysis operator, the `process_tuple` function can be broken down into three mini-steps. The following code snippet demonstrates a new implementation that transforms the processing of a single tuple into an iterator with three steps.

```
next_step = "tokenize"
def has_next() =
    return next_step != "end"

def get_next(tuple) =
    if (next_step == "tokenize"):
        this.tokens = tokenize(tuple)
        next_step = "tagging"
    else if (next_step == "tagging"):
        this.annotations = tag(this.tokens)
        next_step = "inference"
    else if (next_step == "inference"):
```

```
13          this.sentiment = infer(taggings)
14          next_step = "end"
15          return this.sentiment
```

Listing 2.3: An iterator-based transformation of the sentiment analysis operator that decomposes the computation into three mini-steps.

As shown in Listing 2.3, the operator logic is transformed into a state machine [1] that utilizes the `next_step` variable to track the upcoming execution step. When the `get_next` function is called, the operator first examines the computation of the current mini-step, as indicated by `next_step`. It then performs the corresponding computation and updates the `next_step` variable, preparing for the engine's subsequent `get_next` function call. Once the `next_step` variable is set to `end`, the operator recognizes that all mini-steps have been completed.

The engine's implementation needs to be modified to accommodate an iterator interface for processing a single tuple, which can consist of multiple steps. As illustrated in Listing 2.4, to process a single tuple, the engine first acquires an iterator from the operator and then consumes the iterator to go through all the mini-steps of the computation. The engine performs pause checks between each mini-step iteration.

```
1 while (there is an input tuple):
2   iterator = process_tuple(tuple)
3   while (iterator.has_next()):
4     iterator.get_next()
5     if (paused flag is set):
6       suspend execution until resumed
```

Listing 2.4: Suspending operator execution between two fine-grained mini-steps of data processing.

This design provides operator developers with full control over the granularity at which the

operator's code checks for user interactions. This approach offers two main benefits. First, since developers have knowledge of the operator's execution speed, they can select proper mini-steps to ensure each step is fast enough to make the operator responsive. Second, the developer can define meaningful pause points in the operator based on the semantics of the application. For example, in the sentiment analysis operator, the developer may naturally treat each major step as a meaningful point to pause and inspect, while other points, such as implementation details within the `tokenize` function, might not warrant inspection.

A noticeable consideration with this design is the potential for increased overhead caused by numerous fine-granularity checks. Each check requires an extra access with a shared variable, which may be concurrently modified by a different thread. Although the time spent on each individual check is rather small, an abundance of these checks could lead to an increase in overhead. One strategy to mitigate this overhead could involve developers offering hints regarding the speeds of operators. For example, the system could reduce the frequency of checks for faster operators like filter or projection. On the other hand, for slower operators, such as user-defined functions like sentiment analysis, the systems could do more fine-grained checks.

In summary, by actively checking and processing user interactions at pre-determined, fine-granularity points, Texera can be responsive to users' pause requests while still supporting user interactions, such as inspecting state or modifying logic, at appropriate points in the operator's execution.

2.2.3 Experiments

In the following experiments, we first evaluated the latency of pausing the computation by comparing the forced interruption method and the voluntary check method. We implemented the forced interruption method using JVM's built-in `Thread.suspend()` function. This la-

tency was measured from the instance an operator received a `Pause` request until the moment when computation actually paused. Figure 2.4 shows the workflow utilized in the experiments. It includes a diverse set of operators such as `Scan`, `Hash Join`, `Sentiment Analysis`, `Keyword Search`, and `Aggregate`. Notably, the `Aggregate` operator is implemented in two phases: a `Partial Aggregate` and a `Global Aggregate`. During the execution, we randomly paused the workflow at various points, and we measured the average pause latency.

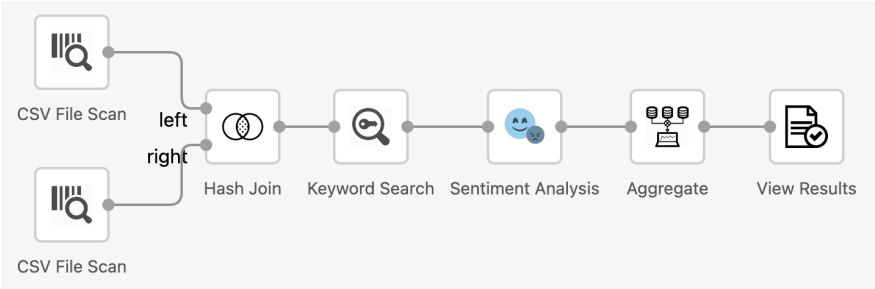


Figure 2.4: An example workflow to evaluate the latency of pausing an operator.

Figure 2.5 shows the experiment results. For the method based on JVM-level thread forced interruption, all operators were able to be paused under a millisecond. The `Sentiment Analysis` operator, being the slowest to pause, required around 0.4 milliseconds, while the Aggregate operator, as the fastest, only needed 0.002 milliseconds. Note that, despite utilizing JVM thread's inherent `Suspend()` method, the more resource-intensive operators still required more time to pause.

As for Texera's voluntary check method, the pause latency depended on the execution duration of each user-defined mini-step of an operator. For example, the implementation of the `Sentiment Analysis` operator is divided into four mini-steps, resulting in a pause latency of roughly 126 milliseconds. Even though this duration is longer than the forced interruption method, it still remains sufficiently responsive for a human user. In summary, the JVM-level forced interruption method offers a markedly shorter latency compared to the voluntary check method. Nonetheless, the voluntary check method can consistently pause all operators in less than 200 milliseconds, fulfilling the response time expectations of the

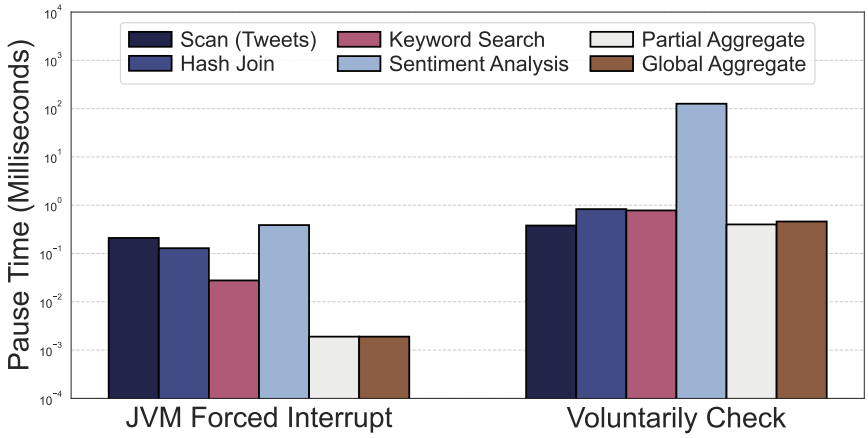


Figure 2.5: Comparison of pause latency between JVM-level forced interruption method and voluntary check method (used in Texera).

users.

Next, we evaluated the performance overhead of the voluntary check method. This was done by testing two operators: a fast filter operator processing one million tuples, implemented by comparing a tuple field to an integer, and a slower sentiment analysis operator processing 100,000 tuples, implemented using the Stanford NLP [75] library. We varied the frequency of the voluntary checks by adjusting the number of tuples processed between each check. This varied from every tuple, to 10 tuples per check, to 100, 1,000, and 10,000 tuples per check, and finally, a baseline with no checks at all. Figure shows the experiment results. The fast filter operator can experience up to 25% overhead with the most fine-grained check (a check with every tuple), with the performance overhead soon diminishes as the check frequency decreases. Conversely, for a slower operator, even the most fine-grained checks did not introduce a noticeable overhead, which is approximately around 3% compared to the baseline.

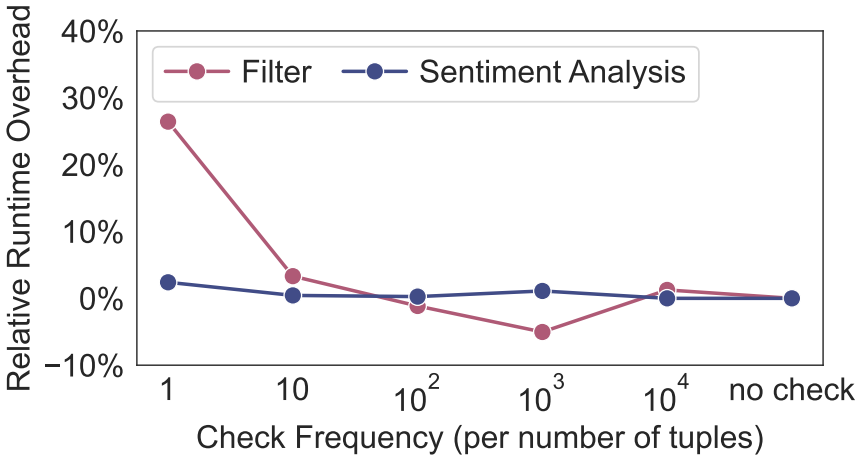


Figure 2.6: Comparison of relative runtime overhead at various voluntary check frequencies, compared to a baseline run with no voluntary checks."

2.3 Designing an Extensible and Fault-tolerant Control Plane

In Section 2.2, we discussed how to handle a specific case of a control command, namely pausing the execution of a workflow. To satisfy various user interaction requirements, such as inspecting internal states, examining intermediate results, and modifying operator logic, Texera needs to be extensible and support a wide range of control messages. In this section, we first provide a comprehensive view of how Texera's worker actor component handles other types of control messages. In addition, Texera is designed as a scalable and parallel engine capable of running on a cluster of machines. Given the common occurrence of failures on clusters, fault tolerance is a critical aspect that must be supported. However, ensuring fault tolerance becomes increasingly complex when Texera also needs to accommodate various user interactions. In the second part, we discuss how Texera recovers user interactions in addition to data computation in case of failures.

2.3.1 Handling Arbitrary Control Commands in a Worker

User interaction requests are executed by sending control messages from the controller through a dedicated channel to a worker actor. We extend the approach outlined in Section 2.2.2 for suspending execution, as illustrated in Listing 2.5. Between two fine-grained mini-step of data processing, we seize the opportunity to check the control channel and process any control messages received by the operator. The main difference in the new implementation lies in calling a special function `handle_control_message` between two mini-steps.

```
1 while (not finished):
2   tuple = data_channel.get()
3   iterator = process_tuple(tuple)
```

```
  while (iterator.has_next()):
    iterator.get_next()
    handle_control_message()

def handle_control_message():
    while (control channel has an message
            OR paused flag is set):
        message = control_channel.get()
        rpc_handler.process(message)
```

Listing 2.5: Handle arbitrary control messages between two mini-steps of data processing

During data-processor execution, any control messages sent to the workflow worker are queued in the control channel as a FIFO queue. The `handle_control_message` function processes all control messages currently in the control channel, one at a time. Each control message is processed by the RPC handler based on its type, which can involve reading the state of the operator, examining execution statistics, modifying the logic or state of an operator, or sending other control messages to the controller and other workers. When there are no control messages left in the control channel, the workflow worker can proceed to process the next mini-step of the data-processing logic.

It is worth noting that the pause and resume logic is integrated into the control handling mechanism. If a control message requests to pause the execution of a worker, the data processing should halt until a `Resume` command is received. When a `Pause` control message is processed by the RPC handler, a paused flag is set. For subsequent processing, we stop taking data messages. In this way, the data processing logic won't be invoked to process the next data. Instead, the `control_channel.get()` statement blocks the worker's execution until a control message is received from the user. This ensures that the workflow actor remains responsive to user interactions when the execution is paused. Additionally, the workflow actor does not consume CPU resources or engage in busy-waiting while it is paused.

Upon receiving a resume message, the paused flag is unset, allowing the workflow actor to exit the `handle_control_message` loop and resume normal processing in the next iteration. While the data processor is paused, any new data messages arriving in the data channel are queued in a FIFO order. As Texera pauses all operator workers in a workflow, the tuples in the data channel will not overflow because both upstream and downstream operators stop processing and producing any tuples.

2.3.2 Fault tolerance with User Interactions

We begin by presenting an example that illustrates possible loss of user interactions (e.g., bug fixes) when a failure occurs, as depicted in Figure 2.7. In this example, a user pauses a UDF (user-defined function) operator after a data tuple `D2` is processed. The user interacts with the operator by sending three control messages: `C1` to pause the execution, `C2` to inspect the operator's internal state, and `C3` to modify the operator's logic in order to fix a discovered bug. After resuming the execution, the subsequent data tuple `D3` is processed using the updated operator logic and is subsequently output to the user.

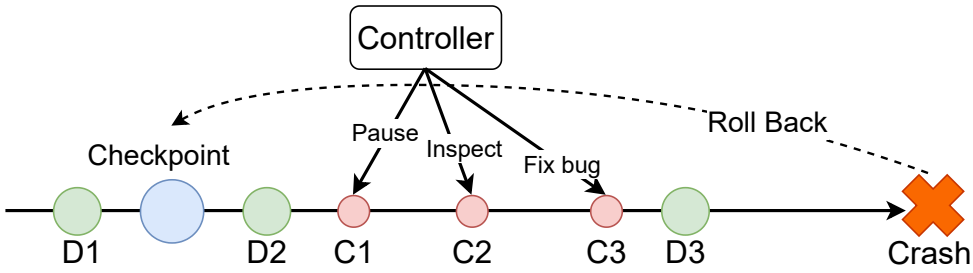


Figure 2.7: A scenario illustrating the impact of failure in the presence of user interactions: a user pauses a UDF operator, inspects and fixes a bug, and resumes the execution before a crash happens.

Consider the case where a failure happens to this operator after the data tuple `D3` is processed. Common approaches in most existing data processing systems, such as Spark, Hadoop, and Flink, involve rolling back to the last checkpoint and rerunning the computation. This method is effective when the data processing logic does not involve user interactions, as

queries are often deterministic, and rerunning them produces the same result. However, in Texera, simply rerunning the data computation without considering user interactions would result in the loss of user interactions. As illustrated in Figure 2.8, if we do not rerun the user control message `C3`, which updates the operator logic, data tuple `D3` will be processed using the old, buggy operator logic instead of the corrected logic.

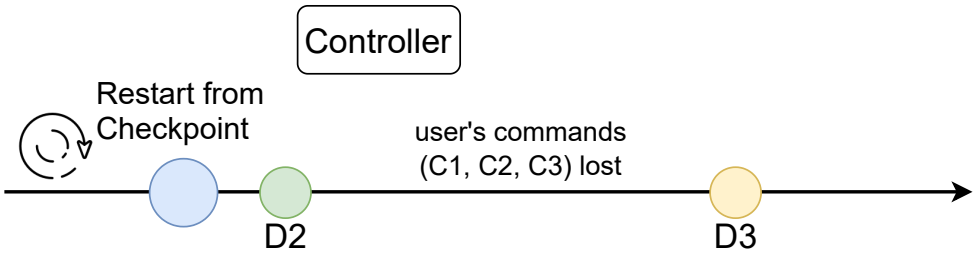


Figure 2.8: Recovery without considering user interactions: data tuple D3 is processed using the old buggy operator logic

In the context of fault tolerance, we aim to provide an "output commit" guarantee, which ensures that any result sent to the user remains committed and unchanged despite any failures. This guarantee necessitates the reapplication of user control commands to the operator during recovery. However, blindly reapplying user commands has issues. If the controller resends user commands to the UDF operator during recovery, the timing of the control commands arriving at the operator might differ from the previous run. For example, the pause command might arrive after the data tuple `D3` is processed, causing the recovery run to produce an output different from the original run and violating the output commit guarantee.

2.3.3 Texera's Approach: Logging Non-determinism from User Interactions

We propose a solution, which logs all non-deterministic factors, including the timing of user interactions and the content of users' control commands, to effectively address the

aforementioned issues.

Logging all input messages. A simple approach is to record all input messages received by an operator worker. We assume that the processing logic of an operator is deterministic. If an operator worker begins with the same initial state, processing the same sequence of messages will deterministically modify its internal state and generate the same output messages. Consequently, a failed operator can be recovered from a log containing all its input messages since the start of execution.

Figure 2.9 illustrates the logging process of an operator worker during execution. The log stores data messages `D1` and `D2`, control messages `C1` and `C2`, followed by data message `D3`. It is important to note that all user interactions are captured within the log. During this process, each data/control message received by the operator is written to disk prior to being processed by the worker. This results in a delay before the processing of each message, which can lead to significant overhead in practice. One commonly adopted optimization involves writing the message to the log and processing it concurrently, buffering the output of each input message until the input message is persisted.

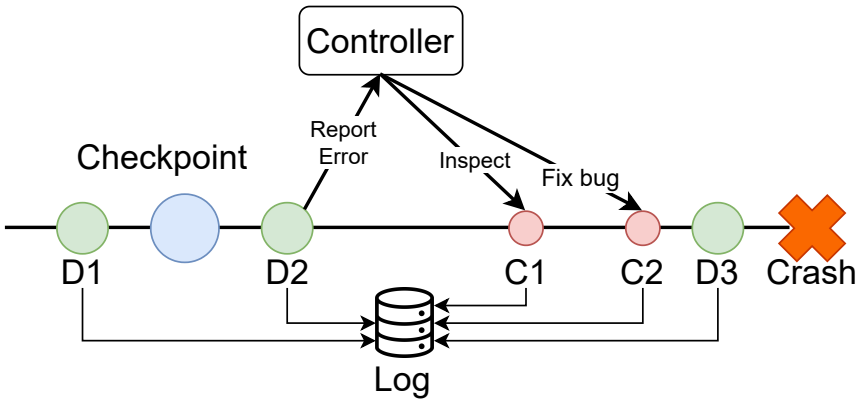


Figure 2.9: Naive logging for all input messages

Logging message arrival order. An alternative approach involves logging only the relative arrival order of input messages, relying on upstream operators to regenerate message content during recovery. This method effectively reduces the log size and the overhead of writing

the log. However, this approach requires rolling back upstream operators to regenerate data messages, as well as rolling back the controller to regenerate control messages. Restarting the controller also requires regenerating all input control messages received from operators. Since the controller serves as the central point of communication with all operators, this action in turn results in rolling back all operators in the workflow. This cascading effect not only significantly increases the recovery time, but also obstructs user interactions with the controller during the rollback process.

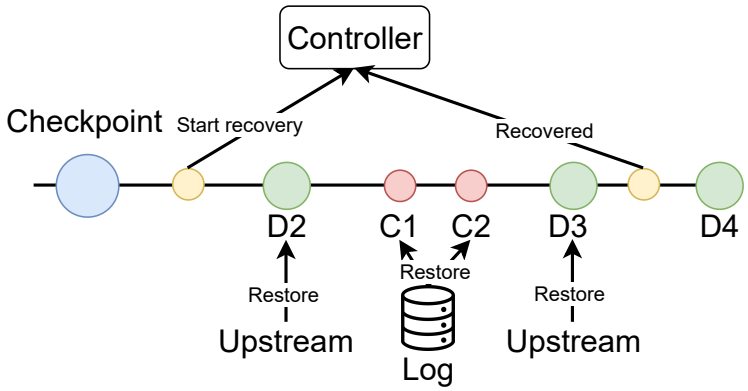


Figure 2.10: Recovery process of operator workers in Texera

Hybrid message logging strategy. To address both the logging overhead and cascade rollback issues mentioned earlier, we adopt a hybrid approach that treats data and control messages differently. Specifically, we log the content and relative order of control messages, while logging only the relative order of data messages. During recovery, we first restart the failed operators. Then we analyze the dataflow graph to find all upstream operators of the failed operators. These upstream operators need to be restarted as well to regenerate the content of input tuples. For control messages, we do not need to restart the controller; instead, we directly restore the content of the control messages from the log. Figure 2.10 illustrates the recovery process of the failed operator worker. It loads the previous checkpoint and notifies the controller about its recovery status. The controller rolls back the upstream operators to regenerate data tuples `D2` and `D3`. The operator worker reprocesses `D2` from upstream, reads control messages `C1`, `C2`, and `C3` from the log, and re-applies the bug fix

before processing `D3`.

Each restarted operator receives the same set of data and control messages, enforces the original order of processing using the log, and re-generates the same output messages, thus satisfying the output commit requirement. This approach has a low logging overhead because the content of control messages is typically much smaller than that of data messages. Additionally, this method avoids the cascading rollback issue by only restarting the upstream dependencies of the failed operators, rather than all the operators in the workflow. The logging method used in Texera is reminiscent of the logical (operational) logging approach, which can be found in algorithms in multi-level recovery [71], where the transaction operations or commands that lead to changes in system states are logged. In Texera, we log and replay both data and control messages that the operators receive and processing these messages in their original order upon restart. This is in contrary from the physical logging method, such as in ARIES [78], where the physical changes made to the system state are logged. In the event of a failure, the system can be restored by undoing or redoing operations directly on the state. Texera employs a logical logging method to save the potential storage cost of the a large amount of intermediate data or large operator sates.

2.3.4 Experiments

We conducted experiments to compare the runtime overhead and the recovery time of different logging methods. We used the workflow shown in Figure 2.11 in this experiment. This workflow reads a tweet table containing one million tweets, filters the tweets, and then joins the tweets with a geo-location table. Subsequent operations included searching for a keyword in the tweets and aggregating the results. During the workflow execution, each operator exchanged dummy control messages with the controller twice every second.

Figure 2.12 illustrates the execution time of the example workflow using different logging

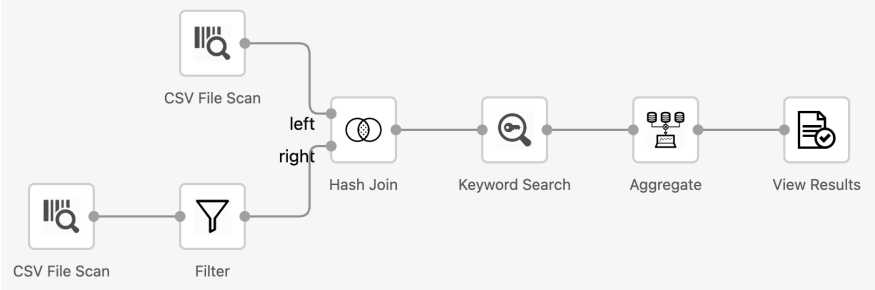


Figure 2.11: Example workflow to evaluate different logging methods.

methods, while Figure 2.13 displays the corresponding log size on disk. Table 2.1 provides the recovery time given different operator failures and different logging methods.

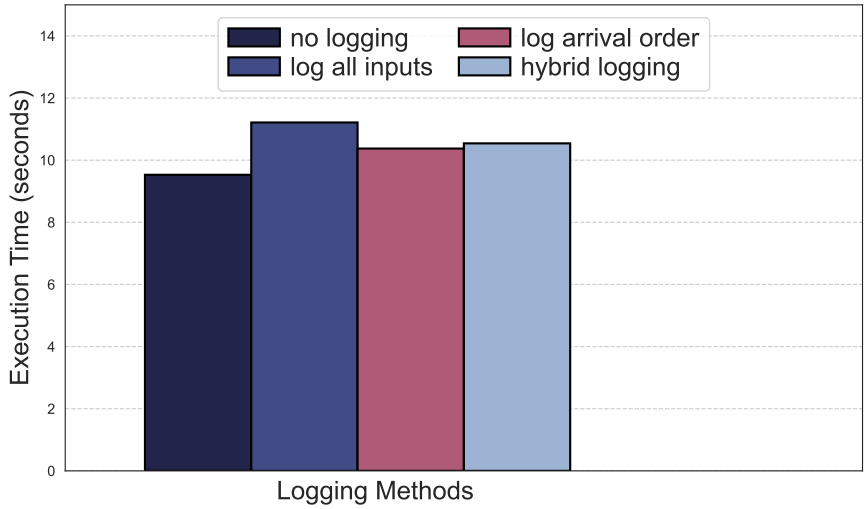


Figure 2.12: Execution time of the example workflow with different logging methods.

Without logging, the workflow completed in approximately 9.5 seconds. When all input messages were logged, including both data and control messages, the execution time increased to 11.2 seconds. This method resulted in the largest log size of around 1.6GB. However, it also enabled the quickest recovery, as the failed operator could directly access the input data from the log without restarting upstream operators. In contrast, the method of logging only the arrival order of messages lowered the execution time to roughly 10.3 seconds, which had less overhead compared to logging all messages. Moreover, it resulted in the smallest log size of around 300KB. However, this method incurred the longest recovery time due to the need for a complete workflow restart. The hybrid logging strategy resulted in a slightly larger

Failed Operator	log all inputs	log arrival order	hybrid logging
Filter	5.27	10.56	10.3
Join	10.12	10.52	10.17

Table 2.1: Recovery time of different logging methods.

log size of approximately 370 KB, and an execution time of around 10.5 seconds, both of which were less than the method of logging all messages. As for recovery time, it was longer than the method of logging all messages, but shorter than the method of logging only the input order. Thus, this hybrid method provided a slightly more balanced solution between runtime overhead and recovery speed.

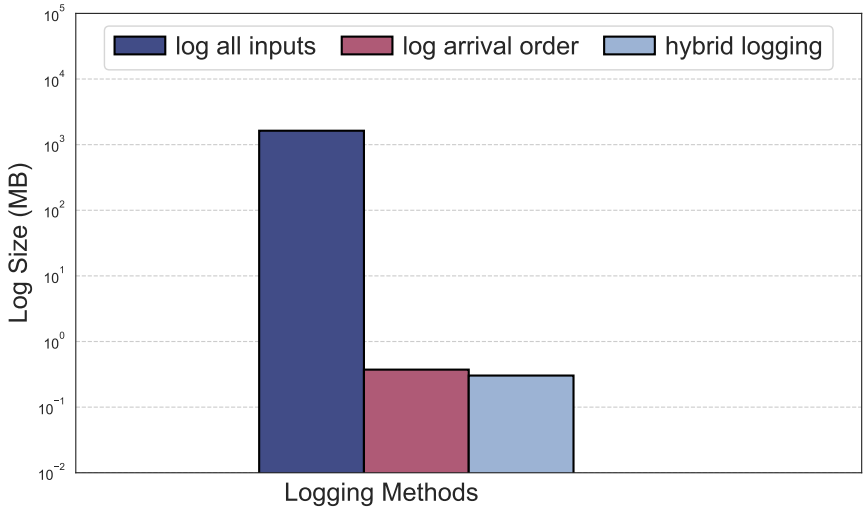


Figure 2.13: Storage overhead of the example workflow with different logging methods.

2.4 Supporting a Collaborative User Interface

So far we have covered Texera's rich user interaction features with low response time. In this section, we discuss the collaborative features supported by Texera. We first discuss the aspects of collaborative workflow editing and construction. We present a few design choices to enable shared editing, and support easy-to-use features such as auto complete. Next, we elaborate on how Texera can efficiently share execution state updates among all users, synchronize execution controls, and support a new user joining an existing execution session.

2.4.1 Collaborative Workflow Construction

A key challenge in building a collaborative workflow editor is to resolve editing conflicts between concurrent users and maintain data consistency. There are two popular techniques to support distributed data consistency and real-time conflict resolution, namely Operational Transformation [90] (OT), and Conflict-free Replicated Data Types [86] (CRDT). We refer interested readers to know how these two technologies work behind the scenes to plenty of online resources [58, 10].

Operational Transformation (OT). It is an approach that enables real-time collaborative editing by transforming operations in such a way that they maintain consistency among different replicas of shared data. In OT, when a user performs an action, this action is transformed by other users before it is applied to the shared data. This transformation ensures that the order of operations does not matter, and the final state of the shared data remains consistent across all the replicas. In a system using OT, each user's frontend client maintains a local copy of the shared data and communicates with a central server for synchronization. In an OT-based system, the central server plays a crucial role in maintaining consistency among clients by transforming and coordinating operations. The server can host

a workflow compiler that listens to changes in the shared data, providing smart auto-complete and suggestions to the frontend clients. As the server maintains the ground truth of the data, it can provide accurate auto-complete suggestions to all its connected clients.

Figure 2.14 illustrates this process with two users, Alice and Bob, concurrently editing a workflow. In the first step, Alice and Bob make simultaneous edits, which are then transmitted to a centralized OT server. Upon receiving these concurrent edits in the second step, the OT server arranges them in the order they are received, transforms each edit to resolve conflicts, and applies them to the workflow stored on the server. Finally, in the third step, the OT server sends each workflow update to the workflow compiler, which recompiles the user's workflow and provides the most recent autocomplete suggestions to both Alice and Bob via their respective web user interfaces.

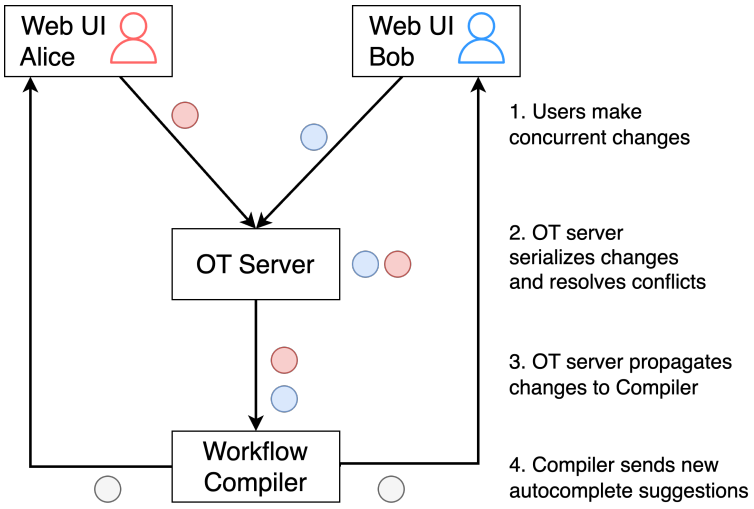


Figure 2.14: Provide conflict resolution and autocomplete suggestions using Operational Transformation (OT).

Conflict-free Replicated Data Types (CRDT). It is an alternative approach for maintaining consistency in distributed systems. CRDTs are data structures designed to be replicated across multiple nodes while allowing concurrent updates without coordination between the nodes. The main idea behind CRDT is that it should always be possible to merge different replicas of the shared data into a consistent state, even when updates happen concurrently.

This is achieved by ensuring that all operations on the CRDT are commutative, associative, and idempotent. In a system using CRDT, clients can update their local copies without the need for coordinating with a central server, such as using a peer-to-peer communication with other clients. Since CRDTs allow concurrent updates without coordination, it is more challenging for the compiler inside the server since the server does not have the ground truth.

Figure 2.15 illustrates this process. In step 1, Alice and Bob make two concurrent edits. Their respective web UIs exchange these edits directly, eliminating the need for a central server to coordinate. In step 2, upon receiving the edits, Alice and Bob independently resolve conflicts and update their documents. The CRDT algorithm ensures that both users' documents will converge after the updates. In step 3, for each workflow update, Alice and Bob's clients independently send the new workflow to the workflow compiler, requesting the most up-to-date autocomplete suggestions.

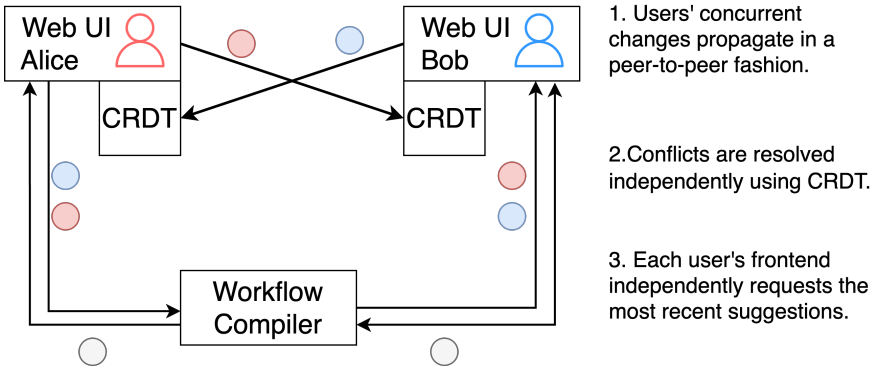


Figure 2.15: Provide conflict resolution and autocomplete suggestions using Conflict-free Replicated Data Types (CRDT).

Comparing OT and CRDT. While Operational Transformation (OT) Conflict-free Replicated Data Type (CRDT) mechanisms function differently, it's worth noting that both support the majority of common edit operations. The disparities between the two primarily stem from their respective implementations rather than from any foundational shortcomings inherent in each mechanism [96, 11]. In terms of performance, OT-based implementations are generally considered more efficient compared to CRDT-based implementations. However,

it's important to acknowledge that many recent CRDT implementations have significantly improved, offering comparable efficiency and speed [6, 4].

In terms of adopting OT or CRDT in a data analytics system, these two approaches mainly differ in their system architecture and coordination requirements. For example, in the OT architecture, the workflow compiler recompiles the workflow only twice in this example, corresponding to each update sent from the OT server. However, in the CRDT architecture, the workflow compiler recompiles the workflow four times, as both Alice and Bob independently request autocomplete suggestions for two updates. This implies that in the OT architecture, the time complexity is proportional to the number of changes, while in the CRDT architecture, the time complexity is the product of the number of changes and the number of concurrent users. For instance, in an execution session involving 10 users, even if a single user makes a single change, the workflow compiler must recompile the workflow 10 times.

Although an OT-based implementation may be more suitable for a data analytics system that typically has a central server, Texera opts for a CRDT-based implementation due to its superior open-source ecosystem and more robust frontend support. A survey of popular open-source conflict resolution tools reveals that most OT-based tools are either deprecated or no longer updated, whereas CRDT-based tools have richer features, more frequent updates, and a thriving user base and community [82, 112].

To address the challenges presented in the CRDT-based architecture, specifically the higher time complexity for the workflow compiler to recompile workflows for providing autocomplete suggestions, we made the following optimizations to increase efficiency.

Optimization 1. First, we only recompile the query if it affects the autocomplete results. This is achieved by distinguishing different types of edits. For instance, edits made purely at the UI level, such as moving an operator to a different position, need to be synchronized across frontends but do not affect the execution logic of the workflows. For these types of

edits, the frontend does not send requests to the workflow compiler. Furthermore, for some operators, only changes to certain properties affect the autocomplete suggestion results. For example, a sentiment analysis operator requires two properties: `InputColumn`, the column to perform sentiment analysis on, and `OutputColumn`, a new column name for storing the result. If the user only changes the value of `InputColumn`, this does not affect the output schema of this operator. Therefore, we only send requests to recompile the workflow when the value of `OutputColumn` changes.

Optimization 2. Second, the workflow compiler maintains a cache of recently compiled workflows and their corresponding schema propagation results. When a new request comes in, the compiler compares the request with the cache. If a match occurs, this indicates that this version of the workflow has been compiled before and the results in the cache will be returned to the frontend. This optimization is especially useful in the CRDT-based architecture. In the case of a single user making an edit, once the workflow compiler has compiled this new version, when all other users request suggestions for this version, we can retrieve the result from the cache and avoid recompiling again.

2.4.2 Collaborative Workflow Execution

In an interactive data processing system, there is a rich set of execution states that can be shared by the users, as shown in Figure 2.16. The execution states include the status of the workflow (e.g., initializing, running, paused, error, or completed), runtime statistics of operators (input and output counts), user interaction history (e.g., user commands, system responses, error messages, logs, and traces), and progressive execution results. Next, we discuss how to share execution states in a collaborative data analytics setting, including sharing execution state updates among users, allowing execution controls to be synchronized among users, and supporting adding a new user to an execution session.

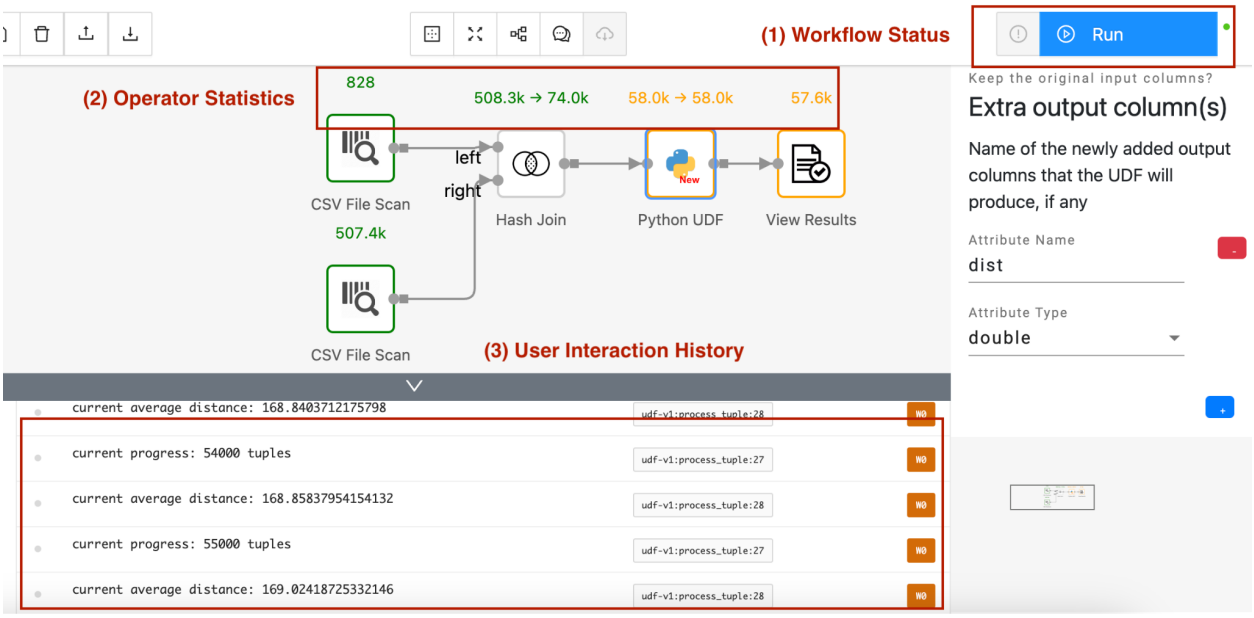


Figure 2.16: Execution states shared by users: (1) workflow status, (2) operator running statistics, and (3) user interaction history, including user commands to the system and system's output messages.

Shared Execution States

A key component of enabling collaborative workflow execution is the sharing of execution states. When a user initiates a workflow, it is essential that other users sharing the same workflow see its running status and receive frequent updates. For instance, suppose Alice and Bob are collaboratively monitoring a workflow. They both should observe the workflow status transitions, such as from 'ready' to 'running', and from "running" to "paused". Figure 2.17 illustrates this process. The "Shared Execution Manager" component plays a crucial role in maintaining this synchronicity. It periodically communicates with the Amber execution engine, querying the status of the entire workflow and collecting operator statistics. Once it obtains the latest state updates, it broadcasts these updates to all web UIs engaged in the execution session. This ensures that all users remain informed and in sync with the ongoing workflow process.

However, a naive approach of periodically sending the complete execution state snapshot to

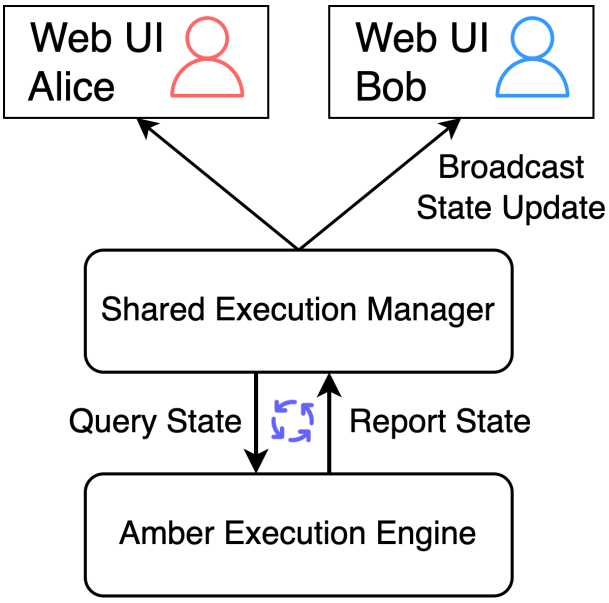


Figure 2.17: Sharing execution states in Texera. The shared execution manager periodically queries the Amber execution engine for the latest execution states and broadcasts the state updates to all web UIs.

the frontend is not viable due to the frontend's limited processing power and the network overhead. Additionally, the interaction history keeps growing as more user interactions occur during execution, and progressive execution results can become large. To provide a smooth user experience, we need incremental delta updates for various execution states. The challenge lies in the fact that different execution states require different incremental methods to be updated to the frontend. Next, we discuss various approaches to making incremental delta updates to the frontend for different execution states.

Workflow Status and Operator Statistics. For the workflow state and operator status, the backend sends only the status of the changed operators. Consider the example workflow state shown in Figure 2.16. The two `CSV File Scan` operators and the `Hash Join` operator have already completed their execution and their statistics will not change any more. In this case, the server will not resend the statistics of these operators to the frontend. The subsequent updates will only contain the new statistics for the `Python UDF` and the `View Results` operators. On the frontend, whenever it receives updates of status changes, it

simply replaces the old status value with the new one.

Interaction History. The interaction history works differently compared to operator status. Since interaction history continually grows, only newly generated entries are sent. Whenever the client sends a request to the engine, and the engine generates a response, both the request and the response are added to the interaction history. In this case, only the newly added entries are sent to the frontend. The frontend then appends the new entries to the history list upon receiving an incremental update. As an example, refer to Figure 2.16 which shows a user interaction history. The `Python UDF` operator continually updates the user about its progress after every 1000 tuples. Suppose it has already processed 55,000 tuples, thus generating 55 entries in the user interaction history. When another 1000 tuples are processed, the new progress report at the 56,000-tuple mark gets added and sent to the frontend. Note that only this additional entry gets transmitted, instead of transmitting all 56 entries from the interaction history.

Progressive Execution Results. Execution results can be extremely large, making it infeasible to send the entire result to the frontend. Naturally, a pagination method is adopted, where the frontend only retains the content of the current page. In this scenario, the pagination metadata is still updating; therefore, only metadata updates are sent to the frontend. Progressive computation has various output modes for early results. In an `append-only` mode, the result only adds new entries as the execution continues. In this case, the backend sends a number of results to the frontend. In a `retractable` mode, the earlier results could be deleted or updated. An example is incrementally updating the results of a group-by aggregation, such as a sum operation. In this case, the sum within each group keeps changing. For example, the sum might be updated from 100 to 200. In this case, updates are sent to the frontend to inform of row number of the dirty rows, meaning the content of the rows are updated, and the frontend needs to re-fetch data from the backend to refresh the results.

Shared Execution Control

In Texera, each collaborator should be able to control the execution, such as pausing and resuming it. It is important for their actions to be immediately visible to other users. For example, if Alice pauses the execution, Bob should instantaneously see on his browser that the workflow is paused. Figure 2.18 illustrates the process of shared execution control. Initially, Alice sends a `Pause` request to the Amber engine via the shared execution manager. Once all operators are paused, Amber sends a `Paused` notification to the state manager. The state manager then updates the current state and broadcasts this new state to all the web UIs. As a result, every collaborator observes the workflow's execution state changing to `Paused`. Collaborators can inspect and modify the workflow, then resume or rerun it according to the new logic. If any editing conflicts occur, these are resolved before the execution resumes.

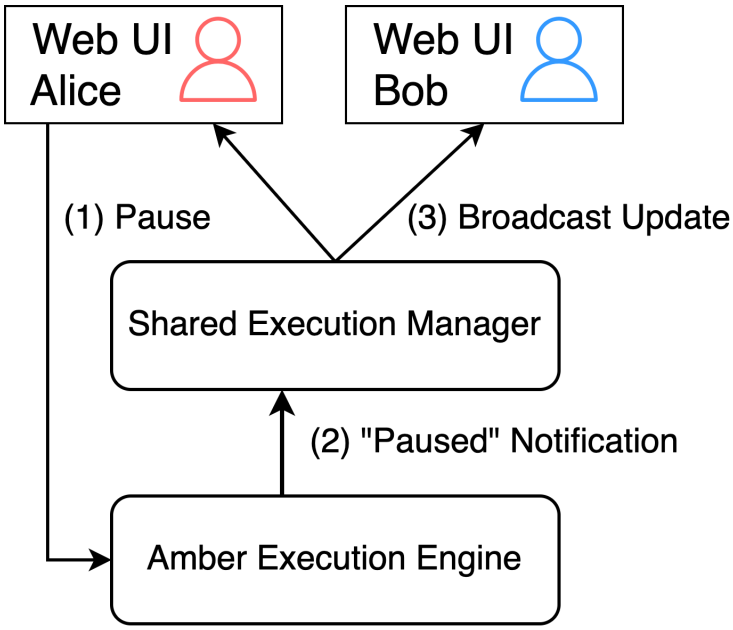


Figure 2.18: When a user pauses a workflow, the states of all web UIs are updated.

Adding a New Collaborator

A user should have the ability to invite a new collaborator to join an execution session at any point, to help with the result analysis or troubleshooting. In this scenario, the execution state should be seamlessly shared with the new collaborator's frontend UI, i.e., any user can attach or detach from the shared execution state at any time. For instance, if Alice invites her colleague, Charlie, to help with investigating a buggy execution, Charlie then joins the ongoing session as a new collaborator. At this point, Charlie should be able to access all the information available to Alice and Bob, including the workflow status, operator statistics, and past user interaction history. Charlie can also inspect the training metrics in detail and evaluate expressions at the exact same execution state as Alice and Bob. Figure 2.19 depicts the process of incorporating new collaborators into a workflow execution session.

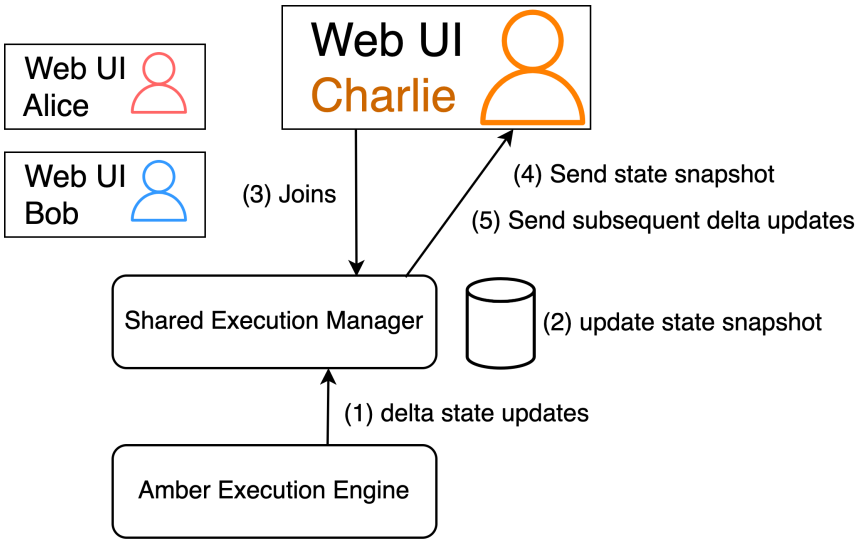


Figure 2.19: (1) Shared Execution Manager receives delta updates from the Amber engine. (2) A state snapshot is maintained by applying these delta updates. (3) A new user joins the execution session. (4) The execution state snapshot is sent to the new user first to update the new user's web UI. (5) The new user's web UI receives subsequent delta updates.

Consider the aforementioned method of sending incremental updates to clients. A problem arises when a new user joins an execution session in the middle of an execution. If the new user's frontend only receives incremental updates from the moment they join, their

frontend state will be incomplete. For example, the new user might only see interactions made after they join, and not be able to view past interaction history, which could be critical for investigating problems and analyzing results. To address this problem, the state manager maintains the current state snapshot of the execution. Whenever the state manager receives a new incremental update from the Amber execution engine, the snapshot stored in the state manager is updated using the same method as the clients. When a new user such as Charlie joins the execution session, the latest state snapshot and result snapshot are sent to Charlie's web UI to bring him up-to-date with the current execution progress. Afterward, all subsequent incremental updates are directed to the new user, ensuring that they have a complete and up-to-date view of the execution state. This architecture also allows an existing user to safely leave the execution session and later reconnect to the execution.

2.4.3 Experiments

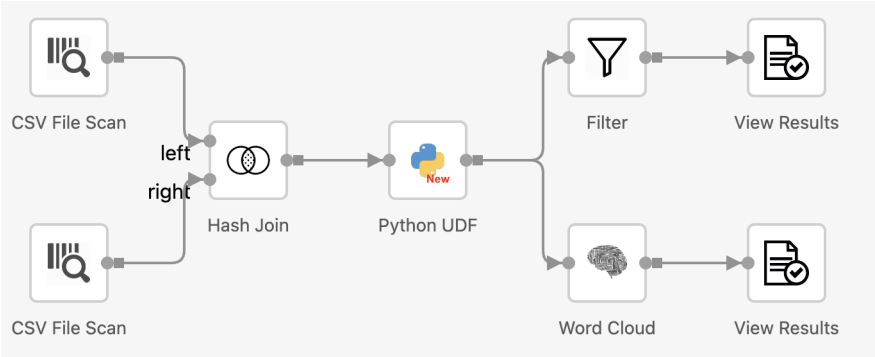


Figure 2.20: Example workflow to evaluate workflow re-compilations in a CRDT-based architecture.

We conducted experiments to evaluate the effectiveness of the optimizations discussed in Section 2.4.1. These optimizations aim to reduce the number of workflow re-compilations within a CRDT-based architecture. The workflow used in the experiment, as shown in Figure 2.20, involved joining a tweet table with a geo-location table, followed by a Python UDF operator. Subsequent operations included filtering the data and displaying the results

via a word cloud operator.

We designed a simulated collaborative editing session from a single user to four concurrent users, each concurrently constructing different parts of a workflow. We maintained the total number of editing steps made by all users at approximately the same value. This setup allowed us to compare the count of workflow re-compilations by the workflow compiler, both with and without the two above-mentioned optimizations.

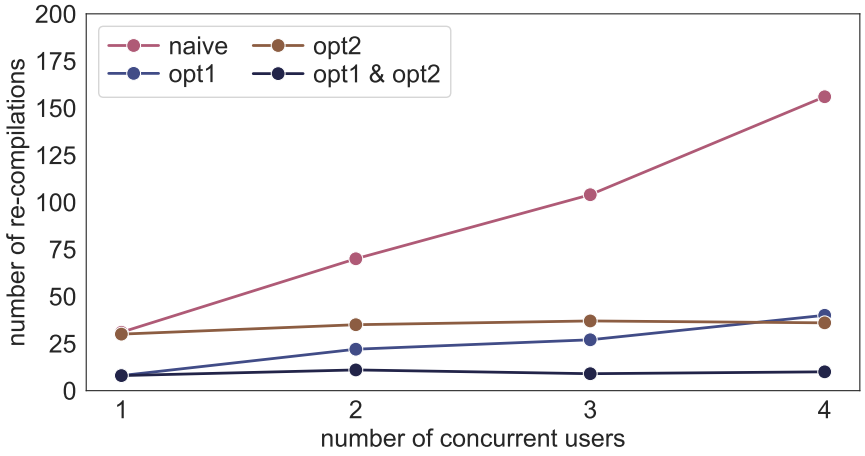


Figure 2.21: Number of workflow re-compilations with different number of users concurrently constructing a workflow.

Figure 2.21 shows the evaluation results. Without any optimizations, the count of workflow re-compilations increased linearly with the user count. For example, in a shared editing session with two users, the workflow compiler performed a total of 70 re-compilations, whereas the number increased to 156 with four users. With the first optimization, where only necessary changes were sent to the workflow compiler, we observed a slower growth rate in the number of workflow re-compilations as the user count increased. However, the growth remained linear relative to the user count.

With the second optimization, the number of edits remained largely unaffected by the number of concurrent users. This is because it is relatively rare to have race conditions involving edits made by different users within a very short time span. Furthermore, users typically

interacted with different components, which minimized conflicts. As a result, most requests could be served directly from the cache, thus avoiding re-compilations. Upon combining the two optimizations, we observed the least number of re-compilations, yielding the best results.

Chapter 3

Fries: Fast and Consistent Runtime Reconfiguration in Dataflow Systems with Transactional Guarantees

3.1 Introduction

Big data systems are widely used to process large amounts of data. Each computation job in these systems can take a long time to run, from hours to days or even weeks to finish. Applications that require timely processing of input data often use pipelined dataflow execution engines [35, 37, 103, 16], for example, in the scenarios of processing real-time streaming data, or answering queries progressively to provide early results to users. In these applications, when a long running job continuously processes ingested data, developers often need to change the computing logic of the job without disrupting the execution, as illustrated in the following example.

Consider a data-processing pipeline for payment-fraud detection shown in Figure 3.1. This simplified dataflow resembles many real-world applications [95, 44]. A stream of payment tuples is continuously ingested into the dataflow, with each tuple containing payment information such as customer, merchant, and amount. The dataflow uses two machine learning (ML) operators FC and FM to detect fraud based on customer and merchant information.

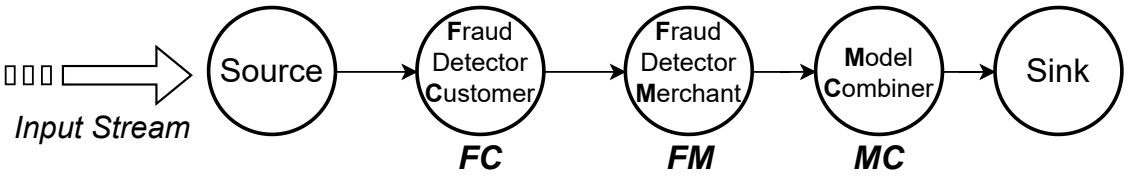


Figure 3.1: An example data-processing pipeline for fraud detection processing continuously ingested data.

Consider two example use cases in this dataflow. *Use case 1: fixing loopholes in operators.* After observing unexpected tuples from the Sink operator, the user identifies a loophole in the operator FM. She wants to update this operator to incorporate new rules to fix the loophole, without stopping the execution. *Use case 2: handling surges of data arrival rate.* Suppose the data arrival rate at the source suddenly increases, and as a result, the end-to-end processing latency becomes larger. The user finds that the ML operator FM is the bottleneck. To reduce

the latency, she wants to "hot-replace" the expensive ML model (e.g., a deep neural network) with a lightweight model (e.g., a decision tree) to improve its performance, thus reduce the processing latency. Again, she wants to make the change without stopping the execution. These examples show the importance of allowing developers to change the dataflow execution "on the fly." We call such changes *runtime reconfigurations*. This problem has gained a lot of interest in the research areas of software engineering [89], mobile computing [61, 102], and distributed systems [64, 72]. Recently, users of dataflow systems also show the need for runtime reconfigurations [95, 47, 46] and more systems start supporting this important feature [34], such as Amber [65], Chi [73], Flink [105], and Trisk [76].

Naturally there is a delay from the time a user requests a reconfiguration to the time its changes take effect in the target operators. This delay is critical to the performance of the system. For example, in use case 1, the user wants to fix the loophole as soon as possible since a large reconfiguration delay can cause financial losses. In use case 2, a large delay in mitigating the surge can cause the system to suffer longer in terms of long latency and wasting of computing resources. Thus we want this delay to be as low as possible.

A main limitation of existing systems supporting runtime reconfigurations is that they could have a long reconfiguration delay. In these systems, after a reconfiguration request is submitted, they need to wait for all the in-flight tuples to be processed by those target reconfiguration operators, as well as those earlier operators in the dataflow, before the requested changes can be applied on the target operators. This delay could be very long, when there are many in-flight tuples, or some of these operators are expensive, especially for operators using advanced machine learning models and those implemented as user-defined functions (UDF's).

In this paper, we develop a novel technique, called "Fries," to perform runtime reconfigurations with a low delay. It leverages the emerging availability of fast control messages in many systems recently. A *fast control message*, "FCM" for short, is a message exchanged between

the controller in the data engine and an operator without being blocked by data messages. Figure 3.2 shows an example of handling a reconfiguration request of two operators FM and MC using FCM's. Upon a reconfiguration request, the controller sends an FCM to each of the two operators, and each of them applies the new configuration immediately after receiving the message. Since FCM's are sent separately from data messages, these changes can reach the target operators much faster.

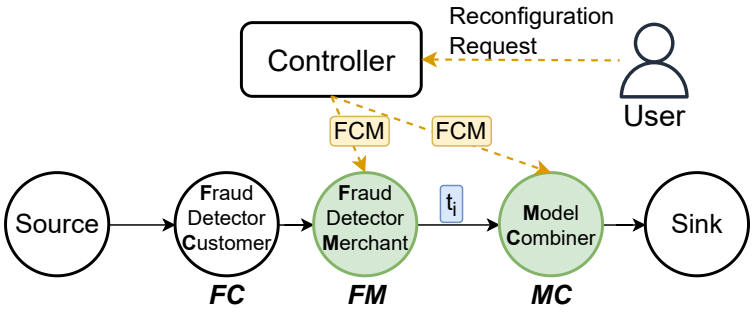


Figure 3.2: Handing a runtime reconfiguration of operators FM and MC using fast control messages (FCM's).

We will show in Section 3.4.1 that the naive way of using FCM's can cause consistency issues in Figure 3.2. It has unexpected side effects, e.g., producing incorrect results on the output tuples, or even causing the operator MC to crash. This example shows several challenges in developing Fries: 1) What is the meaning of "consistency" in this reconfiguration context? 2) How to ensure this consistency while reducing their delay? 3) How to deal with different types of operators and support parallel executions? We study these challenges and make the following contributions.

- We analyze epoch-based reconfiguration schedulers and show their limitations (Section 3.3).
- We formally define consistency of a reconfiguration based on transactions (Section 3.4).
- We first consider a simple class of dataflows that have one-to-one operators only, and develop a Fries scheduler that guarantees consistency (Section 3.5).

- We then consider the general class of dataflows with one-to-many operators, and extend the Fries scheduler (Section 3.6).
- We extend Fries to more general cases, such as dataflows with blocking operators and multiple workers. We also discuss how to support fault-tolerance (Section 3.7).
- We conduct an extensive experimental study to evaluate Fries in various scenarios and show its superiority compared to epoch-based schedulers (Section 3.8).

3.1.1 Related Work

Reconfiguration systems. Recently, many data-processing systems have started to support reconfigurations. Flink [35] supports reconfiguration by taking a savepoint [45], killing the running job, then restarting the job with the new configuration. This approach is disruptive to the dataflow execution. Spark Streaming [22, 116] uses a mini-batch-based execution strategy and supports reconfiguration between mini-batches. Chi [73] enables runtime reconfiguration by propagating epoch markers in its data stream. Trisk [76] provides an easy-to-use programming API for reconfigurations. The approaches in these systems are all based on epochs, which can have a long reconfiguration delay, as analyzed in Section 3.3. Fries relies on FCM's to perform reconfigurations with a low delay. Noria [51] is a system that uses dataflows to incrementally maintain materialized views. The system supports reconfigurations of view definitions, which require the new views to be recomputed from entire base tables. In Fries, an update of a dataflow only affects the future tuples. The input tuples that are already processed by the dataflow are not recomputed using the new configuration.

Re-scaling systems. Some systems [43, 57, 79] support updating the dataflow for re-scaling. For example, Megaphone [57] based on timely dataflow [81] supports a fine-granularity re-scaling and Rhino [79] based on Flink supports re-scaling with very large states. Fries

focuses on reconfiguring the computation functions of operators, which is different from re-scaling.

Transactions in dataflow systems. S-Store [77] and the work in [30] are systems that allow streaming dataflows and OLTP workloads to access a shared mutable state. Although both systems do not support reconfigurations directly, we could map a reconfiguration to these systems. S-Store defines transactions on the processing of each input batch on a single operator. This model cannot express our consistency requirements in reconfigurations. The work in [30] treats a dataflow as a black box, thus it has the limitation of not being able to utilize the properties of the dataflow and its operators to reduce the reconfiguration delay. Fries can do so to achieve this reduction. Both earlier systems include a transaction scheduler to manage the processing of data, which creates scheduling overhead even when there is no reconfiguration. The Fries scheduler has no such overhead before receiving a reconfiguration request. Additionally, both earlier systems are only on a single node, while the Fries scheduler can run on a distributed engine on a cluster.

Transactions in database systems. Transactions are widely studied in traditional database systems (e.g., [27, 109, 28]). A uniqueness in transactions in our work is that they treat operations in a reconfiguration as a separate transaction, which is handled differently from data transactions. In addition, Fries does optimizations by utilizing special properties in our problem setting, including the DAG shape of a dataflow, and types of operators, e.g., one-to-one and one-to-many. Moreover, the Fries scheduler uses FCM's and epoch markers to schedule transactions without locking.

3.2 Problem Settings

3.2.1 Data-Processing Model

A data-processing system runs a computation dataflow job represented as a directed acyclic graph (DAG) of operators. Each operator receives tuples from its input edges, processes them, and sends tuples through its output edges. An operator contains a computation function f represented as

$$f : (s, t) \rightarrow \big(s', \{(t'_1, o'_1), \ldots, (t'_n, o'_n)\}\big).$$

The function processes a tuple t at a time with a state s of the operator, produces a set of zero or more output tuples $\{t'_1, \ldots, t'_n\}$, where each tuple t'_i has a receiving operator o'_i. The operator also updates its state to s'. The system has a module called *controller* that manages the execution of the job, handles requests from the user, and exchanges messages with operators during the execution.

For simplicity, we first focus on dataflows under the following assumptions. (1) A dataflow contains pipelined operators only, such as selection, projection, union, and other tuple-at-a-time operators. We consider a class of join operators where the operator first collects all the tuples from one input (e.g., the "build" input of a hash join), then starts processing tuples from the other input (e.g., the "probe" phase of a hash join). We consider the processing of tuples from the second input of join. (2) Each operator has a single worker. We relax these assumptions in Section 3.7.

As an example, consider the data-processing pipeline for payment-fraud detection shown in Figure 3.1. The example dataflow uses two machine learning (ML) operators for fraud detection. The first one, denoted as FC, keeps a state of the 5 recent tuples of each customer.

For each input tuple, FC updates the state and feeds the 5 recent tuples of the customer into an ML model. The predicted probability $p_c(5)$ is attached as a new column of the tuple. The second one, denoted as FM, keeps a state of the 5 recent tuples of each merchant. Similarly, it uses an ML model to generate a predicted probability $p_m(5)$, and attaches it as a new column of the tuple. Finally, the model combiner MC uses $p_c(5)$ and $p_m(5)$ of each tuple to compute the final average probability with the weights $[0.4, 0.6]$.

3.2.2 Runtime Reconfiguration

Definition 3.1 (Runtime reconfiguration)**.** *During the execution of a dataflow, an update to the computation functions of its operators is a* runtime reconfiguration *of this execution.*

Formally, a reconfiguration $\mathcal{R}$ is a set of operators with a function update $\mu(o_i)$ for each operator o_i, i.e.,

$$\mathcal{R} = \{(o_1, \mu(o_1)), \ldots, (o_n, \mu(o_n))\}.$$

Each operator o_i has a *function-update operation* $\mu(o_i)$. This operation applies a pair $\langle f'_{o_i}, \mathcal{T}_{o_i} \rangle$ to the operator, where f'_{o_i} is a new computation function of the operator. $\mathcal{T}_{o_i}$ is a state transformation that converts the operator's original state s to a new state $s^* = \mathcal{T}_{o_i}(s)$, which can be consumed by f'_{o_i}. In this paper, we consider the case where there is one reconfiguration at a time.

In the running example, suppose the user identifies a flaw in the dataflow and wants to reconfigure the two operators FM and MC. Specifically, the user wants to change FM to output an additional probability value $p_m(10)$, which is predicted using the 10 recent tuples of each merchant. The operator MC needs to be updated to combine all three probabilities ($p_c(5)$, $p_m(10)$, and $p_m(5)$) with the new weights $[0.4, 0.4, 0.2]$. Table 3.1 shows the old and new configurations of the two reconfiguration operators.

	FM's output	MC weights
Old configuration	$p_m(5)$	[0.4, 0.6]
New configuration	$p_m(10)$, $p_m(5)$	[0.4, 0.4, 0.2]

Table 3.1: Operator executions during a reconfiguration.

Note that the new configuration of an operator can require a state different from that of the old configuration. In this case, the reconfiguration can use a state transformation to migrate the old state to the new one. In the running example, the old configuration of operator FM uses a state with the last 5 payment tuples for each merchant. However, the new configuration of FM needs a list of last 10 tuples for each merchant. The user provides a state transformation $\mathcal{T}$ for operator FM, to instruct the system in transferring operator FM's old state to the new one. In this example, the user chooses to fill the new state with the 5 tuples from the old state and 5 additional *null* values.

3.3 Epoch-Based Reconfiguration Schedulers and Limitations

In this section, we explain epoch-based reconfiguration schedulers and show their limitation of long delays.

3.3.1 Epoch-Based Schedulers

Dataflow epoch. A stream of tuples processed by the system can be divided into consecutive sets of tuples, where each set is called an *epoch* [32]. One way to create epochs is to use epoch markers. At the start of a new epoch, an epoch marker is injected to each source operator. The epoch marker is then propagated along the data stream using the following protocol [32]. When an operator receives an epoch marker from an input channel, it performs epoch alignment by waiting for all its inputs to receive an epoch marker, then sends the marker downstream. As an example, Figure 3.3 shows two epochs during the execution of the fraud-detection dataflow. An epoch marker injected between t_4 and t_5 divides the input stream into two epochs. The epoch marker indicates the end of epoch 1 and the start of epoch 2.

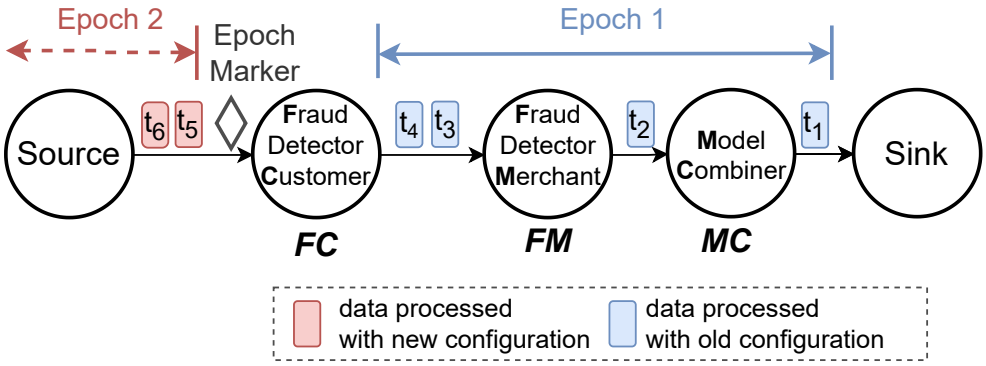


Figure 3.3: An epoch-based reconfiguration scheduler in Chi [73]. It uses an epoch barrier to apply the new configuration to operators FM and MC at the start of Epoch 2.

Definition 3.2 (Epoch-based Scheduler)**.** *An* epoch-based scheduler *schedules a reconfiguration request between two epochs. That is, for each reconfiguration operator O, all the tuples in the old epoch are processed with the old configuration of O, and all the tuples in the new epoch are processed with the new configuration of O.*

Considering the aforementioned method to generate epochs, the following is an implementation adopted by Chi [73]. We call this implementation "Epoch Barrier Reconfiguration" scheduler, or "EBR" in short. Upon a reconfiguration request, the controller starts a new epoch and piggybacks the reconfiguration in the epoch marker. When a reconfiguration operator receives epoch markers from all its inputs, it applies the new configuration. The operator then processes the input tuples in the next epoch using the new configuration. Figure 3.3 shows the process of handling a reconfiguration of operators FM and MC using the EBR scheduler. When operator FM receives the epoch marker, it applies the new configuration, and propagates the marker to operator MC. When operator MC receives the epoch marker, it also applies the new configuration.

System	**Epoch creation**	**Reconfiguration Strategy**
Chi [73]	Epoch makers	Piggybacking control messages in epoch markers
Flink [45]	Epoch makers	Stop-and-restart
Spark Streaming [22]	Mini-batch	Stop-and-restart

Table 3.2: Epoch-based reconfiguration schedulers.

Table 3.2 shows different epoch-based reconfiguration schedulers.

In Flink [45], upon a reconfiguration, a new epoch is immediately triggered using an epoch marker. At the end of the old epoch, each operator saves its state into a checkpoint. After the old epoch is processed by all the operators, Flink kills the execution, updates the dataflow graph, loads the saved states, and restarts the dataflow. In Spark Streaming [22], epochs are created by dividing the input data stream into small mini-batches, each of which is an epoch. A mini-batch is processed one at a time by launching a separate computation job.

Upon a reconfiguration, the system modifies the dataflow graph before starting the job of the next mini-batch.

3.3.2 Limitations: Long Reconfiguration Delays

A major limitation of epoch-based reconfiguration schedulers is a long reconfiguration delay, which is from the time a request is submitted to the time the new configuration takes effect in the target operators. In particular, the system needs to process all the in-flight tuples before the new epoch. Take the EBR scheduler in Figure 3.3 as an example. Operator FM needs to finish processing the in-flight tuples t_3 and t_4. In general, this delay could be long due to the following reasons. First, the dataflow can contain multiple expensive operators that make the processing of an epoch slow. Second, the number of in-flight tuples could be large, especially when the system is under high workload. We may want to reduce the number of in-flight tuples by decreasing the buffer size. However, a smaller buffer can be easily filled by a minor fluctuation in the input ingestion rate. When the buffer is full, the system triggers back-pressure, which can decrease the throughput. Moreover, a small buffer size causes the networking layer to transmit data in small batches, which introduces additional transmission overhead. Compared to the EBR approach, the Flink approach suffers from an additional delay of stopping and restarting the dataflow. Spark Streaming can also have a long reconfiguration delay. The delay is determined by the processing time of a mini-batch, with a predefined interval usually set to a few seconds. However, the delay can be higher when the processing speed cannot keep up with a surge of the data ingestion rate.

3.4 Scheduling Reconfigurations Using Fast Control Messages

In this section, we introduce a new type of reconfiguration schedulers based on fast control messages (FCM's). We present a naive scheduler and show its issues. We then formally define consistency of a reconfiguration.

Definition 3.3 (Fast Control Message)**.** *A* fast control message*, "FCM" for short, is a message exchanged between the controller and an operator without being blocked by data messages.*

There are many ways to implement fast control messages. For instance, to send an FCM from the controller to the fraud detector in our running example, one approach is to set up a new communication channel between the controller and the fraud detector. The channel is separate from existing data channels, and the FCM can bypass data messages. Another way is to transmit the FCM using existing data channels, but assigning a higher priority to the FCM. The FCM is first sent to a source operator of the workflow, then propagated along the edges to the fraud detector, and it bypasses data messages in each data channel.

3.4.1 FCM-based Schedulers

Naive FCM scheduler. A main benefit of using FCM's to schedule reconfigurations compared to epoch-based schedulers is that FCM's have a much smaller delay. A naive scheduler leverages this benefit as follows. The controller sends an FCM directly to each reconfiguration operator. When an operator receives an FCM, it applies the new configuration immediately after finishing the processing of its current tuple. We use Figure 3.2 to explain how the naive scheduler works in a reconfiguration of two operators FM and MC. Using this scheduler,

the controller sends an FCM directly to each of the two operators FM and MC. The FCM carries the new function f' and the state transformation $\mathcal{T}$ of the corresponding operator. These operators update their configuration after receiving their FCM.

While this naive scheduler has a low reconfiguration delay, it could generate an undesirable reconfiguration schedule in this example. Notice that the scheduler does not coordinate the updates to these two operators that run independently. Consider the in-flight tuple t_i, which is processed by FM using its old configuration. Suppose the MC switches to the new configuration before the arrival of t_i. Then tuple t_i is processed by MC using its new configuration. The tuple contains two probability values $p_c(5)$ and $p_m(5)$, but the new configuration of MC expects three probability values. This schema mismatch could have unexpected side effects, such as an incorrect result on the produced output tuple, or even causing the operator MC to crash. This example shows the importance for the reconfiguration to be performed in a synchronized manner on the two reconfiguration operators. In particular, we want a tuple to be processed by the two operators either using the old configuration or using the new configuration.

FCM multi-version scheduler. To ensure a tuple is processed by the same configuration of multiple operators, we can use the following FCM-based multi-version scheduler that maintains multiple configurations of an operator at the same time. The controller first sends an FCM to each reconfiguration operator. Each operator keeps both the old configuration and the new one. After all operators have received the FCM, each source operator increments its version number, which is tagged to each source tuple. For each input tuple, an operator checks the tuple's tagged version number, chooses the corresponding configuration version to process the tuple, and tags the same version number to the output tuples. As an example, in Figure 3.4, after the new configuration is sent to operator E, the source operators then tag subsequent output tuples t_3 and t_4 with the new version v_2.

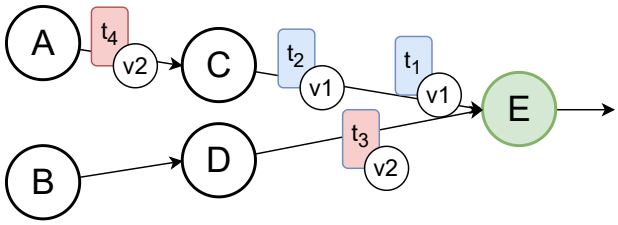


Figure 3.4: Using an FCM multi-version scheduler, an operator processes a tuple based on its version tag.

This scheduler has two problems. First, each reconfigured operator may need to keep two sets of states for two configurations, and these states could be very large (e.g., large hash tables or machine learning models). Second, this scheduler still suffers from a possible high reconfiguration delay. In particular, similar to the case of the EBR scheduler, there can be a large amount of in-flight tuples that are already tagged with the old version and they still need to be processed with the old configuration (e.g., t_1 and t_2 in Figure 3.4).

3.4.2 Reconfiguration Consistency

We formally define the consistency requirements in this context. At a high level, we treat the processing of a single source tuple by multiple operators as one *transaction*, and a reconfiguration as another transaction. We use conflict-serializability to define the consistency of a schedule of a reconfiguration.

Definition 3.4 (Scope of a source tuple)**.** *The* scope *of a source tuple t of a dataflow W, denoted as $\mathcal{S}(W, t)$, is a pair $(\mathcal{S}, \preccurlyeq_{\mathcal{S}})$, where $\mathcal{S}$ is a set of tuples and $\preccurlyeq_{\mathcal{S}}$ is a partial order on $\mathcal{S}$, defined as follows:*

1. *The source tuple is in $\mathcal{S}$.*
2. *For each tuple s in $\mathcal{S}$, if an operator processes the tuple s and produces zero or more output tuples $\{s'_1, \ldots, s'_n\}$, all the produced tuples are also in $\mathcal{S}$. For each tuple s'_i, we have the order $s \prec s'_i$ in $\preccurlyeq_{\mathcal{S}}$.*

For instance, in Figure 3.5, a source tuple t is ingested into the dataflow from the source operator A and processed by operators C, D, E, F, and H. The scope of t includes the tuples on the highlighted edges and their partial order defined as their edges on the DAG.

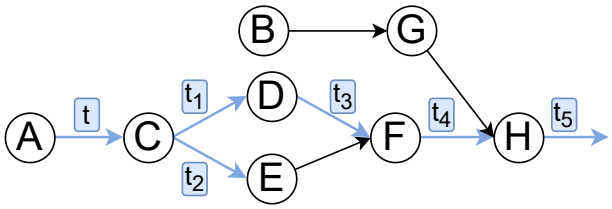


Figure 3.5: Scope of a source tuple in a dataflow.

Definition 3.5 (Data operation)**.** *The* data operation *of a tuple s is the processing of s by its receiving operator o, denoted as $\phi(s, o)$.*

Definition 3.6 (Data transaction)**.** *For a dataflow W and a source tuple t in W, let $(\mathcal{S}, \preccurlyeq_{\mathcal{S}})$ be the scope of t. The* data transaction *of t is a pair $(\Phi, \preccurlyeq_{\Phi})$, where Φ is the set of data operations of the tuples in $\mathcal{S}$, and $\preccurlyeq_{\Phi}$ is a partial order on Φ. For two data operations $\phi(t_i, o_i)$ and $\phi(t_j, o_j)$ in Φ, we have $\phi(t_i, o_i) \prec \phi(t_j, o_j)$ in $\preccurlyeq_{\Phi}$ if and only if $t_i \prec t_j$ is in $\preccurlyeq_{\mathcal{S}}$.*

For instance, in Figure 3.2, tuple t has the following data transaction T_1:

$$T_1 : [\phi(t, FC), \phi(t, FM), \phi(t, MC)].$$

In the data transaction, "$\phi(t, FC)$" is a data operation representing the processing of this tuple t by the FC operator.

Definition 3.7 (Function-update transaction)**.** *The* function-update transaction *of a reconfiguration $\mathcal{R} = \{(o_1, \mu(o_1)), \ldots, (o_n, \mu(o_n))\}$ on a dataflow W is the set $\{\mu(o_1), \ldots, \mu(o_n)\}$, where each $\mu(o_i)$ is a function-update operation in $\mathcal{R}$.*

For instance, the reconfiguration in Figure 3.2 has the following function-update transaction T_2:

$$T_2 : \{\mu(FM), \mu(MC)\}.$$

In the function-update transaction, "$\mu(FM)$" is a function-update operation representing that the operator FM switches to the new configuration. Note that the order of different operations in a function-update transaction does not matter because they update different operators and are independent of each other.

Definition 3.8 (Conflicting operations)**.** *A data operation $\phi(t, o)$ and a function-update operation $\mu(o')$ are said to be* conflicting *if $o = o'$, i.e., they are on the same operator. They are said to be* not conflicting *if $o \neq o'$.*

For instance, in Figure 3.2, operations $\phi(t, FM)$ and $\mu(FM)$ are conflicting because they are on the same operator. Operations $\phi(t, FC)$ and $\mu(FM)$ are not conflicting as they are on different operators.

Definition 3.9 (Schedule)**.** *A* schedule *of a set of transactions $T_1, \ldots, T_k$ is the set of all the operations in those transactions with a* partial order*. The schedule is called* serial *if for each pair of transactions T_i and T_j, T_i's operations in the schedule are either all before those in T_j or all after those in T_j.*

In this thesis we only consider schedules that include one function-update transaction and many data transactions.

Definition 3.10 (Conflict-equivalence)**.** *Two schedules S_1 and S_2 of the same set of transactions are said to be* conflict-equivalent *if $\forall o_i, o_j \in S_1$, if o_i and o_j are conflicting, and o_i is before o_j in S_1, then o_i is also before o_j in S_2.*

Definition 3.11 (Conflict-serializable)**.** *A schedule is said to be* conflict-serializable *if it is conflict-equivalent to a serial schedule of the same set of transactions.*

In the rest of the paper, when a partial order of a data transaction or a schedule defines a total order, for simplicity, we just show the transaction or the schedule as a sequence. We use the running example in Figure 3.1 to explain these concepts.

- S_1 is a schedule of the two transactions T_1 and T_2:

$$S_1 : [\phi(t, FC), \mu(FM), \phi(t, FM), \mu(MC), \phi(t, MC)].$$

- S_2 is a serial schedule of the two transactions:

$$S_2 : [\mu(FM), \mu(MC), \phi(t, FC), \phi(t, FM), \phi(t, MC)].$$

  In particular, all T_2's operations in this schedule are before those in T_1.

- S_1 and S_2 are conflict-equivalent. For example, for the conflicting pair $\mu(FM)$ and $\phi(t, FM)$, the former is before the latter in both schedules.

- S_1 is conflict-serializable because it is conflict-equivalent to the serial schedule S_2.

- S_3 is not a conflict-serializable schedule:

$$S_3 : [\phi(t, FC), \phi(t, FM), \mu(FM), \mu(MC), \phi(t, MC)].$$

  We can show that S_3 is not conflict-equivalent to any serial schedule. Intuitively, it has two pairs of conflicting operations, namely [$\phi(t, FM)$, $\mu(FM)$] and [$\mu(MC)$, $\phi(t, MC)$], and their corresponding transaction orders are different.

S_3 is the "bad" schedule generated by the naive FCM scheduler in Section 3.4.1, in which tuple t is processed using the old configuration of FM and the new configuration of MC. Schedule S_1 is a "good" schedule since t is processed entirely using the new configurations of both operators FM and MC and the aforementioned schema-mismatch issue does not happen.

Consistency of epoch-based schedulers. Consider the example in Figure 3.1. The aforementioned schedule $S1$ in Section 3.4.2 is produced by the EBR epoch-based scheduler,

where the epoch marker is propagated before tuple t. We show that the EBR approach can always produce a conflict-serializable schedule in Lemma 3.1. We also show that in general, an epoch-based scheduler always produces conflict-serializable schedules in Lemma 3.2.

Lemma 3.1. *Every schedule produced by the EBR epoch-based scheduler is conflict-serializable.*

Proof. Let S be a produced schedule. We construct a serial schedule S' using the following steps. Consider the function-update transaction U and each data transaction T for a tuple t. Since the epoch marker serve as a barrier dividing the input stream into two epochs, t can be in only one of the following two cases:

- t is before the epoch marker. For each conflict in S between a data operation ϕ in T and a function-update operation μ in U, ϕ is before μ. We place T before U in S'. Thus, in S', ϕ is also before μ.
- t is after the epoch marker. Similarly, we place T after U in S'. Each conflict order in S remains the same in S'.

For those data transactions before U, we order them in S' following the order of their first data operations in S. For data transactions after U, we order them in S' following the order of their first data operations in S. Notice that there are no conflicts between two data transactions.

The schedule S is conflict-equivalent to the constructed serial schedule S' because all conflicting pairs in S have the same order in S'. Therefore, S is conflict-serializable. □

Lemma 3.2. *Every schedule produced by a general epoch-based scheduler is conflict-serializable.*

Proof. Consider a function-update transaction U and each data transaction T for a source tuple t. For a general epoch-based scheduler, all the tuples in the scope of t are in one

epoch E_t and U is scheduled between two epochs E_i and E_{i+1}. We can compare E_t with the boundary between E_i and E_{i+1} to determine the position of T in a serial schedule. We can prove this claim using steps similar to those in the proof of Lemma 3.1. □

3.5 Dataflows with one-to-one Operators Only

In this section, we consider the case where a dataflow contains one-to-one operators only. We propose a scheduler called Fries, which uses FCM's to achieve low reconfiguration delay and still guarantees conflict-serializability of produced schedules.

Definition 3.12 (One-to-one operator)**.** *An operator is called* one-to-one *if its processing function emits at most one (tuple, receiving operator) pair for each input tuple.*

This type includes operators such as projection, filter, map function, equi-join on key attributes, and union.

Definition 3.13 (One-to-many operator)**.** *An operator is called* one-to-many *if its processing function can emit more than one output (tuple, receiving operator) pair for an input tuple.*

This type includes operators such as join on non-key attributes and flatten function. In the rest of this section, we consider dataflows where all operators in the dataflow are one-to-one.

3.5.1 Conflict-Serializable Schedules Produced by the Naive FCM-based Scheduler

Section 3.4.1 shows an example dataflow and a reconfiguration where the naive FCM-based scheduler produces a non-conflict-serializable schedule. Next we use an example to show that the naive scheduler can still guarantee conflict-serializability for some types of dataflows and reconfigurations.

Example 3.1. *Suppose we want to use the naive FCM-based scheduler to handle a reconfiguration of the two operators C and D as shown in Figure 3.6. Operator X is a one-to-one*

operator that route each output tuple to either operators C and D. In this case, we have a data transaction $T_3 = [\phi(t_1, X), \phi(t_1, C])$, another data transaction $T_4 = [\phi(t_2, X), \phi(t_2, D])$, and a function-update transaction $U = [\mu(C), \mu(D)]$. The controller sends two separate FCM's to C and D. Consider a possible schedule with T_3, T_4, and U:

$$S_4 : [\phi(t_1, X), \mu(C), \phi(t_1, C), \phi(t_2, X), \mu(D), \phi(t_2, D)].$$

Schedule S_4 is conflict-serializable because it is conflict-equivalent to the serial schedule $[U, T_3, T_4]$. Interestingly, we can show that all schedules produced by the naive FCM-based scheduler in Figure 3.6 are conflict-serializable.

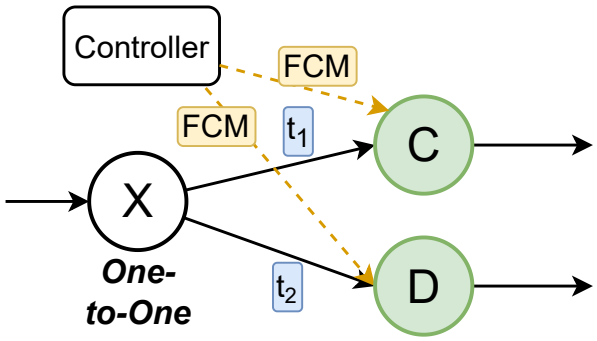


Figure 3.6: An example dataflow with a reconfiguration on operators C and D. The naive FCM-based scheduler always produces a conflict-serializable schedule.

One might wonder why the two examples in Figure 3.2 and Figure 3.6 are different in the conflict-serializability of the produced schedules. The main reason is that in Figure 3.2, a tuple can be processed by operators FM and MC, and both of them are in the reconfiguration. But there is no synchronization between the data operations and the function-update operations, causing the non-conflict-serializability. While in Figure 3.6, a tuple is processed by only one of the two paths through either C or D. On each path, there is a single operator in the reconfiguration, thus the data operations and the function-update operations are always synchronized.

Next, we introduce a concept called "minimal covering sub-DAG," which is used to represent

the synchronization components. We then describe the Fries scheduler using this concept, and prove that this scheduler can always produce a conflict-serializable schedule.

3.5.2 Minimal Covering Sub-DAG (MCS)

Definition 3.14 (Minimal covering sub-DAG)**.** *Given a DAG $G = (V, E)$, and a set of vertices $M \subseteq V$, a minimal covering sub-DAG $G' = (V', E')$ is defined as follows:*

1. *$M \subseteq V'$;*
2. *$\forall A, B \in M$, if there is a path from A to B, then all the vertices and edges on the path are in V' and E', respectively;*
3. *G' is minimal, i.e., no proper sub-DAG of G' can satisfy the above two conditions.*

Lemma 3.3. *There is a unique MCS given a DAG and a set of vertices.*

Proof. Suppose G'_1 and G'_2 are two distinct MCS's of a DAG and a set of vertices M. Consider the sub-DAG G'_3 that is the "intersection" of G'_1 and G'_2, i.e., the vertices of G'_3 are the intersection of the two sets of vertices in G'_1 and G'_2, the edges of G'_3 are the intersection of the two sets of edges in G'_1 and G'_2. Since M is a subset of the sets of vertices in G'_1 and G'_2, M is also a subset of the vertices in G'_3. Thus G'_3 satisfies property (1). $\forall A, B \in M$, if there is a path p from A to B, p is also in both G'_1 and G'_2, so p is also in the G'_3. Thus G'_3 also satisfies property (2). Therefore, G'_3 is also an MCS, which contradicts the minimality property (3) of G'_1 and G'_2. □

Algorithm 1 shows an algorithm for finding the minimal covering sub-DAG (MCS) given a DAG G and a set of vertices M. In lines 5- 10, we iterate through the DAG in a topological order. For each vertex v, we mark v in "red" if v is in M or any parent vertex of v is marked

in "red." After this iteration, a vertex marked in "red" is either 1) in M, or 2) a descendant of a vertex in M. Next in lines 11- 16, we iterate through the DAG in a reverse topological order. For each vertex v, we mark v in "blue" if v is in M or any child of v is marked in "blue." After this iteration, a vertex marked in "blue" is either 1) in M, or 2) is an ancestor of a vertex in M. Finally, in lines 18- 20, we add all the vertices marked in both "red" and "blue" to the MCS because these vertices are either 1) in M, or 2) on a path between two operators in M. Then we add all the edges connecting these vertices to the MCS.

The time complexity of this algorithm is $O(V + E)$, specifically:

- The topological ordering in line 4 takes $O(V + E)$ [41];
- The loop of marking "red" in lines 5- 10 takes $O(V + E)$ because we iterate through each vertex once and look at every edge once;
- Similarly, the loop of marking "blue" in lines 11- 16 also takes $O(V + E)$;
- The loop in lines lines 18- 20 takes $O(V)$ and the loop in lines lines 22- 24 takes $O(E)$.

Figure 3.7 shows the minimal covering sub-DAG for the dataflow graph in Figure 3.5 and the set of operators $\{C, F, G\}$ in the reconfiguration. The sub-DAG is: $V' = \{C, D, E, F, G\}$ and $E' = \{C \rightarrow D, C \rightarrow E, D \rightarrow F, E \rightarrow F\}$. In general, we can show that there is a unique MCS given a DAG and a set of vertices, and we can compute the MCS using an algorithm with an $O(V + E)$ time complexity.

3.5.3 The Fries Scheduler

The Fries scheduler uses components of the MCS to schedule the reconfiguration. A *component* is a maximal sub-DAG of the MCS where every pair of vertices in the component are connected by a path, ignoring the direction of edges. For example, the sub-DAG in

Algorithm 1 Find Minimal Covering SubDAG

Input: dataflow DAG $G = (V, E)$
Input: $M = \{m_1, \ldots, m_n\}$

$D \leftarrow \varnothing$
for each $v \in V$ **do**
    $D[v] \leftarrow \varnothing$
let $v_1, \ldots, v_{|V|}$ be a topological ordering of G
for each $v \leftarrow v_1, \ldots, v_{|V|}$ **do**
    if $v \in M$ **then**
        add "red" to $D[v]$
    for each incoming edge e of v **do**
        if "red" $\in D[e.from]$ **then**
            add "red" to $D[v]$
for each $v \leftarrow v_{|V|}, \ldots, v_1$ **do**
    if $v \in M$ **then**
        add "blue" to $D[v]$
    for each outgoing edge e of v **do**
        if "blue" $\in D[e.to]$ **then**
            add "blue" to $D[v]$
$V' = \{\}$
for each $v \in V$ **do**
    if $D[v] = \{$"red", "blue"$\}$ **then**
        add v to V'
$E' = \{\}$
for each $e \in E$ **do**
    if $e.from \in V' \wedge e.to \in V'$ **then**
        add e to E'
return (V', E')

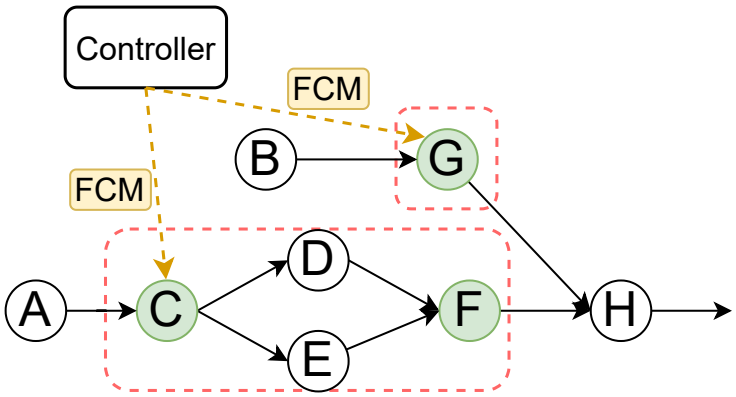


Figure 3.7: Two components of the minimal covering sub-DAG used in the Fries scheduler are highlighted in red.

Figure 3.7 has two components, each marked in a red box. The components of the MCS can be also computed using an algorithm [42] with an $O(V + E)$ time complexity.

The Fries scheduler is formally described in Algorithm 2. We first construct the minimal covering sub-DAG from the original dataflow DAG and operators in the reconfiguration (lines 1 and 2). We compute the components within the MCS (line 3). For each component in the MCS, the controller sends an FCM to the "head" operators, i.e., those with no input edges in the component. The head operators then start propagating an epoch marker within the component (lines 4 to 6). Specifically, when an operator receives an epoch marker, it performs marker alignment on the input edges in its component. An operator sends an epoch marker only to its downstream operators in its component.

Algorithm 2 The Fries Scheduler (for dataflows with one-to-one operators only)

Input: $G = (V, E)$
Input: $\mathcal{R} = \{(o_1, U_1), \ldots, (o_n, U_n)\}$
1: $M \leftarrow \{o_1, \ldots, o_n\}$
2: $G' \leftarrow findMCS(G, M)$
3: $\mathcal{C}_1, \ldots, \mathcal{C}_p \leftarrow findComponents(G')$
4: **for** each $\mathcal{C} \leftarrow \mathcal{C}_1, \ldots, \mathcal{C}_p$ **do**
5: send an FCM to the each head operator in $\mathcal{C}$
6: start propagating an epoch marker within $\mathcal{C}$

As an example, in Figure 3.7, the controller sends an FCM to operator C, which is the only head operator of the first component. The controller also sends an FCM to operator G, which is the only head operator of the second component. When C receives the FCM, it applies the new configuration and starts propagating an epoch marker to operators D and E. These operators then forward the marker to operator F. When F receives the marker from both D and E, it applies the new configuration and stops the marker propagation. When operator G receives the marker, it applies the new configuration and does not send out an epoch marker.

Next, we show that the Fries scheduler can always produce a conflict-serializable schedule.

Lemma 3.4. *Consider a dataflow graph G with one-to-one operators only, with a reconfiguration $\mathcal{R}$, and the MCS G' generated by Algorithm 2. Each component of G' contains at least one reconfiguration operator.*

Proof. By the construction of Algorithm 2, the set of reconfiguration operators M in $\mathcal{R}$ are used to construct the MCS G'. Suppose all the vertices in a component $\mathcal{C}$ of G' are not in M. We construct a new DAG G'' by removing all the vertices and edges in C from G'. Using similar steps as in Lemma 3.3, we can show that G'' is still a minimal covering sub-DAG of G and M. This result contradicts the minimality property of G'. □

Lemma 3.5. *In dataflows with one-to-one operators only, consider a dataflow graph G with a reconfiguration $\mathcal{R}$, and the MCS G' generated by Algorithm 2. For each source tuple, the operators in its data transaction overlap with at most one component of G'.*

Proof. Suppose the operators processing a tuple t overlap with two components $\mathcal{C}_1$ and $\mathcal{C}_2$ in the MCS. Based on Lemma 3.4, there is an operator A in $\mathcal{C}_1$ and another operator B in $\mathcal{C}_2$, where $A, B \in M$. In dataflows with one-to-one operators only, tuple t goes through a *chain* of operators and there must be a path between A and B in G. By Definition 3.14, the path must also be in G'. By the definition of components, A and B must be in the same component of G', which contradicts the assumption. □

Theorem 3.1. *In dataflows with one-to-one operators only, the Fries scheduler in Algorithm 2 always produces a conflict-serializable schedule.*

Proof. Let S be a produced schedule. We can construct a serial schedule S' using the following steps. Consider the function-update transaction U and each data transaction T for a tuple t. Based on Lemma 3.5, T can be in only one of the following two cases. (1) Operators in T do not overlap with any component in the MCS. In this case, T does not have any conflict with U. We can place T before U in S'. (2) Operators in T overlap with

one component in the MCS. In this case, we place T in S' by comparing the position of a tuple and the epoch marker of this component. In both cases, for those data transactions before U, we order them in S' following the order of their first data operations in S. For data transactions after U, we order them in S' following the order of their first data operations in S. Notice that there are no conflicts between two data transactions. The schedule S is conflict-equivalent to the constructed serial schedule S' because all conflicting pairs in S have the same order in S'. Therefore, S is conflict-serializable. □

The reconfiguration delay of the Fries scheduler is decided by the size of each MCS component, which is the number of edges in the component. Compared to the EBR scheduler, the FCMs sent to the head of each MCS component are not blocked by the processing of data by the upstream operators. Within each MCS component, the Fries scheduler still relies on epoch markers. In the extreme case where the MCS covers the entire dataflow graph, the Fries scheduler essentially becomes the epoch-based scheduler, where the FCMs are sent to all source operators and the epoch markers need to be propagated through the entire dataflow.

3.6 Dataflows with One-to-Many Operators

In this section we consider dataflows with one-to-many operators.

3.6.1 Challenges

Figure 3.8 shows a part of a dataflow with a one-to-many Join operator, which joins each input tuple with the Merchants table. When a tuple contains purchases from multiple merchants, Join generates multiple output tuples. For instance, the tuple t_1 joins with three merchants and produces the tuples t_2, t_3, and t_4. The Split operator splits the stream based on merchant information and sends different tuples to the two merchant fraud-detector operators FMX and FMY. The prediction results are combined by a Union operator.

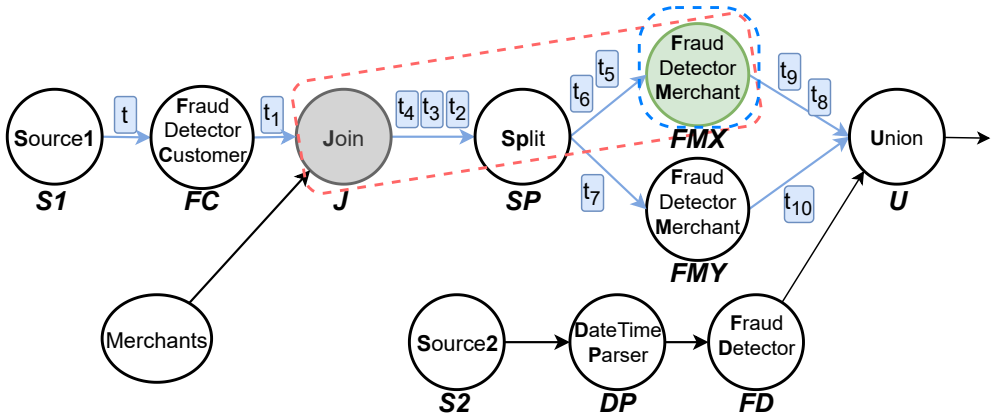


Figure 3.8: Reconfiguration of operator FMX in a dataflow with a one-to-many Join operator. An incorrect MCS generated by Algorithm 2 is highlighted in blue. The correct MCS generated by Algorithm 3 is highlighted in red.

Based on Definition 3.6, the source tuple t has the following data transaction T_5.

$$\Phi \; in \; T_5 : \{\phi(FC, t), \phi(J, t_1), \phi(SP, t_2), \phi(SP, t_3), \phi(SP, t_4),$$
$$\phi(FMX, t_5), \phi(FMX, t_6), \phi(FMY, t_7), \phi(U_1, t_8), \phi(U_1, t_9), \phi(U_1, t_{10})\}.$$

We use an example to show that when reconfiguring a dataflow with one-to-many operators, a naive adoption of the Fries scheduler in Algorithm 2 can produce a non-conflict-serializable schedule. Consider a reconfiguration of operator FMX in Figure 3.8. Algorithm 2 adds the only reconfiguration operator FMX to the set M and computes the MCS with one component, which contains the operator FMX and no other edges. Algorithm 2 ignores the Join operator because it is not in the reconfiguration. The method sends an FCM to FMX. This operator does not propagate the FCM to its downstream operators because it is the only operator in the MCS component. Suppose the FCM sent to operator FMX arrives *after* the tuple t_5 and before the tuple t_6 in the same transaction. Then this scheduler produces the following schedule with a total order of the data operations and the function-update operations:

$$S_5 : [\phi(FC, t), \phi(J, t_1), \phi(SP, t_2), \phi(SP, t_3), \phi(SP, t_4), \boldsymbol{\phi(FMX, t_5)},$$
$$\boldsymbol{\mu(FD_1)}, \boldsymbol{\phi(FMX, t_6)}, \phi(FMY, t_7), \phi(U, t_8), \phi(U, t_9), \phi(U, t_{10})].$$

We can show that the schedule S_5 is not conflict-serializable. Intuitively, as indicated in the operations in bold, tuple t_5 is processed by FMX with the old configuration, and tuple t_6 in the same transaction is processed by FD_2 with the new configuration.

3.6.2 Extending the Fries scheduler

We extend the Fries scheduler Algorithm 2 to produce a conflict-serializable schedule for a dataflow with one-to-many operators and a function-update transaction. Intuitively, for a one-to-many operator, each of its descendant operators could receive multiple input tuples that belong to the same data transaction. In Figure 3.8, operator SP receives three tuples (t_2, t_3, and t_4), and operator FMX receives two tuples (t_5 and t_6) in the same data transaction.

Consider a reconfiguration that includes the operator FD_1. The function-update operation $\mu(FD_1)$ can be conflicting with the data operations of tuples t_5 and t_6 (in the same data transaction) in the same operator. To guarantee a conflict-serializable schedule, these two data operations must synchronize with $\mu(FMX)$ to ensure that both data operations are either before $\mu(FMX)$ or after $\mu(FMX)$. In other words, $\mu(FMX)$ cannot be scheduled in the middle of these two data operations. Notice that the Join operator is the earliest ancestor one-to-many operator of the reconfiguration operator FMX. If an FCM is sent to an operator O after the Join operator, since the operator O could possibly generate multiple data operations for the same data transaction, the FCM can be injected in the middle of these data operations, causing the schedule to be not conflict-serializable. Based on these observations, to guarantee the conflict-serializability, we can start the synchronization from the Join operator using an epoch marker. Recall that the Fries scheduler starts the epoch marker propagation from the head operators of a component in the MCS. The MCS is constructed using a set of operators M, which includes the reconfiguration operator FMX. To make sure the Join operator is treated as a head operator in a component, we add the operator to M before computing the MCS.

Algorithm 3 The Fries Scheduler (for general dataflows with one-to-many operators)

Input: A dataflow $G = (V, E)$
Input: A reconfiguration $\mathcal{R} = \{(o_1, U_1), \ldots, (o_n, U_n)\}$
1: $M = \{o_1, \ldots, o_n\}$
2: **for** each reconfiguration operator o_i in $\{o_1, \ldots, o_n\}$ **do**
3: $\quad \mathcal{A} \leftarrow$ set of ancestor one-to-many operators of o_i
4: $\quad \mathcal{E} \leftarrow computeEarliestAncestors(\mathcal{A})$
5: $\quad M \leftarrow M \cup \mathcal{E}$
6: ...same as Algorithm 2 line 2-6

Algorithm 3 shows the extended Fries scheduler, with the part in the box showing the differences compared to the original Fries scheduler in Algorithm 2. When constructing the MCS, apart from adding the operators in the reconfiguration to M (line 1), we also add to M all the earliest one-to-many ancestor operators of each reconfiguration operator o_i (lines 2

to 5). This step is done by first finding the set of ancestor one-to-many operators of o_i, denoted as $\mathcal{A}$, then finding the earliest ones in $\mathcal{A}$. Notice that a reconfiguration operator could have more than one earliest ancestor one-to-many operator. For example, in Figure 3.8, suppose the operators FMX and FMY are the only one-to-many operators in the dataflow. Then the reconfiguration operator U has both FMX and FMY as its earliest ancestor one-to-many operators according to the partial order of the DAG. We do the modification in the box because we want to start the synchronization from these one-to-many operators with the reconfiguration operators using epoch markers. The remaining steps are the same as in Algorithm 2.

As an example, in Figure 3.8, the only one-to-many operator is the Join operator J. Because the reconfiguration operator FMX's earliest ancestor one-to-many operator is J, we add J to M when constructing the MCS. The resulting MCS includes a single component with operators J, SP, and FMX, together with their edges. The controller injects an FCM to operator J, which propagates an epoch marker within the component to operator FMX.

We show that the extended Fries scheduler still guarantees conflict-serializability of its produced schedule.

Lemma 3.6. *(Corresponding to Lemma 3.4.) Consider a dataflow graph G with a reconfiguration $\mathcal{R}$, and the MCS G' generated by Algorithm 3. Each component of G' contains at least one reconfiguration operator.*

Proof. Let M be the set of operators used in Algorithm 3 to compute the MCS in line 1. Using steps similar to those in Lemma 3.4, we can show each component contains at least one operator in M. By the construction in Algorithm 3, an operator P in M is either (1) a reconfiguration operator, or (2) an earliest one-to-many ancestor operator of a reconfiguration operator O. In the latter case, by the construction in Algorithm 3, O is also in G'. By the definition of components, O is in the same component as P. Therefore, in both cases, each

component of G' contains at least one reconfiguration operator. □

Lemma 3.7. *(Corresponding to Lemma 3.5.) Consider a dataflow graph G with a reconfiguration $\mathcal{R}$, and the MCS G' generated by Algorithm 3. For each source tuple, the operators in its data transaction overlap with at most one component of G'.*

Proof. Suppose the operators processing a tuple t overlap with two components $\mathcal{C}_1$ and $\mathcal{C}_2$ in the MCS. By Lemma 3.6, there is a reconfiguration operator A in $\mathcal{C}_1$ and another reconfiguration B in $\mathcal{C}_2$. Let $(\mathcal{S}, \preccurlyeq_{\mathcal{S}})$ be the scope of t. Let t_A and t_B be two tuples (in $\mathcal{S}$) processed by operators A and B, respectively. Notice that the partial order $\preccurlyeq_{\mathcal{S}}$ of the scope forms a tree. Let t_L be the latest common ancestor tuple of t_A and t_B in the tree. By the definition of the scope $(\mathcal{S}, \preccurlyeq_{\mathcal{S}})$, the receiving operator L of t_L must be a one-to-many operator because there is more than one child of t_L in the tree. Notice that L is a common ancestor of A and B because it has paths to both operators in G. For operator A, by the construction in Algorithm 3, L is either (1) an earliest one-to-many operator of A, or (2) on the path between A and an earliest one-to-many operator of A. In both cases, L is in G'. By the definition of components, since L is in G' and L is connected to both A and B, A and B must be in the same component of G', which contradicts the assumption. □

Lemma 3.8. *Consider a dataflow graph G with a reconfiguration $\mathcal{R}$, and the MCS G' generated by Algorithm 3. A head operator H in a component of G' receives at most one input tuple.*

Proof. Let M be the set of operators used in 3 to compute the MCS. Suppose a head operator H of a component in G' is not in M. We construct a new sub-DAG G'' by removing S and its edges from G'. Using similar steps as in Lemma 3.3, we can show that G'' is still a minimal covering sub-DAG of G and M, which contradicts the minimality property of G'.

By the construction in Algorithm 3, operator H is either (1) a reconfiguration operator with no ancestor one-to-many operators, or (2) an earliest one-to-many operator of a reconfigura-

tion operator, which also has no ancestor one-to-many operators. Therefore, H can receive at most one input tuple in T. □

Theorem 3.2. *(Corresponding to Theorem 3.1.) For a workflow possibly with one-to-many operators and a reconfiguration request, Algorithm 3 always produces a conflict-serializable schedule.*

Proof. Consider the function-update transaction U of a reconfiguration $\mathcal{R}$ in the algorithm and the data transaction T for a source tuple t. Lemma 3.7 shows that the operators in T overlap with at most one component of the MCS G' produced in the algorithm. Consider a head operator H in a component of G'. Lemma 3.8 shows that a head operator H in a component of G' can receive at most one input tuple in T. We compare the position of the epoch marker on H with a possible single input tuple of H to determine the position of T in a serial schedule. We can prove this claim using steps similar to those in the proof of Theorem 3.1. □

3.6.3 Reducing delay by MCS pruning

For dataflows with one-to-many operators, the reconfiguration delay can be long when there are many intermediate operators between the head of an MCS component and a reconfiguration operator in the component. To address this limitation, we improve the Fries scheduler in Algorithm 3 by using pruning rules to remove one-to-many operators that do not need to be synchronized. Algorithm 4 shows the addition of a pruning step. In line 4, we call a function pruneAncestors that applies pruning rules to each of the ancestor one-to-many operators to decide it can be pruned.

Next, we introduce two pruning rules that are used in the improved Fries scheduler.

1. Edge-wise one-to-one pruning rule. Figure 3.9 (I) shows a part of a dataflow with a

Algorithm 4 The Fries Scheduler with a Pruning Process

1: $M = \{o_1, \ldots, o_n\}$
2: **for** each reconfiguration operator o_i in $\{o_1, \ldots, o_n\}$ **do**
3: $\quad \mathcal{A} \leftarrow$ set of ancestor one-to-many operators of o_i
4: $\quad \boldsymbol{pruneAncestors(\mathcal{A})}$
5: $\quad \mathcal{E} \leftarrow computeEarliestAncestors(\mathcal{A})$
6: $\quad M \leftarrow M \cup \mathcal{E}$
7: . . . same as Algorithm 2 line 2-6

Replicate operator, denoted as RE. This operator replicates each input tuple to produce two output tuples and sends each of them to operators C and D. RE is a one-to-many operator by Definition 3.13. Suppose all other operators in this dataflow are one-to-one operators. Using Algorithm 3, the Fries scheduler includes operators RE, C, and E in the MCS, as shown in the red box in Figure 3.9 (I). This is because RE is the earliest one-to-many ancestor operator of the reconfiguration operator E.

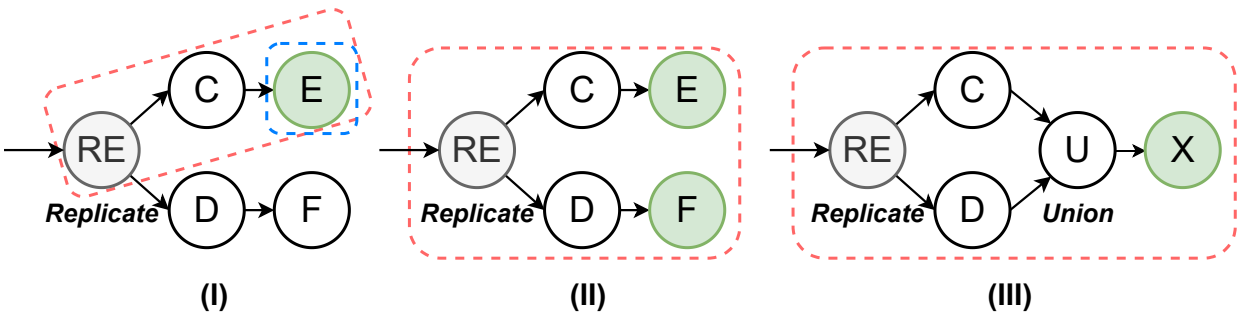


Figure 3.9: Example reconfigurations on dataflows with a replicate operator. (I): The MCS can be pruned. (II) and (III): the MCS's cannot be pruned.

Although operator RE is a one-to-many operator, for an input tuple, the operator outputs a single tuple on each edge. For the reconfiguration operator E, it only receives a single tuple in each data transaction. Therefore, there is no need for operator E to synchronize with operator RE. The MCS only contains operator E, as shown in the blue box in Figure 3.9. Figure 3.9 (II) and (III) show dataflows where the MCS with a replicate operator cannot be pruned. In Figure 3.9 (II), for each tuple processed by operator E, the corresponding replicated tuple must be processed by the same version of operator F. We can achieve

the goal by starting the synchronization from RE. In Figure 3.9 (III), operator X receives all the replicated tuples in each data transaction. Therefore we also need to start the synchronization from the one-to-many operator RE.

Next, we formally describe the pruning rule. We prune an ancestor one-to-many operator A of a reconfiguration operator o_i if the following conditions are true. (1) On each of its output edges, A emits at most one tuple for each input tuple. (2) The A has only one output edge e connected to a downstream reconfiguration operator, and this output edge e is connected to o_i. Intuitively, condition (1) ensures that A behaves like a one-to-one operator on each of its output edges. Condition (2) ensures that the reconfiguration transaction of o_i affects only one output tuple of A sent on edge e. As analyzed in Section 3.6.2, a one-to-many operator O needs to be included in the MCS to ensure multiple output tuples of O are processed using the same configuration. In this case, only a single output tuple of A is affected by the reconfiguration. Therefore, A can be pruned from the set of operators used to construct the MCS.

2. Uniqueness pruning rule.

Next, we show another example of pruning an one-to-many operator. In Figure 3.10, suppose we want to reconfigure operator E. Each input tuple is first replicated by operator RE. The replicated tuples are sent to operators C and D. They are then combined to a single tuple using a Self-Join operator SJ on the primary key. Algorithm 3 computes the sub-DAG from operator RE to operator E as the MCS, as shown in the red box in Figure 3.10. However, notice that operator SJ ensures that it generates at most one output tuple for input tuple from the source. Therefore, RE does not need to be synchronized and the MCS only needs to contain E without RE, as shown in the blue box. In general, we prune an ancestor one-to-many operator A of a reconfiguration operator o_i if on each path from A to o_i, there exists an operator O that has the following uniqueness property: operator O generates at

most one output tuple for each data transaction. In the running example, SJ is such an O operator and RE can be pruned.

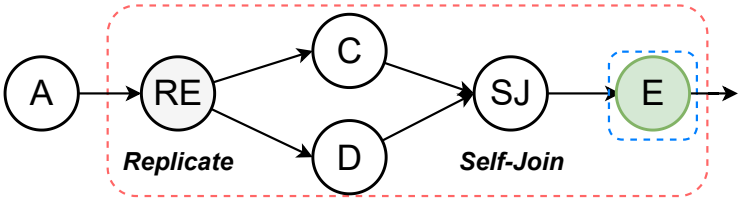


Figure 3.10: Operator RE can be pruned from the set of operators used to construct the MCS.

3.7 Extensions

In this section, we consider how the Fries scheduler works in more general cases, including the case of workflows with blocking operators and the case where an operator has multiple workers. Moreover, we discuss how to support fault tolerance in the Fries scheduler.

3.7.1 Dataflows with Blocking Operators

We now consider how the Fries scheduler works on dataflows containing blocking operators, such as aggregation and sort. Consider a blocking operator B. All operators before B need to run to their completion before the operators after B start to run. In other words, the operators before B and those after B never execute at the same time. Based on this observation, we can use the blocking operators in a dataflow to divide the dataflow into multiple sub-dataflows, with each of them containing pipelined operators only. Then we run Fries on each sub-dataflow during its execution.

3.7.2 Multiple Workers for an Operator

In a parallel execution engine, each operator can have multiple workers, with each worker processing a data partition. We map a single-worker dataflow $G = (V, E)$ to a parallel dataflow $G^* = (V^*, E^*)$, where each operator v in V is mapped to multiple parallel workers $v^1, \ldots, v^p$ in V^*, where p is the number of workers of the operator. We map a reconfiguration $\mathcal{R}$ specified on the single worker dataflow G to a new reconfiguration $\mathcal{R}^*$ on the parallel dataflow G^* of G. Figure 3.11 shows part of a parallel dataflow based on Figure 3.8, where each operator runs using two workers. For each function update $\mu(o_i)$ on an operator o_i in R, we map it to a set of function updates on all the workers of o_i, i.e., $\{(o_i^1, \mu(o_i)), \ldots, (o_i^p, \mu(o_i))\}$ in R^*. For example, the reconfiguration on operator FMX is mapped to a reconfiguration

on the corresponding workers FMX_1 and FMX_2.

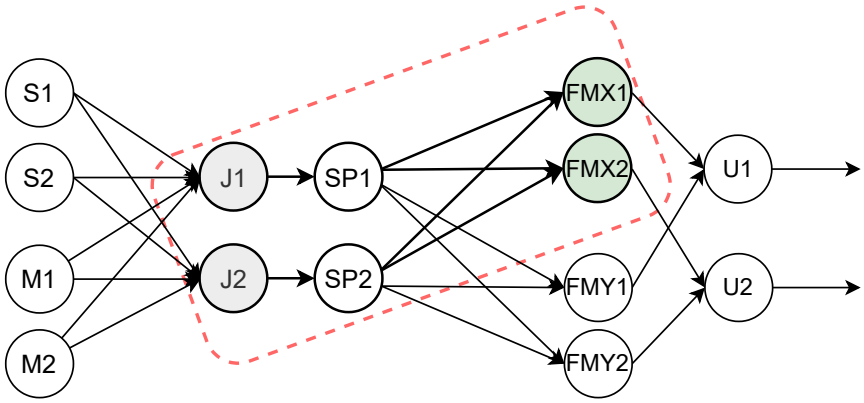


Figure 3.11: A reconfiguration on a parallel dataflow with two workers per operator.

Notice that the parallel dataflow G^* is also a DAG. The Fries scheduler in Algorithm 4 can be directly run on G^* with $\mathcal{R}^*$. The operators and edges in the generated MCS are highlighted in red in Figure 3.8 The Fries scheduler treats a worker of an operator to have the same property (one-to-one or one-to-many) as the operator in hash and range partitioning. For example, both workers of the Join operator are treated as one-to-many operators. When using the broadcast strategy, a worker broadcasts an output tuple to all its downstream operators, same as the Replicate operator described in Section 3.6.3. In this case, the Fries treats it as if a Replicate operator is added after the worker. The pruning techniques described in Section 3.6.3 can still be used.

3.7.3 Fault Tolerance Using the Fries Scheduler

Fault tolerance requires that a system can recover to a consistent state in case of failures. For a dataflow G and a reconfiguration $\mathcal{R}$, the execution of the dataflow is not in a consistent state if some operators in $\mathcal{R}$ are updated, and some operators in $\mathcal{R}$ are not. In an epoch-based scheduler, fault-tolerance can be supported using epoch-based checkpointing [33, 32]. However, such checkpointing cannot guarantee fault tolerance for the Fries scheduler. Consider the reconfiguration in Figure 3.7 and the following sequence of events: (1) G receives

a checkpoint marker from B; (2) G and C receive the reconfiguration FCM's; and (3) C receives a checkpoint marker from A. The checkpoint contains the old configuration of G and the new configuration of C, which is not in a consistent state. Next we discuss two methods to support fault tolerance in Fries.

Checkpoint-based fault tolerance. When a reconfiguration request arrives at the controller, the controller cancels all in-flight checkpoints because they could produce inconsistent states. The controller then blocks any new checkpoints to be started until all head operators of each MCS component have received their FCM's. In this way, the subsequent epoch markers will always be after the FCM's, thus the subsequent checkpoints only contain the fully updated configuration. The blocking period is short because the FCM's are not blocked by any data messages.

Logging-based fault tolerance. The FCMs introduce non-determinism in the execution of an operator. We can log all the non-determinism factors of each operator, including the arrival order of data tuples and the FCMs. During recovery, each operator is deterministically replayed and the FCMs are injected following the original order. We can leverage an existing logging-based fault-tolerance approach such as the one in Clonos [93], which is built on top of Flink. FCMs can be modeled as RPC calls received by an operator in Clonos, which are recorded in the logs.

3.8 Experiments

In this section, we present the results of experiments of different reconfiguration schedulers and show the benefits of Fries.

3.8.1 Setting

Datasets. We used three datasets shown in Table 3.3. Dataset 1 had 24M tuples of credit card payments with 12 attributes [85], such as the customer, merchant, date, amount, and chip usage. Dataset 2 was constructed by grouping the credit card payments per user in dataset 1. Each record had a user and a list of payments by the user. We used this dataset to utilize a one-to-many unnest operator to split a payment list into multiple records. Dataset 2 was generated using the TPC-DS benchmark [104] with a scale factor of 100.

Workflows. We constructed workflows as shown in Figure 3.12. Workflow W_1 simulated a fraud detection application, and it detected fraud of a user based on the user's historical payment amounts. By default, the source operator read the payment table with a rate of 1,000 tuple/s. The user-based inference operator saved 10 most recent payment amounts for each user as its internal state. For each input tuple, the operator updated the user's state and used an LSTM auto-encoder [110] to predict the probability of fraud. Workflow W_2 was constructed based on TPC-DS query 40. For all items with a price between 0.99 and 1.49, this workflow computed the item id and location of the warehouse the item was delivered from in a 60-day period. Workflow W_3 was constructed based on TPC-DS query 71. It produced the brands managed by a given manager that sold their products across three sales channels at either breakfast or dinner time for a given month. All the join operators in these workflows were one-to-one operators because they join a primary key with a foreign key. We only considered the pipelined sub-DAG of each dataflow. For example, if a hash join has a

build phase and a probe phase, we only consider the pipelined probe phase. In Figure 3.12, we highlighted all the pipelined edges considered in the experiment in red.

On top of W_1, workflow W_4 included an additional merchant-based inference operator for users who made a large amount of payments. For each merchant, the inference operator saved 50 most recent payments, and used a similar LSTM auto-encoder to do inference. A payment record was processed by both inference operators. If one of them produced a probability greater than a threshold, the payment record was flagged as fraud. Workflow W_5 replicates the payment tuples to both the user-based inference operator and the merchant-based operator. After each operator makes fraud predictions, a Self-Join operator is used to combine the replicated tuples into a single one.

Dataset	Table	Attribute #	Tuple #
1	Credit card payment	12	24M
2	Credit card payment aggregated per user	2	20K
3	Catalog sales	34	144M
	Store sales	23	288M
	Web sales	34	71M

Table 3.3: Datasets used in the experiments.

Reconfigurations. For workflow W_1, we performed configurations with one operator. For workflows W_2, W_3, and W_4, we performed reconfigurations with multiple operators. The methods of choosing reconfiguration operators will be described in each experiment.

Schedulers. We implemented two epoch-based schedulers. The first one performed a reconfiguration with a savepoint, which was natively supported by Flink (described in Section 3.3). The second one was the scheduler of Chi [73] (described in Section 3.3). As Chi was not open source, we implemented this scheduler on top of Flink, and used Flink's aligned checkpoint barriers as epoch markers. The first scheduler always stopped and restarted the execution *after* the propagation of checkpoint barriers to apply reconfiguration. The second scheduler applied reconfiguration *during* the barrier propagation, and did not require an additional

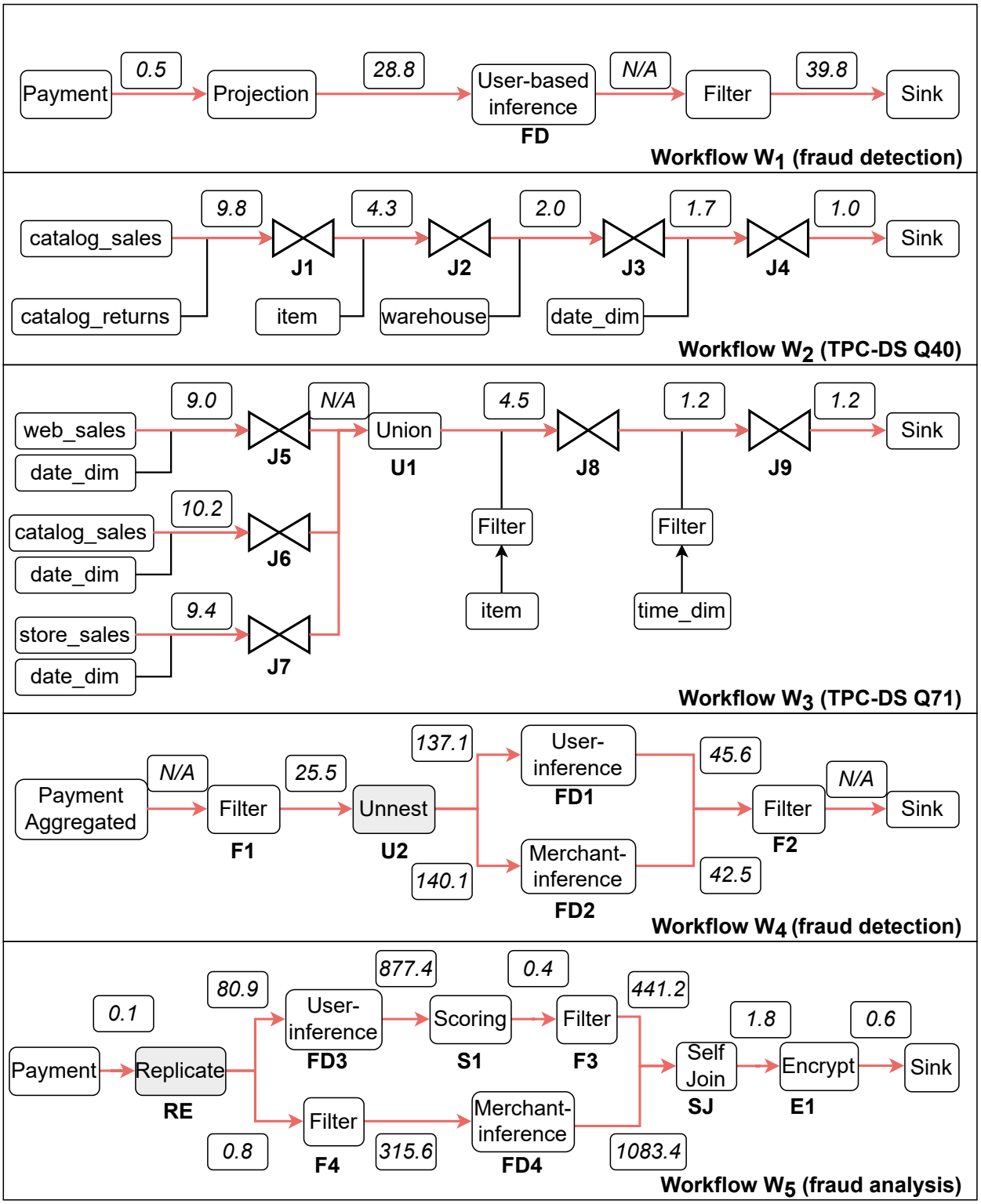


Figure 3.12: Workflows used in the experiments. Pipelined edges are highlighted in red.

stop-and-restart of the system. As a consequence, the second scheduler always had a shorter reconfiguration delay than the first, as verified in our experiments. Therefore, between these two schedulers, we only report the results of the second, denoted as "Epoch scheduler."

For fair-comparison purposes, we implemented Fries also on top of Flink. In the implementation, FCM's between the controller and a specific worker of an operator were sent in special network channels (available in Flink). For each MCS C computed in Fries for a reconfiguration, the controller sent FCM's to the workers of C's head operators. These workers pushed checkpoint barriers to the workers of their downstream operators in C. To let every operator know which downstream operators were in C, the checkpoint barrier also included C and the reconfiguration operators in C. Every reconfiguration operator in C applied the reconfiguration after receiving checkpoint barriers from all its upstream operators in C. The reconfiguration for this MCS C completed after all C's reconfiguration operators applied the reconfiguration.

System environment. All the experiments were conducted on the Google Cloud Platform (GCP). The execution was on a GCP dataproc cluster with 1 coordinator machine and 10 worker machines. All the machines were of type n1-highmem-4 with Ubuntu 18.04. The job controller of Flink ran on the coordinator. The coordinator machine had a 2TB HDD, while each worker machine had a 250GB HDD. To separate computation and storage, we stored the datasets in an HDFS file system on another cluster with 6 e2-highmem-4 machines, each with 4 vCPU's, 32 GB memory, and a 500GB HDD. For all the schedulers, we used Flink release 1.13 and Java 8.

3.8.2 Choke Point Analysis of Workflows

In the execution there were various choke points in the workflow where the reconfiguration delay between two operators was very high. We analyzed these choke points in the experiment

workflows by computing the average reconfiguration delay between two operators using the epoch scheduler and showed the numbers on top of each edge in Figure 3.12. The numbers represented the delay from the time when the upstream operator applied the reconfiguration and sent checkpoint barriers to the time when the downstream operator aligned all the checkpoint barriers and applied the reconfiguration. Some edges are marked as N/A because the two connected operators were fused to a single operator chain in Flink. Edges marked with a number perform re-partition operations, thus the two connected operators are not chained.

We had the following observations. 1) Expensive operators usually created choke points in the workflow. For example, in $W4$, both inference operators applied the reconfiguration from $U1$ after the checkpoint barriers were sent out for around 140s. The inference operators accumulated input tuples in their input data channel, which blocked the checkpoint barrier to be processed. 2) Stragglers also created choke points. For example, in $W5$, there was a delay of 877.4s between $FD3$ and $S1$, because one of the $FD3$ workers was a straggler. Recall that due to the epoch alignment step, $S1$ had to receive all the checkpoint barriers before applying the reconfiguration. $S1$ was blocked when waiting for the straggler $FD3$ worker to finish. 3) If operators had similar costs, choke points depended on the amount of data in each operator's input data channel. For example, in both $W2$ and $W3$, the first several joins had larger delays of reconfiguration. This is because the data was filtered by every join, and the joins near the sink received less data so they had a lower reconfiguration delay.

3.8.3 Benefits of Short Reconfiguration Delay: Reducing End-to-end Tuple Latency

A main advantage of Fries was its short reconfiguration delay compared to epoch-based schedulers. To show the benefits of this advantage, we considered a scenario for W_1 as shown in Figure 3.13, where the developer needed to hot-replace the model in the user-based inference operator FD during the execution to deal with a sudden surge of input data. In W_1, we set the number of workers for operators (except for the source and sink) to 40 to utilize all the cores in the cluster. The time for FD to process a tuple was about 25ms, and the maximum throughput of this operator was around $1,600$ tuple/s. We used a single worker of the source operator and another single worker of the sink operator on the same machine so that they can use the same clock. The source operator started with an initial ingestion rate of $1,000$ tuples/s. After 100 seconds, we increased the ingestion rate to $2,000$ tuples/s. The developer saw an increasing trend of the end-to-end tuple latency. He requested a reconfiguration to replace the original LSTM auto-encoder model in FD with another LSTM auto-encoder with fewer parameters at $t = 120s$ to speed up the processing. The developer continuously monitored the end-to-end tuple latency. After another 100 seconds, we further increased the rate to $9,000$ tuples/s. The developer decided to further decrease the cost of FD to reduce the latency. At $t = 220s$, he requested another reconfiguration to replace the LSTM auto-encoder model with a simple decision-tree model.

To compute the end-to-end latency of each input tuple, we attached a timestamp at the moment when the source operator generated this tuple. When the tuple was received by the sink operator, the latency was computed as the difference between the current time and the attached timestamp. Figure 3.13 shows the average end-to-end latency of output tuples received for every 10-second sliding window. (1) For the case without reconfiguration, after 100 seconds, the end-to-end latency began to increase because FD was not fast enough to process all the incoming tuples. Many tuples were buffered in the network channel. The

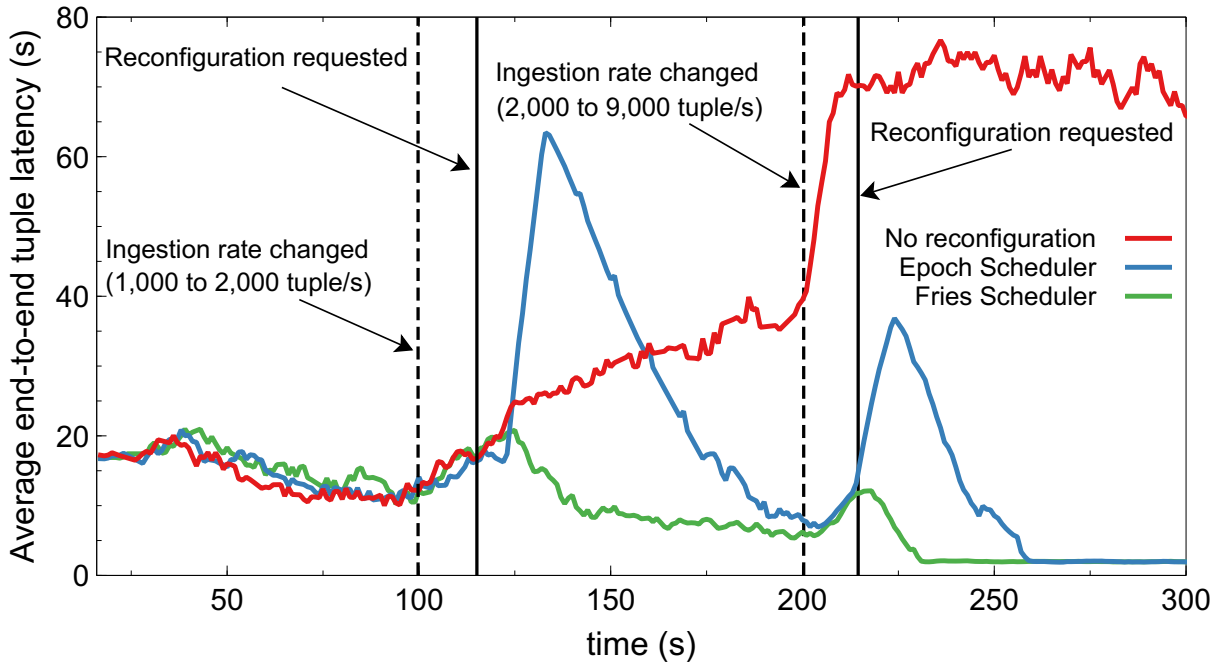


Figure 3.13: Effect of mitigating surges of data-ingestion rate by different schedulers (W_1 on dataset 1).

latency increased continuously and stabilized at around 70s when backpressure from FD was propagated to the source operator to slow down the data ingestion rate. (2) For the case of using the Epoch scheduler, the latency rapidly increased to above 60 seconds until around $t = 135s$ due to the surge. The main reason for the increase in the end-to-end latency was the blocking in the epoch alignment step. Since the sink operator had only one worker, it needed to wait until all 40 upstream FD workers completely processed all tuples in the old epoch before it could process any tuples in the new epoch. Note that the delay was determined by the slowest FD worker. We observed that the average delay of all workers was 30 seconds. However, there were two straggler workers that took 58 seconds and 69 seconds to finish processing the old epoch, respectively. The two straggler workers suffered from data skew. On average, each worker processed 35,000 tuples in the old epoch. However, the slowest worker processed 62,000 tuples.

(3) For the case of using the Fries scheduler, the latency immediately decreased after $t = 120s$, indicating that FD applied the reconfiguration and was able to quickly process the

buffered tuples. Compared to Epoch, Fries required less time to mitigate the surge. In this reconfiguration, the MCS component contained operator FD only. Therefore, FCMs are directly sent to all FD workers and no epoch markers were propagated to any downstream operators. This eliminated the aforementioned delay caused by the epoch alignment step.

3.8.4 Benefits of Short Reconfiguration Delay: Reducing Wasted Computing Resources

Another benefit of a short reconfiguration delay can be illustrated in the following scenario. When processing data with unexpected content or formats, the workflow can produce invalid output tuples to be collected and reprocessed. A large number of invalid output tuples not only wastes computation in the current execution, but also requires more resources in the future. A short reconfiguration delay can effectively reduce the amount of wasted computing resources. To illustrate the benefit, we considered workflow W_1, and attached a version number $V1$ to every source tuple. The user-based inference operator FD had another version number $V2$ for its processing logic. For every input tuple, FD expected $V1$ of the tuple to match with $V2$; otherwise, the operator produced an invalid output tuple. For every 50 seconds, we increased $V1$, and the developer realized the version changed 20 seconds after that. Then, he requested a reconfiguration of FD to also increase its version $V2$. We measured the number of invalid output tuples produced by the workflow over time under different reconfiguration schedulers as a metric of wasted computing resources.

Figure 3.14 showed the result. (1) For the case without reconfiguration, all the output tuples after the first input version update were invalid. Thus the number of invalid tuples increased continuously. After 300 seconds, the number of invalid tuples reached 302K. (2) For the case of using the Epoch scheduler, operator FD had to process all the tuples prior to the epoch marker before applying the reconfiguration. So all the tuples after the input version update

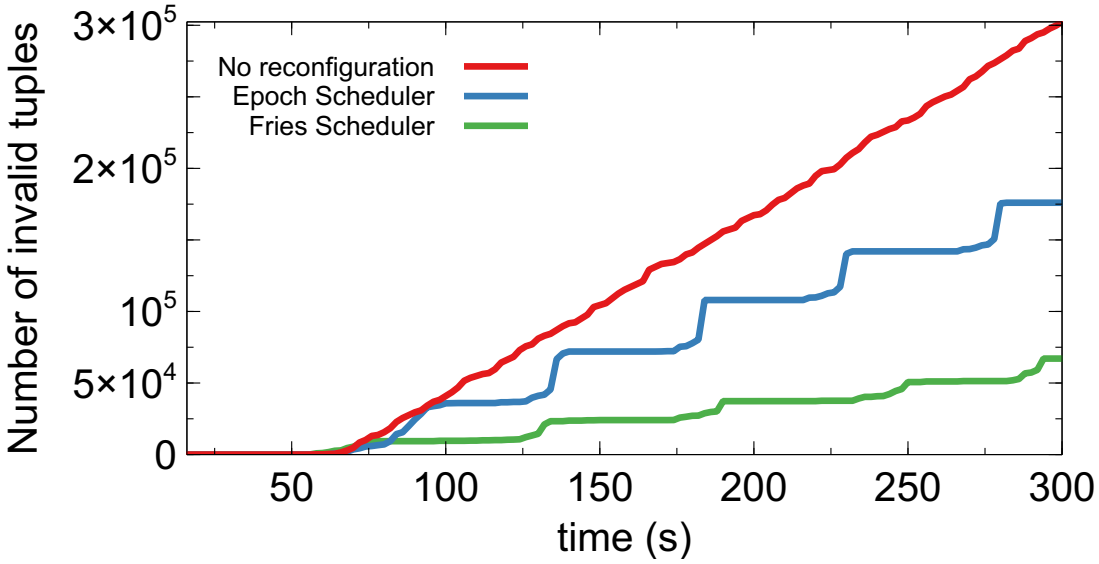


Figure 3.14: Effect of schedulers on the number of invalid output tuples (W_1 on dataset 1).

and prior to the marker were invalid. The workflow produced many invalid tuples after each input version update, resulting in reprocessing of 176K tuples after 300 seconds.(3) For the case of using the Fries scheduler, the operator quickly applied the reconfiguration, so fewer invalid tuples were generated for each input version change. At the end, only 67K tuples needed to be reprocessed.

3.8.5 Effect of Data Ingestion Rates on Reconfiguration Delays

Next we evaluated the effect of different factors on the delay. We first considered data-ingestion rate. For workflow W_1, we gradually increased the rate at the source operator from 500 tuples/s to $2,500$ tuples/s. For each configuration, after the execution of 120 seconds, we applied a dummy reconfiguration on operator FD and measured the delay under the two schedulers. As shown in Figure 3.15 (with a log scale for the y-axis), when the ingestion rate increased, the delay of the Epoch scheduler also increased due to the larger amount of in-flight tuples prior to the epoch marker. Since the Fries scheduler sent FCM's directly to FD, its delay grew much slower than the Epoch scheduler.

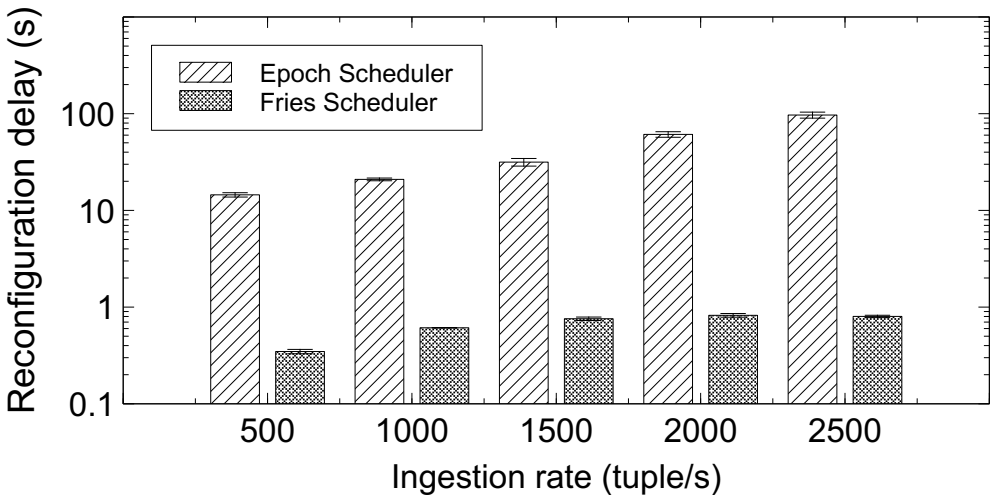


Figure 3.15: Effect of data ingestion rate on the reconfiguration delay (with 95% confidence intervals) (W_1 on dataset 1).

3.8.6 Effect of Operator Costs on Delays

To evaluate the effect of operator cost on the reconfiguration delay, for workflow W_1, we gradually increased the cost of the user-based inference operator FD to process each input tuple. The FD operator maintained a bounded queue of recent payment amounts of each user. When an input tuple was received by FD, the operator passed the payment amounts in the queue to its ML model. In different runs of experiments, we gradually increased the size of this queue from 10 to 50 so that the operator took more time to process each input tuple. Again, for each configuration, after the execution ran for 120 seconds, we applied a dummy reconfiguration on FD and measured the delay under the two schedulers. As shown in Figure 3.16, when the FD's cost increased, the delay of the Epoch scheduler also increased because each in-flight data tuple prior to the epoch marker took more time to be processed. On the other hand, the delay of the Fries scheduler grew much slower than the Epoch scheduler.

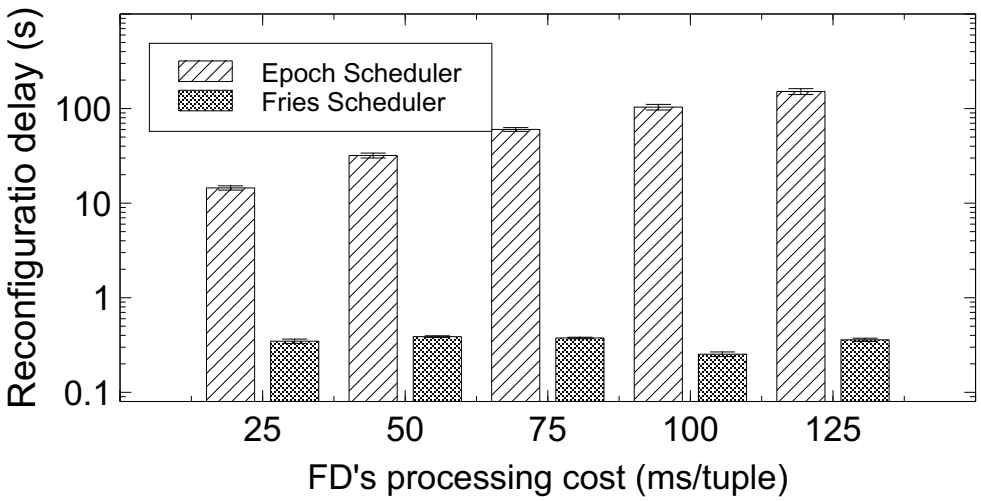


Figure 3.16: Effect of operator cost on the reconfiguration delay with 95% confidence intervals (W_1 on dataset 1).

3.8.7 Effect of Reconfigurations on Delays

We wanted to evaluate the effect of reconfigurations on the delay under the two schedulers. We varied the number of reconfiguration operators in both workflows W_2 and W_3. For both workflows, we used 40 workers for each operator. For every 10 seconds, we requested a reconfiguration and measured the average reconfiguration delay. The results are shown in Table 3.4. For each reconfiguration, we show its operators, the MCS components generated by the Fries scheduler, the length of a longest path of each component, the delay of using the Fries scheduler, and the delay of using the Epoch scheduler. We reported the path length because it affected the delay in the Fries scheduler.

We have the following observations from the results. (1) The delay of the Fries scheduler was always significantly lower than the delay of the Epoch scheduler. For example, for the W_2 configuration including $J1$ and $J4$, the delay of the Fries scheduler was $1,702$ms, compared to $12,361$ms of the Epoch scheduler. (2) For reconfigurations with multiple operators, if each operator formed a component in MCS, the delay of Fries was very low. For example, for the W_3 reconfiguration of $J5$ and $J6$, each of them formed their own component. The Fries

scheduler had a delay of 127ms, comparable to 87ms in the case with $J5$ as the only reconfiguration operator. This low delay was because the Fries scheduler sent FCM's separately to both operators and their reconfiguration happened in parallel. (3) When the length of the longest path in a component increased, as expected, the delay of the Fries scheduler also increased. For example, for the reconfiguration of $J1$ and $J3$, the longest path in their MCS had a length of 2, and the delay was $1,664$ms. For the reconfiguration of $J1$ and $J4$, the longest path in their MCS had a length of 3, and the delay increased to $1,702$ms.

Workflow	Reconfiguration operators	MCS components	Longest path length	Fries Scheduler delay (ms)	Epoch Scheduler delay (ms)
W_2	J1	{**J1**}	0	46	11,432
	J2	{**J2**}	0	44	11,709
	J1, J3	{**J1**, J2, J3}	2	1,664	12,339
	J1, J4	{**J1**, J2, J3, J4}	3	1,702	12,361
	J3, J4	{**J3**, J4}	1	387	13,767
W_3	J5	{**J5**}	0	87	4,127
	J5, J6	{**J5**} {**J6**}	0 0	127	8,352
	J5, J6, J7, J8	{**J5**, **J6**, **J7**, U1, J8}	3	447	19,608
	J5, J6, J7, J9	{**J5**, **J6**, **J7**, U1, J8, J9}	4	526	19,717
	J7, J8, J9	{**J7**, U1, J8, J9}	3	1,340	20,532

Table 3.4: Reconfiguration operators, corresponding MCS, and reconfiguration delay in W_2 and W_3 on dataset 3. Head operators in each component are highlighted in bold.

3.8.8 Reconfiguration Delays in Workflows with One-to-many Operators

We used workflow W_4 to evaluate the effect of reconfiguration operators on the reconfiguration delay in workflows with a one-to-many operator $U2$. This operator split all the payments of a user and sent them to both $FD1$ and $FD2$. Table 3.5 shows the results. We have the following observations. (1) The delay of the Fries scheduler was still always lower than the Epoch scheduler. (2) The reconfiguration of $FD1$ took a long time (47,892ms) in

Fries because $FD1$ was not the head operator of its component. The epoch markers had to go through the data channels of $FD1$ (from multiple workers). Since $FD1$ processed tuples slowly, many of its input tuples were buffered in its data channels, which delayed the propagation of the epoch markers. (3) The reconfiguration of $F2$ took a long delay ($221,353$ms) in Fries because its generated MCS contained every operator on the path from $U2$ and $F2$ with the one-to-many $U2$ operator and both $FD1$ and $FD2$ were slow.

Reconfiguration operators	MCS components	Longest path length	Fries Scheduler delay (ms)	Epoch Scheduler delay (ms)
F1, U2	{**F1**, U2}	1	69	151
FD1	{**U2**, FD1}	1	47,892	131,103
F2	{**U2**, FD1, FD2, F2}	5	221,353	236,153

Table 3.5: Reconfiguration operators, corresponding MCS, and reconfiguration delay in W_4 on dataset 2. Head operators in each component are highlighted in bold.

3.8.9 Effect of MCS Pruning on Delays in Workflows with One-to-many Operators

We used workflow W_5 to evaluate the effect of the MCS pruning method proposed in Section 3.6.3 on the reconfiguration delay in workflows with a one-to-many Replicate operator and a Self Join operator. For each reconfiguration, we compare the Fries scheduler with the pruning step turned on and turned off. Table 3.6 shows the results. We have the following observations. (1) In general, when pruning is possible, the size of MCS components was reduced and the delay with pruning was significantly lower than the delay without pruning. For example, the reconfiguration of operator $FD4$ and the reconfiguration of operator $F3$ benefited from the edge-wise one-to-one pruning rule. (2) In the case of reconfiguring both $FD3$ and $FD4$, the pruning rules could not prune the one-to-many Replicate operator. Therefore the delays were similar. (3) The reconfiguration of operator $E1$ benefited from the uniqueness pruning rule. This reconfiguration had the largest benefit in delay because

Reconfiguration operators	MCS with pruning	MCS without pruning	Fries with pruning delay (ms)	Fries without pruning delay (ms)
FD4	{**FD4**}	{**RE**, F4, FD4}	158	450,149
F3	{**F3**}	{**RE**, FD3, S1, F3}	94	383,781
F4	{**F4**}	{**RE**, F4}	10	446
FD3, FD4	{**RE**, FD3, F4, FD4}	{**RE**, FD3, F4, FD4}	661,892	663,460
E1	{**E1**}	{**RE**, FD3, S1, F3, F4, FD4, SJ, E1}	85	1,122,686

Table 3.6: The effect of MCS pruning on delays in W_5.

the number of edges in the MCS reduced from eight to zero, which greatly reduced the synchronization time.

3.8.10 Effect of Multiple Workers on Delays

To evaluate the effect of the worker number per operator on the reconfiguration delay, we considered workflow W_2 and increased the worker number per operator from 1 to 40. After the workflow ran for 20 seconds, we requested a dummy reconfiguration of $J1$ and $J4$. We measured the reconfiguration delay of the two schedulers.

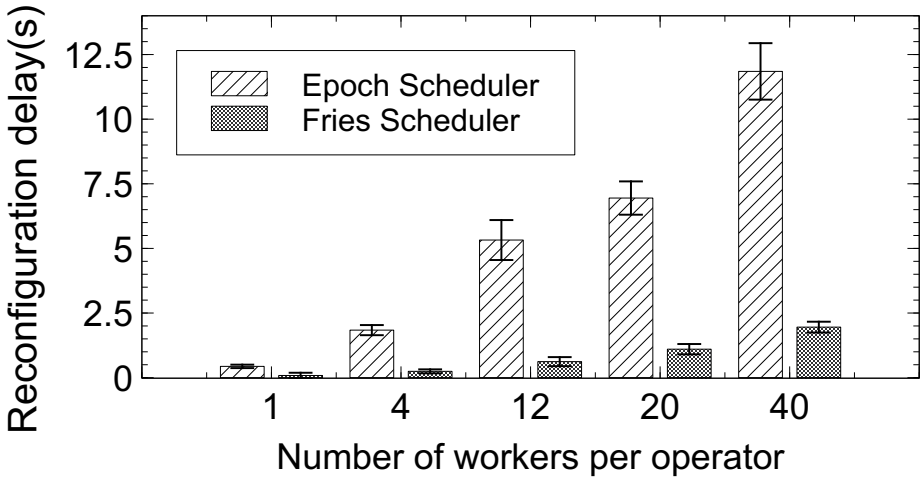


Figure 3.17: Effect of worker number on reconfiguration delay with 95% confidence intervals (W_2 on dataset 3).

As shown in Figure 3.17, as the worker number increased, the delay increased for both

schedulers. This was because between each pair of join operators, the data was shuffled and every join worker needed to receive an epoch marker from all its upstream workers. So the number of data channels between the joins was the product of their numbers of workers. When each worker number increased, the number of epoch markers to collect also increased. The fact that the delay of Fries scheduler was again lower than the Epoch scheduler can be explained using Table 3.7. In particular, the Fries scheduler propagated epoch markers only through the data channels between MCS workers, while the Epoch scheduler had to propagate epoch markers through all the data channels. The table shows that the number of channels between MCS workers was always less than the number of channels between all workers.

Worker # per operator	Total # of data channels between all workers	Total # of data channels between MCS workers
1	5	3
4	68	48
12	588	432
20	1,620	1,200
40	6,440	4,800

Table 3.7: Effect of number of workers on data channels for reconfiguration of $J1$ and $J4$ (W_2 on dataset 3).

3.9 Conclusions

In this paper we studied the problem of runtime configurations in data-intensive workflow systems with a low delay. We showed limitations of existing epoch-based reconfiguration schedulers on the delay. We developed a new technique called Fries that uses fast control messages to do reconfigurations. We formally defined consistency in runtime reconfigurations, and developed a Fries scheduler with consistency guarantee. The technique also works for parallel executions and supports fault tolerance. Our experimental evaluation showed the advantages of this technique compared to epoch-based schedulers.

Chapter 4

Tempura: a General Cost-based Optimizer Framework for Incremental Data Processing

4.1 Introduction

Data analytics often involves handling large datasets and complex queries. Processing such massive amounts of data traditionally could be time-consuming, leaving users without feedback on results until the very end. This lack of real-time visibility can impede user experience, as users cannot view results, identify workflow issues, or implement quick fixes and iterations to improve the data workflows. As a result, it is crucial for the system to provide progressive updates and allow users to observe and interact with the operators during execution [70]. Incremental computation is a technique that can be used to allow systems to deliver preliminary results swiftly based on a subset of the data, refining them over time as more data gets processed. These incremental updates enable users to inspect the results and monitor the workflow's progress in real time, potentially expediting insights and enhancing efficiency.

New advancements in big data systems make data ingestion more real-time and analysis increasingly time sensitive, which boost the adoption of the incremental processing model. Beyond the use case of interactive data analytics, incremental processing is widely used in many other scenarios. Another emerging application that relies on incremental processing is Progressive Data Warehouse [108]. Enterprise data warehouses usually have a large amount of data that need to be collected, enriched, and analyzed by various business logic. For example, at a large corporation such as Alibaba, daily report queries are scheduled after 12 am when the previous day's data has been fully collected, and the results must be delivered by 6 am sharp before the bill-settlement time. Some routine analysis jobs are handled using batch processing, causing dreadful "rush hour" scheduling patterns. This approach puts pressure on resources during traffic hours, and leaves the resources over-provisioned and wasted during the off-traffic hours. Incremental processing can answer routine analysis jobs progressively as data gets ingested, and its scheduling flexibility can be used to smoothen the resource skew. Incremental processing can be adapted in several ways. One approach involves running jobs periodically [106]. Alternatively, raw data can be processed incrementally as it

arrives in real-time using streaming systems such as Apache Flink [35]. Another strategy is to process data just-in-time in response to interactive queries, as demonstrated in JENNER [50] and EnrichDB [49].

Incremental processing has also been adopted in various application domains such as database incremental view maintenance (IVM) [40, 56, 67, 14], stream processing [21, 12, 48, 80, 100], active databases [15], resumable query execution [36], approximate query processing [38, 117, 60], etc. **A key problem behind these applications is how to generate an efficient incremental plan for a query while satisfying the unique requirements of different applications**. We can identify these distinctions primarily through the expected query results to be delivered in each incremental run, as well as the frequency at which updates are expected. Figure 4.1 illustrates the different frequency requirements for incremental updates in the context of streaming applications, interactive data analytics (Texera), and progressive data warehouses. We delve further into these distinctions in the following analysis.

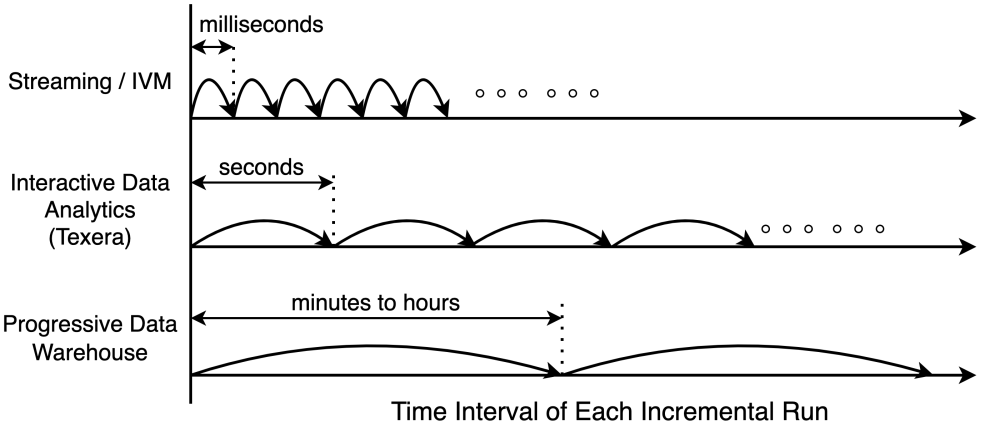


Figure 4.1: A comparison of different incremental update frequency requirements in different applications.

In the context of stream processing, the streaming query results need to be updated and delivered in real-time, whenever new data arrives. The on-demand nature of stream processing requires immediate result update, ideally on a millisecond scale. In the context of incremental view maintenance, the system must promptly update the view as per the view

definition whenever alterations occur in a base table. In terms of update frequency, the view maintenance system responds rapidly to any changes, keeping the view as up-to-date as possible.

In contrast, in interactive data analytics, all data is readily available at the beginning. A data processing system such as Texera [70] can compute and deliver any part of the result to the user progressively. However, the output from each run also needs to cater to the user's practical needs. For example, the output must be substantial enough to provide meaningful information for the user, avoiding too few results that may not be useful. In terms of update frequency, a data analyst interacting with a workflow in this context usually requires updates at a slower pace, typically on a scale of seconds rather than milliseconds. This ensures that the user has ample time to interpret each new set of results, without being overwhelmed.

Progressive data warehouses have another different set of requirements. In terms of results, the warehouse may compute only a part of a query and may not need to produce output after each computation. This is attributed to the business logic, which primarily demands the final results by the specified deadline, with no requirement for any intermediate outcomes. In terms of update frequency, the data is ingested into the warehouse on a much coarser time scale, often ranging from minutes to even hours. Similarly, incremental queries also run at a slower pace, such as every hour.

As different applications have unique requirements about the expected results and the frequency of incremental updates, it is challenging to generate an efficient incremental plan that can adapt to such varied requirements. Note that there exists a plethora of incremental computation algorithms. The choice among them is not a simple one-size-fits-all decision and no algorithm is always optimal, since the optimal plan is *data dependent and application dependent*. As an example, consider a routine analysis job as described in Example 4.1, which reports the gross revenue by consolidating sales orders with returned ones. Figure 4.2 illustrates the workflow of this job.

Example 4.1 (Reporting consolidated revenue)**.**

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

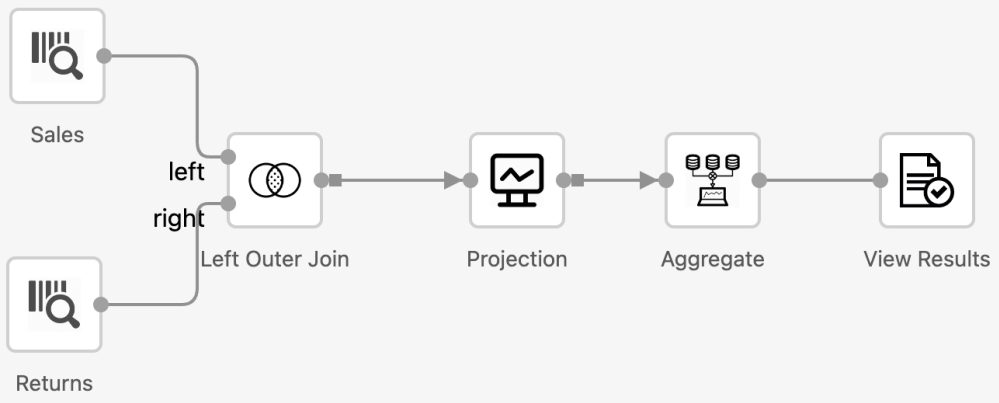


Figure 4.2: An example workflow to calculate gross revenue by joining data from sales and returned orders.

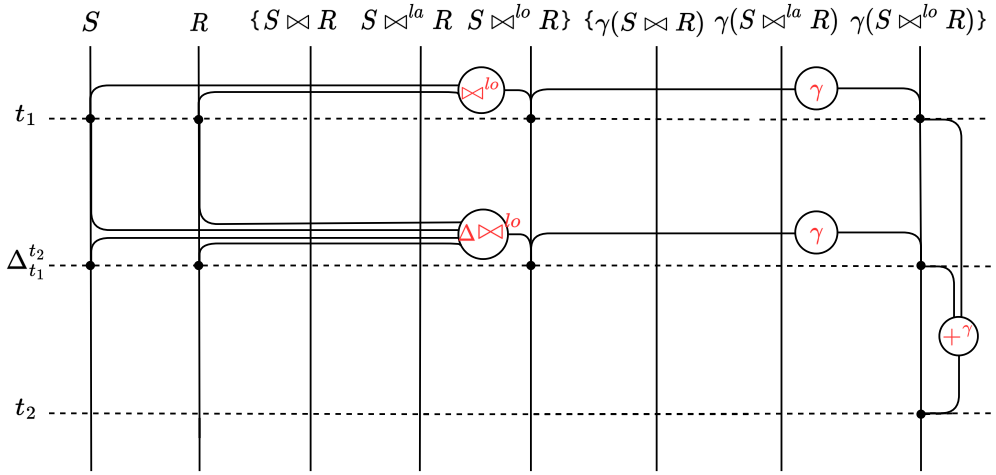
This query can be incrementally computed in different ways as the data in tables sales and returns becomes available gradually. Consider two basic methods used in IVM and stream computing. (1) A typical view maintenance approach (denoted as **IM-1**) treats summary as views [40, 55, 56, 117]. It always maintains summary as if it is directly computed from the data of sales and returns seen so far. Therefore, even if a sales order will be returned in the future, its revenue is counted into the gross revenue temporarily. Figure 4.3(a) shows the execution plan using this approach. In Figure 4.3, each vertical line represents a relation that evolves over time and each horizontal line represents a specific time point. At time t_1, the plan computes the left-outer join and aggregation results. After the new data from t_1 to t_2 arrive, the query plan computes the incremental left-outer join and aggregation results. Finally, the results at t_1 and the incremental results from t_1 to t_2 of the aggregation are

combined to produce the final results.

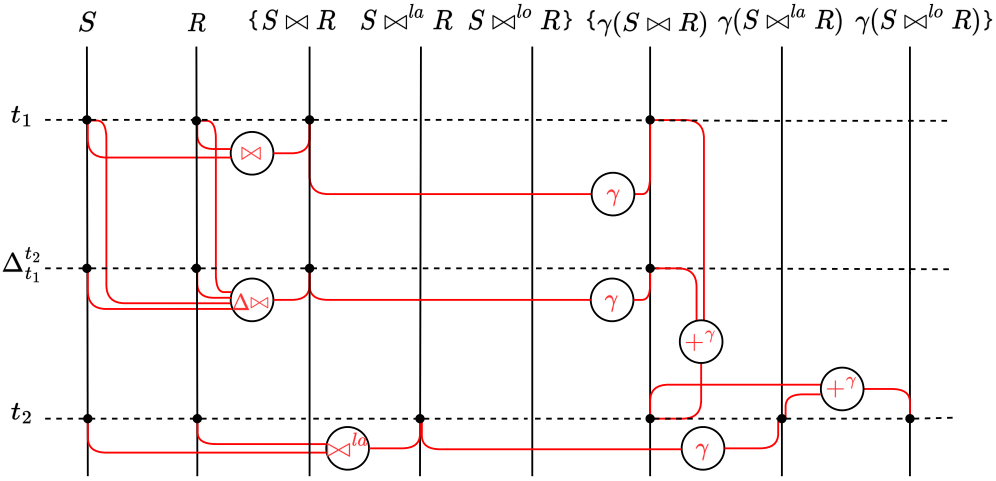
(2) A typical stream-computing method (denoted as **IM-2**) avoids such retraction [59, 68, 74, 99]. It holds back sales orders that do not join with any returns orders until all data is available. Figure 4.3(b) shows the execution plan using this approach. At t_1, the plan computes only the aggregation results of an inner join instead of a left-outer join. After the data from t_1 to t_2 arrive, the plan incrementally updates the aggregation results of the inner join. Next, the complete aggregation results of a left-anti join are computed using all input data up to t_2. Finally, the results of inner join aggregation and left-anti join aggregation are combined to produce the final results.

In the progressive data warehouse scenario, if returned orders are rare, `IM-1` can maximize the amount of early computation and thus deliver better resource-usage plans. Otherwise, if there are many returned orders, `IM-2` can avoid unnecessary re-computation caused by retraction and thus be better. (See Section 4.2.2 for a detailed discussion.) In the interactive data analytics scenario, if returned orders are rare, `IM-2` holds the output results and does not output any retractions to the user. This approach might yield limited meaningful information in each incremental output, making `IM-1` a more preferred option. Otherwise, if returned orders are common, `IM-2` is able to consistently provide substantial results in each incremental run and can still be more efficient than `IM-1`. This analysis shows that different data statistics and application requirements can lead to different preferred methods.

Since the optimal plan for a query given a user-specified optimization goal is data and application dependent, a natural question is how to develop a principled cost-based optimization framework to support efficient incremental processing. To the best of our knowledge and also to our surprise, no such a framework in the literature has been explored. In particular, existing solutions still rely on users to empirically choose from individual incremental techniques, and it is not easy to combine the advantages of different techniques and find the plan that is truly cost optimal. When developing this framework, we face more challenges compared to



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



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

Figure 4.3: Comparison of incremental query plans produced by approach `IM-1` and approach `IM-2`. The symbol $\bowtie^{lo}$ refers to left outer join and the symbol $\bowtie^{la}$ refers to left anti join.

traditional query optimization [54, 94] (see Section 4.2.2): (1) Incremental query planning requires tradeoff analysis on more dimensions than traditional query planning, such as different incremental computation methods, data arrival patterns, which states to materialize, etc. (2) The plans for different incremental runs are correlated and may affect each other's optimal choices. It is essential to jointly consider the runs across the entire timeline.

In this thesis we propose a unified cost-based query optimization framework, which allows users to express and integrate various incremental computation techniques and provides a turn-key solution to decide optimal incremental execution plans subject to various objectives. We make the following contributions.

- We propose a new theory called the *TIP model* on top of time-varying relation (TVR) that formulates incremental processing using TVR. The TIP model describes a formal algebra for TVRs, which includes a definition of TVRs on top of relations, semantics of querying on TVRs, and basic operations on TVRs such as TVR difference and merge operations (Section 4.3). This model serves as a theoretical foundation of our optimization framework.
- We provide a rewrite-rule framework under the TIP model to describe different incremental computation techniques, and unify them to explore in a single search space for an optimal incremental plan (Section 4.4). This framework allows these techniques to work cooperatively, and enables cost-based search among possible plans.
- We build a Cascade-style optimizer named Tempura. It supports cost-based optimization for incremental query planning based on the TIP model. We discuss how to explore the plan space (Section 4.5) and search for an optimal incremental plan (Section 4.6).
- We give a detailed specification on how to integrate Tempura into a traditional optimizer (Section 4.7).
- We propose multiple techniques to improve the query planning speed, such as template copying (Section 4.8.1), plan space pruning (Section 4.8.2), and optimizations of the rule engine (Section 4.8.3).

- We discuss how to practically integrate Tempura into existing systems, particularly in the cases of inaccurate data statistics estimation in (Section 4.9). We elaborate how Tempura can initially conduct several incremental runs for statistics collection and then perform dynamic re-optimization of incremental plans.
- We conduct a thorough experimental evaluation of the Tempura optimizer in various application scenarios. The results show the effectiveness and efficiency of Tempura (Section 4.10).

4.2 Problem Formulation

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

4.2.1 Incremental Query Planning

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

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

The goal is to generate an *incremental plan* $\mathbb{P} = [P_1, \ldots, P_k]$ where P_i defines the task (a physical plan) to execute at time t_i, such that (1) $\forall 1 \leq i \leq k$, P_i can deliver the results defined by Q_i, and (2) the cost $\tilde{\mathfrak{c}}(\mathbb{P})$ is minimized. Next we use a few IQP scenarios to demonstrate how they can be modeled using the above definition. Note that we discuss the problem in a static setting for simplicity. In this setting, the optimizer generates a complete incremental plan using static information about the incrementally arrived data and the query requirement. Discussions about extending the optimizer's capability to perform

re-optimizations in a dynamic setting will be deferred to Section 4.9

Incremental View Maintenance (`IVM-PD`). Consider the problem of incrementally maintaining a view defined by query Q. Instead of using any concrete physical time, we can use two logical time points $\vec{T} = [t_i, t_{i+1}]$ to represent a general incremental update at t_{i+1} of the result computed at t_i. We assume that the data available at t_i is the data accumulated up to t_i, whereas at t_{i+1} the new delta data (insertions/deletions/updates) between t_i and t_{i+1} is available, denoted by $\vec{D} = [D, \Delta D]$. At both t_i and t_{i+1} we want to keep the view up to date, i.e., $\vec{Q}$ is defined as $Q_i = Q(D), Q_{i+1} = Q(D + \Delta D)$. As the main goal is to find the most efficient incremental plan, we set $\tilde{\mathfrak{c}}$ to be the cost of P_{i+1}, i.e., the execution cost at t_{i+1}. (For a formal definition see $\tilde{c}_v$ in Section 4.6.2.) Note that if Q involves multiple tables and we want to use different incremental plans for updates on different tables, we can optimize multiple IQP problems by setting ΔD to the delta data on only one of the tables at a time.

Progressive Data Warehouse (`PDW-PD`). We model this scenario by choosing $\vec{T}$ as physical time points of the planned incremental runs. Note that we only require the incremental plan to deliver the results defined by the original analysis job Q at the last run, that is, at the scheduled deadline of the job, without requiring output during the early runs. Thus, $\vec{Q} = [\emptyset, \cdots, \emptyset, Q]$. We set $\tilde{\mathfrak{c}}$ as a weighted sum of the costs of all plans in $\mathbb{P}$ (see $\tilde{c}_w(O)$ in Section 4.6.2).

4.2.2 Plan Space and Search Challenges

We elaborate different plans to answer the query in Example 4.1 using the `PDW-PD` definition. Suppose the query summary is originally scheduled at t_2, but the progressive data warehouse decides to schedule an early execution at t_1 on partial inputs. Assume the records visible at t_1 and t_2 in sales and returns are those in Fig. 1(a). In this IQP problem, we have $\vec{T} = [t_1, t_2]$ and $\vec{Q} = [\emptyset, q]$, where q is the summary query, $\vec{D}$ is shown in Fig. 1(a), and $\tilde{\mathfrak{c}}$ is the cost function

that takes the weighted sum of the resources used at t_1 and t_2. Many existing incremental methods (e.g., view maintenance, stream computing, mini-batch execution [40, 56, 14, 21]) can be used here. Consider two common methods `IM-1` and `IM-2`.

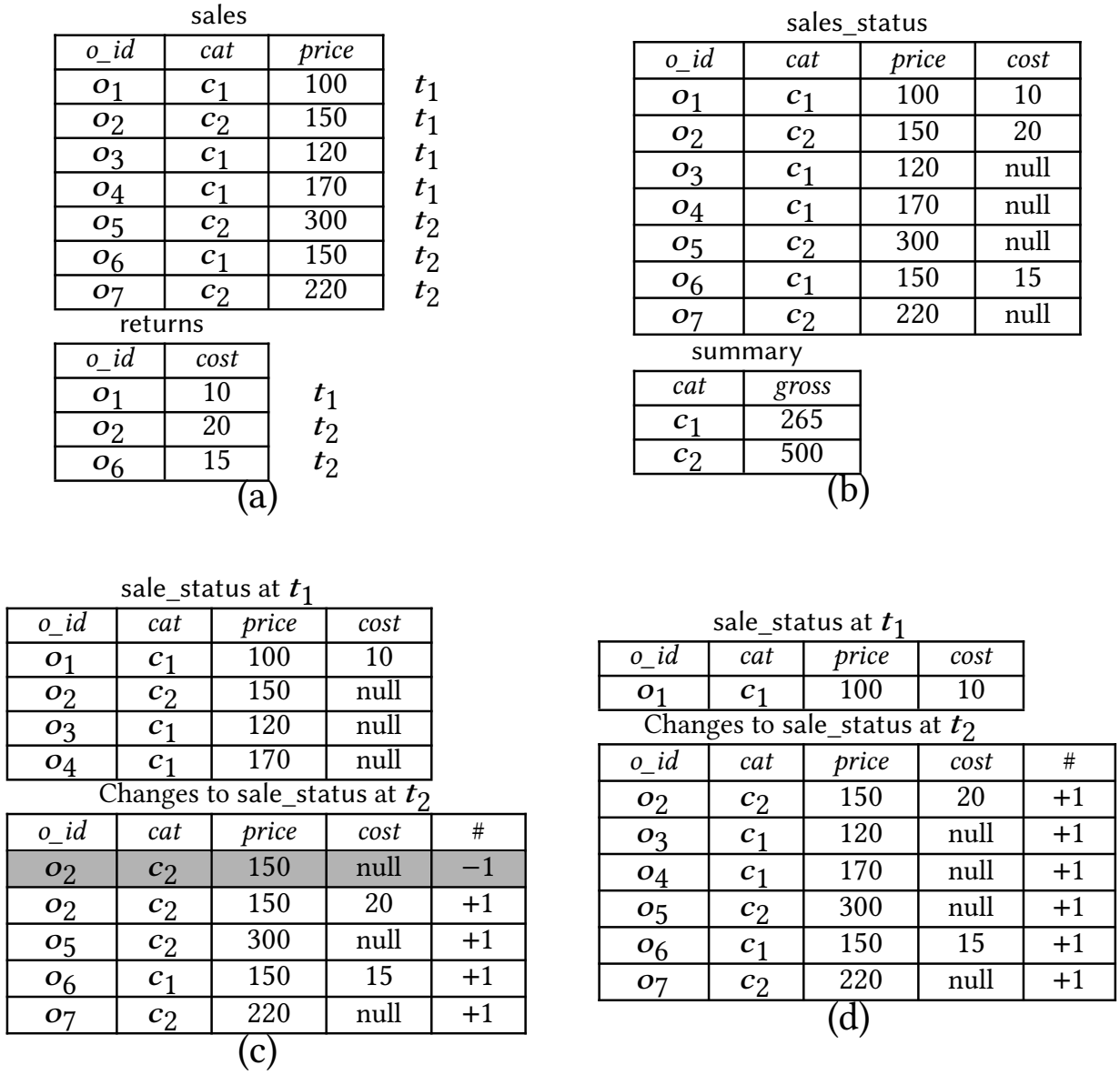


Figure 4.4: (a) Data arrival patterns of sales and returns, (b) results of sales_status and summary at t_2, (c) incremental results of sales_status produced by `IM-1` at t_1 and t_2, and (d) incremental results of sales_status produced by `IM-2` at t_1, t_2.

Method `IM-1` treats sales_status and summary as views, and uses incremental computation to keep the views up to date with respect to the data seen so far. The incremental compu-

tation is done on the delta input. For example, the delta input to sales at t_2 includes tuples $\{o_5, o_6, o_7\}$. Fig. 1(c) depicts sales_status's incremental outputs at t_1 and t_2, respectively, where $\# = +/-1$ denote insertion or deletion respectively. Note that a returns record (e.g., o_2 at t_2) can arrive much later than its corresponding sales record (e.g., the shaded o_2 at t_1). Therefore, a sales record may be output early as it cannot join with a returns record at t_1, but retracted later at t_2 when the returns record arrives, such as the shaded tuple o_2 in Fig. 1(c).

Method `IM-2` can avoid such retraction during incremental computation. Specifically, in the outer join of sales_status, tuples in sales that do not join with tuples from returns for now (e.g., o_2, o_3, and o_4) may join in the future, and thus are held back at t_1. Essentially the outer join is computed as an inner join at t_1. The incremental outputs of sales_status are shown in Fig. 1(d).

In addition to these two, there are many other methods as well. Generating one plan with a high performance is non-trivial due to the following reasons. *(1) The optimal incremental plan is data dependent, and should be determined in a cost-based way.* In the running example, `IM-1` computes 9 tuples (5 tuples in the outer join and 4 tuples in the aggregate) at t_1, and 10 tuples at t_2. Suppose the cost per unit at t_1 is 0.2 (due to fewer queries at that time), and the cost per unit at t_2 is 1. Then its total cost is $9 \times 0.2 + 10 \times 1 = 11.8$. `IM-2` computes 6 tuples at t_1, and 11 tuples at t_2, with a total cost of $6 \times 0.2 + 11 \times 1 = 12.2$. `IM-1` is more efficient, since it can do more early computation in the outer join, and more early outputs further enable summary to do more early computation. On the contrary, if retraction is often, say, with one more tuple o_4 at t_2, then `IM-2` is more efficient, as it costs 12.2 versus 13.8 of `IM-1`. This is because retraction wastes early computation and causes more recomputation. Notice that the performance difference of these two approaches can be arbitrarily large.

(2) The entire space of possible plan alternatives is very large. Different parts within a query

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

(3) Complex temporal dependencies between different incremental runs can also impact the plan decision. For instance, during the continuous ingestion of data, query sales_status may prefer a broadcast join at t_1 when the returns table is small, but a shuffled hash join at t_2 when the returns table gets bigger. But the decision may not be optimal, as shuffled hash join needs data to be distributed by the join key, which broadcast join does not provide. Thus, two join implementations between t_1 and t_2 incur reshuffling overhead. IQP needs to jointly consider all runs across the entire timeline.

Such complex reasoning is challenging, if not impossible, even for very experienced experts. To solve this problem, we offer a cost-based solution to systematically search the entire plan space to generate an optimal plan. Our solution can unify different incremental computation techniques in a single plan.

4.3 The TIP Model

The core of incremental computation is to deal with relations changing over time, and understand how the computation on these relations can be expanded along the time dimension. In this section, we introduce a formal theory based on the concept of *time-varying relation* (TVR) [21, 25, 91], called the *TVR-based Incremental query Planning (TIP) Model*. The model naturally extends the relational model by considering the temporal aspect to formally describe incremental execution. It also includes various data-manipulation operations on TVRs, as well as rewrite rules of TVRs in order for a query optimizer to define and explore a search space to generate an efficient incremental query plan. To the best of our knowledge, the proposed TIP model is the first one that not only unifies different incremental computation methods, but also can be used to develop a principled cost-based optimization framework for incremental execution. We focus on definitions and algebra of TVRs in this section, and dwell on TVR rewrite rules in Section 4.4.

4.3.1 Time-Varying Relations

Definition 4.1. *A* time-varying relation (TVR) *R is a mapping from a time domain $\mathcal{T}$ to a bag of tuples belonging to a schema.*

A *snapshot* of R at a time t, denoted R_t, is the instance of R at time t. For example, due to continuous ingestion, table sales (S) in Example 4.1 is a TVR, depicted as the blue line in Fig. 4.5. On the line, tables ① and ② show the snapshots S_{t_1} and S_{t_2} respectively. Traditional data warehouses run queries on relations at a specific time, while incremental execution runs queries on TVRs.

Definition 4.2 (Querying TVR)**.** *Given a TVR R on time domain $\mathcal{T}$, applying a query Q on R defines another TVR $Q(R)$ on $\mathcal{T}$, where $[Q(R)]_t = Q(R_t), \forall t \in \mathcal{T}$.*

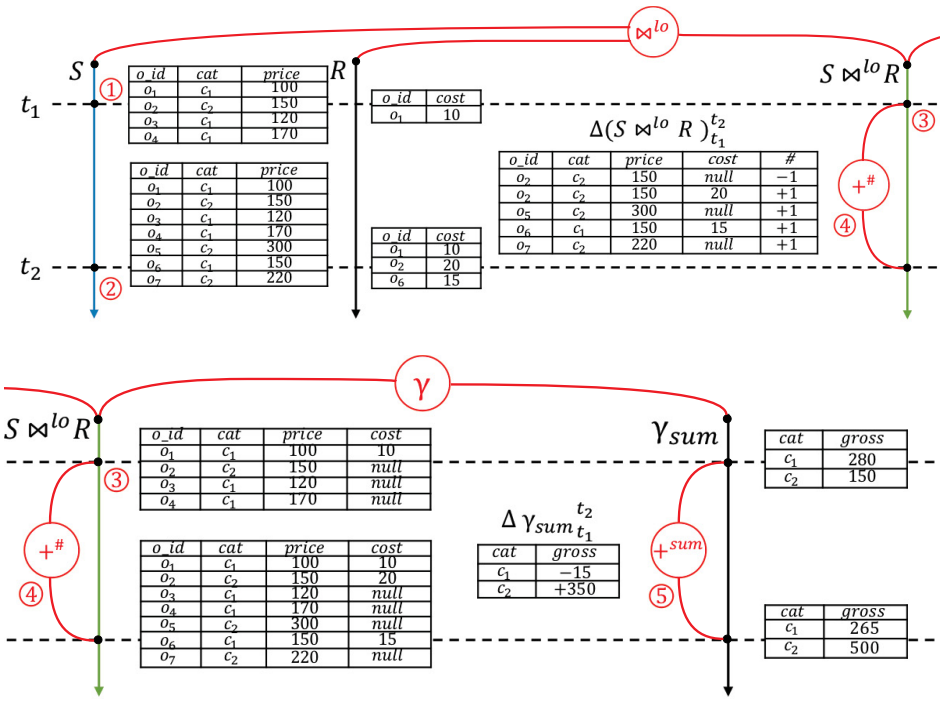


Figure 4.5: Example TVRs and their relationships. We denote left outer-join as $\bowtie^{lo}$, left anti-join as $\bowtie^{la}$, left semi-join as $\bowtie^{ls}$, and aggregate as γ.

In other words, the snapshot of $Q(R)$ at t is the same as applying Q as a query on the snapshot of R at t. For instance, in Fig. 4.5, joining two TVRs sales (S) and returns (R) yields a TVR ($S \bowtie^{lo} R$), depicted as the green line. Snapshot $(S \bowtie^{lo} R)_{t_1}$ is shown as table ③, which is equal to $St_1 \bowtie^{lo} R_{t_1}$. We denote left outer-join as $\bowtie^{lo}$, left anti-join as $\bowtie^{la}$, left semi-join as $\bowtie^{ls}$, and aggregate as γ. For brevity, we use "Q" to refer to the "TVR $Q(R)$" when there is no ambiguity.

4.3.2 Basic Operations on TVRs

Besides as a sequence of snapshots, a TVR can be encoded from a delta perspective using the changes between two snapshots. We denote the difference between two snapshots of TVR R at $t, t' \in T$ $(t < t')$ as the *delta* of R from t to t', denoted $\Delta R_t^{t'}$, which defines a second-order TVR.

Definition 4.3 (TVR difference)**.** *$\Delta R_t^{t'}$ defines a mapping from a time interval to a bag of tuples belonging to the same schema, such that there is a merge operator "+" satisfying* $R_t + \Delta R_t^{t'} = R_{t'}$.

Table ④ in Fig. 4.5 shows $\Delta(S \bowtie^{lo} R)_{t_1}^{t_2}$, which is the delta of snapshots $(S \bowtie^{lo} R)_{t_1}$ and $(S \bowtie^{lo} R)_{t_2}$. Here multiplicities (#) represent insertion and deletion of the corresponding tuple, respectively. The merge operator + is defined as additive union on relations with bag semantics, which adds up the multiplicities of tuples in bags.

Interestingly, a TVR can have different snapshot/delta views. For instance, the delta $\Delta\gamma_{sum}{}_{t_1}^{t_2}$ can be defined differently as Table ⑤ in Fig. 4.5. Here the merge operator + directly sums up the partial `SUM` values (the *gross* attribute) per *category*. For *category* c_1, summing up the partial `SUM`'s in $\gamma_{sum t_1}$ and $\Delta\gamma_{sum}{}_{t_1}^{t_2}$ yields the value in $\gamma_{sum t_2}$, i.e., $280 + (-15) = 265$. To differentiate these two merge operators, we denote the merge operator for $S \bowtie^{lo} R$ as $+^{\#}$, and the merge operator for γ_{sum} as $+^{sum}$. This observation shows that the way to define TVR deltas and the merge operator + is not unique. In general, as studied in previous research [66, 117], the difference between two snapshots R_t and $R_{t'}$ can have two types:

(1) *Multiplicity Perspective.* R_t and $R_{t'}$ may have different multiplicities of tuples. R_t may have less or more tuples than $R_{t'}$. In this case, the merge operator (e.g., $+^{\#}$) combines the same tuples by adding up their multiplicities.

(2) *Attribute Perspective.* R_t may have different attribute values in some tuples compared to $R_{t'}$. In this case, the merge operator (e.g., $+^{sum}$) groups tuples with the same primary key, and combines the delta updates on the changed attributes into one value. Aggregation operators usually produce this type of snapshots and deltas. Formally, distributed aggregation in data-parallel computing platforms is often modeled using four methods [114]:

1. `Initialize`: It is called once before any data is supplied with a given key to initialize

the aggregate state.

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

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

Furthermore, for some merge operator $+$, there is an inverse operator $-$, such that $R_{t'} - R_t = \Delta R_t^{t'}$. For instance, the inverse operator $-^{sum}$ for $+^{sum}$ is defined as taking the difference of `SUM` values per *category* between two snapshots.

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

4.4 TVR Rewrite Rules

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

To capture this more general form of equivalence, in this section, we introduce *TVR rewrite rules* in the TIP model, focusing on logical plans. We propose a trichotomy of TVR rewrite rules, namely *TVR-generating rules*, *intra-TVR rules*, and *inter-TVR rules*, and show how to model existing incremental techniques using these three types of rules. This modeling enables us to unify existing incremental techniques and leverage them uniformly when exploring the plan space; it also allows IQP to evolve by adding new TVR rewrite rules.

4.4.1 TVR-Generating and Intra-TVR Rules

Most existing work on incremental computation revolves around the notion of a delta query that can be described as Eq. 4.1 below.

$$Q(R_{t'}) = Q(R_t + \Delta R_t^{t'}) = Q(R_t) + \mathfrak{d}Q(R_t, \Delta R_t^{t'}) \tag{4.1}$$

Intuitively, when an input delta $\Delta R_t^{t'}$ arrives, instead of recomputing the query on the new input snapshot $R_{t'}$, one can directly compute a delta update to the previous query result $Q(R_t)$ using a new delta query $\mathfrak{d}Q$. Essentially, Eq. 4.1 contains two key parts—the delta query $\mathfrak{d}Q$ and the merge operator $+$, which correspond to the first two types of TVR rewrite

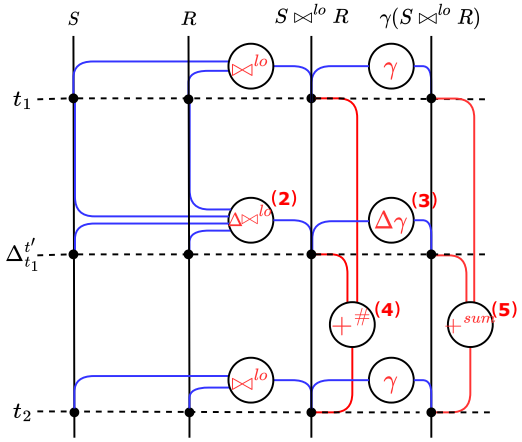


Figure 4.6: Examples of TVR-generating and Intra-TVR rules. Eq. 4.2: incrementally compute the delta of $S \bowtie^{lo} R$. Eq. 4.3: incrementally compute the delta of $\gamma(S \bowtie^{lo} R)$ from the delta of $S \bowtie^{lo} R$. Eq. 4.4 and 4.5: merge a snapshot at t_1 and a delta to a generate a new snapshot at t_2.

rules, namely *TVR-generating rules* and *intra-TVR rules*.

TVR-Generating Rules. Formally, TVR-generating rules define for each relational operator on a TVR, how to compute its deltas from the snapshots and deltas of its input TVRs. In other words, TVR-generating rules define $\mathfrak{d}Q$ for each relational operator Q such that $\Delta Q_t^{t'} = \mathfrak{d}Q(R_t, \Delta R_t^{t'})$. Many previous studies on deriving delta queries under different semantics [29, 31, 40, 55, 56] fall into this category. Some example TVR-generating rules used by `IM-1` in Example 4.1 are shown as follows:

$$\begin{aligned}\Delta(S \bowtie^{lo} R)_{t_1}^{t_2} &= \Delta S^+ \ltimes^{lo} R_{t_2} + S_{t_2} \ltimes \Delta R^+ \\ &+(S_{t_1} - \Delta S^-) \ltimes^{ls} (\Delta R^- \ltimes^{la} R_{t_2}) - \Delta S^- \ltimes^{lo} R_{t_1} \\ &+ (S_{t_1} - \Delta S^-) \ltimes^{ls} (\Delta R^- \ltimes^{la} R_{t_2}) - \Delta S^- \ltimes^{lo} R_{t_1},\end{aligned} \tag{4.2}$$

$$\Delta\gamma(S \ltimes^{lo} R)_{t_1}^{t_2} = \gamma(\Delta(S \ltimes^{lo} R)_{t_1}^{t_2}). \tag{4.3}$$

In the rules, we denote left outer-join as $\bowtie^{lo}$ and left semi-join as $\bowtie^{ls}$. The rules for left

outer-join (Eq. 4.2) and aggregate (Eq. 4.3) are from [55] and [56], respectively. In the rules, we use ΔS^- and ΔR^- to denote deletions in the delta, and ΔS^+ and ΔR^+ to denote insertions in the delta for simplicity. In Eq. 4.2, for brevity, padding nulls to match outer join's schema is omitted in Fig. 4.6 and Fig. 4.8. This padding can simply be implemented using a project operator. The blue lines in Fig. 4.6 demonstrate these TVR-generating rules in a plan space.

Intra-TVR Rules. Intra-TVR rules define the conversions between snapshots and deltas of a single TVR. As in Eq. 4.1, the merge operator + defines how to merge Q's snapshot Q_t and delta $\Delta Q_t^{t'}$ into a new snapshot $Q_{t'}$. Other examples of intra-TVR rules include rules that take the difference between snapshots/deltas if the merge operator + has an inverse operator $-$, e.g., $R_{t'} - R_t = \Delta R_t^{t'}$. In Fig. 4.6, the intra-TVR rules by `IM-1` in Example 4.1 are marked as red lines. These rules are shown as follows:

$$(S \bowtie^{lo} R)_{t_2} = (S \bowtie^{lo} R)_{t_1} +^{\#} \Delta(S \bowtie^{lo} R)_{t_1}^{t_2} \tag{4.4}$$

$$\gamma(S \bowtie^{lo} R)_{t_2} = \gamma(S \bowtie^{lo} R)_{t_1} +^{sum} \Delta\gamma(S \bowtie^{lo} R)_{t_1}^{t_2} \tag{4.5}$$

Note that when merging the snapshot/delta of $S \bowtie^{lo} R$, we use $+^{\#}$ (Eq. 4.4), whereas when merging the snapshot/delta of $\gamma(S \bowtie^{lo} R)$ (query summary), we use $+^{sum}$ (Eq. 4.5).

4.4.2 Inter-TVR Rules

There are incremental methods that cannot be modeled using the two aforementioned types of rules alone. The `IM-2` approach in Example 4.1 is such an example. Different from `IM-1`, approach `IM-2` does not directly deliver the snapshot of $S \bowtie^{lo} R$ at t_1. Instead, it delivers only the tuples that will not be retracted in the future, essentially the results of $S \bowtie R$. At t_2

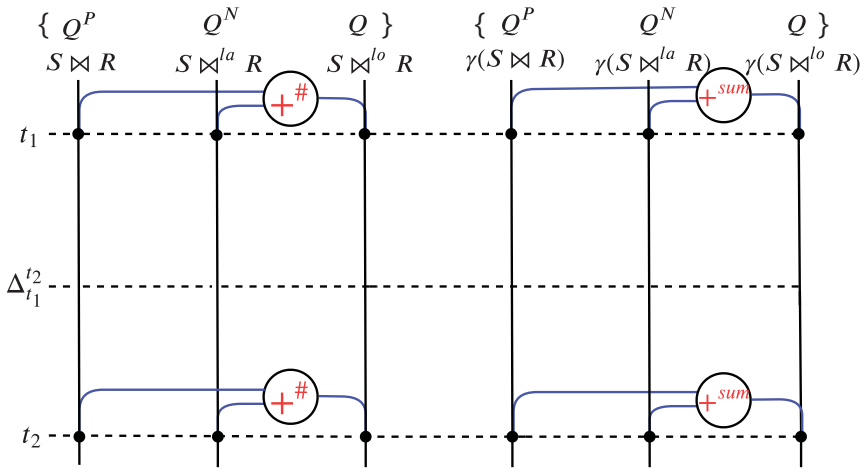


Figure 4.7: Examples of inter-TVR equivalence rules in `IM-2`. Each operator in original query Q is decomposed into two parts: Q^P (positive-only updates) and Q^N (possibly negative updates).

when the data is known to be complete, `IM-2` computes the rest part of $S \bowtie^{lo} R$, essentially $S \bowtie^{la} R$, then pads with nulls to match the output schema.

This observation shows another family of incremental methods: without computing Q directly, one can incrementally compute a set of queries $\{Q'_1, \cdots, Q'_k\}$, and then apply another query P on their results to get Q, formally described as Eq. 4.6. The intuition is that $\{Q'_1, \cdots, Q'_k\}$ may be more amenable to incremental computation:

$$Q(R) = P(Q'_1(R), \cdots, Q'_k(R)). \tag{4.6}$$

Eq. 4.6 describe a general family of methods: they all rely on certain rewrite rules describing the equivalence between snapshots/deltas of multiple TVRs. We summarize this family of methods into *inter-TVR rules*. Next we demonstrate using a couple of existing incremental methods how they can be modeled by inter-TVR rules.

(1) ***`IM-2`***: Let us revisit `IM-2` using the terminology of inter-TVR rules. Formally, $Q = S \bowtie^{lo}$

R is decomposed into Q^P and Q^N:

$$Q^P{}_t = S_t \bowtie R_t,\ Q^N{}_t = S_t \bowtie^{la} R_t,\ Q_t = Q^P{}_t +^{\#} Q^N{}_t \tag{4.7}$$

where Q^P is a positive part that will not retract tuples if both S and R are append-only, whereas Q^N represents a part that could retract tuples. The inter-TVR rule in Eq. 4.7 states that any snapshot of Q can be decomposed into snapshots of Q^P and Q^N at the same time. Similar decomposition holds for the aggregate γ in *summary* too, just with a different merge operator $+^{sum}$. Fig. 4.7 depicts these rules in a plan space. As it is easier to incrementally compute inner join than left outer join, Q^P can be incrementalized more efficiently than Q with rules in Section 4.4.1, whereas Q^N cannot be easily incrementalized, and is not computed until the completion time.

(2) Outer-join view maintenance (`OJV`): Larson et al. [67] proposed a method to incrementally maintain outer-join views.

Query decomposition. The main idea is to decompose a query into three parts given an update to a single input table: a directly affected part Q^D, an indirectly affected part Q^I, and an unaffected part Q^U. Intuitively, an insertion (deletion) in the input table will cause insertions (deletions) to Q^D and deletions (insertions) to Q^I, but leave Q^U unaffected. These parts are formally defined using the join-disjunctive normal form of Q and its subsumption graph. We refer the readers to [67] for details. This decomposition can be expressed using the following inter-TVR rule:

$$Q_t = Q^D{}_t +^{\#} Q^I{}_t +^{\#} Q^U{}_t. \tag{4.8}$$

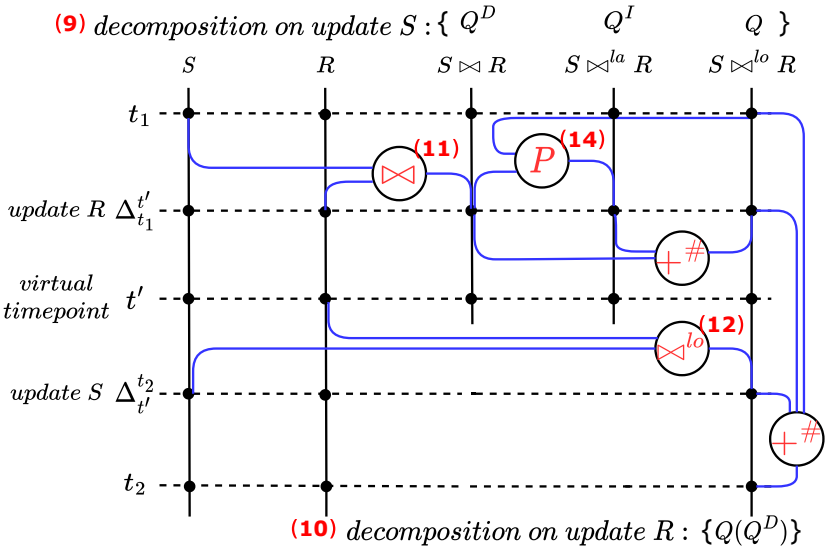


Figure 4.8: Supporting outer-join view maintenance. A virtual timepoint t' is inserted to model updating one base table at a time. Eq. 4.9, 4.10: decompose the query into Q^D and Q^I. Eq. 4.11, 4.12: compute the delta of directly affected parts. Eq. 4.14: compute the delta of indirectly affected parts.

Take query sales_status as an example. As the algorithm in [67] considers updating one input table at a time, we insert a virtual time point t' between t_1 and t_2 to model that R and S are updated separately at t' and t_2. The query sales_status is decomposed as follows:

$$Q^D{}_{t'} = S_{t'} \ltimes R_{t'}, \qquad Q^I{}_{t'} = S_{t'} \ltimes^{la} R_{t'}, \qquad Q^U{}_{t'} = \emptyset, \tag{4.9}$$

when R is updated at t',

$$Q^D{}_{t_2} = S_{t_2} \ltimes^{lo} R_{t_2}, \qquad Q^I{}_{t_2} = \emptyset, \qquad Q^U{}_{t_2} = \emptyset \tag{4.10}$$

when S is updated at t_2.

Note that there is no unaffected part in this example. Unaffected parts only exist when a query joins three or more tables according to the algorithm in [67].

Delta computation. The outer-join view maintenance algorithm maintains the directly

affected parts and the indirectly affected parts separately.

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

$$\begin{aligned} \Delta Q^{D}{}_{t_1}^{t'} &= S_{t_1} \bowtie^{lo} \Delta R_{t_1}^{t'} \\ &= \{(o_2, c_2, 150, 20, +1)\}, \end{aligned} \tag{4.11}$$

$$\begin{aligned} \Delta Q^{D}{}_{t'}^{t_2} &= \Delta S_{t'}^{t_2} \ltimes R_{t'} \\ &= \{(o_5, c_2, 300, null, +1), \\ &\qquad (o_6, c_1, 150, 15, +1), \\ &\qquad (o_7, c_2, 220, null, +1)\}. \end{aligned} \tag{4.12}$$

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

To compute the delta of the indirectly affected parts, OJV combines the delta of Q^D and the previous snapshot of Q, as shown in Eq. 4.13. Compared to computing the detla directly from the base tables, this algorithm can reuse the already computed delta of the directly affected parts. This rule can be expressed using the following inter-TVR rule:

$$\Delta Q^{I}{}_{t}^{t'} = P(\Delta Q^{D}{}_{t}^{t'}, Q_t). \tag{4.13}$$

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

$$\Delta Q^{I}{}_{t_1}^{t'} = -[\sigma_{cost=null}(Q_{t_1}) \bowtie^{ls} \Delta Q^{D}{}_{t_1}^{t'}] \tag{4.14}$$
$$= \{(o_2, c_2, 150, null, -1)\}.$$

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

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

- `IM-2` considers updating all tables at the same time. It decomposes the query into two parts, Q^P and Q^N.
- `OJV` considers updating one base table at a time. It decomposes the query into a finer granularity of three parts: ${Q^D}_{t'}$ on updating R, ${Q^I}_{t'}$ on updating R, and ${Q^D}_{t_2}$ on updating S.
- In `IM-2`, each Q^P part contains the tuples that will never be retracted if the base tables are append-only. As an example, $\Delta Q^{P}{}_{t_1}^{t_2}$ contains only two tuples, o_2 and o_5.
- In `OJV`, each Q^D part contains the tuples that are positive when its corresponding base table is updated. As an example, $\Delta Q^{D}{}_{t'}^{t_2}$ contains tuples o_5, o_6, and o_7. Tuples o_5 and o_7 could potentially be retracted in the future.
- `IM-2` computes the delta using the TVR-generating rules. `OJV` introduces a new rule (Eq. 4.14) to compute the delta of indirectly affected parts.

(3) Higher-order view maintenance: [14, 83] proposed a higher-order view-maintenance algorithm, which can also be expressed by inter-TVR rules. The main idea is to treat the deltas of a query Q as another TVR, and continue applying TVR rewrite rules to incrementally compute it. Formally, considering a query Q and updates to one of its inputs R, the algorithm can be summarized as the following inter-TVR rule:

$$\Delta Q_t^{t'} = \mathfrak{d}Q(R_t, \Delta R_t^{t'}) = P(M_t, \Delta R_t^{t'}). \tag{4.15}$$

The rule decomposes the delta query into two parts: the delta update $\Delta R_t^{t'}$, and an update-independent subquery M that does not involve $\Delta R_t^{t'}$. The two parts are combined using a query P to get the delta of Q. If M is a query involving input relations other than R, it can be further decomposed again with respect to updates to each of its input relations according to Eq. 4.15, until it becomes a constant. We refer the readers to [14] for a detailed algorithm. Take the summary query and updates to sales (S) as an example (we denote returns as R). Applying Eq. 4.15, we can decompose it as

$$\begin{aligned} \Delta Q_t^{t'} &= \gamma_{category;\texttt{SUM}(r)}(\Delta S_t^{t'} \bowtie^{lo} M_t), \\ \text{where} \quad M_t &= \gamma_{o\_id;total=\texttt{SUM}(cost)}(R_t), \\ r &= \texttt{IF}(total\ \texttt{IS NULL}, price, -total). \end{aligned} \tag{4.16}$$

M essentially preprocesses returns by computing the total cost per o_id,[2] and P computes the gross revenue per category by summing up the precomputed total cost in M or the prices

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

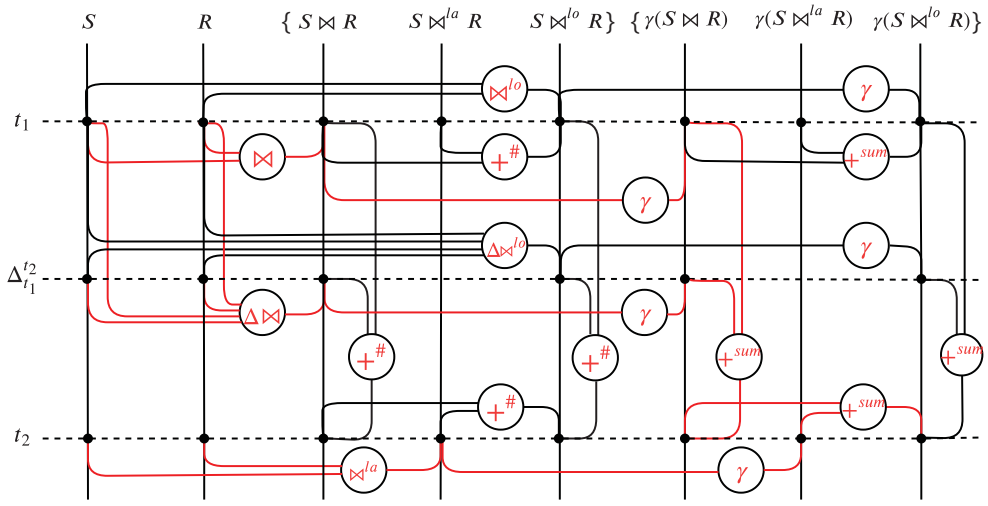


Figure 4.9: The combined incremental plan space of Example 4.1.

of the new orders added to S. Then M is materialized as a higher-order view and can be further incrementally maintained with respect to updates to `returns` by repeatedly applying the inter-TVR rule to generate higher-order views.

4.4.3 Putting Everything Together

The above TVR rewrite rules lay a theoretical foundation for our IQP framework. Different TVR rules can be extended individually and work together automatically. For example, TVR-generating rules can be applied on any TVR created by inter-TVR rules. By jointly applying TVR rewrite rules and traditional rewrite rules, we can explore a plan space much larger than any individual incremental method. Fig. 4.9 shows an example plan space by overlaying Fig. 4.6 and 4.7. Any tree rooted at $\gamma(S \bowtie^{lo} R)_{t_2}$ is a valid incremental plan for Example 4.1, e.g., `IM-2`'s plan is shown in red.

In the next two sections, we discuss how to build an optimizer framework based on the TIP model, including plan-space exploration (Section 4.5) and selecting an optimal incremental plan (Section 4.6).

4.5 Plan-Space Exploration

In this section we study how Tempura explores the incremental plan space. Existing query optimizers explore plans only for a specific time. For incremental processing, we need to explore a much bigger plan space by considering not only relations at different times, but also transformations between them. We illustrate how to incorporate the TIP model into a Cascades-style optimizer [52, 54], and develop a cost-based optimizer framework for IQP called Tempura.

We focus on the key adaptations on two main modules. (1) *Memo*: it keeps track of the explored plan space, i.e., all plan alternatives generated, in a succinct data structure, typically represented as an AND/OR tree, for detecting redundant derivations and fast retrieval. (2) *Rule engine*: it manages all the transformation rules, which specify algebraic equivalence laws and physical implementations of logical operators, and monitors new plans generated in the memo. Whenever there are changes, the rule engine fires applicable transformation rules on the newly-generated plans to add more plan alternatives to the memo.

4.5.1 Memo: Capturing TVR Relationships

The memo in the traditional Cascades-style optimizer only captures two levels of equivalence relationship: *logical equivalence* and *physical equivalence*. A logical equivalence class groups operators that generate the same result set; within each logical equivalence class, operators are further grouped into physical equivalence classes by their physical properties such as sort order, distribution, etc. The "Traditional Memo" part in Fig. 4.10(a) depicts the traditional memo of the sales_status query. For brevity, we omit the physical equivalence classes. For instance, *LeftOuterJoin*[0,1] has Groups G0 and G1 as children, and it corresponds to the plan tree rooted at $\bowtie^{lo}$. G2 represents all plans logically equivalent to *LeftOuterJoin*[0,1].

However, the above two equivalences are not enough to capture the rich relationships in the TIP model. For example, the relationship between snapshots and deltas of a TVR cannot be modeled using the logical equivalence due to the following reasons. Two snapshots at different times produce different relations, and the snapshots and deltas do not even have the same schema (deltas have an extra # column). To solve this problem, on top of logical/physical equivalence classes, we explicitly introduce TVR nodes into the memo, and keep track of the following relationships, shown as the "Tempura Memo" part in Fig. 4.10(a): (1) **Intra-TVR relationship** specifies the snapshot/delta relationship between logical equivalence classes of operators and the corresponding TVRs. The traditional memo only models scanning the full content of S, i.e., S_{t_2}, represented by G0, while the Tempura memo models two more scans: scanning the partial content of S available at t_1 (S_{t_1}), and scanning the delta input of S newly available at t_2 ($\Delta S_{t_1}^{t_2}$), represented by G3 and G5. The memo uses an explicit TVR-0 to track these intra-TVR relationships. (2) **Inter-TVR relationship** specifies the relationship between TVRs described by inter-TVR equivalence rules. For example, the `IM-2` approach decomposes $S \bowtie^{lo} R$ (TVR-2) into two parts Q^P (TVR-3) and Q^N (TVR-4) as in Section 4.3. Note that the above relationships are transitive. For instance, as G7 is the snapshot at t_2 of TVR-3, and TVR-3 is in turn the Q^P part of TVR-2, G7 is also related to TVR-2.

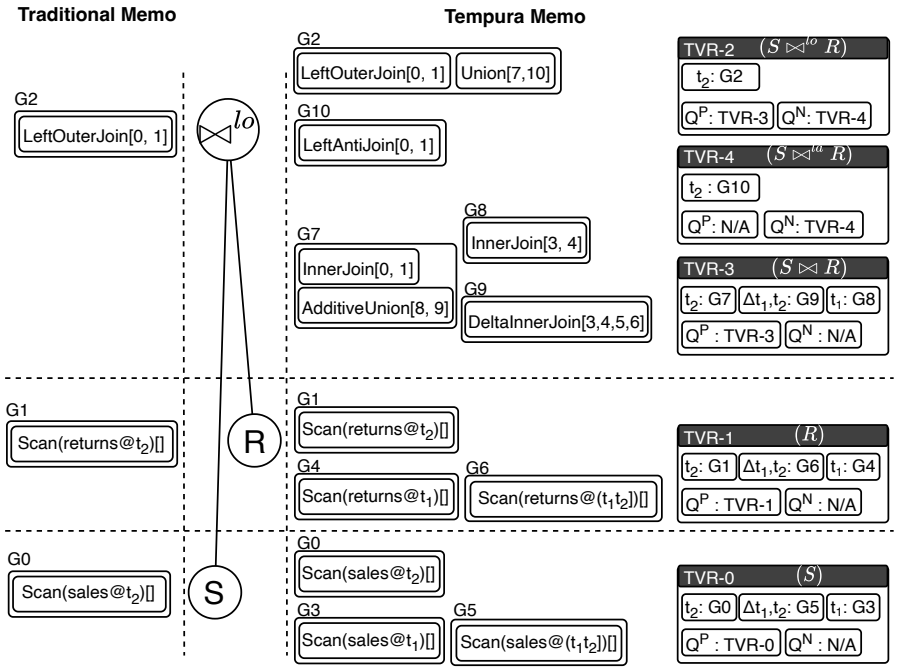
4.5.2 Rule Engine: Enabling TVR Rewritings

As the memo of Tempura strictly subsumes a traditional Cascades memo, traditional rewrite rules can be adopted and work without modifications. Besides, the rule engine of Tempura supports TVR rewrite rules. Tempura allows optimizer developers to define TVR rewrite rules by specifying a graph pattern on both relational operators and TVR nodes in the memo. A TVR rewrite rule pattern consists of two types of nodes and three types of edges: (1) *operator operands* that match relational operators; (2) *TVR operands* that match TVR nodes;

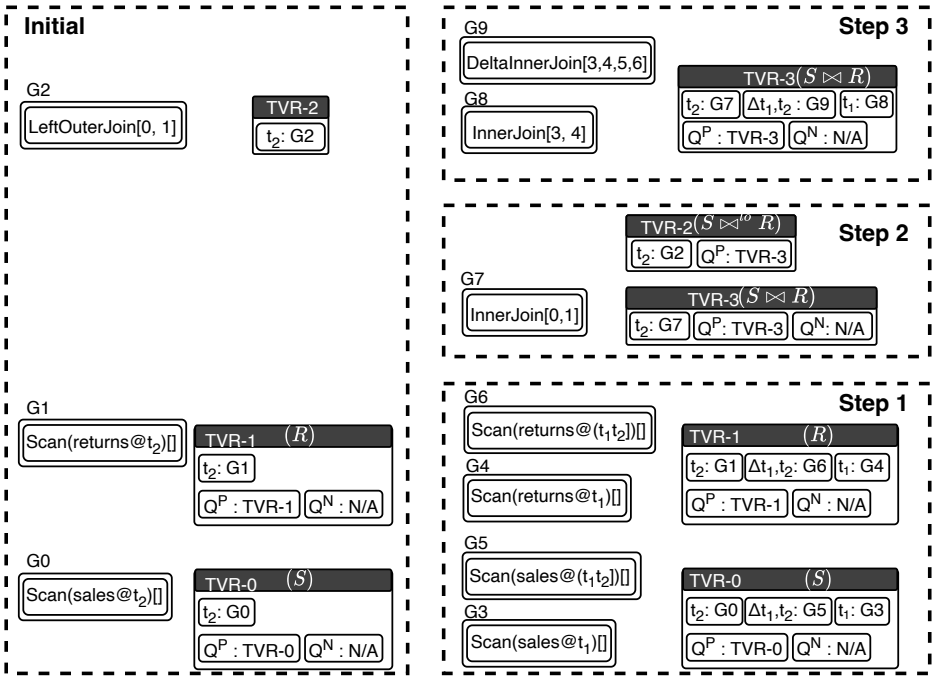
(3) *operator edges* between operator operands that specify the traditional parent-child relationship of operators; (4) *intra-TVR edges* between operator operands and TVR operands that specify intra-TVR relationships; and (5) *inter-TVR edges* between TVR operands that specify inter-TVR relationships. All nodes and intra/inter-TVR edges can have predicates. Once fired, TVR rewrite rules can register new TVR nodes and intra/inter-TVR relationships.

Fig. 4.11(a)-4.11(b) depict two TVR rewrite rules, where solid nodes and edges specify the patterns to match, and dotted ones are newly registered by the rules. In the figures, we also show an example match of these rules when applied on the memo in Fig. 4.10(a). **Rule 1** is the TVR-generating rule to delta compute an inner join. It matches a snapshot of an *InnerJoin*, whose children L, R each have a delta sibling L', R'. The rule generates a *DeltaInnerJoin* taking L, R, L', R' as inputs, and register it as a delta sibling of the original *InnerJoin*. **Rule 2** is an inter-TVR rule of `IM-2`. It matches a snapshot of a *LeftOuterJoin*, whose children L, R each have a Q^P snapshot sibling L', R'. The rule generates an *InnerJoin* of L' and R', and register it as the Q^P snapshot sibling of the original *LeftOuterJoin*.

Fig. 4.10(b) demonstrates the growth of a memo in Tempura. For each step, we only draw the updated part due to space limitation. The memo starts with G0 to G2 and their corresponding TVR-0 to TVR-2. In step 1, we first populate the snapshots and deltas of the *scan* operators, e.g., G3 to G6, and register the intra-TVR relationship in TVR-0 and TVR-1. We also populate their Q^P and Q^N inter-TVR relationships as in `IM-2` (for base tables these relationships are trivial). In step 2, in Fig. 4.11(b). rule 2 matches the tree rooted at *LeftOuterJoin[0,1]* in G2, generates the inner join of G7, and registers G7 to TVR-3 as the snapshot at t_2, and TVR-3 to TVR-2 as Q^P. In step 3, rule 1 matches *InnerJoin[0,1]* in G7 in Fig. 4.11(a) and generates *DeltaInnerJoin*[3,4,5,6] as the delta of TVR-3. By applying other TVR rewrite rules, we eventually get the memo in Fig. 4.10(a).

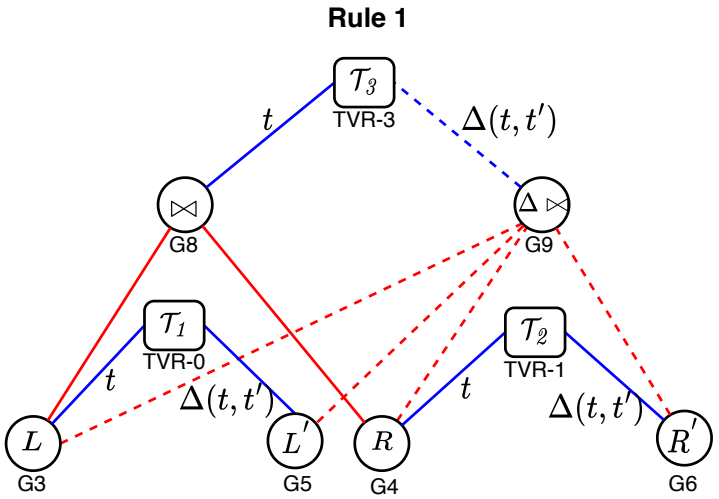


(a) An example memo of subquery sales_status. Compared to the traditional memo, the Tempura memo: (1) maintains more equivalence groups of snapshots at earlier time points and deltas, and (2) has additional TVR nodes to keep track of the intra-TVR and inter-TVR relationships.

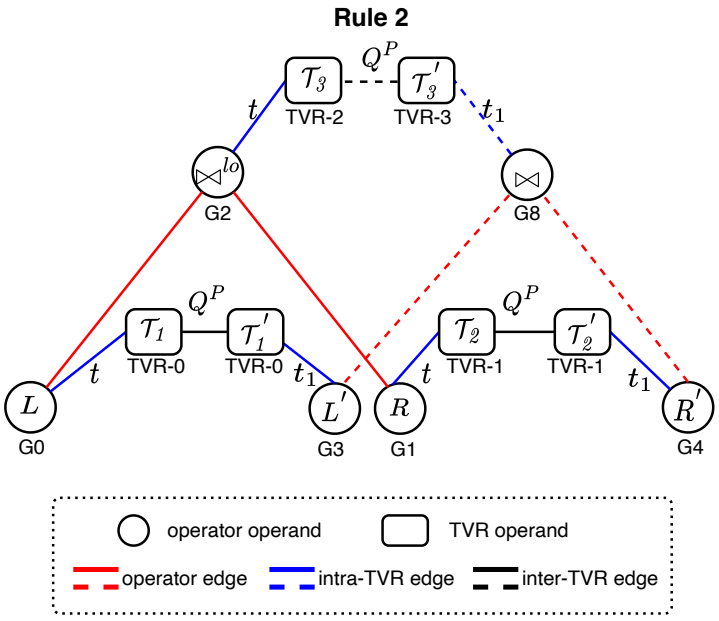


(b) A step-wise illustration of the growth of the memo. Step 1: for each source operator, its TVR, snapshots, and deltas are populated. Step 2: Rules of `IM-2` decompose left outer join to an inner join. Step 3: TVR-generating rules generate the operators to incrementally compute the inner join.

Figure 4.10: Examples of the memo structure in Tempura.



(a) A TVR-generating rule pattern



(b) An inter-TVR rule pattern

Figure 4.11: Example TVR rewrite-rule patterns in Tempura.

4.6 Selecting an Optimal Plan

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

1. In a traditional query plan, all physical operators are executed at the same time point in a single query. In Tempura, physical operators in an incremental plan might be executed at different time points. In Section 4.6.1, we discuss how to assign a valid execution time point of each physical operator.
2. Similarly, in a traditional query plan, the cost function represents the cost of a single time point. In Section 4.6.2, we discuss how to extend the cost function to consider the costs at different time points.
3. Finally, an incremental plan often needs to maintain intermediate states between the executions of different time points. In Section 4.6.3, we discuss how to find the optimal states to materialize.

4.6.1 Time-Point Annotations of Operators

Costing the plan alternatives is not trivial because the temporal dimension is involved. Fig. 4.12(a) depicts one physical plan rooted at $(S \bowtie^{lo} R)_{t_2}$, as shown in red in Fig. 4.9. This plan only specifies the concrete physical operations taken on the data, but does not specify when they are executed. Actually, each operator in the plan usually has multiple choices of execution time. In Fig. 4.12(a), the time points annotated alongside each operator shows the possible temporal domain of its execution. For instance, snapshots S_{t_1} and R_{t_1} are available at t_1, and thus can execute at any time after that, i.e., t_1 or t_2. Deltas $\Delta R_{t_1}^{t_2}$ and $\Delta S_{t_1}^{t_2}$ are not available until t_2, and thus any operators taking it as input, including the *IncrHashInnerJoin*, can only be executed at t_2. The temporal domain of each operator O,

denoted t-dom(O), can be defined inductively: (1) **For a base relation** R, t-dom(R) is the set of execution time points that are no earlier than the time point when R is available. (2) **For an operator** O **with inputs** $I_1, \ldots, I_k$, t-dom(R) is the intersection of its inputs' temporal domains: t-dom(R) = $\cap_{1 \leq j \leq k}$t-dom(I_j).

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

Definition 4.4 (Valid Temporal Assignment)**.** *An assignment of execution time points to a physical plan is valid if and only if for each operator O, its execution time $\tau(O)$ satisfies $\tau(O) \in$ t-dom(O) and $\tau(O) \geq \tau(O')$ for all operators O' in the subtree rooted at O.*

Fig. 4.12(b) demonstrates a valid temporal assignment of the physical plan in Fig. 4.12(a). At t_1, the plan computes *HashInnerJoin* of S_{t_1} and R_{t_1}, and shuffles S_{t_1} and R_{t_1} to prepare for *IncrHashInnerJoin*. At t_2, the plan shuffles the new deltas $\Delta S_{t_1}^{t_2}$ and $\Delta R_{t_1}^{t_2}$, finishes *IncrHashInnerJoin*, and unions the results with that of *HashInnerJoin* computed at t_1. Note that if an operator O and its input I have different execution time points, then the output of I needs to be saved first at $\tau(I)$, and later loaded and fed into O at $\tau(O)$, e.g., *Union* at t_2 and *HashInnerJoin* at t_1. The cost of *Save* and *Load* needs to be properly included in the plan cost. It is worth noting that some operators save and load the output as a by-product, for which we can spare *Save* and *Load*, e.g., *Exchange* of S_{t_1}, R_{t_1} at t_1 for *IncrHashInnerJoin*.

4.6.2 Time-Point-Based Cost Functions

The cost of an incremental plan is defined under a specific assignment of execution time points. Therefore, the optimization problem is formulated as: given a plan space, find a

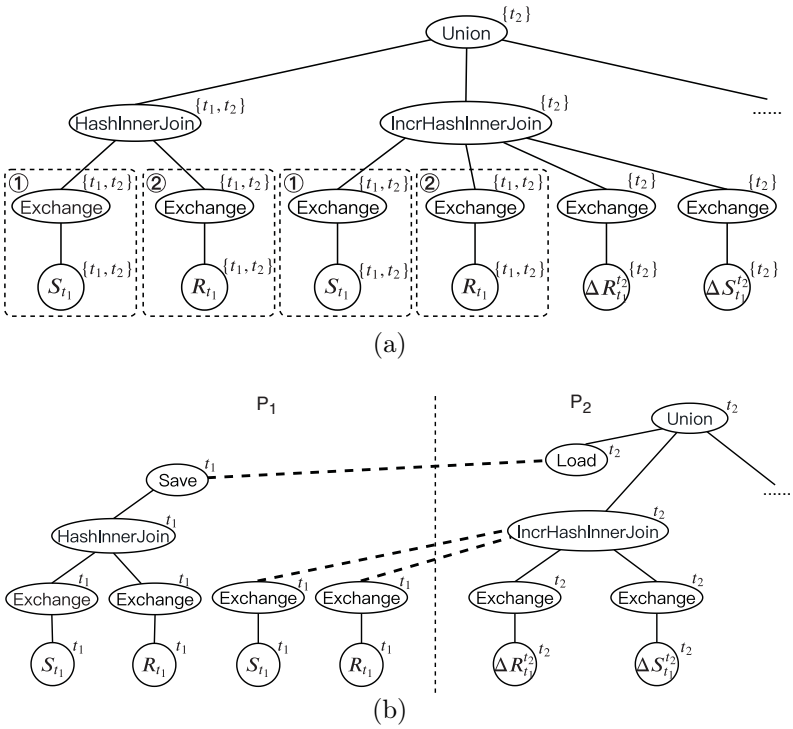


Figure 4.12: Examples of (a) the temporal plan space, and (b) a temporal assignment for subquery sales_status's plan.

physical plan and temporal assignment that achieve the lowest cost. In this section, we discuss the cost model and optimization algorithm for this problem without considering sharing common sub-plans. We will discuss the problem of how to decide which states to materialize in Section 4.6.3.

As an incremental plan can span across multiple time points, the cost function $\tilde{\mathfrak{c}}$ in an IQP problem (as in Section 4.2.1) is extended to a function taking into consideration of costs at different time points. For the cost at each time point, we inherit the general cost model used in traditional query optimizers, i.e., the cost of a plan is the sum of the costs of all its operators. Below we give two examples of $\tilde{\mathfrak{c}}$. We denote traditional cost functions as c, and c_i is the cost at time t_i. As before, c can be a number, e.g., estimated monetized resource cost, or a structure, e.g., a vector of CPU time and I/O.

1. $\tilde{c}_w(O) = \sum_{i=1..T} w_i \cdot c_i(O)$. The extended cost of an operator is a weighted sum of its cost

at each time t_i. For the example setting in Section 4.2.2, $w_1 = 0.2$ for t_1 and $w_2 = 1$ for t_2.

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

Consider the plan in Fig. 4.12(a) as an example. To get the result of *HashInnerJoin* at t_2, we have two options: (i) compute the join at t_2; or (ii) as in Fig. 4.12(b), compute the join at t_1, save the result, and load it back at t_2. Assume the cost of computing *HashInnerJoin*, saving the result, and loading it are 10, 5, 4, respectively. Then for option (i) $(c_1, c_2) = (0, 10)$, for option (ii) $(c_1, c_2) = (15, 4)$. Say that we use $\tilde{c}_w$ as the cost function. If $w_1 = 0.6$ and $w_2 = 1$ then option (i) is better, whereas if $w_1 = 0.2$ and $w_2 = 1$, option (ii) becomes better.

Dynamic programming (DP) used predominantly in existing query optimizers [92, 69, 54] also need to be adapted to handle the cost model extensions. In existing query optimizers, the DP state space is the set of all operators in the plan space, represented as $\{O\}$. Each operator O records the best cost of all the subtrees rooted at O. We extend the state space by considering all combinations of operators and their execution time points, i.e., $\{O\} \times \text{t-dom}(\{O\})$. Instead of recording a single optimum, each O records multiple optima, one for each execution time $\tau(O)$, which represents the best cost of all the subtrees rooted at O if O is generated at τ. During optimization, the state-transition function is the following:

$$\tilde{\mathfrak{c}}[O, \tau] = min_{\forall \text{ valid } \tau_j}\big(\sum_j \tilde{\mathfrak{c}}[I_j, \tau_j] + c_\tau(O)\big). \tag{4.17}$$

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

We have the following observation of the above DP algorithm: the optimization problem

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

4.6.3 Deciding States to Materialize

In an incremental plan, a delta computation often requires intermediate states to be saved from earlier computation. As an example, in Fig. 4.12(b), the incremental hash join at t_2 needs to use the saved intermediate states Shuffle(S_{t_1}) and Shuffle(R_{t_1}). However, an alternative plan is to re-compute these states from base tables instead of reusing materialized states. Whether to save the intermediate states or to recompute the states needs to be decided in a cost-based manner.

We model the problem of choosing the optimal intermediate states to materialize as a multi-query optimization problem by treating the plan at each time point as an independent mini-query and finding sharing states between the mini-queries at different time points. In the example in Fig. 4.12, we treat the whole incremental query as two independent mini-queries at two time points: query1 computes the join result at t_1 and query2 computes the delta join result from t_1 to t_2. These two mini-queries both need the states Shuffle(S_{t_1}) and Shuffle(R_{t_1}): query1 uses the states to produce the hash join result at t_1, and query2 uses these states to compute the incremental hash join result at t_2. The parts ① and② circled in dashed lines in Fig. 4.12(a) depict the shareable candidates. Therefore, computing the shuffle once and materializing these states once can benefit two reuse opportunities and reduce the overall cost of the incremental plan. Note that materializing a state is not always beneficial because the overhead of materialization might be higher than the cost of re-computing it. In order to choose the best sub-plans to materialize, we feed query1 and query2 together to

a multi-query optimization (MQO) algorithm [88, 118, 62]. In other words, a materialized shared sub-plan between two mini-queries Q_i and Q_j at two time points of an incremental plan is essentially an intermediate state that is saved by Q_i and reused by Q_j.

In this chapter, we extend the MQO algorithm in [62], which proposes a greedy framework on top of Cascade-style optimizers for MQO. For the sake of completeness, we list the algorithm in Algo. 5, by highlighting the extensions for progressive planning. The algorithm runs in an iterative fashion. In each iteration, it picks one more candidate from all possible shareable candidates, which if materialized can minimize the plan cost (line 4), where $bestPlan(\mathbb{S})$ means the best plan with $\mathbb{S}$ materialized and shared. The algorithm terminates when all candidates are considered or adding candidates can no longer improve the plan cost. As IQP needs to consider the temporal dimension, the shareable candidates are no longer solely the set of shareable sub-plans, but pairs of a shareable sub-plan s and a choice of its execution time $\tau(s)$. Pair $\langle s, \tau(s)\rangle$ means computing and materializing the sub-plan s at time $\tau(s)$, which can only benefit the computation that happens after $\tau(s)$. For instance, considering the physical plan space in Fig. 4.12(a), the sharable candidates are $\{\langle ①, t_1\rangle, \langle ①, t_2\rangle, \langle ②, t_1\rangle, \langle ②, t_2\rangle\}$. The optimizations in [62] are still applicable to Algo. 5.

Algorithm 5 Greedy Algorithm for Choosing Optimal States to Materialize

```
1: 𝕊 = ∅
2: ℂ = shareable candidate set consisting of all shareable nodes and their potential execution
   time points {⟨s, τ(s)⟩}
3: while ℂ ≠ ∅ do
4:     Pick ⟨s, τ(s)⟩ ∈ ℂ that minimizes c̃(bestPlan(𝕊′)) where 𝕊′ = {⟨s, τ(s)⟩} ∪ 𝕊
5:     if c̃(bestPlan(𝕊′)) < c̃(bestPlan(𝕊′)) then
6:         ℂ = ℂ − {⟨s, τ(s)⟩}
7:         𝕊 = 𝕊′
8:     else
9:         ℂ = ∅
10: return 𝕊
```

As expanded with execution time options, the enumeration space of the shareable candidate set becomes much larger than the original algorithm in [62]. Interestingly, we find that under certain cost models we can reduce the enumeration space down to a size comparable to the original algorithm, formally summarized in Theorem 4.1. This theorem relies on

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

Theorem 4.1. *For an extended cost function $\tilde{c}_w$ satisfying $w_i < w_j$ if $i < j$, or an extended cost function $\tilde{c}_v$ satisfying the property that an entry i has a lower priority than an entry j if $i < j$ in the lexical order, we only need to consider the earliest valid execution time for each shareable sub-plan. That is, for each shareable sub-plan s, we only need to consider the shareable candidate $\langle s, min(t\text{-}dom(s))\rangle$ in Algorithm 5.*

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

4.7 Integrating into Traditional Query Optimizers

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

We implemented Tempura based on Apache Calcite. Without loss of generality, we use Calcite's terminologies in this section: an operator is called a *RelNode*, and a logically equivalent group of operators is called a *RelSet*. A *Trait* represents a physical property of physically equivalent classes. On top of these, Tempura introduces a new data structure called *TvrMetaSet* to store relevant information about a TVR: the time domain of the TVR, Intra-TVR relationships, and Inter-TVR relationships. Next we elaborate more on these using the subquery sales_status from Example 4.1 as an example.

TVR Time Points and Intervals By now we used single time points to identify data versions, in which all (intermediate) results are computed from input relations all at the version of the same time point. Whereas in many computing methods, one need to reason about results computed from input relations at different time points. For instance in both Outer-Join View Maintenance and Higher-Order View Maintenance (both described earlier in Section 4.4.2), to model a partial update of relation S from t_1 to t_2 in $S \bowtie^{lo} R$ with R unchanged, we need to represent the join result of S at t_2 and R at t_1, or vice versa.

Consequently, Tempura keeps track of the time version of every input relations respectively to allow for incremental computation using combinations of input relations at different time points. For a query with k input relations $[I_1, \cdots, I_k]$, we define a TVR time point to be a vector $\vec{t} = [t^1, \cdots, t^k]$. Each time point in the vector represents the time version of the k-th input relation. For example, for a query with two input relations R and S, the TVR time point $[t_1^R, t_2^S]$ represents a state where the result is computed from R in version t_1 and S in version t_2. A TVR time interval is defined as $(\vec{t}, \vec{t'})$, the interval between two TVR time

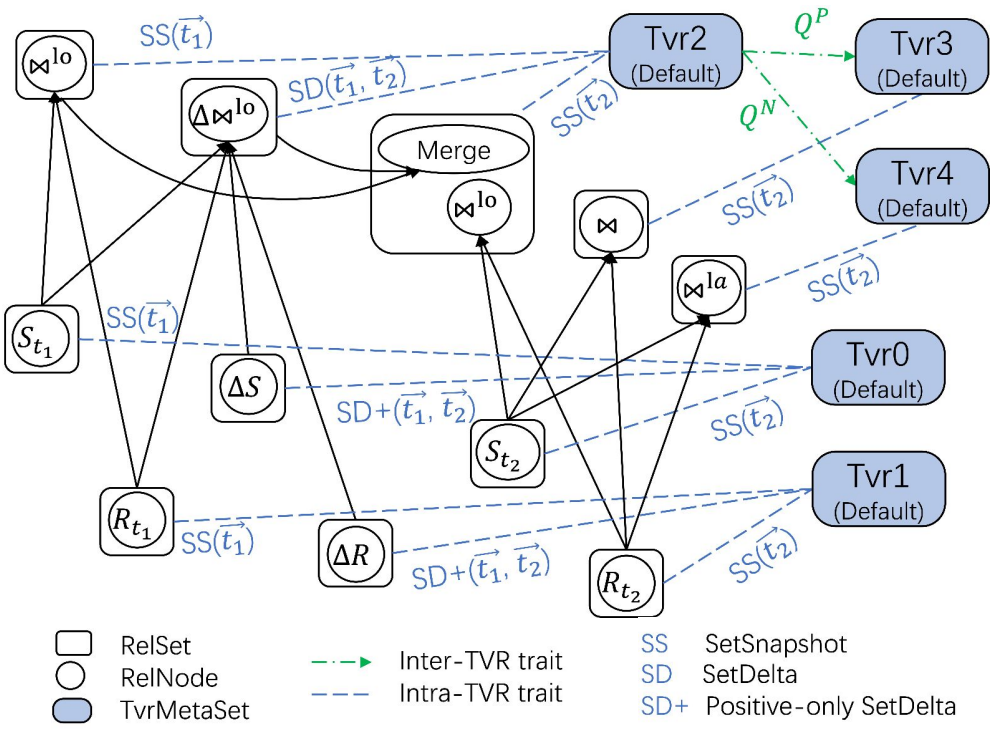


Figure 4.13: Partial memo of subquery sales_status from Example 4.1 in Tempura.

points $\vec{t} = [t^1, \cdots, t^k]$ and $\vec{t'} = [t'^1, \cdots, t'^k]$, where $\forall i\ t^i \leq t'^i$. When the context is clear, we use $\vec{t}$ to denote the TVR time point $\vec{t} = [t, \cdots, t]$. For instance in Fig. 4.13, all five TVRs contain two TVR time points $\vec{t_1} = [t_1^R, t_1^S]$ and $\vec{t_2} = [t_2^R, t_2^S]$.

Time Domain of a TVR The time domain $\mathcal{T}$ of a TVR (introduced earlier in Section 4.3.1) defines relevant time points of the TVR. Specifically, it consists of a list of valid TVR time points and intervals, specified in a data structure called *TvrMetaSetType*. Tempura allows a TVR to have an incomplete time domain, which means not all TVR time points and intervals are required to be present in a TVR.

With two input relations R and S and three time points of $[t_1, t_2, t_3]$, Fig. 4.14 visualizes two types of TVR meta set: *Default* and *Partial*, where each blue point is a valid TVR time point and each yellow arrow is a valid TVR time interval in the time domain of the TVR.

1. The Default TvrMetaSetType only allows TVR time points where all input relations are

at the same time version. On top of that, it only allows TVR time intervals involving adjacent TVR time points to avoid a combinational number of delta intervals. For example in Fig. 4.14(a), there are three TVR time points $[t_1^R, t_1^S]$, $[t_2^R, t_2^S]$, and $[t_3^R, t_3^S]$, and two intervals ($[t_1^R, t_1^S]$, $[t_2^R, t_2^S]$) and ($[t_2^R, t_2^S]$, $[t_3^R, t_3^S]$). This policy has a complexity linear to number of time points, which helps limit exploration space and improve optimization speed.

2. The Partial TvrMetaSetType of certain input relation (e.g. R or S) only allows TVR time intervals where only the corresponding input relation is updated. For example, assuming R is always updated before S in very time step, then Fig. 4.14(b) shows the partial TvrMetaSetType on updating relation R only, where only two TVR time intervals ($[t_1^R, t_1^S], [t_2^R, t_1^S]$) and ($[t_2^R, t_2^S], [t_3^R, t_2^S]$) are allowed. Similarly, Fig. 4.14(c) shows the partial TvrMetaSetType on updating relation S only. Note that although each partial type is incomplete, the two partial types can work together to constitute a valid update path from $[t_1^R, t_1^S]$ to $[t_3^R, t_3^S]$. This policy is useful for incremental computation algorithms that only consider updating one input relation at a time, such as Outer-Join View Maintenance and Higher-Order View Maintenance.

In the memo example in Fig. 4.13, all TvrMetaSets are of the Default TvrMetaSetType, with two TVR time points $\vec{t_1}$ and $\vec{t_2}$, and one interval $(\vec{t_1}, \vec{t_2})$.

Intra-TVR traits A TVR has a mapping from its time domain $\mathcal{T}$ to many relations, e.g. snapshots and deltas. A TvrMetaSet stores these Intra-TVR relationships using Intra-TVR traits, which are bidirectional edges between a TvrMetaSet and a RelSet. For example, Fig. 4.13 plots Intra-TVR traits in blue dotted lines, which connect TvrMetaSets to their related RelSets. Custom Intra-TVR traits can be defined and used by various incremental computing methods. Earlier in Section 4.3 we described the multiplicity and attribute perspectives of a TVR. They corresponds to the example Intra-TVR traits as follows.

1. *SetSnapshot.* This Intra-TVR trait represents that a RelSet is a multiplicity perspective

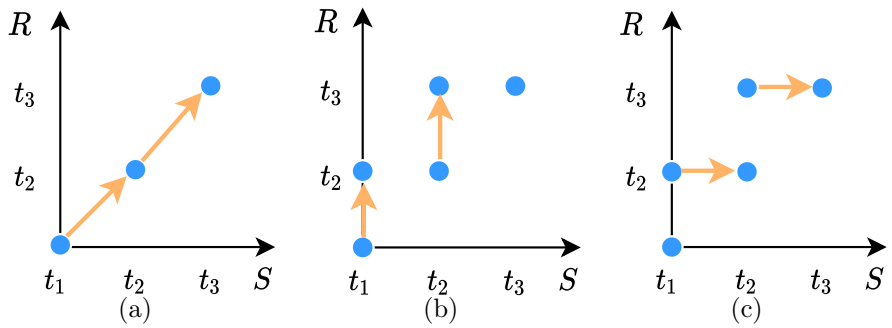


Figure 4.14: TvrMetaSetType Examples: (a) Default, (b) Partial updating R only, and (c) Partial updating S only. Each blue point is a valid TVR time point and each yellow arrow is a valid TVR time interval in the time domain of the TVR.

snapshot at a TVR time point of the connecting TvrMetaSet. The specific TVR time point is stored in the SetSnapshot trait.

2. *SetDelta.* This Intra-TVR trait represents that a RelSet is a multiplicity perspective delta for a TVR time interval. The specific TVR time interval is stored in the Intra-TVR trait. Additionally, SetDelta has two variations, namely positive-only SetDelta and retractable SetDelta. Each variation has a different merge function (see Definition 4.3 in Section 4.3.2) for snapshots and deltas. For retractable delta, the information of the specific column that encodes insertion or deletion is stored in the SetDelta trait.
3. *ValueSnapshot.* This Intra-TVR trait represents that a RelSet is an attribute perspective snapshot at a TVR time point. An attribute perspective snapshot is produced by an aggregation operator. It can be converted to a SetSnapshot by applying the *Final* aggregation function (see Section 4.3.2). The information of the conversion is stored in the ValueSnapshot trait, including the group-by keys and the *Final* aggregation functions.
4. *ValueDelta.* This Intra-TVR trait represents that a RelSet is an attribute perspective delta for a TVR time interval. Similar to SetDelta, there are positive-only ValueDelta and retractable ValueDelta. Similar to ValueSnapshot, necessary information on conversions to SetSnapshot is also stored in ValueDelta.

Inter-TVR Trait. A TvrMetaSet stores Inter-TVR relationships using Inter-TVR traits,

which are directed edges between two TvrMetaSets. Custom Inter-TVR traits can be defined to annotate the information needed by an incremental computation algorithm. For example in Fig. 4.13, the green Q^P and Q^N edges are two Inter-TVR traits used in the `IM-2` approach.

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

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

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

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

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

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

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

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

Tempura designates one time point as the special anchor time point $t\!\downarrow$ of a TVR. The anchor time point must ensure that all rules applied at the anchor time point are time-independent and can be generalized to all time points of a TVR.

$$R'_{t\downarrow} = R_{t\downarrow} \wedge \mathcal{T}(R') = \mathcal{T}(R) \implies R' = R.$$

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

4.8 Improving Query Optimization Speed

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

4.8.1 Translational symmetry of TVRs

Generating a plan for many time points imposes a challenge for the optimization speed. With an increasing number of time points, the memo size and the overhead of the rule pattern matching and firing grows larger. We have an observation that the TVR rules generate the same patterns when applied on operators of different time points of the same TVR. For instance, in Fig. 4.11(b), if we let $t' = t_1$ instead, L' (R') matches G0 (G1) instead of G3 (G4), and we generate the *InnerJoin* in G7 instead of G8. In other words, *InnerJoin*[0,1] in G7 and *InnerJoin*[3,4] are translation symmetric, modulo the fact that G0, G1, and G7 (G3, G4, and G8) are all snapshot t_1 (t_2) of the corresponding TVRs.

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

Template Copying. Before the copying starts, we need to first decide a template and a

copy mapping. Out of all time points and intervals, we first choose one consecutive pair of a time point t for snapshot and a time interval (t, t') for delta as the copying template. Then we define a copy mapping from the template time point/interval to the rest of time points/intervals. For example, if there are three time points $[t_1, t_2, t_3]$ and two time intervals $[(t_1, t_2), (t_2, t_3)]$, we could choose t_1 as the template time point and (t_1, t_2) as the template time interval. The copying mapping for time point is $t_1 \mapsto \{t_2, t_3\}$ and the mapping for time interval is $(t_1, t_2) \mapsto \{(t_2, t_3)\}$. Next, we explain the template generation phase and the copying phase step-by-step.

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

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

Time-dependent rules generates operators that are based on time-specific properties that vary across time. For example, an input relation being empty at time point t_1 does not

imply that the relation will be empty for at all times. If an empty pruning rule is applied to an operator at t_1, it cannot be generalized to the entire TVR. As a result, time dependent rules cannot be fired when constructing the template. Tempura defers the firings of time-dependent rules after the copying phase has ended. Recall that the rule engine performs rule matching for every structrual change in the memo. Tempura always enables such rules for pattern matching during the template generation and copying phase, but any successful matches are put into a separate queue for deferred firing after the copying phase has ended.

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

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

4.8.2 Pruning Plan Exploration Space.

Pruning non-promising alternatives. There are multiple ways to compute a TVRs snapshot or delta, within which certain ways are usually more costly than others. We can prune the non-promising alternatives. For instance, to compute a delta, one can take the difference of two snapshots, or use TVR-generating rules to directly compute from deltas of the inputs. Based on the experience of previous research on incremental computation [63], we know that the plans generated by TVR-generating rules are usually more efficient. Therefore,

for operators that are known to be easily incrementally maintained, such as filter and project, we assign a lower importance to intra-TVR rules for generating deltas to defer their firing. Once we find a delta that can be generated through TVR-generating rules, we skip the corresponding intra-TVR rules altogether. To implement this optimization, we can give this subset of intra-TVR rules a lower priority than all other rules, and thus other TVR rewrite rules and traditional rewrite rules will always be ranked higher. Each intra-TVR rule also has an extra skipping condition, which is tested to see whether the target delta is already generated before firing the rule. If so, the rule is skipped.

Guided exploration. Inside a TVR, snapshots and deltas consecutive in time can be merged together, leading to combinatorial explosion of rule applications. However, the merge order of these snapshots and deltas usually do not affect the cost of the final plan. Thus, we limit the exploration to a left-deep merge order. Specifically, we disable merging of consecutive deltas, and only allow patterns that merge a snapshot with its immediately consecutive delta. In this way, we always use a left-deep merge order.

4.8.3 Optimization of the Rule Engine

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

For example, the TVR-generating rule in Fig. 4.11(a) matches five operators, three TvrMetaSets, and seven edges. The inter-TVR rule in Fig. 4.11(b) matches five operators, five TvrMetaSets,

and nine edges. Rule matching in Tempura is a subgraph isomorphism problem that matches the given rule pattern against the memo graph. The subgraph isomorphism problem is NP-complete and could bring a major performance overhead. In this section, we first explain the general rule matching and firing process in Tempura, then show how Tempura speeds up the rule matching process using techniques including indexing, pre-compliation, and multiple heuristics on match order.

In Tempura, the rule matching process is triggered by any structural changes in the memo, for example, adding a new operator, TvrMetaSet, or edge, including intra/inter-TVR edges and edges between operators. Merging of RelSets or TvrMetaSets is also a structural change. For each rule, Tempura tries to match it starting from the location of the triggering change. As shown in Fig. 4.11(b) starting from the example triggering TvrMetaSet vertex, Tempura follows a depth first search (DFS) matching order in the rule pattern. Whenever the matching fails at one point, it backtracks and moves on to the next candidate in the traversal. Upon finding a successful match, the rule with all matched vertices and edges are added to a rule queue, waiting to be applied.

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

Determining Matching Order. The matching order has a major impact on optimizer

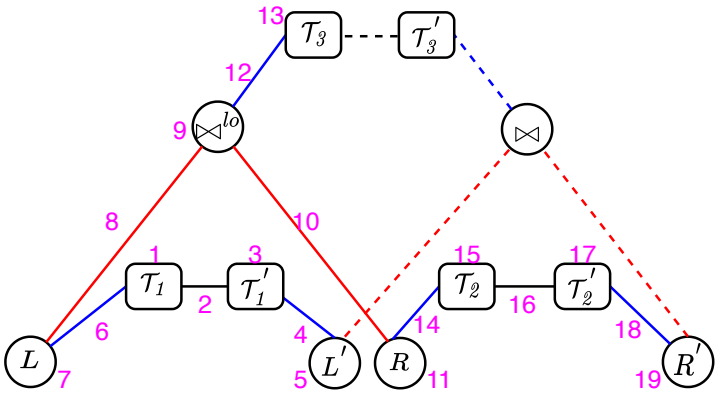


Figure 4.15: A possible matching order of the rule in Fig. 4.11(b) starting from a TVR vertex.

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

1. *Operator Vertex*: At an operator vertex, we can either match connected operators or connected TVRs. Tempura prioritizes on matching operators. It first follows Calcite's traversal order for matching the operator tree, which is to travel up to root and then a pre-order traversal for the rest of the operators. When the operator tree is fully matched, it then follow the intra-TVR edges to expand the search to connected TVRs. The rationale is that operators associated with many predicates are less likely to find a match and thus can abort early.
2. *TVR Vertex*: At a TVR vertex, we can either match operators belong to this TVR, or other connected TVRs via inter-TVR edges. Tempura prioritizes checking the inter-TVR trait and the connected TVRs. After matching all inter-TVR edges and connected TVRs, we then traverse the inter-TVR edges to match operators. This is because inter-TVR traits are less likely to find a match, which can cause the rule matching to terminate earlier.

3. *Intra-TVR Edge*: At an Intra-TVR edge, which connects an operator and a TVR, we need to choose the match order when both connected operator and TVR are not matched yet, i.e. the Intra-TVR edge itself is the triggering point. In this case, we can either match the connected operator first, or the connected TVR first. Tempura prioritizes on matching the connected TVR because this enables expanding the inter-TVR edges faster.
4. *Inter-TVR Edge*: At an Inter-TVR edge, which connects two TVRs, we need to prioritize which side to match first. Tempura compares the number of inter-TVR edges of the two TVRs, and prioritizes the TVR with more inter-TVR edges.

Fig. 4.15 shows a possible match order of the example rule starting from a TVR vertex.

4.9 Tempura in Action

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

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

Data statistics estimation. IQP scenarios usually involve planning for future logical times (e.g., `IVM-PD`) or physical times (e.g., `PWD-PD`) as described in Section 4.2.1, for which estimating the data statistics becomes very challenging. Since these scenarios typically involve recurring queries, we can use historical data arrival patterns to estimate future data statistics. Having inaccurate statistics is not a new problem to query optimization, and many techniques have been proposed [113] to tackle this issue. Note that we can always re-optimize the plan when we find that the previously estimated statistics is not accurate. Also, techniques such as robust planning [24, 53, 111] can be adopted to IQP too. These are out of the scope of this thesis.

4.10 Experiments

In this section, we study the effectiveness and efficiency of Tempura. We used the query optimizer of Alibaba Cloud MaxCompute [18], which was built on Apache Calcite 1.17.0 [19], as a traditional optimizer baseline. We implemented Tempura on the optimizer of MaxCompute. We integrated four commonly used incremental methods into Tempura using TVR-rewrite rules: (1) **IM-1** in Section 4.2.2, (2) **IM-2** in Section 4.2.2 and Section 4.4.2, (3) **OJV** the outer-join view maintenance algorithm in Section 4.4.2, (4) **HOV** the higher-order view maintenance algorithm in Section 4.4.2. By default, Tempura jointly considered all four methods in planning. In the experiments, we used Tempura to simulate each method by turning off the inter-TVR rules of the other methods.

We used two incremental processing scenarios, **`PDW-PD`** and **`IVM-PD`** described in Section 4.2.1, to demonstrate Tempura. `PDW-PD` uses the cost function $\tilde{c}_w(O)$ (in Section 4.6.2), where c_i was a linear function of the estimated CPU/IO/memory/network costs, and $w_i \in [0.25, 0.3]$ for early runs and $w_i = 1$ for the last run. The weight values for early runs are determined based on the typical resource utilization of cluster at non-peak hours Alibaba. This is done to simulate the execution of early computations during off-peak periods.

We used the TPC-DS benchmark [101] ($1TB$) to study the effectiveness (Section 4.10.1) and performance (Section 4.10.3) of Tempura. To further demonstrate the effectiveness of the plans, in Section 4.10.2 we used two real-world analysis workloads consisting of recurrent daily jobs from Alibaba's enterprise data warehouse, denoted as `W-A` and `W-B`.

Table 4.1 shows statistics of the two workloads.

Table 4.1: Statistics of two workloads at Alibaba

	# Queries	Avg. # Joins	Avg. # Aggregates	# Queries ($\geq$ 1 join)	# Queries ($\geq$ 2 joins)
`W-A`	274	1.14	1.77	167	83
`W-B`	554	1.18	1.99	357	144

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

4.10.1 Effectiveness of IQP

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

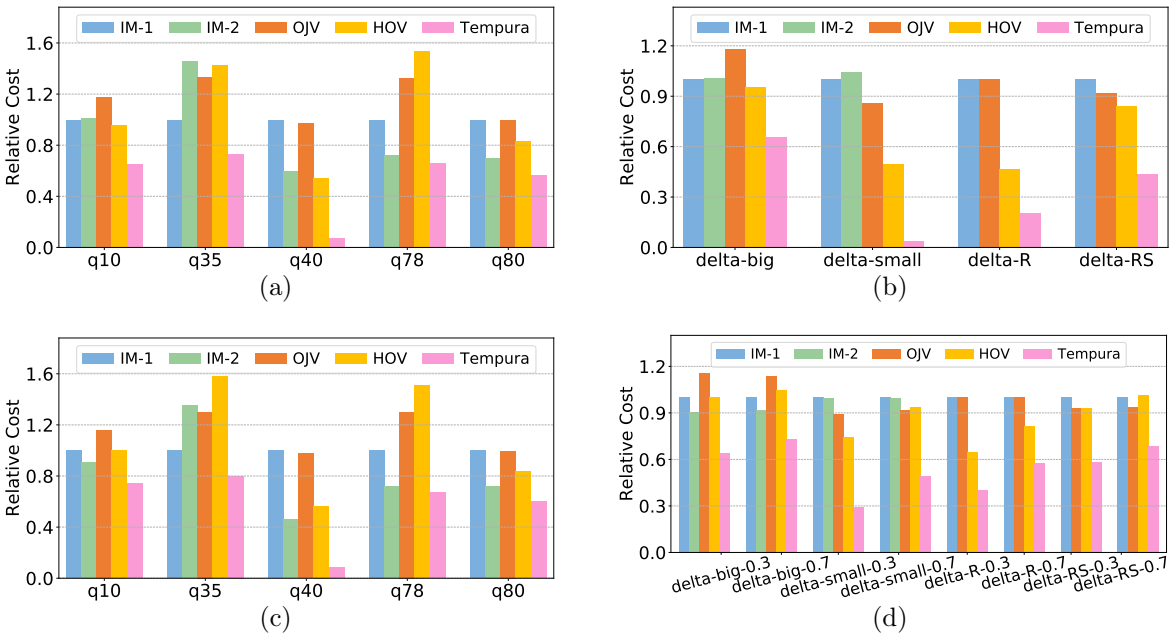


Figure 4.16: (a)(b) The optimal estimated costs of incremental plans in `IVM-PD` for different queries and data-arrival patterns. (c)(d) The optimal estimated costs of incremental plans in `PDW-PD` for different queries, data-arrival patterns and cost weights.

IVM-PD. We first fixed the data-arrival pattern to delta-big and varied the queries. The optimal-plan costs are reported in Fig. 4.16(a). As shown, different queries prefer different incremental methods. For example, `IM-1` outperformed both `OJV` and `HOV` for complex queries such as q35. This is because `OJV` computed Q^I by computing the left-semi join of the delta of Q^D with the previous snapshot (Section 4.4.2), and a bigger delta incurred a higher cost of computing Q^I. Whereas for simpler queries such as q80, `OJV` degenerated to a similar plan as `IM-1`, and thus had similar costs. Note that `HOV` costs much less than both `OJV` and `IM-1`, due to the fact that the maintained higher-order views avoid many repeated joins (e.g., catalog_sales inner joining warehouse, item and date_dim) as in `OJV` and `IM-1`.

Next we chose q10 as a complex query with multiple left outer joins, and varied the data-arrival patterns. The results are plotted in Fig. 4.16(b). Again, the data-arrival patterns affected the preference of incremental methods. For example, `IM-2` could not handle input data with retractions. Compared to delta-big, `HOV` and `OJV` started to outperform `IM-1` by

a large margin in delta-small, as both of them could use different join orders when applying updates to different input relations, and joining a smaller delta earlier could significantly reduce the join cost.

For both experiments, Tempura consistently delivered the best plans. For q40 in Fig. 4.16(a) and the delta-small case in Fig. 4.16(b), Tempura delivered a plan 5-10X better than others. Tempura combines all three of `HOV`, `IM-2` and `IM-1` to generate a mixed optimal plan, and thus leveraged all their advantages. E.g., in q40 Tempura used a similar incremental plan to `HOV`, but Tempura used the `IM-2` approach to join the higher-order views M and ΔR, and applied `IM-1` to incrementalize the Q^N part in `IM-2`.

PDW-PD. For the `PDW-PD` scenario, we conducted the same experiments as in `IVM-PD`, and in addition tried different weights used in the cost functions ($w_1 = 0.3$ vs. $w_1 = 0.7$). We have similar conclusions as in `IVM-PD`, and the results are reported in Figures 4.16(c) and 4.16(d). We make two remarks. (1) Since `PDW-PD` did not require any outputs at earlier runs, Tempura automatically avoided unnecessary computation, e.g., `IM-2` avoided computing the Q^N part, and thus performed better for q10, q35, q40 than in `IVM-PD`. (2) The cost function can also affect the choice of the optimizer. For instance, in Fig. 4.16(d), q10 preferred `HOV` to `OJV` when $w_1 = 0.3$, but the other way when $w_1 = 0.7$. This was because with the cost of early execution increasing, it was less preferable to store many intermediate states as in `HOV`. Tempura exploited this fact and adjusted the computation in each run, and moved some early computation from the first incremental run to the second.

Real CPU Costs. We reported the real CPU costs in Fig. 4.17(a)-4.17(d) for the experiments in Fig. 4.16(a)-4.16(d). The CPU costs are in the unit of `number_of_cores` · `time_of_each_core_in_minutes` ("CPU·Min" in short). As an example, if a query runs with 2 cores for 3 minutes, the total CPU cost is 6 CPU·Min's. In the `PDW-PD` experiments (Figs. 4.17(c) and 4.17(d)), the CPU costs were weighted with the cost function in `PWD-PD`.

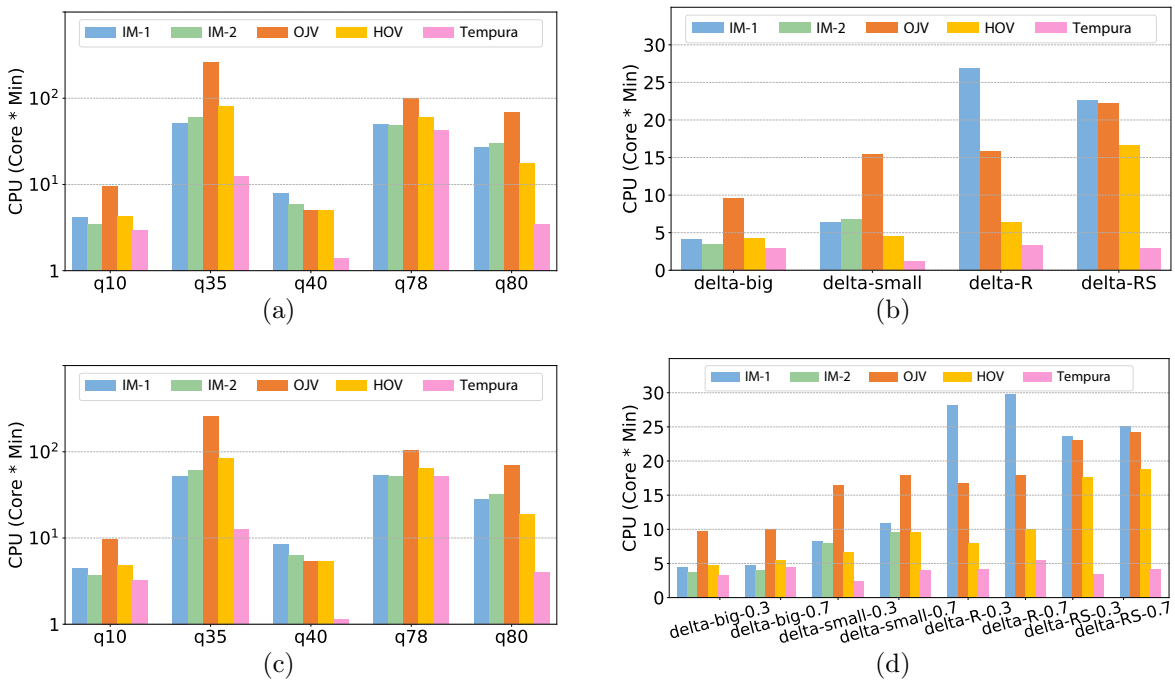
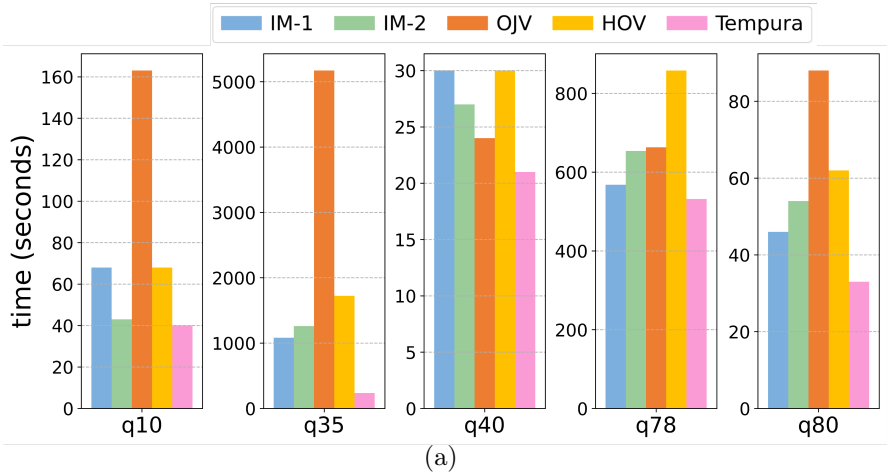


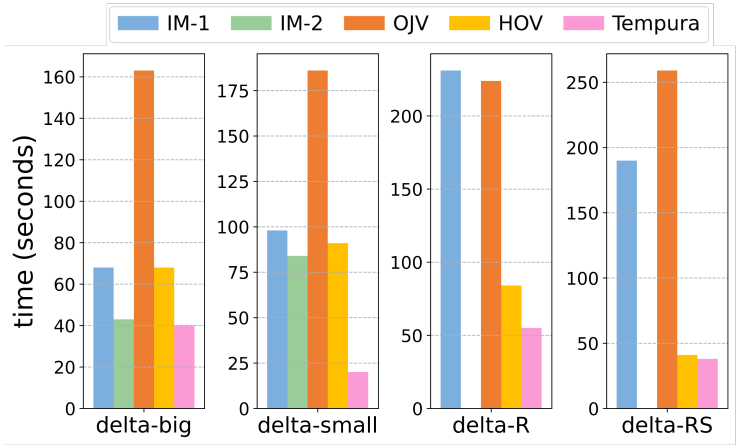
Figure 4.17: (a)(b) The optimal real CPU costs of different incremental plans in `IVM-PD` for different queries and data-arrival patterns. (c)(d) The optimal real CPU costs of different incremental plans in `PDW-PD` for different queries, data-arrival patterns and cost weights.

Note that Fig. 4.17(a) and 4.17(c) are plotted in log scale due to the huge differences in CPU costs for different queries. As we can see, the real CPU costs agreed with the planner's estimation (Fig. 4.16(a)-4.16(d)) pretty well. Some of the real costs were different from the estimated ones because of the inaccuracy of the cost model. But note that Tempura consistently delivered the best plans with the lowest CPU consumption across all experiments.

Real Wall-clock Execution Time. We reported the real wall-clock execution time (in seconds) in Fig. 4.18 in the `IVM-PD` scenario. Due to the huge differences in execution time for different queries, each query is plotted with a separate scale on the y-axis. We noticed that the initial runs of a few jobs failed and the jobs were restarted by the fault-tolerance mechanism of the platform. To make the comparison fair, we only counted the execution time of successful runs and excluded the additional time of the failed runs. The real wall-clock execution time was mostly similar to the real CPU costs (Figs. 4.17(a)-4.17(b)). Tempura



(a)



(b)

Figure 4.18: (a) The wall-clock execution time in `IVM-PD` for different queries (corresponding to Fig. 4.17(a)). (b) The wall-clock execution time in `IVM-PD` for different data-arrival patterns for TPC-DS q10 (corresponding to Fig. 4.17(b)).

still consistently delivered the lowest wall-clock execution time across all experiments. Some of the wall-clock execution times were different from the CPU costs because some plans were easier to be parallelized and the query optimizer assigned a higher degree of parallelism to such queries. In such a case, although the wall-clock execution time was lower, the total CPU costs could be similar as more cores were used.

State Sizes. In this set of experiments, we study the storage costs of materialized states between Tempura and each individual incremental methods. We first fixed the data-arrival pattern to delta-big and tested different queries under `IVM-PD` settings respectively. The re-

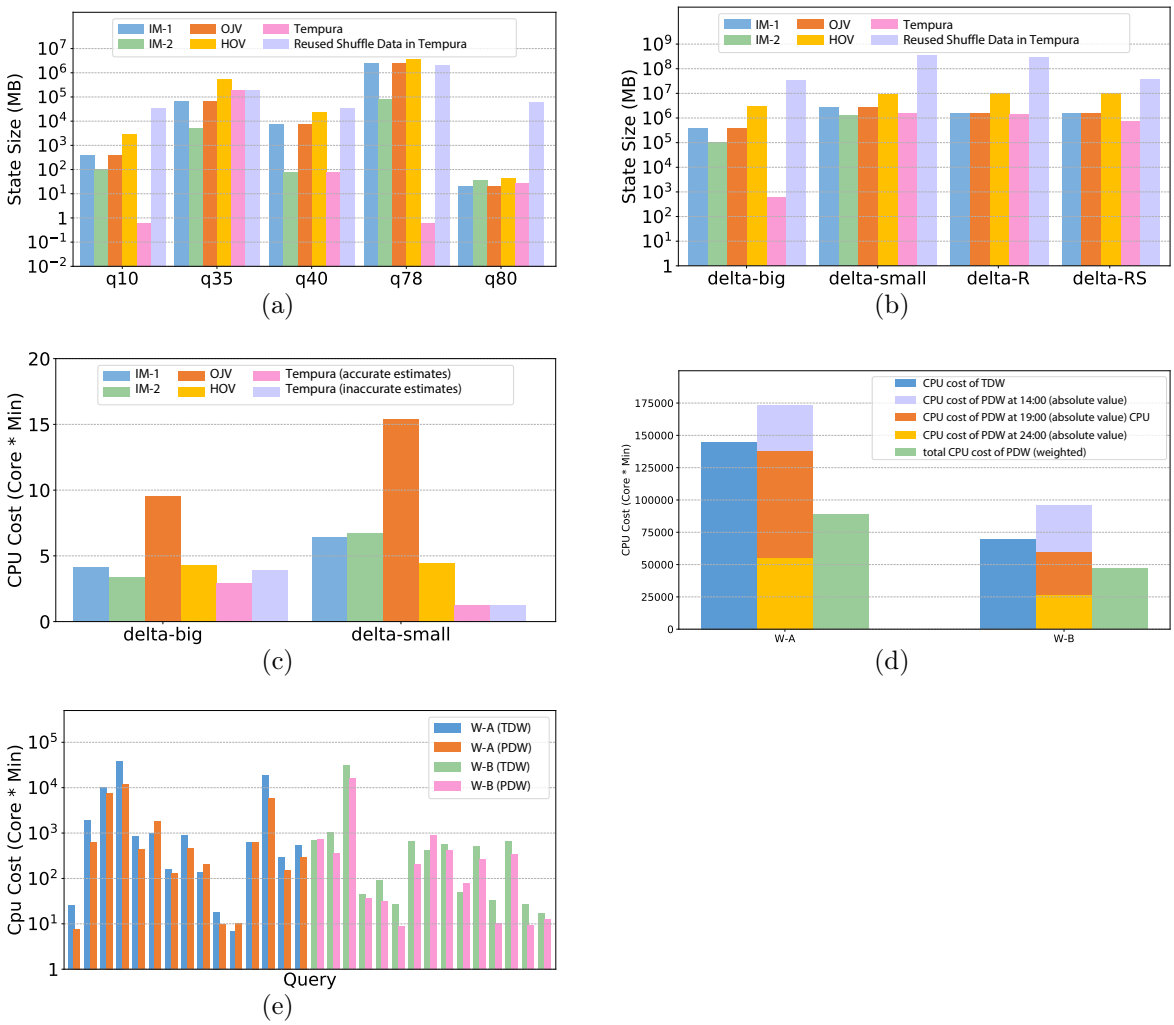


Figure 4.19: (a)(b) The state sizes of different incremental plans in `IVM-PD` for different queries and data-arrival patterns. (c) The plan quality of Tempura under inaccurate cardinality estimation. (d) The comparison between TDW and PDW on the CPU cost of all queries in `W-A` and `W-B`, and (e) a detailed comparison of 30 randomly sampled queries in `W-A` and `W-B`.

sults are reported in Fig. 4.19(a). As shown, for most queries, the sizes of states materialized by Tempura were smaller than or comparable to each individual incremental algorithms. This is due to the fact that Tempura is able to reuse the shuffled data as the states without incurring additional storage overheads (see Section 4.6.1). Thus, we further reported the sizes of the shuffled data reused by Tempura in the figures. Next we chose query q10 and varied the data-arrival patterns. The results are reported in Fig. 4.19(b). Again, the stor-

age costs of Tempura were lower than or comparable to that of each individual incremental algorithms.

Sensitivity to Inaccurate Estimates. Next, we evaluated the sensitivity of Tempura to inaccurate cardinality estimation. We used q10 in the `IVM-PD` scenario. We gave Tempura the estimation of delta-small when running q10 with input delta-big, and gave the estimation of delta-big when running q10 with input delta-small. Fig. 4.19(c) reported the real CPU costs. For delta-big, Tempura with the inaccurate estimation ran slower compared to Tempura with accurate estimation. This is expected because Tempura chose a plan that is optimal to the inaccurate cost model. Nevertheless, Tempura was still faster than `IM-1`, `OJV`, `HOV`, and comparable to `IM-2`. For delta-small, inaccurate estimation had a small impact on execution time, and Tempura was still faster than each individual incremental method.

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

4.10.2 Case Study: Progressive Data Warehouse

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

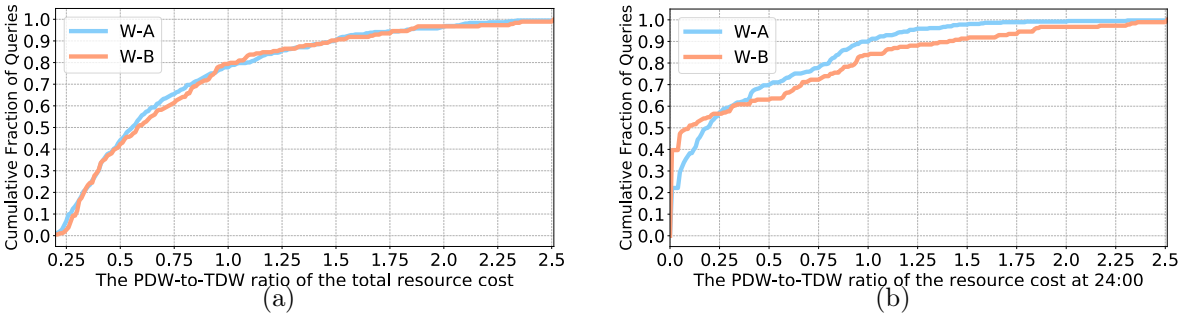


Figure 4.20: (a)(b) The PDW-to-TDW ratio of the real total CPU cost and CPU cost at 24:00 for the data warehouse workloads respectively.

Fig. 4.20(a) shows the real CPU cost of executing the workloads (scored using the cost function in the `PDW-PD` setting), where we plotted the cumulative distribution of the ratio between the CPU cost in PDW versus that in TDW. We can see that PDW delivered better CPU cost for 80% of the queries. For about 60% of the queries, PDW was able to cut the CPU cost by more than 35%. Remarkably, PDW delivered a total cost reduction of 56.2% and 55.5% for `W-A` and `W-B`, respectively. Note that Tempura searched plans based on the estimated costs which could be different from the real execution cost. As a consequence, for some of the queries (less than 10%) we see more than 50% cost increase. Accuracy of cost estimation is not within the scope of the thesis. We further reported the PDW-to-TDW ratio of the CPU cost at 24:00 in Fig. 4.20(b), as this ratio indicated the resource reduction during the "rush hours." As shown, for both workloads, PDW reduced the resource usage at peak hours for over 85% of the queries, and for over 70% of the queries we can see significant reduction of more than 25%.

We also reported the absolute values of CPU costs of `W-A` and `W-B`. However, as `W-A` and `W-B` have 274 and 554 queries each, it is not realistic to show all of them. Instead we reported the total CPU cost breakdowns for TDW and PDW in Fig. 4.19(d). Specifically for PDW, we reported the absolute values of CPU costs at each time, and the total CPU costs weighted according to the cost function in `PDW-PD`. As we can see, Tempura indeed picked better plans with less resource consumption: PDW saved 38.7% and 32.6% CPU costs compared to TDW

for `W-A` and `W-B` respectively. On the other hand, with incremental computation, PDW had relatively low overheads compared to TDW, 19.6% and 37.6% for `W-A` and `W-B` respectively. The PDW overheads are computed by summing up the absolute values of CPU costs at each time, minus the CPU costs of TDW. We further randomly selected 15 queries from `W-A` and `W-B` respectively, and reported their CPU costs in TDW and PDW in Fig. 4.19(e). Again, for most queries PDW reduced the CPU costs by a significant amount.

4.10.3 Performance of IQP

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

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

Table 4.2: Statistics of selected representative queries

Query	Q22	Q20	Q43	Q67	Q27	Q99	Q85	Q91	Q5	Q33
# Joins	2	2	2	3	4	4	6	6	7	9
# Aggregates	1	1	1	1	1	1	1	1	4	4
# Sub-Queries	0	0	0	2	0	0	0	0	7	7

(1) *Query complexity*: How does Tempura perform when queries become increasingly complex, e.g., with more joins or subqueries? (2) *Size of IQP*: How does Tempura perform when $|\vec{T}|$ changes? (3) *Number of incremental methods*: How does Tempura perform when users integrate more incremental methods into it? (4) *Optimization breakdown*: How effective are the speed-up optimizations in Section 4.8?

To study the above four aspects, we selected ten representative TPC-DS queries with different

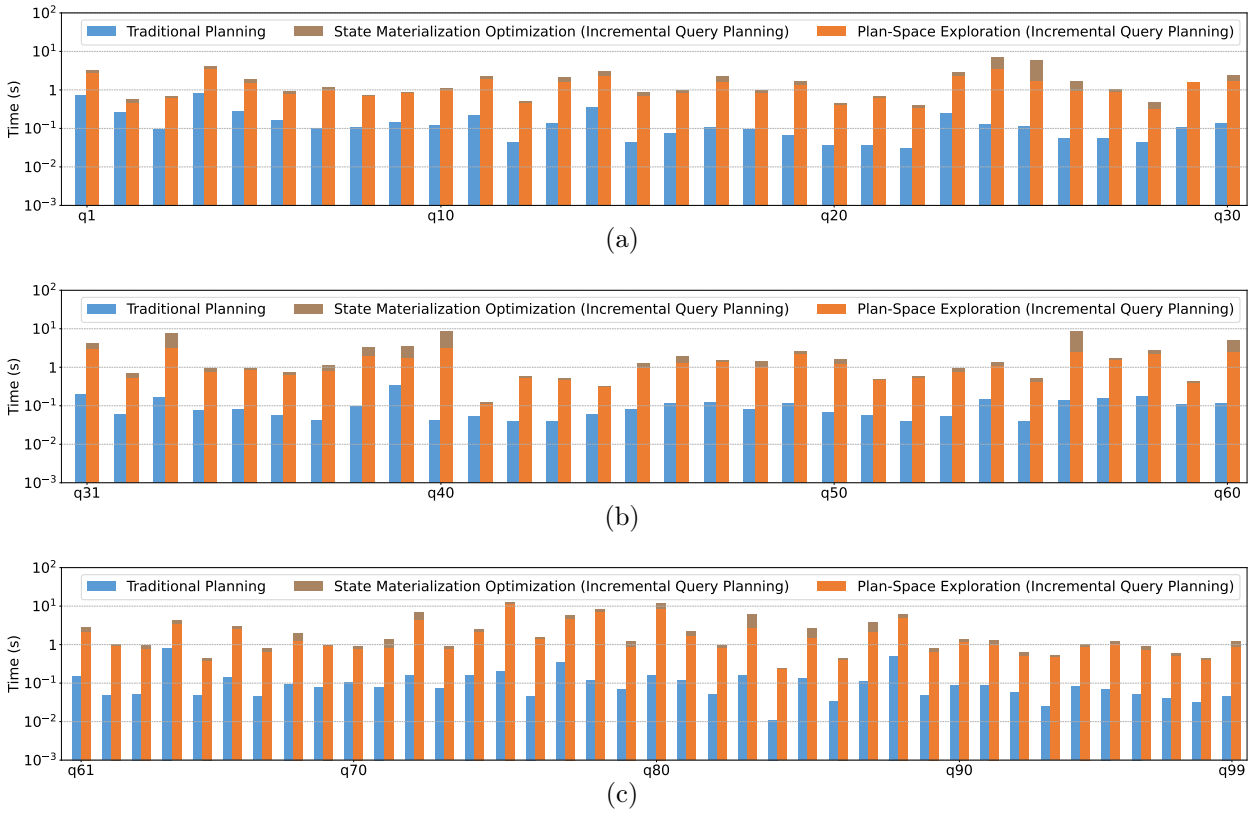


Figure 4.21: Comparing overall planning performance on all TPC-DS queries between traditional and incremental query planning.

numbers of joins, aggregates, and subqueries. The selected queries are shown in Table 4.2.

Overall Planning Performance. We first studied the overall planning performance by comparing Tempura with traditional planning. Fig. 4.21 shows the end-to-end planning time on all TPC-DS queries. As shown, although planned a much bigger plan space, Tempura still delivered high planning performance: IQP finished within 3 seconds for 80% queries, and for all queries finished within 14 seconds. For over 80% queries, the IQP optimization time was less than 24X of the traditional planning time. Even though slower than traditional planning at optimization time, IQP generated much better incremental plans that brought significant benefit in resource usage and query latency. We can further reduce the planning time by adopting a parallel optimizer [94].

As a reference, we also reported the real CPU cost used by TDW, the CPU costs saved by

PDW compared to TDW, and the planning time in Fig. 4.22. We can see that for most queries, the CPU time on planning was 2-3 orders of magnitude smaller than the saved CPU costs. This shows that the planning cost is negligible compared to the execution cost. Thus the benefit of a better plan outweighs the extra time spent on planning.

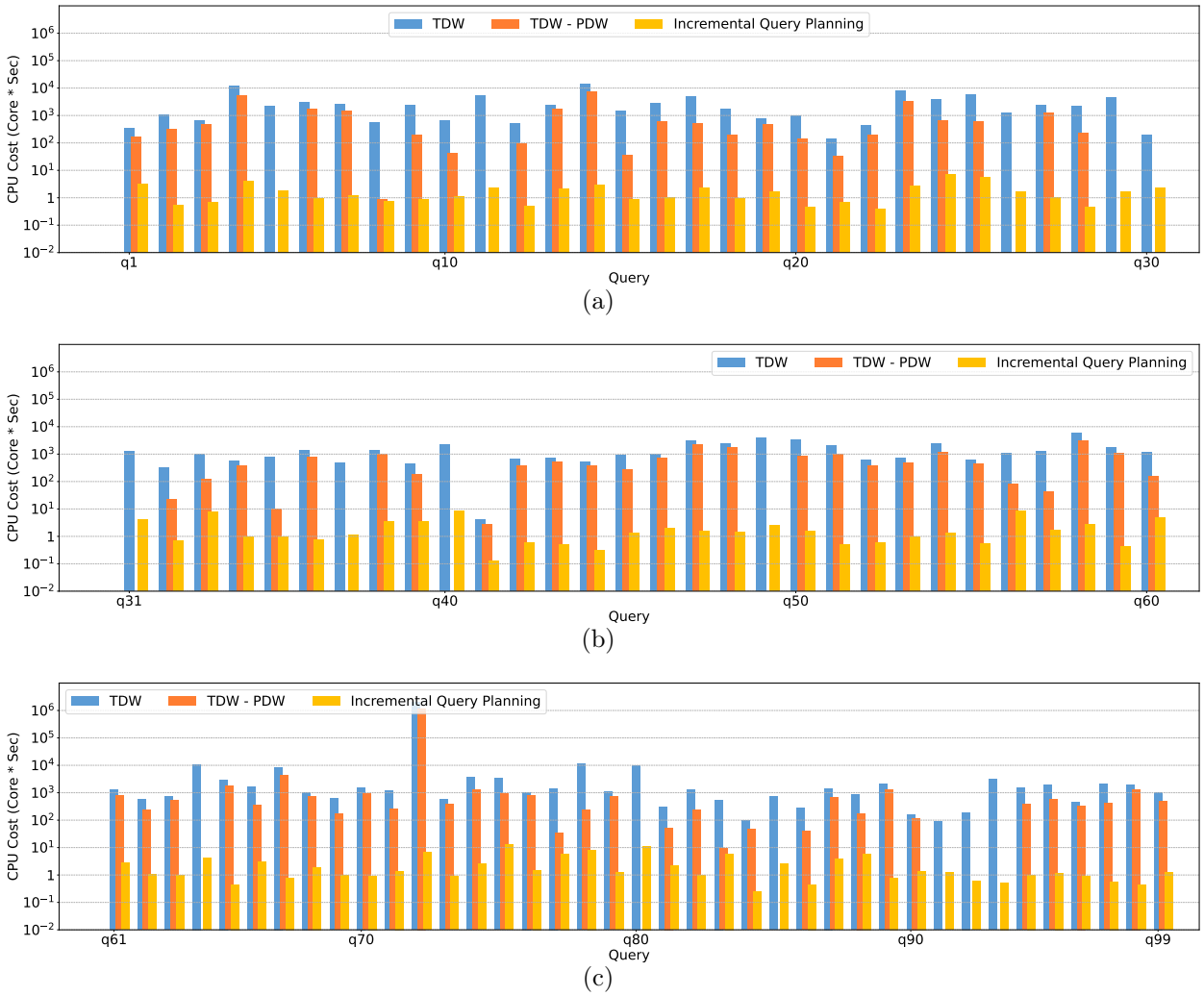


Figure 4.22: Real resource consumption of Tempura's plan as in Fig. 4.21 on all queries in the 1T TPC-DS benchmark.

Query Complexity. To study the impact of query complexity, we reported the planning time break-down on the selected TPC-DS queries in Table 4.2 in Fig. 4.23(a). As shown, the planning time increased when the query complexity increased, because the plan space grew larger for complex queries. The time spent on PSE was less than that spent on SMO in

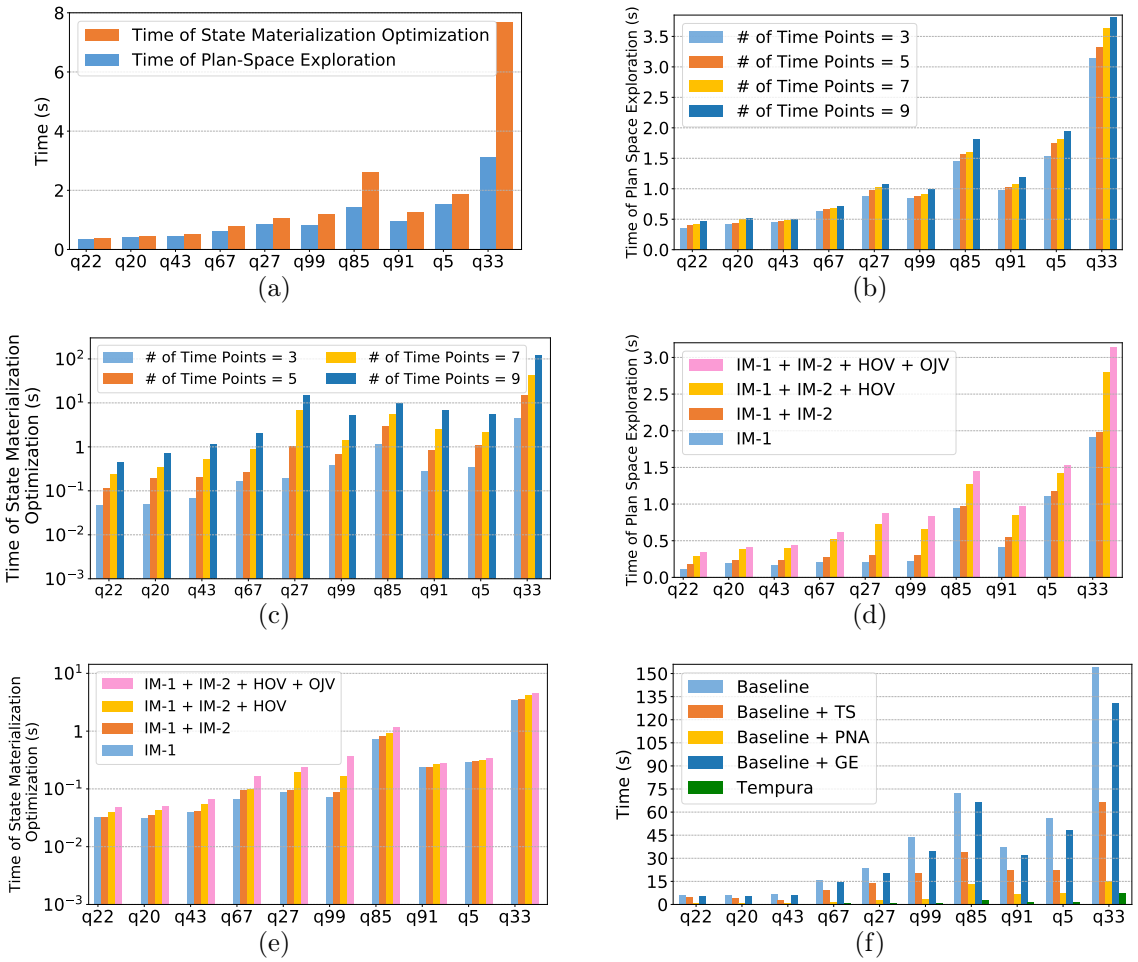


Figure 4.23: Impact of query planning performance of various factors: (a) query complexity, (b) (c) the size of IQP, and (d)(e) the number of incremental methods. (f) Effectiveness of the speed-up optimization techniques. Note that the selected queries are ordered by their query complexity(as listed in Table 4.2).

general, and also grew with a slower pace. This shows that query complexity has a smaller impact on PSE.

Size of IQP. To study the impact of the size of the planning problem, we gradually increased the number of incremental runs planned from 3 to 9, and reported the time on PSE and SMO in Fig. 4.23(b) and 4.23(c). As depicted, the time on PSE stayed almost constant as the size of IQP changed. E.g., when the number of incremental runs grew 3X, the time for q33 only slightly increased by 20%. This was mainly due to the effective speed-up optimization techniques introduced in Section 4.8. In comparison, the SMO time increased

superlinearly with increasing number of incremental runs, due to the time complexity of the MQO algorithm we chose [62].

Number of Incremental Methods. To study the impact of more incremental methods, we gradually added methods `IM-1`, `IM-2`, `HOV` and `OJV` into Tempura. Fig. 4.23(e) and 4.23(f) show the time on PSE and SMO, respectively. As illustrated, the time on both PSE and SMO increased with more incremental methods, due to the increased plan space. There are two interesting findings. (1) The PSE time did not grow linearly with the number of incremental methods, but rather the the plan space size that each method newly introduces. E.g., the increase of PSE time at adding `HOV` was bigger than that at adding `OJV`. This was because both `HOV` and `OJV` update a single relation at a time, which are very different from `IM-1` and `IM-2` that update all relations each time. (2) The number of incremental methods had less impact than the size of the IQP problem, which can be observed on the SMO time. This is because the plan space explored by different incremental methods often have overlaps, whereas the plan spaces of different incremental runs do not.

Exploration Optimization Breakdown.

We evaluated the effectiveness of the speed-up optimizations of exploring the plan space discussed in Section 4.8, i.e., translational symmetry (TS), pruning non-promising alternatives (PNA), and guided exploration (GE). Fig. 4.23(f) reports the PSE times of different combinations of the speed-up optimizations. We compared the implementations with no optimization (Baseline), with each individual optimization (Baseline+TS, +PNA, +GE), and with all three optimizations (Tempura). The optimizations together brought up to 20X speed-up, among which the most effective ones were PNA and TS, bringing 5-12X and 1.5-2.5X improvements each.

Effect of Exploiting TVR Translational Symmetry. We evaluated the memo-copying process using the TVR translational symmetry in Section 4.8.1. Fig. 4.24 shows the break-

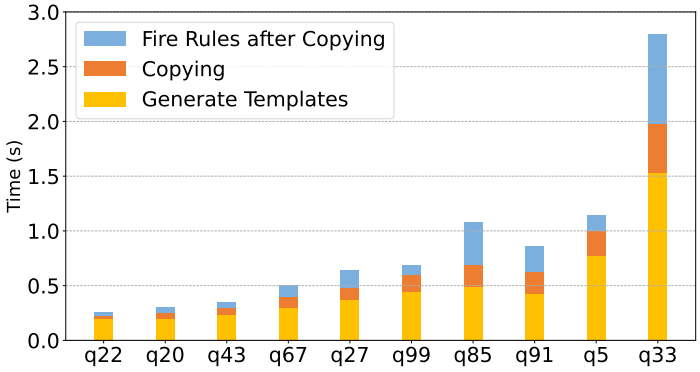


Figure 4.24: Time breakdown of three steps in the memo copying process: template generation, template copying, and firing non-translational symmetric rules after copying.

down of planning time of the three steps used in the copying process, with two time points in the initial template-generation phase, and three additional time points in the template-copying phase. We have the following observations: 1) the time of the template-generation phase varied and it was determined by the complexity of each query; 2) the time spent on copying the template to three additional time points was much less than generating the template on two time points; and 3) the time taken by firing non-translational symmetric rules after the copying were usually small, but it took a long time in a few queries. This is because the new operators generated by non-translational symmetric rules further triggered many traditional rewrite rules such as enforcer rules.

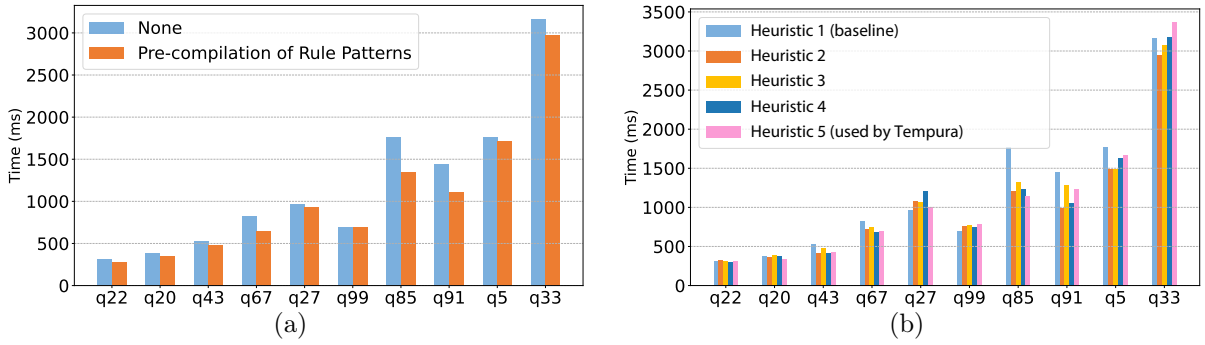


Figure 4.25: Effect of different rule engine optimization techniques on overall planning performance: (a) pre-compilation of rule patterns and (b) different match order heuristics.

Effect of Rule Engine Optimizations. We study the effect of the optimization techniques to speed up the rule-matching process of the rule engine in Section 4.8.3. Fig. 4.25(a)

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

Table 4.3: Configurations of different match order heuristics used in Fig. 4.25(b).

shows the benefit of pre-compilation of rule patterns. We observed that the pre-compilation optimization introduced performance gains on almost all queries tested. Recall that pre-compilation can avoid re-computing the match order on each rule firing. Since the number of rule firings was very large, this optimization cumulatively saved a large amount of time. Fig. 4.25(b) shows the effect of different match-order heuristics on the optimization speed. The specific configuration of each heuristic can be found in table 4.3. We had the following observations. (1) The experiment showed that different match orders indeed had different impacts on the optimization speed. For example, in query 85, heuristic 1 was about 30% slower than heuristic 5. Therefore it is important to choose a good match order to accelerate the optimization. (2) The best heuristic was query-dependent and there was no single heuristic match order that performs the best in all the cases. The heuristic we chose out-performs the baseline for most queries. Based on these observations, Tempura allows a developer to tune the match-order heuristics based on the query and workload.

4.11 Related Work

Incremental Processing. There are rich research works on incremental processing, ranging from incremental view maintenance, stream computing, to approximate query answering and so on. Incremental view maintenance has been studied under both the set [29, 31] and bag [40, 56] semantics, for queries with outer joins [55, 67], and using higher-order maintenance methods [14]. Previous studies mainly focused on delta propagation rules for relational operators. Stream computing [12, 48, 80, 100] adopts incremental processing and sublinear-space algorithms to process updates and deltas. Many approximate query answering studies [13, 23, 39] focused on constructing optimal samples to improve query accuracy. Proactive or trigger-based incremental computation techniques [117, 38] were used to achieve low query latency. These studies proposed incremental techniques in isolation, and do not have a general cost-based optimization framework. In addition, they can be integrated into Tempura.

Query Planning for Incremental Processing. Previous work studied some optimization problems in incremental computation. Viglas et al. [107] proposed a rate-based cost model for stream processing. The cost model is orthogonal to Tempura and can be integrated. DBToaster [14] discussed a cost-based approach to deciding the views to materialize under a higher-order view maintenance algorithm. Tang et al. [97] focused on selecting optimal states to materialize for scenarios with intermittent data arrival. They proposed a dynamic programming algorithm for selecting states to materialize given a fixed physical incremental plan and a memory budget, by considering future data-arrival patterns. These optimization techniques all focus on the optimal materialization problem for a specific incremental plan or incremental method, and thus are not general IQP solutions. Tang et al. [98] discussed the idea of eagerly (or lazily) executing parts of a query that is more (or less) amenable to incremental execution. Tempura can also support this style of execution in the `PDW-PD` setting, where the final results are delivered only at the last run. At earlier runs, the optimizer

can choose to incrementally execute only a sub-part of the query based on cost. In fact, we often observed this behavior in the `PDW-PD` setting in the experiments. [98] analyzes the cost of incremental execution based on the concept of incrementability. This can be adopted in Tempura as a new cost function following the discussion in Section 4.6.2.

Flink [20] uses Calcite [26] as the optimizer to support stream queries, which only provides traditional optimizations on the logical plan generated by a fixed incremental method, but cannot combine multiple incremental methods, nor consider correlations between incremental runs. On the contrary, Tempura provides a general framework for users to integrate various incremental methods, and searches the plan space in a cost-based approach.

Semantic Models for Incremental Processing. CQL[21] exploited the relational model to provide strong query semantics for stream processing. Sax et al. [91] introduced the Dual Streaming Model to reason about ordering in stream processing. The key idea behind [21, 91] is the duality of relations and streams, i.e., time-varying relations can be modeled as a sequence of static relations, or a sequence of change logs. The recent work [25] proposed to integrate streaming into the SQL standard, and briefly mentioned that TVRs can serve as a unified basis of both relations and streams. However, their models do not include a formal algebra and rewrite rules on TVRs. To the best of our knowledge, our TIP model for the first time formally defines an algebra on TVRs, providing a principled way to model different types of snapshots/deltas and operators between them. The trichotomy of TVR rewrite rules subsumes many existing incremental methods, laying a theoretical foundation for Tempura.

Chapter 5

Conclusion and Future Work

In this thesis, we first presented the Texera system in Chapter 2, discussing the design choices and the associated trade-offs of several key components within Texera that enable real-time collaborations and user interactions.

For future work, regarding user interaction support, we aim to incorporate more powerful line-level debugging capabilities and controls, both in the Java-based system and operators, as well as Python UDF operators. Another area of focus is our logging-based fault tolerance, where we plan to leverage deterministic computation to improve the reproducibility of Texera workflows and support time-travel debugging. In terms of enhancing real-time collaboration features, we are actively developing the frontend to be smarter and more user-friendly, such as by adding IDE-like code editing experiences while still providing real-time collaborative editing. Texera is being very actively developed, with numerous ongoing features planned or in progress that can greatly enhancing the system's scalability, extensibility, and security, and user experience.

In Chapter 3, we examined a specific use case of user interaction: modifying the logic of operators in a workflow, also known as reconfigurations. We developed an algorithm called

Fries, which can schedule these reconfigurations with minimal delay while maintaining transactional guarantees, especially when a reconfiguration involves multiple operators.

In this work, we focused solely on modifying the logic of operators in a workflow, leaving the alteration of the workflow graph topology for future research. In many use cases, the ability to change the workflow graph topology is essential and highly beneficial, such as when users want to add or remove operators, adjust operator parallelism (add/remove workers), or modify the data exchange policy. Another area for future work is addressing the potential for high reconfiguration delays in Fries when dealing with certain reconfigurations in workflows involving one-to-many operators. Investigating real-world workloads to determine how Fries performs in these cases and offering multiple levels of transactional consistency guarantees could provide trade-offs between delay and consistency. Fries has been integrated into Texera's master branch and is currently implemented as a research prototype on Apache Flink. Further work could involve exploring the possibility of fully integrating the Fries algorithm into Flink.

In Chapter 4, we introduced Tempura, a cost-based optimization framework specifically designed for incremental processing. As a general framework, Tempura can support a wide range of incremental computation requirements for various applications and use cases, extending beyond Texera's scope. Tempura can select the most suitable incremental computation algorithm based on the specific queries and data involved.

Tempura can jointly consider a multitude of factors when choosing an optimal plan. These include different incremental computation algorithms, physical operator implementations, and identifying the best places to materialize intermediate states. Yet, there are several other facets in incremental computation that Tempura does not currently consider, such as determining the frequency at which an operator or a sub-query should be executed, or prioritization among different inputs for multi-input operators such as the Join operator. As operators possess varying degrees of incremental computation overhead, instituting a

prioritization strategy can potentially mitigate this overhead while still catering to the user's requirements.

Currently, the Tempura optimizer framework works the best with a single query. However, in a data warehouse environment, there are often large pipelines of dependent queries. One area for future work is exploring how to incrementally run a complex data warehouse pipeline. This would involve determining which queries to run incrementally and scheduling the incremental runs. Tempura is implemented on top of Apache Calcite, an open-source query optimizer framework. We have successfully incorporated part of Tempura's changes into Calcite. Another direction for future work is integrating more of Tempura's advanced incremental computation features into the Calcite framework.

Bibliography

[1] Behind the scenes of the C sharp yield keyword — startbigthinksmall.wordpress.com. https://startbigthinksmall.wordpress.com/2008/06/09/behind-the-scenes-of-the-c-yield-keyword/. [Accessed 20-May-2023].

[2] Data Science and Analytics Automation Platform | Alteryx — alteryx.com. https://www.alteryx.com/. [Accessed 17-Apr-2023].

[3] Deepnote: Analytics and data science notebook for teams. — deepnote.com. https://deepnote.com/. [Accessed 17-Apr-2023].

[4] GitHub - dmonad/crdt-benchmarks: A collection of CRDT benchmarks — github.com. https://github.com/dmonad/crdt-benchmarks. [Accessed 06-Jun-2023].

[5] Google Colab — research.google.com. https://research.google.com/colaboratory/faq.html. [Accessed 17-Apr-2023].

[6] I was wrong. CRDTs are the future — josephg.com. https://josephg.com/blog/crdts-are-the-future/. [Accessed 06-Jun-2023].

[7] Introduction to Databricks notebooks | Databricks on AWS — docs.databricks.com. https://docs.databricks.com/notebooks/index.html. [Accessed 17-Apr-2023].

[8] Open for Innovation | KNIME — knime.com. https://www.knime.com/. [Accessed 17-Apr-2023].

[9] RapidMiner | Amplify the Impact of Your People, Expertise and Data — rapidminer.com. https://rapidminer.com/. [Accessed 17-Apr-2023].

[10] Visualization of OT with a central server — operational-transformation.github.io. https://operational-transformation.github.io/. [Accessed 20-May-2023].

[11] 262588213843476. Response to "Real Differences between OT and CRDT for Co-Editors" — gist.github.com. https://gist.github.com/LionsAd/19673619c2fa438e475ddca9aa2841b3. [Accessed 06-Jun-2023].

[12] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The design of the borealis stream processing engine. In *Cidr*, volume 5, pages 277–289, 2005.

[13] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The aqua approximate query answering system. In *ACM Sigmod Record*, volume 28, pages 574–576. ACM, 1999.

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

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

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

[17] Akka Website, `https://akka.io/`.

[18] `https://www.alibabacloud.com/product/maxcompute`.

[19] `https://calcite.apache.org`.

[20] `https://flink.apache.org`.

[21] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[22] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured streaming: A declarative API for real-time applications in apache spark. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 601–613. ACM, 2018.

[23] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 539–550. ACM, 2003.

[24] S. Babu, P. Bizarro, and D. DeWitt. Proactive re-optimization. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 107–118, 2005.

[25] E. Begoli, T. Akidau, F. Hueske, J. Hyde, K. Knight, and K. L. Knowles. One SQL to rule them all - an efficient and syntactically idiomatic approach to management of streams and tables. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1757–1772. ACM, 2019.

[26] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 221–230, New York, NY, USA, 2018. ACM.

[27] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[28] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing for Systems Professionals*. Morgan Kaufmann, 1996.

[29] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 61–71, New York, NY, USA, 1986. ACM.

[30] I. Botan, P. M. Fischer, D. Kossmann, and N. Tatbul. Transactional stream processing. In E. A. Rundensteiner, V. Markl, I. Manolescu, S. Amer-Yahia, F. Naumann, and I. Ari, editors, *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*, pages 204–215. ACM, 2012.

[31] O. P. Buneman and E. K. Clemons. Efficiently monitoring relational databases. *ACM Trans. Database Syst.*, 4(3):368–382, Sept. 1979.

[32] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in apache flink®: Consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 10(12):1718–1729, 2017.

[33] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *CoRR*, abs/1506.08603, 2015.

[34] P. Carbone, M. Fragkoulis, V. Kalavri, and A. Katsifodimos. Beyond analytics: The evolution of stream processing systems. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2651–2658. ACM, 2020.

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

[36] B. Chandramouli, C. N. Bond, S. Babu, and J. Yang. Query suspend and resume. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 557–568, 2007.

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

[38] B. Chandramouli, J. Goldstein, and A. Quamar. Scalable progressive analytics on big data in the cloud. *Proc. VLDB Endow.*, 6(14):1726–1737, Sept. 2013.

[39] S. Chaudhuri, G. Das, and V. Narasayya. Optimized stratified sampling for approximate query processing. *ACM Transactions on Database Systems (TODS)*, 32(2):9, 2007.

[40] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of the Eleventh International Conference on Data Engineering*, ICDE '95, pages 190–200, Washington, DC, USA, 1995. IEEE Computer Society.

[41] T. H. Cormen. *Algorithms Unlocked*. MIT Press, 2013.

[42] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani. *Algorithms*. McGraw-Hill, 2008.

[43] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. R. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In K. A. Ross, D. Srivastava, and D. Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 725–736. ACM, 2013.

[44] Advanced Flink Application Patterns Vol.1: Case Study of a Fraud Detection System, `https://flink.apache.org/news/2020/01/15/demo-fraud-detection.html`.

[45] Savepoints in Apache Flink, `https://ci.apache.org/projects/flink/flink-docs-master/docs/ops/state/savepoints/`.

[46] Support dynamically changing CEP patterns in Flink, `https://issues.apache.org/jira/browse/FLINK-7129`.

[47] Advanced Flink Application Patterns Vol.2: Dynamic Updates of Application Logic, `https://flink.apache.org/news/2020/03/24/demo-fraud-detection-2.html`.

[48] T. M. Ghanem, A. K. Elmagarmid, P.-Å. Larson, and W. G. Aref. Supporting views in data stream management systems. *ACM Transactions on Database Systems (TODS)*, 35(1):1, 2010.

[49] D. Ghosh, P. Gupta, S. Mehrotra, and S. Sharma. A case for enrichment in data management systems. *SIGMOD Rec.*, 51(2):38–43, 2022.

[50] D. Ghosh, P. Gupta, S. Mehrotra, R. Yus, and Y. Altowim. JENNER: just-in-time enrichment in query processing. *Proc. VLDB Endow.*, 15(11):2666–2678, 2022.

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

[52] G. Graefe. The cascades framework for query optimization. *Data Engineering Bulletin*, 18, 1995.

[53] G. Graefe, W. Guy, H. A. Kuno, and G. Paullley. Robust query processing (dagstuhl seminar 12321). In *Dagstuhl Reports*, volume 2. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[54] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*, pages 209–218. IEEE Computer Society, 1993.

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

[56] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 328–339, New York, NY, USA, 1995. ACM.

[57] M. Hoffmann, A. Lattuada, F. McSherry, V. Kalavri, J. Liagouris, and T. Roscoe. Megaphone: Latency-conscious state migration for distributed streaming dataflows. *Proc. VLDB Endow.*, 12(9):1002–1015, 2019.

[58] L. Hupel. CRDTs: Part 1 — lars.hupel.info. `https://lars.hupel.info/topics/crdt/01-intro/`. [Accessed 20-May-2023].

[59] `https://databricks.com/blog/2018/03/13/introducing-stream-stream-joins-in-apache-sp html`.

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

[61] K. Kakousis, N. Paspallis, and G. A. Papadopoulos. A survey of software adaptation in mobile and ubiquitous computing. *Enterp. Inf. Syst.*, 4(4):355–389, 2010.

[62] T. Kathuria and S. Sudarshan. Efficient and provable multi-query optimization. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '17, pages 53–67, New York, NY, USA, 2017. ACM.

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

[64] F. Kon and R. H. Campbell. Supporting automatic configuration of component-based distributed systems. In M. V. Devarakonda, editor, *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies & Systems, May 3-7, 1999, The Town & Country Resort Hotel, San Diego, California, USA*, pages 175–188. USENIX, 1999.

[65] A. Kumar, Z. Wang, S. Ni, and C. Li. Amber: A debuggable dataflow system based on the actor model. *Proc. VLDB Endow.*, 13(5):740–753, 2020.

[66] W. Lang, R. V. Nehme, E. Robinson, and J. F. Naughton. Partial results in database systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1275–1286, New York, NY, USA, 2014. ACM.

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

[68] Y.-N. Law, H. Wang, and C. Zaniolo. Query languages and data models for database sequences and data streams. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, page 492–503. VLDB Endowment, 2004.

[69] M. K. Lee. Implementing an interpreter for functional rules in a query optimizer. 1988.

[70] X. Liu, Z. Wang, S. Ni, S. Alsudais, Y. Huang, A. Kumar, and C. Li. Demonstration of collaborative and interactive workflow-based data analytics in texera. *Proc. VLDB Endow.*, 15(12):3738–3741, 2022.

[71] D. B. Lomet. MLR: A recovery method for multi-level systems. In M. Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 2-5, 1992*, pages 185–194. ACM Press, 1992.

[72] X. Ma, L. Baresi, C. Ghezzi, V. P. L. Manna, and J. Lu. Version-consistent dynamic reconfiguration of component-based distributed systems. In T. Gyimóthy and A. Zeller, editors, *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 245–255. ACM, 2011.

[73] L. Mai, K. Zeng, R. Potharaju, L. Xu, S. Suh, S. Venkataraman, P. Costa, T. Kim, S. Muthukrishnan, V. Kuppa, S. Dhulipalla, and S. Rao. Chi: A scalable and programmable control plane for distributed stream processing systems. *Proc. VLDB Endow.*, 11(10):1303–1316, 2018.

[74] D. Maier, J. Li, P. Tucker, K. Tufte, and V. Papadimos. Semantics of data streams and operators. In T. Eiter and L. Libkin, editors, *Database Theory - ICDT 2005*, pages 37–52, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[75] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014, June 22-27, 2014, Baltimore, MD, USA, System Demonstrations*, pages 55–60. The Association for Computer Linguistics, 2014.

[76] Y. Mao, Y. Huang, R. Tian, X. Wang, and R. T. B. Ma. Trisk: Task-centric data stream reconfiguration. In C. Curino, G. Koutrika, and R. Netravali, editors, *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, pages 214–228. ACM, 2021.

[77] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Çetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, M. Stonebraker, K. Tufte, and H. Wang. S-store: Streaming meets transaction processing. *Proc. VLDB Endow.*, 8(13):2134–2145, 2015.

[78] C. Mohan, D. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[79] B. D. Monte, S. Zeuch, T. Rabl, and V. Markl. Rhino: Efficient management of very large distributed state for stream processing engines. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2471–2486. ACM, 2020.

[80] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. CIDR, 2003.

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

[82] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma. Yjs: A framework for near real-time P2P shared editing on arbitrary data types. In P. Cimiano, F. Frasincar, G. Houben, and D. Schwabe, editors, *Engineering the Web in the Big Data Era - 15th International Conference, ICWE 2015, Rotterdam, The Netherlands, June 23-26, 2015, Proceedings*, volume 9114 of *Lecture Notes in Computer Science*, pages 675–678. Springer, 2015.

[83] M. Nikolic, M. Dashti, and C. Koch. How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 511–526, New York, NY, USA, 2016. ACM.

[84] Orleans Website, `https://dotnet.github.io/orleans/`.

[85] I. Padhi, Y. Schiff, I. Melnyk, M. Rigotti, Y. Mroueh, P. Dognin, J. Ross, R. Nair, and E. Altman. Tabular transformers for modeling multivariate time series. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3565–3569. IEEE, 2021.

[86] N. M. Preguiça, C. Baquero, and M. Shapiro. Conflict-free replicated data types crdts. In S. Sakr and A. Y. Zomaya, editors, *Encyclopedia of Big Data Technologies*. Springer, 2019.

[87] M. Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In K. Huff and J. Bergstra, editors, *Proceedings of the 14th Python in Science Conference 2015 (SciPy 2015), Austin, Texas, July 6 - 12, 2015*, pages 126–132. scipy.org, 2015.

[88] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 249–260, New York, NY, USA, 2000. ACM.

[89] A. Sadeghi, N. Esfahani, and S. Malek. Ensuring the consistency of adaptation through inter- and intra-component dependency analysis. *ACM Trans. Softw. Eng. Methodol.*, 26(1):2:1–2:27, 2017.

[90] K. Santosh and A. Khunteta. A survey on operational transformation algorithms: Challenges, issues and achievements. *International Journal of Computer Applications*, 3, 07 2010.

[91] M. J. Sax, G. Wang, M. Weidlich, and J.-C. Freytag. Streams and tables: Two sides of the same coin. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*, BIRTE '18, New York, NY, USA, 2018. Association for Computing Machinery.

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

[93] P. F. Silvestre, M. Fragkoulis, D. Spinellis, and A. Katsifodimos. Clonos: Consistent causal recovery for highly-available streaming dataflows. In G. Li, Z. Li, S. Idreos, and D. Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1637–1650. ACM, 2021.

[94] M. A. Soliman, L. Antova, V. Raghavan, A. El-Helw, Z. Gu, E. Shen, G. C. Caragea, C. Garcia-Alvarado, F. Rahman, M. Petropoulos, F. Waas, S. Narayanan, K. Krikellas, and R. Baldwin. Orca: A modular query optimizer architecture for big data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 337–348, New York, NY, USA, 2014. ACM.

[95] StreamING Machine Learning Models: How ING Adds Fraud Detection Models at Runtime with Apache Flink, `https://www.ververica.com/blog/real-time-fraud-detection-ing-bank-apache-flink`.

[96] C. Sun, D. Sun, A. Ng, W. Cai, and B. Cho. Real differences between OT and CRDT under a general transformation framework for consistency maintenance in co-editors. *Proc. ACM Hum. Comput. Interact.*, 4(GROUP):06:1–06:26, 2020.

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

[98] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin. Thrifty query execution via incrementability. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 1241–1256. ACM, 2020.

[99] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, SIGMOD '92, page 321–330, New York, NY, USA, 1992. Association for Computing Machinery.

[100] H. Thakkar, N. Laptev, H. Mousavi, B. Mozafari, V. Russo, and C. Zaniolo. Smm: A data stream management system for knowledge discovery. In *2011 IEEE 27th International Conference on Data Engineering*, pages 757–768. IEEE, 2011.

[101] `http://www.tpc.org/tpcds/`.

[102] A. Tiwari, B. Ramprasad, S. H. Mortazavi, M. Gabel, and E. de Lara. Reconfigurable streaming for the mobile edge. In A. Wolman and L. Zhong, editors, *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications, HotMobile 2019, Santa Cruz, CA, USA, February 27-28, 2019*, pages 153–158. ACM, 2019.

[103] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. V. Ryaboy. Storm@twitter. In C. E. Dyreson, F. Li, and M. T. Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 147–156. ACM, 2014.

[104] TPC-DS http://www.tpc.org/tpcds/.

[105] Upgrading Applications and Flink Versions, `https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/ops/upgrading/`.

[106] P. Vassiliadis. A survey of extract-transform-load technology. *Int. J. Data Warehous. Min.*, 5(3):1–27, 2009.

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

[108] Z. Wang, K. Zeng, B. Huang, W. Chen, X. Cui, B. Wang, J. Liu, L. Fan, D. Qu, Z. Ho, T. Guan, C. Li, and J. Zhou. Grosbeak: A data warehouse supporting resource-aware incremental computing. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, Portland, Oregon, USA, 2020. ACM.

[109] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[110] B. Wiese and C. Omlin. Credit card transactions, fraud detection, and machine learning: Modelling time with lstm recurrent neural networks. In *Innovations in neural information paradigms and applications*, pages 231–268. Springer, 2009.

[111] F. Wolf, N. May, P. R. Willems, and K.-U. Sattler. On the calculation of optimality ranges for relational query execution plans. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 663–675, New York, NY, USA, 2018. Association for Computing Machinery.

[112] P. Yin, H. Lai, S. Zhao, R. Fu, I. Cisneros, R. Ge, J. Zhang, H. Choset, and S. A. Scherer. Automerge: A framework for map assembling and smoothing in city-scale environments. *CoRR*, abs/2207.06965, 2022.

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

[114] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 247–260, 2009.

[115] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In E. M. Nahum and D. Xu, editors, *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. USENIX Association, 2010.

[116] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: fault-tolerant streaming computation at scale. In M. Kaminsky and M. Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 423–438. ACM, 2013.

[117] K. Zeng, S. Agarwal, and I. Stoica. iolap: Managing uncertainty for efficient incremental olap. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1347–1361, New York, NY, USA, 2016. ACM.

[118] J. Zhou, P.-A. Larson, P.-A. Larson, J.-C. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 533–544, New York, NY, USA, 2007. ACM.