Theories and programming systems for weakly consistent distributed systems



Junghun Yoo
St Hugh's College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy

Michaelmas 2022

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor Hongseok Yang, whose expertise, mentorship, and encouragement were instrumental in shaping the direction of this research. His valuable insights and constructive feedback have significantly enriched the quality of this thesis. I have been extremely fortunate to be supervised by him. I am deeply grateful to Professor Sam Stanton for all his support and help until the end of my DPhil journey.

My special thanks go to Professor Andrzej Murawski and Professor Peter Van Roy, who gave me the constructive and detailed feedback throughout the completion of the thesis. I also thank Professor Luke Ong and Professor Dan Olteanu for their helpful comments during the Transfer and the Confirmation of Status.

I would like to express my appreciation to my family, my better half Dr Min Young Tak and my son Zane, for their unwavering support, love, and understanding throughout this demanding journey. I also thank my parents-in-law, Professor Ki Hyun Tak and Professor Byung Eun Cho, for their huge support. I owe my thanks to my brother, Seoung Hoon Yoo. Lastly, I am deeply indebted to my parents, Hyung Du Yoo, Young Nam Kim, for their unconditional love and support. I dedicate this thesis to my family.

Abstract

Weakly consistent distributed systems are systems that provide a weak form of guarantee about the sameness of the replicated data. They allow using the replicated data from any replicas without the coordination between them and can thus enhance the performance and availability of the system. However, due to the lack of coordination, programs developed on weakly consistent distributed systems might generate counter-intuitive results.

The aim of this research is to i) address those glitches in developing programs on weakly consistent distributed systems and ii) provide formal guarantee on the executions of programs run on such systems.

In order to overcome the counter-intuitive nature of the results generated by weakly consistent distributed systems, this thesis proposes a new programming language Lasp_{PC}, a variant of Lasp that uses provenance and cover, and its runtime system. Optimisation techniques are also introduced for the effective execution of the programs for the language and its runtime system.

This thesis provides formal guarantee for the programs run on weakly consistent distributed systems by defining key principles for the new language and presenting proofs for the theoretical properties of the language.

The new language introduced in the thesis uses a variant of OR sets that incorporates provenance and cover, OR_{PC} sets. The use of provenance and cover enables the language to offer the guarantee of a new property, strong consistency in weakly consistent distributed systems. The new property ensures the exclusion of the counter-intuitive results by enabling the retrieval of computational results from the systems. There is no provision of comparable guarantees in the existing research studies on Lasp. The OR_{PC} sets are used to define client-level data types, which retain the guarantee of the new property. The new property is also used to extend the new language to support aggregation operations that are considered particularly difficult to be supported in weakly consistent distributed systems.

The approach taken in this research to remedy the glitches and provide the formal guarantee has been implemented, optimised and evaluated in

three substantial example programs, namely, the replicated leader board, the MapReduce program and the BigBench queries.

This thesis also identifies a case study of one of the extensively-used blockchain consensus protocols, the Stellar Consensus Protocol (SCP). It is currently difficult to be adopted for the newly developed language and its runtime system but the protocol has been rigorously studied using the formal modelling and verification techniques.

Contents

1 Introduction **1**

1.1 Context 1

1.2 Contributions 6

1.3 Roadmap 7

2 Background and Related Work **9**

2.1 Approaches to Resolve the Programmability Issue 9

2.2 Replicated Data Types and Programming Languages for Replicated Data 11

2.3 The Research Context 12

3 Rationale for Introducing a New Programming Language **14**

3.1 Specification of the example program 15

3.2 Guaranteeing strong convergence 16

3.2.1 Problems 16

3.2.2 Basic Concepts for the Solutions 16

3.2.3 How Provenance Works 18

3.2.4 How Cover Works 19

3.2.5 Consistent Results 22

4 Observed-Remove Set with Provenance and Cover **24**

4.1 Basic concepts 24

4.2 Definition of OR_{PC} Sets 28

4.3 Operations of OR_{PC} Sets 38

5 Dataflow Programs Expressed by Directed Acyclic Graphs **42**

5.1 Distribution Architecture of the System 42

5.2 Dataflow Programs 42

5.3 Operational Semantics 43

5.4 Consistent Execution and Correspondence Property 47

5.5 Consistent Read 50

5.5.1 Definition 50

5.5.2 Guarantee . . . 51
5.6 Proof of the Consistent Read Theorem . . . 53
5.6.1 Shadow State . . . 53
5.6.2 Dataflow Tree . . . 66
5.6.3 Definitions and examples for Complete-Dot Set . . . 75
5.6.4 Lemma for Complete-Dot Set . . . 78
5.6.5 Properties between Executions . . . 89
5.6.6 Proof of the Theorem . . . 106

6 Data Types Built Upon OR_{PC} Sets 108
6.1 Data Type Constraints, Partially Ordered Events and Mixed Data Type Operations . . . 108
6.2 Last-Writer-Wins Register . . . 116
6.3 Grow-only Counter . . . 118
6.4 Observed-Remove Bag . . . 119

7 Operations for Aggregation 121
7.1 Details of Aggregation Operations . . . 122
7.2 Proof of Theorem 5.5.1 for Aggregation Operations . . . 125
7.3 The Example Program for Aggregation Operations . . . 132

8 Experiments 135
8.1 Implementation Details . . . 135
8.2 Example Programs . . . 136
8.2.1 The Replicated Leader Board . . . 136
8.2.2 The MapReduce Program . . . 139
8.2.3 BigBench Queries . . . 141
8.3 Three General Optimisations . . . 149
8.3.1 Removal of Redundant Information . . . 150
8.3.2 Synchronisation Filter . . . 151
8.3.3 Delta State with the Incremental Computation Mode and Caches 153
8.4 Experiment Results . . . 154
8.4.1 The Replicated Leader Board . . . 154
8.4.2 The MapReduce Program . . . 155
8.4.3 BigBench Queries . . . 155
8.4.4 Discussion of the Performance . . . 157
8.5 Further Experiments . . . 157
8.5.1 Glitch Removal . . . 157
8.5.2 Convergence . . . 158

8.5.3 Fault Tolerance . . . 159

9 Case Study **160**
9.1 The Stellar Consensus Protocol . . . 161
9.1.1 The Nomination Protocol . . . 162
9.1.2 The Ballot Protocol . . . 163
9.2 Formal Modelling for the SCP . . . 164
9.2.1 Four Design Templates for Vote-based Consensus Protocols . . . 164
9.2.2 Design Templates for the SCP . . . 166
9.3 Verification Results for the SCP . . . 169

10 Conclusion **172**
10.1 Summary . . . 172
10.2 Future Work . . . 174

Bibliography **177**

A Source Code of Experiments **183**
A.1 The Replicated Leader Board . . . 183
A.2 The MapReduce Program . . . 188
A.3 BigBench Queries . . . 191
A.3.1 Query 1 . . . 191
A.3.2 Query 24 . . . 196
A.3.3 Query 29 . . . 209

List of Figures

1.1 System architecture of the example program 4

3.1 Dataflow of the example program 14

8.1 Optimisation 1: Removal of redundant information 151
8.2 Optimisation 2: Synchronisation filter 152
8.3 Optimisation 3: Delta state with the incremental computation mode and caches 154
8.4 Experiment Results for the Replicated Leader Board 155
8.5 Experiment Results for the MapReduce Program 155
8.6 Experiment Results for BigBench Queries 156
8.7 Metadata Size of the final node for the Replicated Leader Board 157

9.1 Templates for Tick Manager and Tick Worker 165
9.2 Template for the Nomination Protocol 166
9.3 Template for the Ballot Protocol 167
9.4 Template for the Slot Manager 167
9.5 Template for the Nomination Protocol Checker 168
9.6 Verification Result for Open Membership 170
9.7 Verification Result for the Cascade Theorem 170

List of Tables

8.1 Results of the consistent read 158
8.2 Results of the normal read 158

9.1 Without the Four Design Templates[49] 169
9.2 With the Four Design Templates 169

Listings

6.1 LWW register in *replica a* . . . 110
6.2 LWW register in *replica b* . . . 110
6.3 OR_{PC} set in *replica a* . . . 110
6.4 OR_{PC} set in *replica b* . . . 110
6.5 G counter in *replica a* . . . 113
6.6 G counter in *replica b* . . . 113
6.7 OR bag in *replica a* . . . 114
6.8 OR bag in *replica b* . . . 114
6.9 A LWW register with `filter`() . . . 116
6.10 LWW register and OR bag with $\times$ . . . 116
7.1 The average calculator . . . 133
7.2 The generalised average calculator . . . 133
8.1 Complete definition of Lasp_{PC} . . . 135
8.2 The *GroupScores* node from *UserData* and *GroupData* . . . 137
8.3 The *RankDivider* node from *GroupData* and *Divider* . . . 138
8.4 The *GroupMedal* node from *GroupScores* and *RankDivider* . . . 139
8.5 The example program for MapReduce . . . 140
8.6 The SQL query for Query 1 . . . 142
8.7 The example program for Query 1 . . . 143
8.8 The SQL query for Query 24 . . . 144
8.9 The example program for Query 24 . . . 146
8.10 The SQL query for Query 29 . . . 148
8.11 The example program for Query 29 . . . 148

Chapter 1

Introduction

This thesis provides a solution to address the problem of counter-intuitive result generation on the weakly consistent distributed systems. Extending earlier work on Lasp [35], the thesis proposes a new programming language Lasp_{PC} [1], which uses provenance and cover, and its runtime system. It also discusses how formal guarantee for the programs run on weakly consistent distributed systems was provided, including the guarantee of a new property *strong consistency in weakly consistent distributed systems*. It defines key principles for the new language and presents proofs for the theoretical properties of the language. Three cases studies are included in the thesis to demonstrate how the problem has been resolved and the formal guarantee has been implemented, optimised and evaluated.

This chapter first provides an overview of the research context, highlighting the gaps in the literature. It then discusses the contributions of the research, followed by the description of the structure of the thesis.

1.1 Context

In order to achieve highly available and sustainable distributed systems, global-scale Internet services and mobile applications use replicated data stored at multiple replicas around the world. The common usage of these replicated data often involves using the so-called weakly consistent distributed systems, which provide only a weak form of guarantee about the sameness of the replicated data: these systems allow using the replicated data from any replicas without requiring the all replicas to have the identical state. The choice of using weakly consistent distributed systems over strong consistent distributed systems is based on one's preference to performance over correctness [13]. When a replica in those systems receives a query that involves an update, it first applies the update to its local state, calculates the outcome of the query based on information from the local state without any coordination with other replicas, and returns the result

[1] PC means provenance and cover

of the calculation. The coordination with other replicas is carried out only later by a separate mechanism, which periodically exchanges the local state of the replica with those of the others. This lazy coordination improves the responsiveness and makes the system tolerate failures such as network partition, but it leads to the loss of the so-called strong consistency (the systems with the strong consistency behave as if the replicated data are stored and computed in a single machine). It means that the systems only guarantee weak consistency.

One of the well-known issues of using such weakly consistent distributed systems is that queries on replicated data may generate counter-intuitive results, since the results are only generated from the local data of a replica. The counter-intuitive results indicate the results that are different from the expected outcomes of user operations. For example, Amazon's shopping cart that is implemented on top of Dynamo [20], one of the distributed systems with weak consistency, has an issue of the reappearance of the removed items. The counter-intuitive results make it difficult to develop programs on these weakly consistent distributed systems. Thus, resolving this programmability issue has been an active research topic in the distributed systems and programming languages communities. Previous approaches to this issue include i) developing data types and programming languages that support data replication [35], ii) devising proof techniques for distributed programs using replicated data [3], iii) designing formal models for data replication and weak consistency [16], and iv) providing new distributed systems that do not exhibit some of counter-intuitive results mentioned above [19]. However, despite all those efforts, the issue is far from being resolved. Only a few solutions for weakly-consistent programming models have been proposed, and both their theoretical and practical implications have not been studied thoroughly. As far as the language design is concerned, the following key factors have not been clarified yet: i) key principles for new language constructs for improving the programmability and the analysis of their theoretical properties, ii) prerequisites for the development of an actual language and its runtime system, and iii) optimisation techniques for supporting these new language constructs with acceptable performance.

In order to fill these gaps in research, this thesis introduces and demonstrates the new programming language Lasp_{PC} and its runtime system we developed, defines key principles for the new language, provides proofs for the theoretical properties of the language, and explains the optimisation techniques we employed. In particular, we define a new property, namely, *strong consistency in weakly consistent distributed systems*. This property is used to retrieve complete computational results from the local data of a replica, and this makes sure that the programs written in our new language do not have the counter-intuitive results. The new property is also used to provide aggregation operations for the new language. Furthermore, we study one of the blockchain

consensus protocols so that it can be used to extend the new programming language for supporting sequential data types such as replicated growable arrays (RGAs) [43].

By filling the gaps, we attempt to answer the following fundamental question: what principles should one consider when developing an ideal programming language for distributed programs with weak consistency? For the development of our new language, we consider a programming language that supports dataflow-style programs and replicated data types, such as Lasp [35], as a candidate for an ideal programming language. We also propose a new way of using the concept of *provenance* used in the database community and introduce a novel concept, *cover*, for the new language to answer the question. We found that the use of provenance and cover leads to a new design of dataflow languages for replicated data and weak consistency, and that this novel design addresses some important language design issues regarding the expressiveness of the languages and the performance and guarantee of its runtime system. This application of provenance also contributes to the extension of the usual formalisation of provenance based on semi-ring [26] and semi-module [4], which can be useful for improving the handling of the provenance for aggregation [4] in the database applications.

The key functionalities of the new language we aim to implement are threefold: decentralized communication, high responsiveness and strong convergence. These functionalities are described using the example program we developed. The example program is an online game that runs on a mobile device which can played by a large number of users. The game assigns each player to a group and maintains various statistics globally with a leader board summarising the performance of players and their groups. We assume that we are building a program that first ranks all groups according to the performance of the players in each group, and then assigns medals (either gold, silver or bronze) to each group based on this ranking. In order for the program to provide correct results, it should meet the following three slightly contradicting functionalities:

- **Decentralized communication.** When running on a mobile device, the program should compute ranks based on the results of nearly all games (not *all* games due to the weak consistency), most of which are played by different users, and on different mobile devices, without the centralised server collecting the complete results. Note that this functionality entails that instances of the program on mobile devices should communicate with one another frequently.

- **High responsiveness.** Even if a mobile device is not connected to the Internet, the program running on that device should still remain available and incorporate all the local plays when updating the ranking of the groups and assigning medals to those groups.

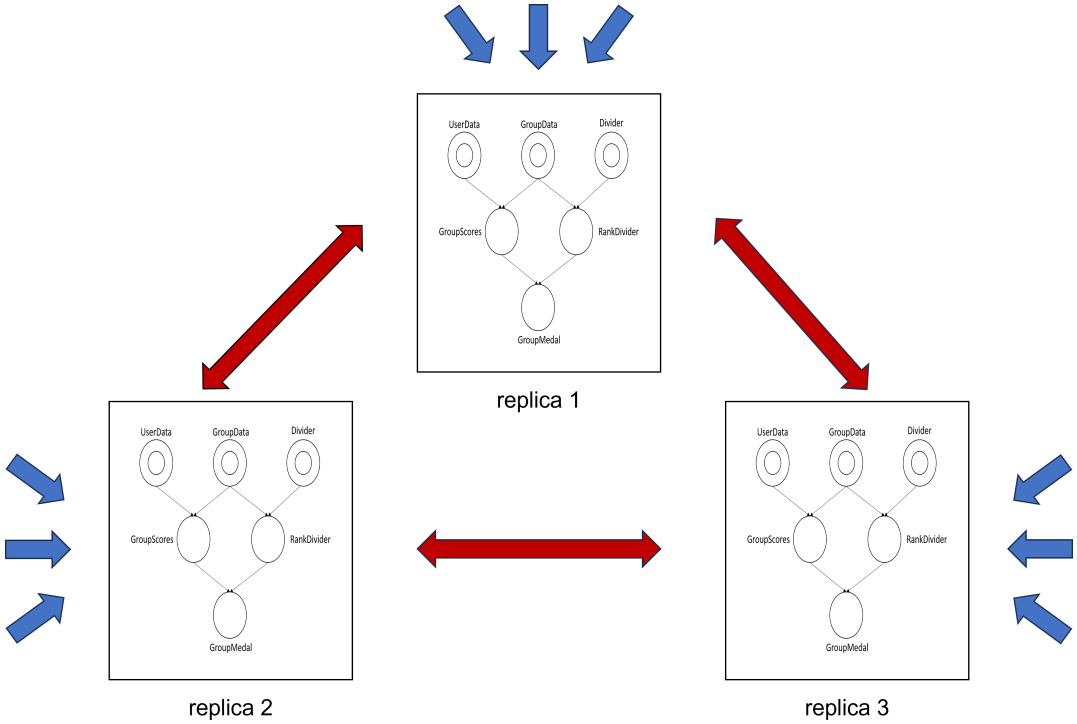


Figure 1.1: System architecture of the example program

- **Strong convergence.** Strong convergence means that the rankings and medal assignments displayed across all devices are identical to the results we would get by running the program on a single device with subsets of play results that are selected using provenance and cover. It aims to ensure that the ranking and the medal assignment are computed consistently across all devices with respect to those subsets of plays.

Lasp_{PC} [2], an extension of the Lasp programming language [35] we developed, uses our new design (provenance and cover). We implemented a runtime system for the new language. Lasp_{PC} is a dataflow-based programming language for weak consistency that supports all the three functionalities mentioned above. Lasp is an example of a dataflow-based language for weak consistency but it only supports the first two functionalities, decentralised communication and high responsiveness. Thus, in Lasp, it is very difficult to write our ranking program that supports the third functionality, strong convergence. To support all the three functionalities, we need a new programming language and its runtime system, such as the one that will be described from Chapter 4 to Chapter 7.

Figure 1.1 illustrates the system architecture of how the dataflow-based example program is executed. Each replica executes a complete dataflow graph that comes from the example program. Then, each replica i) receives input data from users (the

[2]The entire implementation can be found at `https://github.com/junghunyoo/lasp/tree/consistent_read`

blue arrows in the figure), ii) processes them without coordination and iii) periodically exchanges the states of every node in the dataflow (the red arrows in the figure).

One of the common scenarios of the counter-intuitive result generation involves the reappearance of the removed values. For example, we can assume that the three replicas in Figure 1.1 have information about the insertion of the value 10 and a user removes that value on replica 3. If replica 1 has not received the information about the removal of the value and the user tries to read all the values, the user will see the value 10 that was previously removed again.

The design of Lasp_{PC} is based on the extension of the concept of an Observed-Remove (OR) set [6] using provenance and cover, the OR_{PC} set. The existing OR set does not have information on how values in the set are calculated and if the values are the results of the complete computation on the dataflow-based programs. Hence, the use of the existing OR set is not enough to guarantee strong convergence, the third functionality mentioned above.

In order to guarantee the theoretical properties including the three functionalities, we designed Lasp_{PC} to track a number of events, such as the identifier of an insert operation executed on a replica, and those events are stored in the form of provenance and cover. Provenance tracks the way the inserted values are included in the results of dataflow computation and cover captures the number of computations completed up to the point of reading the results. The following shows a possible (and naive) representation of an OR_{PC} set:

$$
\begin{aligned}
&(\{(i_1,_),(i_2,_),(i_3,_),(i_4,_),(i_5,_),(j_1,l),(j_1,r),(j_2,l),(j_2,r),(j_3,l),(d_1,_)\},\\
&\ \{(i_1,_),(i_2,_),(i_3,_),(i_4,_),(j_1,l),(j_1,r),(j_2,l),(j_2,r),(j_3,l),(d_1,_)\},\\
&\ \{\{(i_1,_),(j_1,l),(j_1,r),(d_1,_)\},\\
&\quad \{(i_2,_),(i_3,_),(i_4,_),(i_5,_),(j_2,l),(j_2,r),(j_3,l),(d_1,_)\}\},\\
&\ [\{(i_1,_),(j_1,l),(j_1,r),(d_1,_)\} \mapsto \qquad (1.1)\\
&\quad \{((g_1,\mathit{bronze}),(i_1,_)\wedge(j_1,l)\wedge(j_1,r)\wedge(d_1,_))\},\\
&\ \{(i_2,_),(i_3,_),(i_4,_),(i_5,_),(j_2,l),(j_2,r),(j_3,l),(d_1,_)\} \mapsto\\
&\quad \{(g_2,\mathit{bronze}),(i_2,_)\wedge(i_3,_)\wedge(j_2,l)\wedge(j_2,r)\wedge(d_1,_)),\\
&\quad\ ((g_1,\mathit{bronze}),(i_2,_)\wedge(i_3,_)\wedge(j_2,l)\wedge(j_2,r)\wedge(j_3,l)\wedge(d_1,_))\}]).
\end{aligned}
$$

The examples of provenance are highlighted in red and an example of cover is highlighted in blue in the representation (1.1). Each value in the set is paired with its provenance and each provenance indicates which values are used for the computation. For example, the provenance $(i_1,_)\wedge(j_1,l)\wedge(j_1,r)\wedge(d_1,_)$ indicates that the value (g_1, bronze) is computed using the values inserted with $(i_1,_)$, (j_1,l), (j_1,r) and $(d_1,_)$. However, provenance cannot indicate whether the values inserted with its member are completely computed. Here, the word 'completely computed' means that those values are computed by every dataflow operation. The subset in cover $\{(i_1,_),(j_1,l),(j_1,r),(d_1,_)\}$ implies that the values inserted with $(i_1,_)$, (j_1,l), (j_1,r) and $(d_1,_)$ are completely computed. Therefore,

$\{(i_1, _), (j_1, l), (j_1, r), (d_1, _)\} \mapsto \{((g_1, \mathit{bronze}), (i_1, _) \wedge (j_1, l) \wedge (j_1, r) \wedge (d_1, _))\}$ implies that the value (g_1, bronze) is computed using the values inserted with $(i_1, _)$, (j_1, l), (j_1, r) and $(d_1, _)$ and those values are completely computed. By using the combination of provenance and cover, it is possible to extract complete computational results only from replicated data on weakly consistent systems so that consistent outcomes can be retrieved at all replicas.

Our experiments on the system we newly implemented are threefold: developing a replicated leader board for an online mobile game, developing a MapReduce program, and developing a replicated relational database program with BigBench queries [21] on weakly consistent distributed systems. The results of the experiments showed that our language and its runtime system perform with the reasonable execution time and memory usage while guaranteeing the theoretical properties.

1.2 Contributions

The contributions of this thesis are as follows:

- We propose a new programming language, Lasp_{PC}, and its runtime system for replicated data with weak consistency using provenance and cover, which allows the extraction of complete computational results from the local state of a replica. The use of our programming language remedies the inherent problem of counter-intuitive result generation on weakly consistent distributed systems.

- We define a new property, *strong consistency in weakly consistent distributed systems*, and it is used to prove that the programs written in our new programming language provide complete computational results (Chapter 5).

- We also define key theoretical properties of our language: the correspondence property (Theorem 5.4.3) and the consistent read property (Theorem 5.6.14). The correspondence property proves that our system satisfies the eventual consistency property and provides a detailed description of how the system looks like at the quiescent state. We prove the correspondence property using the concept of consistent executions. The consistent executions are the executions that process all input streams and exchange those processed input streams among all replicas so that every replica has the information about all input values. Then, every replica executes dataflow operations without communicating with each other. The consistent read property proves that the results of the $\texttt{read}_c()$ operation satisfy the *strong consistency in weakly consistent distributed systems* property. It means that every value in the results of the operation is the one that is completely computed by every dataflow operation in the program (Chapter 5).

- We prove that the programs implemented using the system-level data type, the OR_{PC} set, satisfy the theoretical properties we defined (Chapter 5).

- We propose three client-level data types defined using the OR_{PC} set. The fact that the OR_{PC} set is used to defined those types implies that any programs implemented using the three client-level data types we propose also satisfy the theoretical properties (Chapter 6).

- The new design provides a new replicated data type that supports aggregation operations, such as `agg_cnt`() and `agg_sum`(), which are considered particularly difficult to support on weakly consistent systems. The aggregation operations only use the values from the complete computational results for aggregation and this ensures that the results of the operations also satisfy the *strong consistency in weakly consistent distributed systems* property (Chapter 7).

- We implement, optimise and evaluate three example programs, the replicated leader board, the MapReduce program and the BigBench queries. In our experiments, the executions of the three programs show promising performance without the generation of any counter-intuitive results (Chapter 8).

- Three optimisation techniques are implemented and evaluated: i) removing the redundant information to reduce the metadata size, ii) using the synchronisation filter mode that makes data exchanges to happen for only selected nodes and iii) using delta state with incremental computation mode and caches to reduce the redundant dataflow computations (Chapter 8).

- We study one of the blockchain consensus protocols, the Stellar Consensus Protocol (SCP) [34] with the aim of extending our new language to support sequential data types. However, the protocol requires some coordination (albeit at a low level) at the early stage and it was not possible to apply it directly to our current implementation. With the expectations that it can be applied to our runtime system in the future and it can also help develop new blockchain consensus protocols of similar nature, we have pursued the study of the protocol. With formal modeling and verification with the UPPAAL model checker program [5], we provide four design templates for modeling voting-based consensus protocols that are similar to the SCP (Chapter 9).

1.3 Roadmap

Chapter 2 introduces the background of this research and reviews the relevant literature. We explain the rationale for introducing a new programming language in Chapter 3.

Chapter 4 introduces the system-level data type, $\mathrm{OR}_{\mathrm{PC}}$, including its definition and its operations. The key theoretical properties of the programs written in our new language including the strong consistency property and their formal proofs are presented in Chapter 5. The client-level data types built upon the $\mathrm{OR}_{\mathrm{PC}}$ are defined in the following chapter including their operations (Chapter 6). Chapter 7 contains the explanation about two operations for aggregation. The explanation includes definitions of the operations and the proof of the main theorem with those operations. Our experiment results and the detailed explanation of the three optimisation techniques are described in Chapter 8. The following chapter (Chapter 9) explains how the SCP works and provides its formal modeling and verification results with the UPPAAL model checker program [5]. Chapter 9 also introduces the design templates for modeling voting-based consensus protocols. The last chapter (Chapter 10) concludes this thesis.

Chapter 2

Background and Related Work

This chapter introduces the background of this research and reviews the relevant literature. After the CAP theorem [22] proves that distributed systems with replicated data cannot achieve strong consistency, high availability and network partition tolerance at the same time, weakly consistent distributed systems are introduced by providing a weak form of guarantee about the consistency of the replicated data. The sacrifice of consistency in large scale distributed systems is often justified for commercial reasons despite program developers seeking stronger properties [47, 23]. Furthermore, weakly consistent distributed systems have a well-known programmability issue, the occurrence of the counter-intuitive results. A famous example of the programmability issue is the reappearance of removed items from Amazon shopping cart that is implemented on top of Dynamo [20].

2.1 Approaches to Resolve the Programmability Issue

There are four approaches to resolve the programmability issue in weakly consistent distributed systems: i) developing data types and programming languages that support data replication, ii) devising proof techniques for distributed programs using replicated data, iii) designing formal models for data replication and weak consistency, and iv) providing new distributed systems that do not exhibit the issue.

First, there have been attempts to resolve the issue by developing data types and programming languages that support data replication [45, 28, 31, 35]. For example, Shapiro *et al.* [45] introduce various Conflict-free Replicated Data Types (CRDTs) which allow local updates without coordination with other replicas by guaranteeing the strong eventual consistency (SEC) property. The advantage of using this approach lies in the provision of properties guaranteed by replicated data types and programming languages. This enables programmers using those types and languages to develop programs without concerns about breaking those properties. However, since properties that are not

provided can still affect the program development, it is important to provide users with as many properties as possible. Our research took this approach and developed a new language which provides new properties. This approach and the rationale for using this approach will be discussed further in the next two sections.

The second approach involves devising proof techniques for distributed programs using replicated data. It is an extensively studied approach [25, 32, 40, 8, 50, 48, 39]. Gotsman *et al.* propose the proof rule for various consistency guarantees to ensure the integrity of a given data invariant and provide an automated SMT-based reasoning tool for the rule. This approach can provide properties that are not supported by the existing programming languages and replicated data types through the use of reasoning tools. The use of this approach involves extra work on the part of programmers, including learning and being familiarised with the reasoning tools and providing extra information such as specs for the properties they want to guarantee.

Another approach to resolving the programmability issue is to design formal models for data replication and weak consistency [12, 14, 15, 27, 44, 24]. For example, Burckhardt *et al.* [15] propose a framework for verifying the implementations of replicated data types using the formal specifications of those types. This approach aids understanding of the way weak consistency operates and facilitates the verification of formal models. Nevertheless, reducing the gap between the models and the implemented system requires considerable skills and expertise and this should be considered when designing the formal models.

The fourth approach involves the development of new weakly consistent distributed systems that can overcome the programmability issue. Distributed systems with replicated data provide five different levels of consistency, namely, quiescent consistency, eventual consistency, causal consistency, sequential consistency, linearisability [12]. The first three levels of consistency are considered weak consistency and the remaining two are considered strong consistency. This approach aims to avoid the programmability issue by developing distributed systems with replicated data that has stronger consistency when compared to the existing systems [46, 9, 30, 7]. Lesani *et al.* [30] introduce a framework called Chapar for replicated key-value stores which guarantee the causal consistency, a stronger level of consistency than the eventual consistency.

Our new programming language Lasp_{PC} provides the level of consistency that falls between eventual consistency and causal consistency. This is because Lasp_{PC} does not guarantee any theoretical properties about the relation between input values. For example, causal consistency guarantees causal relations between input values but Lasp_{PC} does not provide this guarantee. However, Lasp_{PC} does provide a level of consistency that is stronger than eventual consistency. Lasp_{PC} can return the completely computed results through the $\texttt{read}_c()$ operation, while eventual consistency does not provide any

guarantee until the system reaches the state of convergence. The results containing glitches commonly occur on eventual consistency but the completely computed results from Lasp_{PC} are free from those glitches.

2.2 Replicated Data Types and Programming Languages for Replicated Data

This section focuses on reviewing studies that involve the development of replicated data types and programming languages for weakly consistent distributed systems.

Conflict-Free Replicated Data Types (CRDTs) [45] are proposed to provide replicated data types that can perform the updates locally without concerns about how two states of those types reconcile local updates after the exchanges of updates. CRDTs guarantee strong eventual consistency (SEC) which ensures the convergence (*i.e.* all the states of the type become identical) when every state has the same set of updates [45]. However, SEC cannot be guaranteed if the system does not converge. In fact, the users might continuously generate updates and the system might never converge. Furthermore, SEC does not provide any guarantees regarding the correctness of functionalities of CRDTs.

There have been some attempts to improve the performance of CRDTs. First, delta state replicated data types [2] are introduced in order to reduce the size of transmitted data for state exchanges. Second, Briquemont *et al.* propose conflict-free partially replicated data types (CPRDTs) [10] in order to use small capacity devices as participants of the system. However, they are based on CRDTs and what they guarantee also remains the same as CRDTs. Third, in order to support data types with non-commutative operations, Mergeable Replicated Data Types (MRDT)[28] are introduced. One of the constraints of CRDTs is that the operations of the data type have to be commutative, and MRDTs aim to eliminate this constraint. By using invertible relational specifications, MRDTs can support the Queue data type and its non-commutative operations such as `pop`() and `push`().

As for the programming languages, there have been two significant developments. First, Lewchenko *et al.* introduced conflict-aware replicated data types (CARD) and its refinement-typed programming language CAROL[31]. The *consistency guard* in CAROL ensures that the invariants programmers intend to guarantee are satisfied. It also allows them to control the consistency of the program. Second, the Lasp programming language [35, 36] uses CRDTs [45] with the dataflow-based computation [37]. The inputs and outputs of dataflow operations are CRDTs and the language supports functional programming operations such as `map`() or `fold`(). Lasp shows its practicality in a large scale evaluation [38]. It is also used for edge computing systems [17].

2.3 The Research Context

A majority of studies attempt to resolve the programmability issue by developing replicated data types, proving the correctness of the functionality of the existing data types, or guaranteeing the correctness of programs developed using those data types. Most studies on the existing replicated data types or programs focus on proving the correctness of individual data types. However, the way the programs are implemented involves the interaction between several data types and there is a dearth of research that properly considers this. For instance, the OR set data type, one of the replicated data types, is the type that has been extensively researched. Nonetheless, despite the $\cup$ operation between two different sets being a common operation in a non-replicated set, it has still not been clearly identified how the output can be computed if the operation is carried out using two OR sets.

Lasp [35], an example of the first approach, is a dataflow-style programming language for distributed programs with data replication and weak consistency. It attempted to address this problem by employing the idea of combining the CRDTs [45] and dataflow-based computations [37]. The example program we developed is based on this programming language. Programs in Lasp are written in a dataflow style - computations are expressed as a directed acyclic graph. The directed acyclic graph consists of a set of input nodes and a set of intermediate nodes. The set of input nodes receives updates from users and it models a mutation against an abstract data type. The set of intermediate nodes computes new values along edges in the graph that are derived from the updates performed against the set of input dataflow nodes.

Operations in Lasp are performed locally, without coordinating with any other replica in the system, and Lasp's runtime system ensures that all replicas of all dataflow nodes in the system propagate their updates to every corresponding replica in the system. The way Lasp operates ensures that all Lasp programs meet the first two functionalities described in Chapter 1: decentralised communication and high responsiveness. The first functionality, decentralised communication, is met because each replica in Lasp sends its updates to all other replicas in the system without the centralised server. The second functionality, high responsiveness, is also fulfilled as each replica in the system runs an instance of the Lasp program with its own copy of data. However, by design, Lasp does not provide a formal guarantee about convergence that is strong enough to entail our convergence requirement (strong convergence, the third functionality). It may even fail to achieve this weak guarantee when extended with higher-order functions and aggregation operations. We demonstrate how this issue occurs in Section 3.2.

Lasp is also close to the general-purpose programming language. Yet, as it is developed using the CRDTs, it also inherited what the CRDTs guarantee (Strong Eventual

Consistency [45]) and their limitations. This implies that it might generate counter-intuitive results before the system reaches the convergence state. Furthermore, the existing notion of CRDTs including the OR set data type [6] is not sufficient to express dataflow-style distributed programs. For example, the information about how input data is propagated, a path of data, is required to check whether the data is completed computed in order to remove the counter-intuitive values from the results.

Hence, this research takes the first approach and aims to overcome those limitations by introducing a new programming language through the extension of Lasp. Having explained which of the four approaches was chosen for this research in this chapter, we explore the limitations in more detail and describes the direction of my research in the next chapter.

Chapter 3

Rationale for Introducing a New Programming Language

In this chapter, we explain in detail the rationale for introducing a new programming language that supports all of dataflow-style computation, data replication, and weak consistency. The key functionalities of the new language are introduced and explained in Section 1, and we give an example program that demonstrates the importance of those functionalities in Section 3.1. The program is an online mobile game which can be played by a large number of users on mobile devices. In Section 3.2, we demonstrate the difficulties of fulfilling the third functionality, strong convergence for Lasp [35] and other languages for weakly consistent systems. We also identify potential challenges, present an informal design of a new language that uses provenance and cover information, and demonstrate how the new language fulfils the key functionalities.

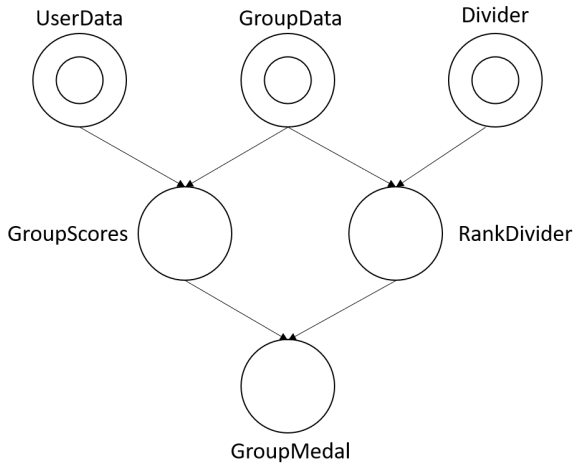


Figure 3.1: Dataflow of the example program

3.1 Specification of the example program

In order to understand how the example program demonstrates the usage of three key functionalities, it is necessary to explore how the dataflow of the program is structured and how the results are computed. Figure 3.1 illustrates the structure of the dataflow of our program as a directed acylic graph. The graph has three nodes where input updates are inserted. Each of these nodes implements a finite map data structure using bag data types, as described below:

- *UserData.* It stores the scores of plays for each user in the form of a finite function from user identifiers to sets of scores-per-play.
- *GroupData.* It stores information about the membership of each group in the form of a finite map from group identifiers to sets of group members.
- *Divider.* It stores the threshold information for gold, silver, and bronze medals in the percentage format, calculated using the ranking of all groups. It is a finite map from medals to percentages in the ranking.

Program users interact with the above data structures by performing operations with APIs. These operations generate unique update identifiers, which get propagated through the directed acyclic graph.

There are three computations that occur in the graph: two that generate intermediate results, and one that generates the final result from the intermediate ones. The two intermediate computations do the following:

- *GroupScores.* For every existing group, `GroupScores` aggregates all the scores from all the users in the group and sums them to compute a total score for the group. The results are stored in a finite map from group identifiers to total scores;
- *RankDivider.* The node `RankDivider` computes the medal thresholds in the global ranking by counting the number of groups and converting the medal thresholds in percentage to position in the ranking. The results are stored in a finite map from medals to positions in ranking.

The final results stored in `GroupMedal` are computed by sorting groups according to their scores and assigning medals to them based on their positions in the ranking and thresholds for medals.

3.2 Guaranteeing strong convergence

3.2.1 Problems

The difficulty of guaranteeing the third functionality, strong convergence, arises from two types of glitch behaviours in dataflow-style programs on distributed systems.

- **Asynchronous computation glitches.** These glitches occur when it is not guaranteed that updates are fully propagated in the dataflow graph before getting a value from the final results. For instance, a user may choose to remove herself from her current group and create a new group containing only herself. If we do not wait for the information about this update to be propagated to all paths from `GroupData` to `GroupMedal` (through `GroupScores` and `RankDivider`), it means that the information goes through `GroupScores` only. In this case, we would get an incorrect medal assignment.

- **Asynchronous communication glitches.** These glitches arise when the copies of a dataflow node at different replicas compute results with overlapping but not identical sets of input updates. For example, consider the case where the instances of `UserData` at two replicas store the same information about *some* input updates from previous synchronization, but the information the two instances have is not identical due to the accumulation of new plays performed only on one of those replicas after synchronization. In this case, the dataflow graphs at these two replicas may have different results at their final nodes, and the system must find a way to reconcile these differences. We can make this example more concrete by assuming the following situation: if both instances of `UserData` at two replicas contain a result showing that Player 1 (in the one-person team) scored 2 points in their penultimate games and then played a game on each device where Player 1 scored 1 point each in their last games, the resulting team score (stored in `GroupScores`) should be 4; however, without additional information such as meta-data, the system is unable to detect that none of the following scores is a correct outcome: 3 (incorrect, assuming equivalence), 5 (incorrect, not enough information to decide this way), and 6 (incorrect, combining the results via addition, assuming the disjointedness of input updates).

3.2.2 Basic Concepts for the Solutions

Having identified the issues with glitch behaviours, it becomes clear that there is a need to address those issues. What should a programming language provide in order to alleviate the problems arising from the two glitches? Our solutions are i) to utilise and maintain the *provenance* of intermediate results in a dataflow using or extending the

formalism based on semi-ring [26] and semi-module [4] from the database community, ii) to keep track of the knowledge about completed computations of intermediate results using *cover* and iii) to provide an operation for querying the state of any one replica so that it returns a *consistent* result. The operation in the third solution works in a way that it uses all meta-data information of the state of the replica (including provenance and cover from the first two solutions) and returns a result corresponding to some consistent execution, a consistent result. Here, consistent executions can be treated as the non-distributed executions on a single node and the details of the consistent execution can be found in Chapter 5.

The first two solutions are related to the kind of information required in order to resolve the issues with glitch behaviours and the third solution is about the way this information is used for the resolution to the issues. In this section, we focus on demonstrating how the first two solutions work using examples and how these solutions are used to return consistent results in the third solution. The formal definitions and proofs for the third are discussed in Chapter 5. The key idea of the first two solutions is that every node in the dataflow stores not just the results of computation but also the information about which end-user updates are used for and fully incorporated to the results. This information is updated or modified by the node's successors in the dataflow. As we also demonstrate using the proofs in the subsequent chapters, such information is used to resolve the issues caused by asynchronous computation and asynchronous communication.

Before we explore the details of our solutions, we need to explain the basic concepts using the node `GroupScores` illustrated in Figure 3.1. `GroupScores` keeps sets of key-value pairs and each pair consists of a group id and a score of the group. The node's high-level objective is to store the intermediate results of the dataflow. In our implementation, the representation of `GroupScores` is enriched with *provenance* and *cover* information. For example, one possible (and naive) representation of the node `GroupScores` is as follows:

$$\begin{aligned}(A, L, \mathcal{C}, K) = (&\{(i_1, _), (j_1, _)\},\\ &\{(i_1, _), (j_1, _)\},\\ &\{\{(i_1, _), (j_1, _)\}\},\\ &[\{(i_1, _), (j_1, _)\} \mapsto \{((g_1, 100), (i_1, _) \wedge (j_1, _))\}]).\end{aligned} \tag{3.1}$$

When a data item is inserted at the input node such as the node `UserData` or `GroupData`, the identifier of the insertion is also stored. The i_k are the identifiers of the insertion that adds a score of a play at the node `UserData`, and the j_k are the identifiers of the insertion that adds a player to a group at the node `GroupData`. We call these identifiers *events*. Each event is stored with its *path* annotation and this path annotation represents a path in the dataflow that goes from the input node, where the event was inserted, to the node

`GroupScores`. The symbol _ is used for the simplicity of the representation. It denotes the unique path from the input node of the associated event to the node `GroupScores`. An *event-history* is a pair of event and its path annotation. The representation (3.1) consists of four components $(A, L, \mathcal{C}, K)$: a set of all known event-histories (A), a set of live event-histories (L), a collection of subsets of A ($\mathcal{C}$), and a map from subsets in $\mathcal{C}$ to sets of data-provenance pairs (K). The component A records all the event-histories that have been propagated to this representation along the dataflow. If delete operations are used, some of these event-histories will be cancelled, and information about this cancellation will also be propagated along the dataflow. The component L records such cancellations indirectly by keeping record of uncancelled events. This indirect approach is a common optimisation technique for saving computation time and space. The third component $\mathcal{C}$ is a collection of subsets of A. The union of all subsets in $\mathcal{C}$ becomes A, and we call $\mathcal{C}$ *cover* of A. The last component K is a map from subsets in $\mathcal{C}$ to sets of provenance-annotated data items called *snapshots*. The details of how cover and snapshots are working are explained later in this section with examples.

3.2.3 How Provenance Works

Having explored the basic concepts used in the representations, we now explain the first solution to resolve the issues with glitch behaviours. As for the details of the solution, our idea of using and maintaining provenance is demonstrated using the representations (3.1)-(3.4) of the node `GroupScores`. The parts that demonstrate the usage and maintenance of the provenance are highlighted in red. A *provenance* is a disjunctive normal form of event-histories, where the event-histories are treated as atomic predicates. Each disjunct in a provenance describes the origin of the associated data item and each event-history in a disjunct represents the data item that is used to compute the associated data item of the disjunct. For example, the provenance of $((g_1, 100), (i_1, _) \wedge (j_1, _))$ in (3.1) has just one disjunct $(i_1, _) \wedge (j_1, _)$, which tells us that $(g_1, 100)$ was computed because one player joined the group (expressed by $(j_1, _)$) and the group gets the score of 100 reflecting the total score of this player (expressed by $(i_1, _)$). If the same player gets another score of 200 and the score is inserted with the event-history $(i_2, _)$, then the updated representation of `GroupScores` would be:

$$\begin{aligned}
&(\{(i_1, _), (i_2, _), (j_1, _)\},\\
&\ \{(i_1, _), (i_2, _), (j_1, _)\},\\
&\ \{\{(i_1, _), (i_2, _), (j_1, _)\}\},\\
&\ [\{(i_1, _), (i_2, _), (j_1, _)\} \mapsto \{((g_1, 300), (i_1, _) \wedge (i_2, _) \wedge (j_1, _))\}]).
\end{aligned} \tag{3.2}$$

Since every node in our dataflow graph, including the node `GroupScores`, uses a *set* data type, there are two occasions when a disjunct is added to the provenance of a data item in such a set: the identical data item is added multiple times to the set by

different insert operations by the end user, or the identical data item is computed by the dataflow operation multiple times with different collections of inputs. For example, assume that another instance of the node `GroupScores` on a different mobile device has the following representation:

$$\begin{array}{l}(\{(i_3, _), (j_1, _)\},\\ \{(i_3, _), (j_1, _)\},\\ \{\{(i_3, _), (j_1, _)\}\},\\ [\{(i_3, _), (j_1, _)\} \mapsto \{((g_1, 300), (i_3, _) \wedge (j_1, _))\}]).\end{array} \tag{3.3}$$

The representation (3.3) indicates that the same player played a new game with this different device and scored 300. After the first device that has the representation (3.2) receives the information about another $(g_1, 300)$ and merges it with its own representation, the merged representation is as follows:

$$\begin{array}{l}(\{(i_1, _), (i_2, _), (i_3, _), (j_1, _)\},\\ \{(i_1, _), (i_2, _), (i_3, _), (j_1, _)\},\\ \{\{(i_1, _), (i_2, _), (i_3, _), (j_1, _)\}\},\\ [\{(i_1, _), (i_2, _), (i_3, _), (j_1, _)\} \mapsto\\ \quad \{((g_1, 300), ((i_1, _) \wedge (i_2, _) \wedge (j_1, _)) \vee ((i_3, _) \wedge (j_1, _)))\}]).\end{array} \tag{3.4}$$

Due to the identical data item $(g_1, 300)$ from another instance, the disjunct $(i_3, _) \wedge (j_1, _)$ is added in the provenance of $(g_1, 300)$ in the representation (3.4).

3.2.4 How Cover Works

We move on to the second solution, our idea of using and maintaining cover. The parts that demonstrate the usage and maintenance of the cover are included in the representations (3.5)-(3.13) and highlighted in red. We explain it using the two nodes `GroupScores` and `UserData` illustrated in Figure 3.1. The details of the node `GroupScores` have been explained in the previous example. `UserData` keeps sets of key-value pairs and each pair consists of a user id and a score of the user. The high-level objective of the node `UserData` is to store the input data of the dataflow. Maintaining cover works differently depending on whether a node stores the input data or intermediate results of the dataflow. Firstly, we explain how cover works at the input node `UserData`. One possible (and naive) representation of the node `UserData` is as follows:

$$\begin{array}{rl}(A, L, \mathcal{C}, K) = & (\{(i_1, _)\},\\ & \{(i_1, _)\},\\ & \{\{(i_1, _)\}\},\\ & [\{(i_1, _)\} \mapsto \{((u_1, 100), (i_1, _))\}]).\end{array} \tag{3.5}$$

The third component $\mathcal{C}$ ($\{\{(i_1, _)\}\}$) in (3.5) is *cover* of the first component A ($\{(i_1, _)\}$). $\mathcal{C}$, cover of A, should satisfy the following three properties:

- Every member of $\mathcal{C}$ is a subset of A;

- The union of all subsets in $\mathcal{C}$ becomes A;
- No two subsets in $\mathcal{C}$ satisfy the $\subseteq$ relation.

Each subset in $\mathcal{C}$ is mapped to a set of provenance-annotated data items (a *snapshot*) by the fourth component K and this relation implies that if every event-history in a subset in $\mathcal{C}$ and its data item are fully computed, the result would be identical to the mapped snapshot. In the representation (3.5), the snapshot $\{((u_1, 100), (i_1, _))\}$ is the result of complete computation of the event-history $(i_1, _)$ and its data item $(u_1, 100)$. Cover $\mathcal{C}$ in every input node has an extra property: $\mathcal{C}$ has only one subset as a member. In order to satisfy the three properties of cover, the only subset in $\mathcal{C}$ on an input node should be the same as A ($\mathcal{C} = \{A\}$). Assume that the new data item $(u_1, 200)$ and its event-history $(i_2, _)$ is inserted in the representation (3.5). Then, the updated representation is:

$$\begin{array}{l}(\{(i_1, _), (i_2, _)\}, \\ \{(i_1, _), (i_2, _)\}, \\ \{\{(i_1, _), (i_2, _)\}\}, \\ [\{(i_1, _), (i_2, _)\} \mapsto \{((u_1, 100), (i_1, _)), ((u_1, 200), (i_2, _))\}]).\end{array} \tag{3.6}$$

The new event-history is added to the existing subset in $\mathcal{C}$ and this still satisfies the extra property for cover in every input node. Assume that another instance of the node `UserData` on a different mobile device has the following representation:

$$\begin{array}{l}(\{(i_3, _)\}, \\ \{(i_3, _)\}, \\ \{\{(i_3, _)\}\}, \\ [\{(i_3, _)\} \mapsto \{((u_1, 300), (i_3, _))\}]).\end{array} \tag{3.7}$$

After the representation (3.6) receives the information about the representation (3.7) and merges it, the merged representation becomes:

$$\begin{array}{l}(\{(i_1, _), (i_2, _), (i_3, _)\}, \\ \{(i_1, _), (i_2, _), (i_3, _)\}, \\ \{\{(i_1, _), (i_2, _), (i_3, _)\}\}, \\ [\{(i_1, _), (i_2, _), (i_3, _)\} \mapsto \{((u_1, 100), (i_1, _)), ((u_1, 200), (i_2, _)), ((u_1, 300), (i_3, _))\}]).\end{array} \tag{3.8}$$

The merged subset $\{(i_1, _), (i_2, _), (i_3, _)\}$ in (3.8) is calculated by the union of two subsets $\{(i_1, _), (i_2, _)\}$ and $\{(i_3, _)\}$ in (3.6) and (3.7), respectively.

Secondly, we now explain how cover works at the intermediate node `GroupScores`. Assume that the representation of the node `GroupScores` is as follows:

$$\begin{array}{l}(\{(i_1, _), (j_1, _)\}, \\ \{(i_1, _), (j_1, _)\}, \\ \{\{(i_1, _), (j_1, _)\}\}, \\ [\{(i_1, _), (j_1, _)\} \mapsto \{((g_1, 100), (i_1, _) \wedge (j_1, _))\}]).\end{array} \tag{3.9}$$

After the new data item $(u_1, 200)$ and its event-history $(i_2, _)$ is added to the node `UserData` and this information is propagated to the node `GroupScores`, the representation (3.9) is updated as follows:

$$\begin{array}{l}(\{(i_1, _), (i_2, _), (j_1, _)\},\\ \{(i_1, _), (i_2, _), (j_1, _)\},\\ \{\{(i_1, _), (i_2, _), (j_1, _)\}\},\\ [\{(i_1, _), (i_2, _), (j_1, _)\} \mapsto \{((g_1, 300), (i_1, _) \wedge (i_2, _) \wedge (j_1, _))\}]).\end{array} \tag{3.10}$$

Note that the addition of the new data item happened at the input node of the same instance. As a result, the new event-history of the newly added item is added to the existing subset of cover. Now, we show how cover is updated after a new data item is added to the different instance. Assume that another instance of the node `GroupScores` on a different mobile device has the following representation:

$$\begin{array}{l}(\{(i_3, _), (j_1, _)\},\\ \{(i_3, _), (j_1, _)\},\\ \{\{(i_3, _), (j_1, _)\}\},\\ [\{(i_3, _), (j_1, _)\} \mapsto \{((g_1, 300), (i_3, _) \wedge (j_1, _))\}]).\end{array} \tag{3.11}$$

The representation (3.11) shows that the new data item $(u_3, 300)$ and its event-history $(i_3, _)$ is added to the different instance of the node `UserData` and this information is propagated to the different instance of the node `GroupScores`. After the representation (3.10) receives the information about the representation (3.11) and merges it, the merged representation would be:

$$\begin{array}{l}(\{(i_1, _), (i_2, _), (i_3, _), (j_1, _)\},\\ \{(i_1, _), (i_2, _), (i_3, _), (j_1, _)\},\\ \{\{(i_1, _), (i_2, _), (j_1, _)\}, \{(i_3, _), (j_1, _)\}\},\\ [\{(i_1, _), (i_2, _), (j_1, _)\} \mapsto \{((g_1, 300), (i_1, _) \wedge (i_2, _) \wedge (j_1, _))\},\\ \{(i_3, _), (j_1, _)\} \mapsto \{((g_1, 300), (i_3, _) \wedge (j_1, _))\}]).\end{array} \tag{3.12}$$

Unlike the input nodes, the intermediate nodes cannot compute the result of the merged cover by the union of their subsets. If it was the case of input nodes, the merged cover would be $\{\{(i_1, _), (i_2, _), (i_3, _), (j_1, _)\}\}$. Instead, at the intermediate nodes such as `GroupScores`, the merged cover $\{\{(i_1, _), (i_2, _), (j_1, _)\}, \{(i_3, _), (j_1, _)\}\}$ is computed by combining two covers $\{\{(i_1, _), (i_2, _), (j_1, _)\}\}$ and $\{\{(i_3, _), (j_1, _)\}\}$. As a result, the subsets in cover remain the same.

If, unlike the example above, the combined result violates the third property of cover, which means there are two subsets C_i and C_j in the combined result such that $C_i \subseteq C_j$, the merged cover removes C_i and the snapshot for C_i is merged into the snapshot for C_j. If the node `UserData` of the original instance receives the information about $(u_3, 300)$ from the different instance, merges it, and propagates it to the node `GroupScores`, then

the representation of the node `GroupScores` becomes:

$$
\begin{aligned}
&(\{(i_1, _), (i_2, _), (i_3, _), (j_1, _)\},\\
&\ \{(i_1, _), (i_2, _), (i_3, _), (j_1, _)\},\\
&\ \{\{(i_1, _), (i_2, _), (i_3, _), (j_1, _)\}\},\\
&\ [\{(i_1, _), (i_2, _), (i_3, _), (j_1, _)\} \mapsto\\
&\qquad \{((g_1, 300), ((i_1, _) \wedge (i_2, _) \wedge (j_1, _)) \vee ((i_3, _) \wedge (j_1, _)))\}]).
\end{aligned}
\tag{3.13}
$$

Note that during the process of computing the merged cover, the subset $\{(i_1, _), (i_2, _), (i_3, _), (j_1, _)\}$ in (3.13) remains and the subsets $\{(i_1, _), (i_2, _), (j_1, _)\}$ and $\{(i_3, _), (j_1, _)\}$ in (3.12) are removed because

$$
\begin{aligned}
\{(i_1, _), (i_2, _), (j_1, _)\} &\subseteq \{(i_1, _), (i_2, _), (i_3, _), (j_1, _)\} \qquad \text{and}\\
\{(i_3, _), (j_1, _)\} &\subseteq \{(i_1, _), (i_2, _), (i_3, _), (j_1, _)\}.
\end{aligned}
$$

3.2.5 Consistent Results

How the third solution uses provenances and cover information is demonstrated using the node `GroupMedal` illustrated in Figure 3.1. `GroupMedal` keeps sets of key-value pairs and each pair consists of a group id and the medal of the group. The node's high-level objective is to store the final result of the dataflow. For example, one possible (and naive) representation of the node `GroupMedal` is as follows:

$$
\begin{aligned}
&(\{(i_1, _), (i_2, _), (i_3, _), (i_4, _), (i_5, _), (j_1, l), (j_1, r), (j_2, l), (j_2, r), (j_3, l), (d_1, _)\},\\
&\ \{(i_1, _), (i_2, _), (i_3, _), (i_4, _), (j_1, l), (j_1, r), (j_2, l), (j_2, r), (j_3, l), (d_1, _)\},\\
&\ \{\{(i_1, _), (j_1, l), (j_1, r), (d_1, _)\},\\
&\quad \{(i_2, _), (i_3, _), (i_4, _), (i_5, _), (j_2, l), (j_2, r), (j_3, l), (d_1, _)\}\},\\
&\ [\{(i_1, _), (j_1, l), (j_1, r), (d_1, _)\} \mapsto\\
&\qquad \{((g_1, \mathit{bronze}), (i_1, _) \wedge (j_1, l) \wedge (j_1, r) \wedge (d_1, _))\},\\
&\quad \{(i_2, _), (i_3, _), (i_4, _), (i_5, _), (j_2, l), (j_2, r), (j_3, l), (d_1, _)\} \mapsto\\
&\qquad \{((g_2, \mathit{bronze}), (i_2, _) \wedge (i_3, _) \wedge (j_2, l) \wedge (j_2, r) \wedge (d_1, _)),\\
&\qquad\ ((g_1, \mathit{bronze}), (i_2, _) \wedge (i_3, _) \wedge (j_2, l) \wedge (j_2, r) \wedge (j_3, l) \wedge (d_1, _))\}]).
\end{aligned}
\tag{3.14}
$$

In the representation (3.14), unlike i_k and d_k which have only one path annotation, there are two path annotations for the events (j_k) from the node `GroupData`: l and r. The former represents the left path, `GroupData - GroupScores - GroupMedal`, and the latter represents the right path, `GroupData - RankDivider - GroupMedal`.

The main goal of the third solution is to provide the read operation that returns *a consistent result*. A consistent result is the result that is returned by the read operation from the so-called glitch-free execution, the execution with a single replica and the complete computation. In the glitch-free execution, there is only one replica in the system. This prevents the result from having asynchronous communication glitches; there are simply no replicas to communicate. Furthermore, every result from the glitch-free execution can only be returned after all dataflow operations are completely computed and this means that there is no asynchronous computation glitch.

Now, we explain how we can identify a consistent result using provenance and cover information. A consistent result can be identified by checking whether the following two properties are satisfied:

- *The property about the input data.* The result should be computed using input data from all input nodes that reach the final node. In case of the final node `GroupMedal` in Figure 3.1, the result should be computed using input data from the node `UserData` is inserted with the events i_k, the node `GroupData` with the events j_k and the node `Divider` with the events d_k.

- *The property about the complete dataflow computation.* Among the input nodes in the first property, if there is an input node that has multiple paths to reach the final node, the result should be computed using all event-histories that have all path annotations for this input event. In case of the final node `GroupMedal` in Figure 3.1, such input events are the events from the node `GroupData` (j_k) and their path annotations are l and r. Therefore, the provenance of elements in the result should contain (j_k, l) and (j_k, r).

The parts related to the two properties above are highlighted in red in (3.14). The consistent result for the first snapshot is (g_1, bronze). The provenance of (g_1, bronze) contains events from all input nodes: i_1, j_1 and d_1 (the first property). In addition, the event-histories that contain j_1 also contain all path annotations to reach the node `GroupMedal`: (j_1, l) and (j_1, r) (the second property). The consistent result for the second snapshot is (g_2, bronze) and it also satisfies the two properties as highlighted.

Chapter 4

Observed-Remove Set with Provenance and Cover

The system-level data type of our programming language for expressing weakly-consistent distributed computations is based on an extension of a set data type, called Observed-Remove (OR) set [6]. Our further extension adds two new features — provenance and cover — to this data type. These features force the Observed-Remove sets with provenance and cover (the OR_{PC} sets) used in a dataflow to track important information about how computation has been processed in the dataflow. The tracked information leads to a principled use of the dataflow which guarantees a formal relationship between the results of the dataflow in our distributed setting and those of the same dataflow in the sequential setting (strong convergence). Such a relationship helps reduce the difficulty of developing distributed programs that run under weak consistency by guaranteeing strong convergence. The details of this relationship and proofs are discussed in the next chapter. This chapter discusses the definition and operations of the system-level data type, the OR_{PC} set.

4.1 Basic concepts

Let $\mathbb{N}$ be a set of non-negative integers, $\mathcal{V}$ a countably infinite set of the ids of dataflow nodes, and $\mathcal{R}$ a finite set of replica ids such that $\mathcal{R}$ and $\mathcal{V}$ are disjoint.

Definition 4.1.1. An *event* is a tuple $e = (v, r, n) \in \mathcal{V} \times \mathcal{R} \times \mathbb{N}$ such that $n > 0$. We use $\mathcal{E}$ to denote the set of all events.

An event (v, r, n) represents the identifier of an operation performed on an OR_{PC} set. Specifically, it denotes the identifier of the n-th `insert`() operation that is executed on the dataflow node v at the replica r. We write $e.v$, $e.r$, and $e.n$ to mean the components of the event e.

Definition 4.1.2. A *dataflow path* t_h for a node $v \in \mathcal{V}$ is defined:

$$t_h ::= (v, \texttt{undef}, \texttt{undef}) \mid (v, t'_h, \texttt{undef}) \mid (v, \texttt{undef}, t'_h) \mid (v, t''_h, \texttt{unk}) \mid (v, \texttt{unk}, t''_h)$$

where t'_h and t''_h are (smaller) dataflow paths for some dataflow nodes. We let $\mathcal{T}_h$ be the set of all dataflow paths.

A dataflow path t_h for a node v in $\mathcal{V}$ represents a path over the dataflow graph that goes from a leaf of the graph, an input node, to the node v. The first case $(v1, \texttt{undef}, \texttt{undef})$ means that the path consists of only one node and this unique node is $v1$ itself. Here, `undef` indicates that the path for the child node does not exist. This implies that the node $v1$ is an input node. The remaining four cases deal with the situation where the path has the depth greater than 2 and has both leaf and non-leaf nodes. The second and third cases, $(v2, t'_h, \texttt{undef})$ and $(v2, \texttt{undef}, t'_h)$, describe that the last operation performed in the path (the one that produces the input for the current node $v2$) is either unary operations or the $\cup$ operation. It is clear that the unary operations produce the paths contain `undef` because they have a single child node as a parameter. The $\cup$ operation also produces the paths contain `undef` (the left child node for the second case and the right child node for the third case) because the values from both child nodes exist in the results independently. However, the results of the $\times$ operation are generated by the combination of two values from both child nodes and this means that the paths for the both child nodes do exist. The final two cases cover this $\times$ operation. The penultimate case expresses that t''_h is a path producing the left input of the $\times$ operation, and the last case that it is a path producing the right input of the operation. `unk` indicates that the path for the child node exists, but it is unknown.

Definition 4.1.3. An *event-history* h is a pair of an event and a dataflow path, i.e. $h = (e, t_h)$, such that $(e.v, \texttt{undef}, \texttt{undef})$ is a leaf of t_h. The event-histories are ordered by

$$(e, t_h) \sqsubseteq (e', t'_h) \iff t_h = t'_h \wedge e \sqsubseteq e'$$

where

$$e \sqsubseteq e' \iff e.v = e'.v \wedge e.r = e'.r \wedge e.n \leq e'.n.$$

We use $\mathcal{H}$ to denote the set of all event-histories.

The dataflow path of an event-history (e, t_h) expresses a path over the dataflow graph through which the event e has been propagated to the node of the path t_h. When a node performs a dataflow operation, it takes not just values from its input nodes but also event-histories that give rise to those values. Then, the node updates the dataflow paths of those event-histories to add information about the node and its operation.

Definition 4.1.4. A *dot* d is a finite set of event-histories. We write $\mathcal{D}$ for the set of all dots.

At a high level, an OR_{PC} set denotes a set of values, just like the standard set data types. But unlike those data types, the OR_{PC} set associates each value in the set with meta-data that makes it possible to deal with some issues from weakly-consistent distributed dataflow computation, such as a conflict resolution. The meta-data for each value u is a finite set of dots $p = \{d_1, \ldots, d_n\}$. Each d_i describes one of the reasons of u's existence in the set: a value u in the set exists because i) the users inserted values with the associated events of the event-histories in d_i into the input nodes that implement OR_{PC} sets, ii) these inserted values and their event-histories got transformed and combined along multiple paths in the dataflow, and iii) the results of these transformations and combinations together gave rise to the value u with d_i in the current set. Notice the sense of conjunction in this reading of d_i, and the sense of disjunction in the reading of $p = \{d_1, \ldots, d_n\}$. This means that even if some reasons for the existence (dots in p) become invalid (due to some `insert`() operations get cancelled by corresponding `delete`() operations) and one reason (a dot in p) survives, the value u still remains in the set. We introduce a terminology, *provenance*, to refer to p.

Definition 4.1.5. A *provenance* p is a finite collection of dots. Provenances are ordered by the subset relation:

$$p \sqsubseteq p' \iff p \subseteq p'.$$

We write $\mathcal{P}$ for the poset of provenances.

Provenances form a commutative semiring $(\mathcal{P}, 0, \oplus, 1, \otimes)$ whose constants and operators are defined as follows:

1. $0 = \{\} \in \mathcal{P}$ and $1 = \{\{\}\} \in \mathcal{P}$.

2. $\oplus$ is a binary operator on $\mathcal{P}$ defined by

$$p \oplus p' = p \cup p' \text{ for all } p, p' \in \mathcal{P}.$$

3. $\otimes$ is a binary operator on $\mathcal{P}$ defined by

$$p \otimes p' = \{d \cup d' \mid d \in p \wedge d' \in p'\} \text{ for } p, p' \in \mathcal{P}.$$

Note that $p \sqsubseteq p'$ if and only if $p \oplus p'' = p'$ for some p''. This equivalence is a standard way of inducing a preorder for a given semiring. In our case, this induced preorder is a partial order. It means that our semiring is *naturally ordered* in the terminologies of database researchers.

Lemma 4.1.6. *The collection of provenances $\mathcal{P}$ is a join semi-lattice in the sense that for every finite subset P of $\mathcal{P}$, it has the least upper bound of P, defined as follows:*

$$\bigsqcup P = \bigcup_{p \in P} p.$$

Notice that

$$p \sqcup p' = p \oplus p' \quad \textit{and} \quad \bot = 0.$$

Proof. $(\mathcal{P}, \sqsubseteq)$ is a poset, because $\subseteq$ is a partial order (i.e., $\subseteq$ is reflexive, transitive, and anti-symmetric). It remains to show that $p \sqcup p'$ is the least upper bound of p and p', and that $\bot$ is the least element in this poset. The latter is immediate because $\bot$ is the empty set ($0 = \{\}$), which is included in all sets. To prove the former, consider $p, p' \in \mathcal{P}$. Since p and p' are finite collections of dots, their union is also a finite collection of dots. Thus, $p \sqcup p' = p \cup p'$ is a provenance. Furthermore, since

$$p \subseteq (p \cup p') = (p \sqcup p') \quad \text{and} \quad p' \subseteq (p \cup p') = (p \sqcup p'),$$

$p \sqcup p'$ is an upper bound of p and p'. The only remaining part of the proof is to show that $p \sqcup p'$ is the *least* upper bound. Consider an upper bound p'' of p and p':

$$p \sqsubseteq p'' \quad \text{and} \quad p' \sqsubseteq p''.$$

Thus,

$$p \subseteq p'' \quad \text{and} \quad p' \subseteq p'',$$

which implies that $p \sqcup p' = p \cup p' \subseteq p''$. We have just shown that $p \sqcup p' \sqsubseteq p''$. □

Definition 4.1.7. Let H be a subset of $\mathcal{H}$. A *cover* $\mathcal{C}$ of H is a finite collection of subsets of H such that

$$\bigcup_{C_i \in \mathcal{C}} C_i = H.$$

This definition is straightforward from the mathematical point of view, but the way that we use it will be less obvious. At this point, we just mention that a node in a dataflow will keep a cover $\{C_0, \ldots, C_n\}$ of the set of all the event-histories known to the node, and that the cover satisfies the following (informal) invariant: if a set of event-histories $\{h_1, \ldots, h_m\}$ is included in some C_i stored in the state of a node, then the node's state also contains the outcome of every dataflow computation that satisfies two conditions: every event in the computation is $h_j.e$ for some $h_j \in C_i$, and it is processed in the computation along the associated path $h_j.t_h$.

4.2 Definition of $\mathbf{OR_{PC}}$ Sets

Let $\mathcal{U}$ be the set of all the possible elements inserted to nodes or computed by nodes in some dataflow programs.

Definition 4.2.1. A *provenance store* S is a function from $\mathcal{U}$ to the set of provenances $\mathcal{P}$ such that $S(u) = \{\}$ except for finitely many u's in $\mathcal{U}$. We use $\mathcal{S}$ to denote the set of all provenance stores.

Intuitively, $S(u) \neq \{\}$ means that the element u is in the provenance store S, and the condition says that every provenance store contains only a finite number of elements.

Definition 4.2.2. An *OR_{PC} set* is a tuple

$$X = (A, L, \mathcal{C}, K)$$

of the following components:

1. A is a finite set of event-histories that is downward-closed, i.e. for all event-histories h and h',
$$h \in A \wedge h' \sqsubseteq h \implies h' \in A.$$
2. L is a subset of A.
3. $\mathcal{C}$ is a cover for A such that for all C_i, C_i' in $\mathcal{C}$,
$$C_i \neq C_i' \implies C_i \not\subseteq C_i'.$$
4. K is a function from $\mathcal{C}$ to the set of provenance stores $\mathcal{S}$ such that for all $C_i \in \mathcal{C}$, $u \in \mathcal{U}$, and $d \in K(C_i)(u)$,
$$d \subseteq L \cap C_i.$$

$\mathrm{OR_{PC}}$ sets are ordered by the following relation:

$$(A, L, \mathcal{C}, K) \sqsubseteq (A', L', \mathcal{C}', K') \iff (A \subseteq A') \wedge (L' \cap A \subseteq L) \wedge ((\mathcal{C}, K) \sqsubseteq (\mathcal{C}', K'))$$

where

$$\begin{aligned} & (\mathcal{C}, K) \sqsubseteq (\mathcal{C}', K') \\ \iff & \forall C_i \in \mathcal{C}.\ (\exists C_j \in \mathcal{C}'.\ C_i \subseteq C_j) \\ & \qquad \wedge \\ & \qquad (\forall C_j \in \mathcal{C}, u \in \mathcal{U}, d \in K(C_i)(u).\ C_i \subseteq C_j \wedge d \subseteq L' \implies d \in K'(C_j)(u)). \end{aligned}$$

We write $\mathcal{X}$ for the poset of $\mathrm{OR_{PC}}$ sets.

In an $\mathrm{OR_{PC}}$ set $X = (A, L, \mathcal{C}, K)$, the first component A represents the event-histories of all the elements inserted by the users that have contributed to make the content of the set. Some of these inserted elements can be removed by the subsequent `delete`() operations, and the event-histories of those live (not cancelled) `insert`() operations form L. The downward closure of A formalises the so called per-object causality, which means that in the case if the effect of an `insert`() operation has been incorporated to the set, those of all the preceding `insert`() operations have been incorporated to it as well. The component $\mathcal{C}$ is a cover of A, and no subset in $\mathcal{C}$ dominates another subset in $\mathcal{C}$ by the $\subseteq$ relation. The content of each subset in $\mathcal{C}$ indicates how much the dataflow has processed the elements inserted to the input nodes of the dataflow by the users. The condition on $K(C_i)(u)$ implies that the dots in the provenance $K(C_i)(u)$ should involve only live event-histories.

Intuitively, the partial order for $\mathrm{OR_{PC}}$ sets indicates the progress of computation: $X \sqsubseteq X'$ means that X' is one of the possible futures of the $\mathrm{OR_{PC}}$ set X by the computation in a dataflow. There are three types of possible futures: i) more elements are inserted to the set, ii) more elements are deleted from the set, and iii) more dataflow computations are completed. Some X' might contain all three possible changes. The condition $A \subseteq A'$ checks for i) and ensures that A' has all the information in A about `insert`() operations performed by the user and possibly more, and $L' \cap A \subseteq L$ checks for ii) and provides a similar guarantee for `delete`() operations executed by the user, which result in the cancellation of some `insert`() operations in A. Lastly, $(\mathcal{C}, K) \sqsubseteq (\mathcal{C}', K')$ checks for iii) and indicates two things: subsets in the cover $\mathcal{C}'$ capture consequences of more interactions among user-inserted elements than $\mathcal{C}$ and the provenance stores that contain only dots whose members are in L' of those subsets of $\mathcal{C}'$ are larger than the corresponding stores of $\mathcal{C}$.

Lemma 4.2.3. *The $\sqsubseteq$ relation on OR_{PC} sets is a partial order, i.e., it is reflexive, anti-symmetric and transitive.*

Proof. Let X, Y and Z in $\mathcal{X}$ be $(A, L, \mathcal{C}, K)$, $(A', L', \mathcal{C}', K')$ and $(A'', L'', \mathcal{C}'', K'')$, respectively.

- Reflexivity: $X \sqsubseteq X$. That is, $(A, L, \mathcal{C}, K) \sqsubseteq (A, L, \mathcal{C}, K)$. This holds because by

the definition of $\sqsubseteq$,

$$\begin{aligned}
& A \subseteq A && \text{(true)} \\
\wedge\ & L \cap A \subseteq L && \text{(true)} \\
\wedge\ & \forall C_i \in \mathcal{C}. \\
& \quad \{C_i' \in \mathcal{C} \mid C_i \subseteq C_i'\} = \{C_i\} \\
& \quad \wedge \\
& \quad \forall u \in \mathcal{U}.\ \{d \in K(C_i)(u) \mid d \subseteq L\} \subseteq K(C_i)(u) && \text{(true).}
\end{aligned}$$

- Anti-symmetry: $X \sqsubseteq Y$ and $Y \sqsubseteq X$ implies $X = Y$. That is,

$$\begin{aligned}
& (A, L, \mathcal{C}, K) \sqsubseteq (A', L', \mathcal{C}', K') \wedge (A', L', \mathcal{C}', K') \sqsubseteq (A, L, \mathcal{C}, K) \\
\Longrightarrow\ & (A, L, \mathcal{C}, K) = (A', L', \mathcal{C}', K').
\end{aligned}$$

We prove the required equality as follows. First,

$$(A \subseteq A') \wedge (A' \subseteq A) \implies A = A'.$$

Next,

$$\begin{aligned}
L' \cap A \subseteq L &\iff L' \cap A' \subseteq L && (A = A') \\
&\iff L' \subseteq L && (L' \subseteq A'),
\end{aligned}$$

$$\begin{aligned}
L \cap A' \subseteq L' &\iff L \cap A \subseteq L' && (A' = A) \\
&\iff L \subseteq L' && (L \subseteq A),
\end{aligned}$$

so that $L = L'$. Finally, if we pick any $C_i \in \mathcal{C}$, then

$$\exists C_j \in \mathcal{C}'.\ C_i \subseteq C_j \quad ((\mathcal{C}, K) \sqsubseteq (\mathcal{C}', K') \text{ of } X \sqsubseteq Y).$$

For C_j, we have

$$\exists C_i' \in \mathcal{C}.\ C_j \subseteq C_i' \quad ((\mathcal{C}', K') \sqsubseteq (\mathcal{C}, K) \text{ of } Y \sqsubseteq X).$$

If $C_i \neq C_i'$, then

$$C_i \subseteq C_i' \quad (C_i \subseteq C_j \wedge C_j \subseteq C_i').$$

This breaks one of the conditions for $\mathcal{C}$ that no subset in $\mathcal{C}$ dominates another subset in $\mathcal{C}$ by the $\subseteq$ relation. Therefore, $C_i = C_i'$. This implies

$$C_i = C_j \quad (C_i \subseteq C_j \wedge C_j \subseteq C_i)$$

and

$$\forall C_i \in \mathcal{C}.\, C_i \in \mathcal{C}'.$$

Thus, $\mathcal{C} \subseteq \mathcal{C}'$. The proof for the other inclusion ($\mathcal{C}' \subseteq \mathcal{C}$) is similar. What we have shown implies $\mathcal{C} = \mathcal{C}'$. Now, for any $C_i \in \mathcal{C}$, we have

$$\begin{aligned}&(\forall u \in \mathcal{U}.\, \{d \in K(C_i)(u) \mid d \subseteq L'\} \subseteq K'(C_i)(u))\\ \wedge\ &(\forall u \in \mathcal{U}.\, \{d \in K'(C_i)(u) \mid d \subseteq L\} \subseteq K(C_i)(u))\end{aligned}$$

and $L = L'$. Thus, for all $u \in \mathcal{U}$,

$$K'(C_i)(u) = \{d \in K'(C_i)(u) \mid d \subseteq L'\} = \{d \in K(C_i)(u) \mid d \subseteq L\} = K(C_i)(u).$$

This means that $\forall C_i \in \mathcal{C}.\, K(C_i) = K'(C_i)$. Therefore, $K = K'$, as desired.

- Transitivity: $X \sqsubseteq Y$ and $Y \sqsubseteq Z$ implies $X \sqsubseteq Z$. That is,

$$\begin{aligned}&(A, L, \mathcal{C}, K) \sqsubseteq (A', L', \mathcal{C}', K') \wedge (A', L', \mathcal{C}', K') \sqsubseteq (A'', L'', \mathcal{C}'', K'')\\ \implies &(A, L, \mathcal{C}, K) \sqsubseteq (A'', L'', \mathcal{C}'', K'').\end{aligned}$$

Assume the LHS of the implication holds. We prove the inequality on the RHS as follows. First,

$$(A \subseteq A') \wedge (A' \subseteq A'') \implies A \subseteq A''.$$

Next,

$$\begin{aligned}& L'' \cap A' \subseteq L' && (Y \sqsubseteq Z)\\ \implies & (L'' \cap A') \cap A \subseteq L' \cap A && (\subseteq)\\ \iff & L'' \cap (A' \cap A) \subseteq L' \cap A && (\text{associativity})\\ \iff & L'' \cap A \subseteq L' \cap A && (A \subseteq A' \text{ of } X \sqsubseteq Y)\\ \iff & L'' \cap A \subseteq L' \cap A \subseteq L && (X \sqsubseteq Y)\\ \implies & L'' \cap A \subseteq L.\end{aligned}$$

Finally, for all $C_i \in \mathcal{C}$,

$$\exists C_j \in \mathcal{C}'.\, C_i \subseteq C_j \quad ((\mathcal{C}, K) \sqsubseteq (\mathcal{C}', K') \text{ of } X \sqsubseteq Y)$$

and for all $C_j \in \mathcal{C}'$,

$$\exists C_k \in \mathcal{C}''.\, C_j \subseteq C_k \quad ((\mathcal{C}', K') \sqsubseteq (\mathcal{C}'', K'') \text{ of } Y \sqsubseteq Z).$$

Therefore, for all $C_i \in \mathcal{C}$, there is $C_k \in \mathcal{C}''$ satisfying $C_i \subseteq C_k$. Now, assume there are $K(C_i), K'(C_j)$ and $K''(C_k)$ satisfying $C_i \subseteq C_j \subseteq C_k$. For all $u \in \mathcal{U}$,

$$\begin{aligned}
&\{d \in K(C_i)(u) \mid d \subseteq L''\} \\
=&\{d \in K(C_i)(u) \mid d \subseteq L'' \cap A\} && (d \in K(C_i)(u) \implies d \subseteq A) \\
\subseteq&\{d \in K(C_i)(u) \mid d \subseteq L'' \cap A'\} && (A \subseteq A' \text{ of } X \sqsubseteq Y) \\
\subseteq&\{d \in K(C_i)(u) \mid d \subseteq L' \wedge d \subseteq L''\} && (L'' \cap A' \subseteq L' \text{ of } Y \sqsubseteq Z) \\
\subseteq&\{d \in K'(C_j)(u) \mid d \subseteq L''\} && ((\mathcal{C}, K) \sqsubseteq (\mathcal{C}', K') \text{ of } X \sqsubseteq Y) \\
\subseteq&K''(C_k)(u) && ((\mathcal{C}', K') \sqsubseteq (\mathcal{C}'', K'') \text{ of } Y \sqsubseteq Z).
\end{aligned}$$

Thus,

$$\forall u \in \mathcal{U}.\ \{d \in K(C_i)(u) \mid d \subseteq L''\} \subseteq K''(C_k)(u).$$

This implies that

$$(\mathcal{C}, K) \sqsubseteq (\mathcal{C}', K') \wedge (\mathcal{C}', K') \sqsubseteq (\mathcal{C}'', K'') \implies (\mathcal{C}, K) \sqsubseteq (\mathcal{C}'', K'').$$

We have just shown that

$$(A \subseteq A'') \wedge (L'' \cap A \subseteq L) \wedge (\mathcal{C}, K) \sqsubseteq (\mathcal{C}'', K''),$$

and this is the definition of $(A, L, \mathcal{C}, K) \sqsubseteq (A'', L'', \mathcal{C}'', K'')$.

□

In order to meet the requirement that no subset of a cover includes another subset of the same cover, we define

$$max(\mathcal{C}) = \{C_i \in \mathcal{C} \mid \forall C_i' \in \mathcal{C}.\ C_i \subseteq C_i' \implies C_i = C_i'\}.$$

Every subset included in another subset in the input $\mathcal{C}$ is removed once the $max()$ operation is invoked. This makes sure that the result of the operation meets the requirement. We combine a set of provenance stores $\mathcal{S}_0$ by the following operator:

$$S' = join(L, C_i, \mathcal{S}_0) \text{ where } S'(u) = \left\{ d \in \bigsqcup_{S \in \mathcal{S}_0} S(u) \;\middle|\; d \subseteq L \wedge d \subseteq C_i \right\} \text{ for all } u \in \mathcal{U}.$$

By the definition of $\sqcup$ for provenances (see Lemma 4.1.6), the $join()$ operation calculates the least upper bound of provenances of each element in the set of provenance stores $\mathcal{S}_0$. Then, it selects dots that are included in the first two parameters, L and C_i.

Lemma 4.2.4. *$\mathcal{X}$ is a join semi-lattice. The least element is $(\{\}, \{\}, \{\}, [])$. For all OR_{PC} sets $X = (A, L, \mathcal{C}, K)$ and $X' = (A', L', \mathcal{C}', K')$ in $\mathcal{X}$, their least upper bound is given by*

$$X \sqcup X' = (A'', L'', \mathcal{C}'', K'')$$

where

$$
\begin{aligned}
A'' &= A \cup A',\\
L'' &= (L \cap L') \cup (L \setminus A') \cup (L' \setminus A),\\
\mathcal{C}'' &= \mathit{max}(\mathcal{C} \cup \mathcal{C}'),\\
K''(C_k) &= \mathit{join}(L'', C_k, \{K(C_i) \mid C_i \in \mathcal{C} \wedge C_i \subseteq C_k\} \cup \{K'(C_j) \mid C_j \in \mathcal{C}' \wedge C_j \subseteq C_k\})\\
&\qquad\qquad \textit{for all } C_k \in \mathcal{C}''.
\end{aligned}
$$

Proof. $(\mathcal{X}, \sqsubseteq)$ is a poset because $\sqsubseteq$ is a partial order (see Lemma 4.2.3). The element $(\{\}, \{\}, \{\}, [])$ in $\mathcal{X}$ is the least element because

$$
\begin{array}{rl}
 & (\{\} \subseteq A) \quad \text{(true for any } A\text{)}\\
 & \wedge\\
(L \cap \{\} \subseteq \{\}) \iff & (\{\} \subseteq \{\}) \quad \text{(always true)}\\
 & \wedge\\
 & ((\{\}, []) \sqsubseteq (\mathcal{C}, K)) \quad \text{(true for any } \mathcal{C}, K\text{)}.
\end{array}
$$

To prove that $X \sqcup X'$ is the least upper bound of X and X', consider

$$X = (A, L, \mathcal{C}, K) \quad \text{and} \quad X' = (A', L', \mathcal{C}', K').$$

Let $(A'', L'', \mathcal{C}'', K'') = X \sqcup X'$ be the $\mathrm{OR}_{\mathrm{PC}}$ set defined in this lemma. The following properties are satisfied by the definition of $\sqcup$:

1. A'' is a finite set of event-histories in $\mathcal{H}$ that is downward closed with respect to the $\sqsubseteq$ relation because $A'' = A \cup A'$ and both A and A' are downward closed.

2. $L'' \subseteq A''$ because

$$L'' = (L \cap L') \cup (L \setminus A') \cup (L' \setminus A)$$

   and

$$
\begin{array}{lll}
 & (L \cap L') \subseteq (A \cup A') & (L \subseteq A \wedge L' \subseteq A')\\
\wedge & (L \setminus A') \subseteq A & (L \subseteq A)\\
\wedge & (L' \setminus A) \subseteq A' & (L' \subseteq A').
\end{array}
$$

3. $\mathcal{C}''$ does not contain C_k and C'_k such that $C_k \neq C'_k$ and $C_k \subseteq C'_k$, because of the use of the $\mathit{max}()$ operation in its definition. We also have

$$
\begin{aligned}
\bigcup\{C_k \mid C_k \in \mathit{max}(\mathcal{C} \cup \mathcal{C}')\} &= \bigcup\{C_i \mid C_i \in \mathcal{C}\} \cup \bigcup\{C_j \mid C_j \in \mathcal{C}'\}\\
&= A \cup A'
\end{aligned}
$$

because the $max()$ operation removes a subset from its parameter only when the subset is included in another subset. This property ensures that the operation does not affect the result of $\bigcup$ because every element in the removed subset exists in the survived (not removed) subset.

4. For all $C_k \in \mathcal{C}''$, $u \in \mathcal{U}$, and $d \in K''(C_k)(u)$, we have $d \subseteq L'' \cap C_k$ because L'' and C_k are the first two parameters given to the $join()$ operation here.

Thus,

$$(X \sqcup X') \in \mathcal{X}.$$

The next part of the proof aims to show that $X \sqcup X'$ is an upper bound of X and X':

1. $A \subseteq A''$ and $A' \subseteq A''$ because $A'' = A \cup A'$.

2. Since $L'' = (L \cap L') \cup (L \setminus A') \cup (L' \setminus A)$, we have

$$\begin{aligned} L'' \cap A &\iff (L \cap L' \cap A) \cup ((L \setminus A') \cap A) \cup ((L' \setminus A) \cap A) \\ &\iff (L \cap L') \cup (L \setminus A') \\ &\subseteq L \\ &\text{and} \\ L'' \cap A' &\iff (L \cap L' \cap A') \cup ((L \setminus A') \cap A') \cup ((L' \setminus A) \cap A') \\ &\iff (L \cap L') \cup (L' \setminus A) \\ &\subseteq L'. \end{aligned}$$

3. For every $C_i \in \mathcal{C}$, there exists $C_k \in \mathcal{C}''$ with $C_i \subseteq C_k$, because $\mathcal{C}'' = max(\mathcal{C} \cup \mathcal{C}')$. Also, for all such $C_k \in \mathcal{C}''$ (i.e. $C_i \subseteq C_k$), we have

$$\forall u \in \mathcal{U}.\ \{d \in K(C_i)(u) \mid d \subseteq L''\} \subseteq K''(C_k)(u),$$

because of the definition of the $join()$ operation used to define K''. The same proof can be applied to show that $(\mathcal{C}', K') \sqsubseteq (\mathcal{C}'', K'')$.

Thus,

$$X \sqsubseteq X \sqcup X' \quad \text{and} \quad X' \sqsubseteq X \sqcup X'.$$

The final part is to prove that $X \sqcup X'$ is the least upper bound of X and X'. Let $X_u = (A_u, L_u, \mathcal{C}_u, K_u)$ be an upper bound of X and X':

$$X \sqsubseteq X_u \quad \text{and} \quad X' \sqsubseteq X_u.$$

1. $A'' \subseteq A_u$ because

$$A \subseteq A_u, \quad A' \subseteq A_u, \quad \text{and} \quad A'' = A \cup A'.$$

2. To see why $L_u \cap A'' \subseteq L''$ holds, note that

$$\begin{aligned} L_u \cap (A \cap A') &\subseteq L \cap L' \quad (L_u \cap A \subseteq L \wedge L_u \cap A' \subseteq L'), \\ L_u \cap (A \setminus A') &\subseteq L \setminus A' \quad (L_u \cap A \subseteq L), \\ L_u \cap (A' \setminus A) &\subseteq L' \setminus A \quad (L_u \cap A' \subseteq L'). \end{aligned}$$

   Thus,

$$\begin{aligned} L_u \cap A'' = L_u \cap (A \cup A') &= L_u \cap ((A \cap A') \cup (A \setminus A') \cup (A' \setminus A)) \\ &\subseteq (L \cap L') \cup (L \setminus A') \cup (L' \setminus A) = L''. \end{aligned}$$

3. For all $C_k \in \mathcal{C}''$, there exists $C_u \in \mathcal{C}_u$ such that $C_k \subseteq C_u$. This is because $\mathcal{C}'' = max(\mathcal{C} \cup \mathcal{C}')$,

$$(\forall C_i \in \mathcal{C}.\ \exists C_u \in \mathcal{C}_u.\ C_i \subseteq C_u) \quad \text{and} \quad (\forall C_j \in \mathcal{C}'.\ \exists C_u \in \mathcal{C}_u.\ C_j \subseteq C_u).$$

   Consider $C_k \in \mathcal{C}''$ and $C_u \in \mathcal{C}_u$ such that $C_k \subseteq C_u$. Let $u \in \mathcal{U}$ and $d \in K''(C_k)(u)$ such that $d \subseteq L_u$. We have to show that

$$d \in K_u(C_u)(u).$$

   By the definition of K'', there exists $C_i \in \mathcal{C}$ such that

$$C_i \subseteq C_k \quad \text{and} \quad d \in K(C_i)(u)$$

   or there exists $C_j \in \mathcal{C}'$ such that

$$C_j \subseteq C_k \quad \text{and} \quad d \in K(C_j)(u).$$

   In the former case, since X_u is larger than X and $C_i \subseteq C_k \subseteq C_u$, we have $d \in K_u(C_u)(u)$. In the latter case, since X_u is larger than X' and $C_j \subseteq C_k \subseteq C_u$, we have $d \in K_u(C_u)(u)$ as well. What we have proved so far implies $(\mathcal{C}'', K'') \sqsubseteq (\mathcal{C}_u, K_u)$.

□

In our programming model, every dataflow node internally implements an $\mathrm{OR}_{\mathrm{PC}}$ set, and end users interact with $\mathrm{OR}_{\mathrm{PC}}$ sets at the leaves of a dataflow graph, which do not have any incoming edges from other nodes in the graph. As a result, such user-facing nodes get updated only by `insert`() and `delete`() operations run by the end users, and their $\mathrm{OR}_{\mathrm{PC}}$ sets have relatively simple provenances and cover. We formalise $\mathrm{OR}_{\mathrm{PC}}$ sets with such provenances and cover in the following definition.

Definition 4.2.5. An $\mathrm{OR}_{\mathrm{PC}}$ set $X = (A, L, \mathcal{C}, K)$ is *simple* if

1. $\mathcal{C}$ is a singleton set so that
$$\mathcal{C} = \{A\},$$

2. for all $u \in \mathcal{U}$ and $d \in K(A)(u)$,
$$|d| \leq 1.$$

Simple $\mathrm{OR_{PC}}$ sets are ordered by the same partial order $\sqsubseteq$ as $\mathrm{OR_{PC}}$ sets. We write $\mathcal{X}_s$ for the poset of simple $\mathrm{OR_{PC}}$ sets.

The interesting point about simple $\mathrm{OR_{PC}}$ sets is that their $\mathcal{C}$ components contain only one subset and the subset is equal to A, the set of all known event-histories. Recall that in general, every subset C_i in the $\mathcal{C}$ component of an $\mathrm{OR_{PC}}$ set encodes the property that if a dataflow computation affecting the set involves only the event-histories in C_i, it has already been processed and its result has been incorporated into the set. Thus, the condition of the $\mathcal{C}$ component of a simple $\mathrm{OR_{PC}}$ set means that the set's content is always up to date: it is the result of the complete computations of all the dataflow operations to the set that use all the event-histories known to the set (i.e. event-histories belong to the A component of the $\mathrm{OR_{PC}}$ set) and update the contents of the set.

Lemma 4.2.6. *$\mathcal{X}_s$ is also a join semi-lattice. The least element is*

$$\bot = (\{\},\ \{\},\ \{\{\}\},\ [\{\} \mapsto (\lambda u \in \mathcal{U}.\ \{\})]).$$

For all simple OR_{PC} sets $X_s = (A, L, \mathcal{C}, K)$ and $X'_s = (A', L', \mathcal{C}', K')$, their least upper bound is given by

$$X_s \sqcup X'_s = (A'', L'', \mathcal{C}'', K'')$$

where

$$\begin{aligned}
A'' &= A \cup A', \\
L'' &= (L \cap L') \cup (L \setminus A') \cup (L' \setminus A), \\
\mathcal{C}'' &= \{A \cup A'\}, \\
K'' &= [A'' \mapsto (\lambda u \in \mathcal{U}.\ \{d \in K(A)(u) \sqcup K'(A')(u) \mid d \subseteq L''\})].
\end{aligned}$$

Proof. $(\mathcal{X}_s,\ \sqsubseteq)$ is a poset, because $\mathcal{X}_s$ uses the same order relation $\sqsubseteq$ as the poset $\mathcal{X}$. Also, by the definition of $\sqsubseteq$, $\bot$ is the least element of $\mathcal{X}_s$. In order to prove that $X_s \sqcup X'_s$ is the least upper bound of X_s and X'_s, let

$$X_s = (A, L, \mathcal{C}, K) \in \mathcal{X}_s, \quad X'_s = (A', L', \mathcal{C}', K') \in \mathcal{X}_s, \quad \text{and} \quad X_s \sqcup X'_s = (A'', L'', \mathcal{C}'', K'').$$

First, we show that $X_s \sqcup X'_s$ is a simple $\mathrm{OR_{PC}}$ set. There are two observations we can make:

1. $\mathcal{C}''$ is a singleton set, so that it does not contain two different subsets with one including the other. Also, we have

$$\bigcup\{C_k \mid C_k \in \{A \cup A'\}\} = A'' \quad (A'' = A \cup A').$$

2. For all $u \in \mathcal{U}$ and $d \in K''(A'')(u)$, we have $d \subseteq L''$ because the definition guarantees $d \subseteq L''$. Also, $d \subseteq L''$ implies $d \subseteq A''$ because $L'' \subseteq A''$.

From these observations, it follows that

$$(X_s \sqcup X'_s) \in \mathcal{X}.$$

We now show that $(X_s \sqcup X'_s) \in \mathcal{X}_s$:

1. $\mathcal{C}''$ is a singleton set by definition. Also, we have

$$\mathcal{C}'' = \{A \cup A'\} = \{A''\} \quad (A'' = A \cup A').$$

2. For all $u \in \mathcal{U}$ and for all $d'' \in K''(A'')(u)$, we have that $|d''| \leq 1$ because

$$\begin{array}{lll} & K''(A'')(u) = K(A)(u) \cup K'(A')(u) & \text{(the definition of } \sqcup) \\ \wedge & \forall d \in K(A)(u).\ |d| \leq 1 & (X_s \in \mathcal{X}_s) \\ \wedge & \forall d' \in K'(A')(u).\ |d'| \leq 1 & (X'_s \in \mathcal{X}_s). \end{array}$$

Second, we prove that $X_s \sqcup X'_s$ is an upper bound of X_s and X'_s. We have $(\mathcal{C}, K) \sqsubseteq (\mathcal{C}'', K'')$, because

$$(A \subseteq A'') \wedge (\forall u \in \mathcal{U}.\ \{d \in K(A)(u) \mid d \subseteq L''\} \subseteq K''(A'')(u))$$

by the definitions of A'' and K''. The same proof can be applied to show that $(\mathcal{C}', K') \sqsubseteq (\mathcal{C}'', K'')$. Also, $A \subseteq A''$, $A' \subseteq A''$,

$$\begin{aligned} L'' \cap A &= (L \cap L') \cap A \cup (L \setminus A') \cap A \cup (L' \setminus A) \cap A \\ &= (L \cap L') \cap A \cup (L \setminus A') \cap A \\ &\subseteq L \end{aligned}$$

and

$$\begin{aligned} L'' \cap A' &= (L \cap L') \cap A' \cup (L \setminus A') \cap A' \cup (L' \setminus A) \cap A' \\ &= (L \cap L') \cap A \cup (L' \setminus A) \cap A \\ &\subseteq L'. \end{aligned}$$

Thus,

$$X_s \sqsubseteq X_s \sqcup X'_s \quad \text{and} \quad X'_s \sqsubseteq X_s \sqcup X'_s.$$

Finally, we prove that $X_s \sqcup X'_s$ is the least upper bound of X_s and X'_s. Let $X_u = (A_u, L_u, \mathcal{C}_u, K_u)$ be a simple $\mathrm{OR_{PC}}$ set such that it upper-bounds X_s and X'_s:

$$X_s \sqsubseteq X_u \quad \text{and} \quad X'_s \sqsubseteq X_u.$$

We have $A'' \subseteq A_u$ because

$$A \subseteq A_u, \quad A' \subseteq A_u, \quad \text{and} \quad A'' = A \cup A'.$$

Also, since $L \cap A_u \subseteq L_u$ and $L' \cap A_u \subseteq L_u$,

$$\begin{aligned} L'' \cap A_u &= (L \cap L') \cap A_u \cup (L \setminus A') \cap A_u \cup (L' \setminus A) \cap A_u \\ &\subseteq L_u. \end{aligned}$$

It remains to show that $(\mathcal{C}'', K'') \sqsubseteq (\mathcal{C}_u, K_u)$. Since X'' and X_u are simple $\mathrm{OR_{PC}}$ sets, $\mathcal{C}'' = \{A''\}$ and $\mathcal{C}_u = \{A_u\}$. Also, as we showed already, $A'' \subseteq A_u$. Thus,

$$\forall C_k \in \mathcal{C}''.\ \exists C_u \in \mathcal{C}_u.\ C_k \subseteq C_u.$$

Also, for all $u \in \mathcal{U}$,

$$\{d \in K''(A'')(u) \mid d \subseteq L_u\} \subseteq K_u(A_u)(u)$$

because

$$\begin{aligned} &\{d \in K(A)(u) \mid d \subseteq L_u\} \subseteq K_u(A_u)(u) \\ &\wedge \\ &\{d \in K'(A')(u) \mid d \subseteq L_u\} \subseteq K_u(A_u)(u) \\ &\wedge \\ &K''(A'')(u) = \{d \in K'(A')(u) \mid d \subseteq L''\} \cup \{d \in K'(A')(u) \mid d \subseteq L''\} \\ &\wedge \\ &L'' \cap L_u = L_u. \end{aligned}$$

What we have shown implies $(\mathcal{C}'', K'') \sqsubseteq (\mathcal{C}_u, K_u)$. □

4.3 Operations of $\mathrm{OR_{PC}}$ Sets

$\mathrm{OR_{PC}}$ sets can be updated or accessed by the dataflow operations provided by Lasp (`map`(), `filter`(), ∪, and ×) and also by usual set operations, namely, `read`(), `insert`(), and `delete`(). The dataflow operations and the `read`() operation can be applied to any $\mathrm{OR_{PC}}$ sets, but the `insert`() and `delete`() operations can only be applied to simple $\mathrm{OR_{PC}}$ sets, which sit at the leaves of the dataflow graph and interact with the end users

directly. Also, our operations may be undefined in some cases - the runtime environment should ensure that such undefined cases never arise during the execution.

For a finite downward closed set of event-histories A, we write

$$next(A) = \{((v, r, n+1), (v, \texttt{undef}, \texttt{undef})) \mid ((v, r, n), (v, \texttt{undef}, \texttt{undef})) \in A \\ \wedge ((v, r, n+1), (v, \texttt{undef}, \texttt{undef})) \notin A\}.$$

Recall the semiring operators for provenances explained in Section 4.1:

$$p \otimes p' = \{d \cup d' \mid d \in p \wedge d' \in p'\}, \qquad p \oplus p' = p \cup p'.$$

We define operations for inserting and deleting elements from simple $\mathrm{OR_{PC}}$ sets as follows:

$$\texttt{insert} : \mathcal{E} \times \mathcal{U} \times \mathcal{X}_s \to \mathcal{X}_s$$

$$\texttt{insert}(e, u, (A, L, \{A\}, [A \mapsto S])) =$$

$$\begin{cases} (A', L', \mathcal{C}', [A' \mapsto S']) & \text{if } (e, (e.v, \texttt{undef}, \texttt{undef})) \in next(A) \\ (A, L, \{A\}, [A \mapsto S]) & \text{otherwise} \end{cases}$$

$$\left[\begin{array}{ll} \text{where} & h' = (e, (e.v, \texttt{undef}, \texttt{undef})), \\ & A' = A \cup \{h'\}, \\ & L' = L \cup \{h'\}, \\ & \mathcal{C}' = \{A \cup \{h'\}\}, \\ & S' = S[u \mapsto S(u) \oplus \{\{h'\}\}] \end{array}\right]$$

$$\texttt{delete} : \mathcal{U} \times \mathcal{X}_s \to \mathcal{X}_s$$

$$\texttt{delete}(u, (A, L, \{A\}, [A \mapsto S])) = (A, L', \{A\}, [A \mapsto S'])$$

$$\left[\begin{array}{ll} \text{where} & L' = L \setminus \bigcup\{d \mid d \in S(u)\}, \\ & S' = S[u \mapsto \{\}] \end{array}\right]$$

When a new element u is inserted in a simple $\mathrm{OR_{PC}}$ set $(A, L, \{A\}, [A \mapsto S])$ with the event e that is the identifier of the `insert`() operation, it first checks whether the new event-history is a member of $next(A)$. The $next()$ operation helps to maintain the component A as a downward closed set of event-histories. If the event-history for e passes this check, it is added to A, the set of all event-histories, as well as to L, the set of live event-histories. In this case, the event-history for e is also added to a single subset of the component $\mathcal{C}$ to ensure that the following nodes in the dataflow notice the completion of computations involving the new element u at the $\mathrm{OR_{PC}}$ set. Lastly, the new provenance of the new element is added to the provenance store using the semiring $\oplus$ operation.

The `delete`() operation removes every event-history in the provenance of the deleted element from the live event-history set L. The downstream computation of the dataflow program subsequently gets rid of dots containing the removed event-histories from the provenance stores of other $\mathrm{OR}_{\mathrm{PC}}$ sets in the dataflow. Changing the provenance of the deleted element to the empty set prevents the `read`() operation from returning the deleted element.

The next operation is `read`():

$$\begin{aligned}
&\texttt{read} : 2^{\mathcal{H}} \times \mathcal{X} \to 2^{\mathcal{H}} \times 2^{\mathcal{U}} \cup \{\texttt{undef}\} \\
&\texttt{read}(C_i, (A, L, \mathcal{C}, K)) = \ \text{if (there is } C_i' \in \mathcal{C} \text{ such that } C_i \subseteq C_i') \text{ then} \\
&\qquad\qquad\qquad\qquad \text{let } C_i' \text{ be such a subset in } \mathcal{C} \text{ in} \\
&\qquad\qquad\qquad\qquad (C_i', \{u \in \mathcal{U} \mid K(C_i')(u) \neq \{\}\}) \\
&\qquad\qquad\qquad \text{else } \texttt{undef}
\end{aligned}$$

The first parameter C_i of the `read`() operation is used to ensure the monotonicity of the result of the operation. Let $X = (A, L, \mathcal{C}, K)$ be an $\mathrm{OR}_{\mathrm{PC}}$ set, and (C_i, U) be the result of the $\texttt{read}(\{\}, X)$ operation. The parameter $\{\}$ here indicates that this `read`() operation is the first among consecutive reads. When we perform a further read on an $\mathrm{OR}_{\mathrm{PC}}$ set $X' = (A', L', \mathcal{C}', K')$, we want to ensure that this new read supersedes the old read. We can do this by calling the `read`() operation on X' with C_i that we obtained from the previous read operation: $\texttt{read}(C_i, X')$. The implementation of `read`() checks whether X' has information more up-to-date than C_i, and returns the contents of a set only if this check succeeds. Note that if $X \sqsubseteq X'$, the second call to `read`() always succeeds, because the definition of $\sqsubseteq$ for $\mathrm{OR}_{\mathrm{PC}}$ sets ensures that there exists $C_i' \in \mathcal{C}'$ satisfying $C_i \subseteq C_i'$. It is possible that more than one such C_i's exist in $\mathcal{C}'$ and this makes the `read`() operation non-deterministic. However, this is not a problem for our proofs because any non-deterministic selection still ensures the monotonicity of the result of the subsequent operations with inputs ordered by $\sqsubseteq$.

We also define the dataflow operations for $\mathrm{OR}_{\mathrm{PC}}$ sets as follows:

$$\begin{aligned}
&\texttt{map} : [\mathcal{U} \to \mathcal{U}] \times \mathcal{X} \to \mathcal{X} \\
&\texttt{map}(g, (A, L, \mathcal{C}, K)) = \Big(A,\ L,\ \mathcal{C},\ (\lambda C_i \in \mathcal{C}.\ K'(C_i))\Big) \\
&\qquad \big[\ \text{where} \quad K'(C_i) = \lambda u.\ \textstyle\bigoplus\{K(C_i)(v) \mid v \in g^{-1}(u)\}\ \big]
\end{aligned}$$

$$\begin{aligned}
&\texttt{filter} : [\mathcal{U} \to \{\texttt{true}, \texttt{false}\}] \times \mathcal{X} \to \mathcal{X} \\
&\texttt{filter}(g, (A, L, \mathcal{C}, K)) = \Big(A,\ L,\ \mathcal{C},\ (\lambda C_i \in \mathcal{C}.\ K'(C_i))\Big) \\
&\qquad \big[\ \text{where} \quad K'(C_i) = \lambda u.\ \text{if } g(u) = \texttt{true} \text{ then } K(C_i)(u) \text{ else } \{\}\ \big]
\end{aligned}$$

$$\cup : \mathcal{X} \times \mathcal{X} \to \mathcal{X}$$

$$(A, L, \mathcal{C}, K) \cup (A', L', \mathcal{C}', K') = (A'', L'', \mathcal{C}'', K'')$$

$$\left[\begin{array}{l} \text{where} \\ A'' = A \cup A', \\ L'' = (L \cap L') \cup (L \setminus A') \cup (L' \setminus A), \\ \mathcal{C}'' = max(\{C_i \cup C_j \mid C_i \in \mathcal{C} \wedge C_j \in \mathcal{C}'\}), \\ K''(C_k)(u) = \\ \quad \{d \in K(C_i)(u) \oplus K'(C_j)(u) \mid d \subseteq L'' \wedge C_i \in \mathcal{C} \wedge C_j \in \mathcal{C}' \wedge C_k = C_i \cup C_j\} \end{array}\right]$$

$$\times : \mathcal{X} \times \mathcal{X} \to \mathcal{X}$$

$$(A, L, \mathcal{C}, K) \times (A', L', \mathcal{C}', K') = (A'', L'', \mathcal{C}'', K'')$$

$$\left[\begin{array}{l} \text{where} \\ A'' = A \cup A', \\ L'' = (L \cap L') \cup (L \setminus A') \cup (L' \setminus A), \\ \mathcal{C}'' = max(\{C_i \cup C_j \mid C_i \in \mathcal{C} \wedge C_j \in \mathcal{C}'\}), \\ K''(C_k)(u) = \\ \quad \left\{\begin{array}{ll} \{d \in K(C_i)(u_1) \otimes K'(C_j)(u_2) \mid d \subseteq L'' \wedge C_i \in \mathcal{C} \wedge C_j \in \mathcal{C}' \wedge C_k = C_i \cup C_j\} & \\ \qquad \text{if } u = (u_1, u_2) \text{ for some } u_1, u_2 & \\ \{\} \qquad \text{otherwise} & \end{array}\right. \end{array}\right]$$

For each of these dataflow operations, we need to update dataflow paths that appear in its inputs, before constructing the output as described above. But this update is simple and automatic, and we decide to omit it for simplicity. The details of how dataflow paths get updated are described in Chapter 5.

The two unary operations, `map`() and `filter`(), do not change the contents of input OR_{PC} sets except the provenance stores. The provenances of the elements that cannot satisfy the condition of the first parameter of the `filter`() operation are removed from the provenance stores, but these removals are different from the ones executed by the `delete`() operation. Since there is no change in the live event-history set L, the filtered elements can be made to reappear by another operation such as $\cup$.

Finally, the important functionality of the two binary operations, $\cup$ and $\times$, is to combine computational information of the two operands. The ways to combine the two components A'' and L'' are the same as the ones for $\sqcup$. When computing the $\mathcal{C}''$ component, these operations pair every subset of $\mathcal{C}$ with every subset of $\mathcal{C}'$, and take the union of subsets in such a pair, which represents the merge of two pieces of information about two fully processed event-histories. The methods of combining the two components K and K' depend on the definitions of the two binary operations.

Chapter 5

Dataflow Programs Expressed by Directed Acyclic Graphs

We consider dataflow programs that are expressed in terms of finite directed acyclic graphs. The dataflows of these graphs use the $\mathrm{OR}_{\mathrm{PC}}$ set data types formulated in the previous chapter and process the streaming inputs obtained from the replicated distributed stores. We also investigate key theoretical properties of these programs and provide formal proofs for them.

5.1 Distribution Architecture of the System

Before we explore dataflow programs implemented on top of our run-time system, we describe how the system is structured. We assume that every replica in the system is aware of the dataflow structure of the program and stores states of every node before the execution. For example, before the execution of the example program in Figure 3.1, every replica knows the structure of dataflow of the program and the states of every node (*UserData*, *GroupData*, *Divider*, *GroupScores*, *RankDivider* and *GroupMedal*) are stored in each replica. This allows a single replica to compute the results even if all other replicas are failed. However, it can have inefficient performance due to a repeat of same dataflow computations at every replica. In order to overcome this inefficiency, we introduce an optimisation technique, the synchronise filter mode, in Section 8.3.2.

5.2 Dataflow Programs

We can formally define a directed graph as $G = (V, E)$, consisting of the set of nodes V and the set of edges E which are the ordered pairs of the elements of V. For each $v \in V$, we write $pred(v)$ for $\{w \mid (w, v) \in E\}$, and $succ(v)$ for $\{w \mid (v, w) \in E\}$.

Definition 5.2.1. A *path* q for G is a finite sequence of nodes such that

$$\forall n \in \{k \in \mathbb{N} \mid 0 < k < |q|\}.\ (q_n, q_{n+1}) \in E$$

where q_n indicates the n-th node of the path q.

Definition 5.2.2. A *dataflow graph* G is a tuple (V_I, V_D, E) of data with the following properties:

1. V_I and V_D are finite disjoint sets of nodes.
2. $(V_I \cup V_D, E)$ is a directed graph.
3. The graph is acyclic. That is, for all paths q for $(V_I \cup V_D, E)$,
$$|q| \geq 2 \implies q_1 \neq q_{|q|}.$$
4. $pred(v) = \emptyset$ for all $v \in V_I$.

The nodes in V_I receive input updates from the end users. The condition that such nodes have no predecessors means that only the end users, not other dataflow nodes, can issue such updates. The nodes in V_D keep the results of dataflow operations. We use the dot notation to denote the components of G. For instance, we write $G.V$ for the set of the nodes in G, and $G.E$ for that of the edges in G.

Let $(\mathcal{X}, \bot, \sqcup, \sqsubseteq)$ be the join semi-lattice of the $\mathrm{OR}_{\mathrm{PC}}$ sets.

Definition 5.2.3. A *dataflow program* P is a pair (G, F) of the following data:

1. G is a dataflow graph.
2. F is a map from nodes v of G to *monotone* functions $F(v)$ of the following type:
$$F(v) : [pred(v) \to \mathcal{X}] \to \mathcal{X}$$
where the domain is the join semi-lattice of maps from $pred(v)$ to $\mathcal{X}$ ordered pointwise.

5.3 Operational Semantics

Recall that we use the symbol $\mathcal{R}$ to represent the finite set of replica ids, and r to range over this set. We also remind the reader that an event is a tuple $(v, r, n) \in \mathcal{V} \times \mathcal{R} \times \mathbb{N}$, and the set of all events is denoted by $\mathcal{E}$.

Let $P = (G, F)$ be a dataflow program. When describing the behaviour of P, we consider events where their object set $\mathcal{V}$ is $G.V_I$, the set of input nodes.

Definition 5.3.1. A dataflow path t_h is *compatible for P* if projecting the first node component of all the triples of t_h in the reverse nested order gives a path over the dataflow graph G. More formally, the condition means that the following function F gives a path over G. (defined inductively):

$$\begin{aligned}
F(v, \texttt{undef}, \texttt{undef}) &= v \\
F(v, t'_h, \texttt{undef}) &= F(t'_h)v \\
F(v, \texttt{undef}, t'_h) &= F(t'_h)v \\
F(v, t'_h, \texttt{unk}) &= F(t'_h)v \\
F(v, \texttt{unk}, t''_h) &= F(t''_h)v
\end{aligned}$$

To refer to a P-compatible dataflow path, we use the phrase *"dataflow path for P"*.

Definition 5.3.2. A *replica state s for P* is a map from nodes of G to $\mathrm{OR}_{\mathrm{PC}}$ sets in $\mathcal{X}$, i.e.,

$$s \in [G.V \to \mathcal{X}].$$

We write $\mathit{RStates}(P)$ for the set of replica states.

Definition 5.3.3. An *input stream α for P* is a finite sequence

$$(v_1, \texttt{insert}, u_1), (v_2, \texttt{delete}, u_2), \ldots, (v_n, \texttt{insert}, u_n)$$

where v_i is a node in $G.V_I$ and u_i is a value inserted to or deleted from an $\mathrm{OR}_{\mathrm{PC}}$ set by an end user via the `insert`() or `delete`() operation. We write $\mathit{Streams}(P)$ for the set of all input streams for P.

Definition 5.3.4. A *replica configuration R for P* is a tuple $(\alpha, s, n) \in \mathit{Streams}(P) \times \mathit{RStates}(P) \times \mathbb{N}$. A *system configuration S for P* is a function from replica ids to replica configurations.

Our operational semantics is defined in terms of two kinds of transition relations, one for system configurations and the other for replica configurations. These relations are denoted by $\Longrightarrow$ and $\Longrightarrow_r$, respectively.

The semantics has three rules for transforming system configurations.

$$\frac{(\alpha, s, n) \Longrightarrow_r (\alpha', s', n')}{S[r \mapsto (\alpha, s, n)] \Longrightarrow S[r \mapsto (\alpha', s', n')]}\ \textsc{Step}$$

$$\frac{v \in G.V_I \qquad X_s, X'_s \in \mathcal{X}_s \qquad X''_s = X_s \sqcup_s X'_s}{S[r_1.s \mapsto s[v \mapsto X_s], r_2.s \mapsto s'[v \mapsto X'_s]] \Longrightarrow S[r_1.s \mapsto s[v \mapsto X''_s]]}\ \textsc{Merge}_{\mathit{input}}$$

$$\frac{v \in G.V_D \qquad X, X' \in \mathcal{X} \qquad X'' = X \sqcup X'}{S[r_1.s \mapsto s[v \mapsto X], r_2.s \mapsto s'[v \mapsto X']] \Longrightarrow S[r_1.s \mapsto s[v \mapsto X'']]}\ \textsc{Merge}_{\mathit{dflow}}$$

where $\sqcup_s$ is the join for simple OR_{PC} sets and $\sqcup$ that for general OR_{PC} sets. Recall that these two operations are different, although simple OR_{PC} sets are also OR_{PC} sets. The STEP rule expresses that a replica's state gets transformed by an operation issued by an end user. The two MERGE rules show that two replica states are merged by one of two join operations, $\sqcup_s$ for simple OR_{PC} sets and $\sqcup$ for general OR_{PC} sets. The choice depends on whether the merge happens at a user-facing node in V_I or a dataflow node in V_D.

There are two types of rules for transforming replica configurations: one for processing the input stream (INPUT rules) and the other for computing the dataflow (DFLOW rule). The INPUT rules are for simple OR_{PC} sets, and the DFLOW rule is for general OR_{PC} sets.

The rules for processing the input stream are given below:

$$\frac{\begin{array}{l} v \in G.V_I \qquad X = s(v) \qquad X.\mathcal{C} = \{X.A\} \\ e = (v, r, n) \qquad h = (e, (v, \texttt{undef}, \texttt{undef})) \\ X'.A = X.A \cup \{h\} \qquad X'.L = X.L \cup \{h\} \qquad X'.\mathcal{C} = \{X'.A\} \\ X'.K = [X'.A \mapsto X.K(X.A)[u \mapsto X.K(X.A)(u) \oplus \{\{h\}\}]] \\ s' = s[v \mapsto X'] \qquad n' = n + 1 \end{array}}{((v, \texttt{insert}, u)\alpha, s, n) \Longrightarrow_r (\alpha, s', n')} \text{INPUT}_{ins}$$

$$\frac{\begin{array}{l} v \in G.V_I \qquad X = s(v) \qquad X.\mathcal{C} = \{X.A\} \\ X'.A = X.A \qquad X'.L = X.L \setminus \bigcup\{d \mid d \in X.K(X.A)(u)\} \qquad X'.\mathcal{C} = X.\mathcal{C} \\ X'.K = [X'.A \mapsto X.K(X.A)[u \mapsto \{\}]] \qquad s' = s[v \mapsto X'] \end{array}}{((v, \texttt{delete}, u)\alpha, s, n) \Longrightarrow_r (\alpha, s', n)} \text{INPUT}_{del}$$

The INPUT rules consume a tuple from the input stream α and update the relevant node states.

The rule for computing the dataflow operations is:

$$\frac{v \in G.V_D \qquad X = s(v) \qquad (v, \text{F}(v), s|_{pred(v)}) \Longrightarrow_s X' \qquad s' = s\,[v \mapsto X \sqcup X']}{(\alpha, s, n) \Longrightarrow_r (\alpha, s', n)} \text{DFLOW}$$

where $s|_{pred(v)}$ is the restriction of the function s to the vertex set $pred(v)$. The DFLOW rule contains the relation $\Longrightarrow_s$ for node states, whose transition depends on the function $F(v)$ of each node v. After computing the transition relation, the rule merges the existing state X and the result of the transition X' using the $\sqcup$ operation.

Recall that we define an operation to meet the requirement that no subset of a cover includes another subset of the same cover:

$$max(\mathcal{C}) = \{C_i \in \mathcal{C} \mid \forall C_i' \in \mathcal{C}.\ C_i \subseteq C_i' \implies C_i = C_i'\}.$$

We transform node states using the following rules for the $\Longrightarrow_s$ relation:

$$\frac{\begin{array}{l}(X, (v, \mathit{self}, \texttt{undef})) \Longrightarrow_{t_h} X' \qquad \{w\} = \mathit{pred}(v) \\ X''.A = X'.A \qquad X''.L = X'.L \qquad X''.\mathcal{C} = X'.\mathcal{C} \\ X''.K(C_i)(u) = \bigoplus\{X'.K(C_i)(u') \mid u' \in g^{-1}(u)\}\end{array}}{(v, \texttt{map}(g), [w \mapsto X]) \Longrightarrow_s X''} \ \mathrm{F}(v)_{\mathit{map}}$$

$$\frac{\begin{array}{l}(X, (v, \mathit{self}, \texttt{undef})) \Longrightarrow_{t_h} X' \qquad \{w\} = \mathit{pred}(v) \\ X''.A = X'.A \qquad X''.L = X'.L \qquad X''.\mathcal{C} = X'.\mathcal{C} \\ X''.K(C_i)(u) = (\text{if } (g(u) = \texttt{true}) \text{ then } X'.K(C_i)(u) \text{ else } \{\})\end{array}}{(v, \texttt{filter}(g), [w \mapsto X]) \Longrightarrow_s X''} \ \mathrm{F}(v)_{\mathit{filter}}$$

$$\frac{\begin{array}{l}(X, (v, \mathit{self}, \texttt{undef})) \Longrightarrow_{t_h} X' \qquad (Y, (v, \texttt{undef}, \mathit{self})) \Longrightarrow_{t_h} Y' \qquad \{w, w'\} = \mathit{pred}(v) \\ Z.A = X'.A \cup Y'.A \qquad Z.L = (X'.L \cap Y'.L) \cup (X'.L \setminus Y'.A) \cup (Y'.L \setminus X'.A) \\ Z.\mathcal{C} = \mathit{max}(\{C_i \cup C_j \mid C_i \in X'.\mathcal{C} \wedge C_j \in Y'.\mathcal{C}\}) \\ Z.K(C_k)(u) = \Big\{d \in \Big(X'.K(C_i)(u) \oplus Y'.K(C_j)(u)\Big) \;\Big|\; \\ \qquad\qquad d \subseteq Z.L \wedge (C_i, C_j) \in (X'.\mathcal{C} \times Y'.\mathcal{C}) \wedge C_k = C_i \cup C_j\Big\}\end{array}}{(v, \cup, [w \mapsto X, w' \mapsto Y]) \Longrightarrow_s Z} \ \mathrm{F}(v)_{\cup}$$

$$\frac{\begin{array}{l}(X, (v, \mathit{self}, \texttt{unk})) \Longrightarrow_{t_h} X' \qquad (Y, (v, \texttt{unk}, \mathit{self})) \Longrightarrow_{t_h} Y' \qquad \{w, w'\} = \mathit{pred}(v) \\ Z.A = X'.A \cup Y'.A \qquad X.L = (X'.L \cap Y'.L) \cup (X'.L \setminus Y'.A) \cup (Y'.L \setminus X'.A) \\ Z.\mathcal{C} = \mathit{max}(\{C_i \cup C_j \mid C_i \in X'.\mathcal{C} \wedge C_j \in Y'.\mathcal{C}\}) \\ Z.K(C_k)(u) = \Big\{d \in X'.K(C_i)(u_1) \otimes Y'.K(C_j)(u_2) \;\Big|\; \\ \qquad\qquad d \subseteq Z.L \wedge (C_i, C_j) \in (X'.\mathcal{C} \times Y'.\mathcal{C}) \wedge C_k = C_i \cup C_j \wedge u = (u_1, u_2)\Big\}\end{array}}{(v, \times, [w \mapsto X, w' \mapsto Y]) \Longrightarrow_s Z} \ \mathrm{F}(v)_{\times}$$

The $\mathrm{F}(v)$-related rules consist of two parts: the first for updating the dataflow paths, and the next for executing the dataflow operations. The former is defined in terms of the relation $\Longrightarrow_{t_h}$ defined by the following rules:

$$\frac{\begin{array}{l}t_0 \in \{\texttt{undef}, \texttt{unk}\} \\ Y.A = \{(e, (v, t_h, t_0)) \mid (e, t_h) \in X.A\} \\ Y.L = \{(e, (v, t_h, t_0)) \mid (e, t_h) \in X.L\} \\ Y.\mathcal{C} = \Big\{\big\{(e, (v, t_h, t_0)) \,\big|\, (e, t_h) \in C_i\big\} \;\Big|\; C_i \in X.\mathcal{C}\Big\} \\ Y.K(C_i)(u) = \Big\{\big\{(e, (v, t_h, t_0)) \,\big|\, (e, t_h) \in d\big\} \;\Big|\; d \in X_v.K(C_i)(u)\Big\}\end{array}}{(X, (v, \mathit{self}, t_0)) \Longrightarrow_{t_h} Y} \ \textsc{Append}$$

$$\frac{\begin{array}{l} t_0 \in \{\texttt{undef}, \texttt{unk}\} \\ Y.A = \{(e, (v, t_0, t_h)) \mid (e, t_h) \in X.A\} \\ Y.L = \{(e, (v, t_0, t_h)) \mid (e, t_h) \in X.L\} \\ Y.\mathcal{C} = \left\{ \{(e, (v, t_0, t_h)) \mid (e, t_h) \in C_i\} \;\middle|\; C_i \in X.\mathcal{C} \right\} \\ Y.K(C_i)(u) = \left\{ \{(e, (v, t_0, t_h)) \mid (e, t_h) \in d\} \;\middle|\; d \in X_v.K(C_i)(u) \right\} \end{array}}{(X, (v, t_0, \mathit{self})) \Longrightarrow_{t_h} Y} \textsc{Append}$$

The Append rule updates the entire dataflow paths in a node state using given parameters. The parameters indicate whether the input node is a left child or a right child of the target dataflow node, as well as the type of the sibling of the input node (either `undef` or `unk`).

Definition 5.3.5. An *execution* τ is a finite sequence of system configurations for $n \in \mathbb{N}$

$$\tau = [S_1, S_2, \ldots, S_n]$$

such that $S_i \Longrightarrow S_{i+1}$ for all $1 \leq i < n$.

We denote the i-th system configuration of an execution τ by $\tau[i]$ and the length of τ by $|\tau|$.

5.4 Consistent Execution and Correspondence Property

In this section, we consider a particular type of executions where all the communications between replicas and the processing of the input streams happen before any calculations made by the dataflow operations. We say that such executions are consistent. Then, we prove that the final result of a fully-computed execution can also be obtained by a consistent execution.

We write $S \Longrightarrow_{\mathit{input}} S'$ to mean that $S \Longrightarrow S'$ can be derived using the $\textsc{Input}_*$ rules for input processing or the $\textsc{Merge}_{\mathit{input}}$ rule for the synchronization of input nodes (i.e. nodes in $G.V_I$) between replicas. Also, we write $S \Longrightarrow_{\mathit{dflow}} S'$ to denote that $S \Longrightarrow S'$ can be derived using the DFlow rule for computing dataflow operations.

Definition 5.4.1. An execution τ is *consistent* if there exists $1 \leq m \leq |\tau|$ such that

1. $\tau[i] \Longrightarrow_{\mathit{input}} \tau[i+1]$ for all $1 \leq i < m$;
2. $\tau[j] \Longrightarrow_{\mathit{dflow}} \tau[j+1]$ for all $m \leq j < |\tau|$;
3. $\tau[m](r).\alpha$ is empty for every replica r; and

4. for all S, $\tau[m] \Longrightarrow_{input} S$ implies that $\tau[m] = S$.

Note that in a consistent execution, replicas first incorporate and communicate all input streams fully, and then perform dataflow operations on the state of each replica while forbidding replicas from communicating one another.

Definition 5.4.2. A system configuration S is *at the initial stage* if for all replicas r and nodes v, (i) $S(r).n = 0$ and (ii) $S(r).s(v) = \bot$. It is *completed* if (i) $S(r).\alpha$ is empty for all replicas r and (ii) $S \Longrightarrow S'$ implies $S = S'$ for all S'.

The following theorem shows that our system ensures replica convergence at the quiescent state, which is one of key properties of eventual consistency. Furthermore, a completed and consistent execution is used to describe the converging state in our system configuration.

Theorem 5.4.3 (Correspondence Property)**.** *Let S_I be a system configuration that is at the initial stage. Every S_I-starting completed execution τ has a corresponding S_I-starting completed and consistent execution τ' such that the last system configurations of τ and τ' are identical.*

Proof. Let S_F be the last system configuration of τ. Since τ is completed, S_F satisfies the following properties:

1. $S_F(r).\alpha$ is empty for all replicas r.
2. $S_F(r).s(v) = S_M(r').s(v)$ for all replicas r, r' and nodes v.
3. $S_F(r).s$ is the least fixpoint of the following function K:

$$K : \mathit{RStates}(P) \to_{mono} \mathit{RStates}(P)$$

$$K(s)(v) = \begin{cases} S_F(r).s(v) & \text{if } v \in G.V_I \\ X & \text{if } v \in G.V_D \text{ and } \exists X.\, (v, F(v), s|_{pred(v)}) \Longrightarrow_s X \\ \bot & \text{otherwise} \end{cases}$$

In order to construct a desired execution τ' in the theorem, we pick all the steps in τ that use the $\textsc{Input}_*$ rules or the $\textsc{Merge}_{input}$ rule. We assume that the step number starts from 1. For each step using the $\textsc{Input}$ rule, we keep a tuple (v, f, u, r, i) where f is either the `insert`() or `delete`() operations, (v, f, u) is an input processed, r is the replica of this input, and i is the step number. For each step using the $\textsc{Merge}_{input}$ rule, we keep a tuple (r, r', v, i) of communicating replicas r and r', the node v of the merged object, and the number i of the step.

We enumerate all these tuples:

$$t_1 < t_2 < \ldots < t_m$$

where t_k is (v, f, u, r, i) or (r, r', v, i), and $t_k < t_l$ holds if and only if $t_k.i < t_l.i$. Using this total order, we can construct an execution τ_1 of length $(m + 1)$ that starts from S_I and transforms the system configuration according to t_j at each j-th step. (By our assumption the step number j starts from 1). Let S_M be the last system configuration of this execution τ_1. It satisfies the following properties:

1. $S_M(r).\alpha$ is empty for all replicas r.
2. For all replicas r and nodes v, if $v \in G.V_I$, then $S_M(r).s(v) = S_F(r).s(v)$; otherwise, $S_M(r).s(v) = \bot$.
3. $S_M(r).n = S_F(r).n$ for all replicas r.

Note that because of the property of S_F mentioned above, the second item here implies that $S_M(r).s(v) = S_M(r').s(v)$ for all replicas r, r' and nodes v.

We now extend the execution τ_1 with transitions using the DFlow rule until the program of the rule no longer changes the system configuration. This extension is possible, since $S_F(r).s$ is the least fixpoint of the function K from above and S_F is obtained from S_I by applying the transition rules finitely many times. Let τ_2 be this extension of τ_1, and S'_F be the last system configuration of τ_2. Then, for all replicas r, $S'_F(r).s$ is the least fixpoint of K. Thus,

$$S'_F(r).s = S_F(r).$$

Furthermore, the DFlow rule does not change the α and n components of replica configurations, so that

$$S'_F(r).\alpha = S_M(r).\alpha = S_F(r).\alpha = [] \quad \text{and} \quad S'_F(r).n = S_M(r).n = S_F(r).n$$

for all replicas r. Therefore, $S'_F = S_F$ and τ_2 is completed. Also, by construction, τ_2 is consistent and S_I-starting. Thus, τ_2 is the required execution τ'. □

Weakly consistent distributed systems are designed to ensure eventual consistency, which in our terminology means that if input streams stop at all replicas, eventually all these replicas will reach the same state. Our correspondence property goes one step further than this convergence guarantee, and characterises what that converging state looks like.

5.5 Consistent Read

Theorem 5.4.3 helps us to understand a system configuration to which a dataflow program converges and all replicas have the same states. However, it does not provide any information for what system configurations look like before this convergence happens. In fact, most distributed dataflow programs running under weak consistency receive inputs from the users constantly, and they never converge. Designing a weakly consistent distributed system with formal guarantees on unconverged programs is one of the most important challenges in research on such systems.

All the complexities of our system are for tackling this challenge. In this section, we introduce the $\texttt{read}_c()$ operation for querying the state of only one replica, but in a way that it uses all the components (or meta-data) of the OR_{PC} set of the replica and returns a result corresponding to some consistent execution. The formal statement of this correspondence is the main result of this section.

5.5.1 Definition

The $\texttt{read}_c()$ operation selects a set of values from an OR_{PC} set such that each value has a dot in its provenance identical to the one from the corresponding consistent execution. The details of this correspondence and the meaning of the existence of the corresponding execution will be explained later in this subsection. The important part of the $\texttt{read}_c()$ operation is to compute dots that exist in the corresponding consistent execution.

We formally define the $\texttt{read}_c()$ operation for OR_{PC} sets as follows: when v is the node where the operation is performed,

$$
\begin{aligned}
&\texttt{read}_c : 2^{\mathcal{H}} \times 2^{\mathcal{D}} \times \mathcal{X} \to 2^{\mathcal{H}} \times 2^{\mathcal{D}} \times 2^{\mathcal{U}} \cup \{\texttt{undef}\} \\
&\texttt{read}_c(C_i, D, (A, L, \mathcal{C}, K)) = \\
&\qquad \text{if (there is } C_i' \in \mathcal{C} \text{ with } C_i \subseteq C_i') \text{ then} \\
&\qquad\qquad \text{let } C_i' \text{ be such an element in } \mathcal{C} \\
&\qquad\qquad (C_i', D', \{u \mid K(C_i')(u) \cap D' \neq \emptyset\}) \quad \text{where } D' = \mathit{fn}_{cds}(v, C_i', D) \\
&\qquad \text{else } \texttt{undef}.
\end{aligned}
$$

The $\texttt{read}_c()$ operation returns a tuple with three components, and the third component is a set of values in the given OR_{PC} set containing identical dots from the corresponding consistent execution. The parameters C_i, D of the $\texttt{read}_c()$ operation and the components C_i', D' of the outcome of the $\texttt{read}_c()$ operation are for ensuring the monotonicity of the results of the operation. They are similar to the corresponding parameters and outputs of the $\texttt{read}()$ operation. Let $X = (A, L, \mathcal{C}, K)$ be an OR_{PC} set, and (C_i, D, U) be the result of the $\texttt{read}_c(\{\}, \{\}, X)$ operation. When we perform a further consistent read on an OR_{PC} set $X' = (A', L', \mathcal{C}', K')$, we want to ensure that this new consistent

read supersedes the old read. We can do this by calling the $\texttt{read}_c()$ operation on X' with C_i, D that we obtained from the first consistent read operation: $\texttt{read}_c(C_i, D, X')$. Note that the success of the second call to $\texttt{read}_c()$ can be guaranteed when $X \sqsubseteq X'$, because the existence of some $C_j \in \mathcal{C}'$ satisfying $C_i \subseteq C_j$ can be ensured by the definition of $\sqsubseteq$ for OR_{PC} sets.

The $\texttt{read}_c()$ operation uses two utility operations: one for computing the collection of valid complete-dot sets which are identical to the completely-computed dots through a dataflow and the other for selecting the orderly-advanced complete-dot set from the collection of valid complete-dot sets

$$\mathit{valid_cds_collection}(v, t, H) = \mathcal{D},$$

$$\begin{aligned} &\mathit{fn}_{\mathit{cds}}(v, H, D) = D' \\ &\quad \text{where } D \subseteq D' \wedge D' \in \mathcal{D} \wedge \mathcal{D} = \mathit{valid_cds_collection}(v, t, H) \\ &\quad\quad \wedge (D' \text{ is non-deterministically selected from } \mathcal{D}). \end{aligned}$$

From a set of event histories H in an OR_{PC} set, we can calculate a collection of valid complete-dot sets $\mathcal{D}$ by using the $\mathit{valid_cds_collection}(v, t, H)$ operation. The details of the complete-dot set and the meaning of *valid* will also be explained later in this section. For now, we can assume that a valid complete-dot set contains information related to the dots in the corresponding consistent execution. The last parameter D of the $\mathit{fn}_{\mathit{cds}}(v, H, D)$ operation, the outcome of the previous $\mathit{fn}_{\mathit{cds}}()$ operation, is used for guaranteeing the monotonicity of the outcomes of the operation. The monotonicity of the outcome implies that $D \subseteq D'$. The result of the non-deterministic selection D' in the $\mathit{fn}_{\mathit{cds}}(v, H, D)$ operation always guarantees all properties of the valid complete-dot set because all members in $\mathcal{D}$ are valid complete-dot sets. For practical reasons, the implementation selects D' that can retrieve the maximum number of values for the output of the $\texttt{read}_c()$ operation.

5.5.2 Guarantee

Fix a dataflow program $P = (G, F)$, a node $v \in G.V$, and a replica id r. Also, pick a system configuration at the initial stage S_I, and an S_I-starting execution τ.

The following theorem provides a formal guarantee on what we get by performing the $\texttt{read}_c()$ operation, which is one of key theoretical properties guaranteed by our system. The results of the $\texttt{read}_c()$ operation are the members of the results of the same operation on the corresponding consistent execution and this indicates that the results are from the complete computation without any glitches. The theorem ensures that The results of the $\texttt{read}_c()$ operation satisfy the stronger consistency than eventual consistency on weakly consistent distributed systems.

Theorem 5.5.1 (Consistent Read)**.** *Let S be the last system configuration of τ, and $(_, _, U)$ be the result of performing* $\texttt{read}_c()$ *on $S(r).s(v)$. Then, there exists a consistent execution τ' such that*

1. *τ' starts at S_I; and*
2. *when S' is the last system configuration of τ' and $(_, _, U')$ is the result of executing* $\texttt{read}_c()$ *on $S'(r).s(v)$, we have that $U \subseteq U'$.*

Intuitively, Theorem 5.5.1 expresses that the result of the $\texttt{read}_c()$ operation can be reproduced by the same operation but performed in a non-replicated setting, i.e., a single node. It considers an execution and the $\texttt{read}_c()$ operation performed on the last system configuration of the execution. Then, the theorem says that the set of values U returned by the operation can be reproduced by some closely-related consistent execution τ' in the following sense: performing the same $\texttt{read}_c()$ operation on the last system configuration of τ' gives a superset of U. Since consistent executions are essentially the non-distributed executions on a single node, the theorem says the reproducibility by a non-distributed execution that we point out.

Note that $U = U'$ can only be satisfied when the system configuration of τ reaches at the quiescent state, all replica receive every input and finish every dataflow computation. For example, one possible (and naive) representation of $S(r).s(v)$ is as follows:

$$\begin{aligned}
&(\{(i_1, _), (i_2, _), (i_3, _), (i_4, _), (i_5, _), (j_1, l), (j_1, r), (j_2, l), (j_2, r), (j_3, l), (d_1, _)\},\\
&\ \{(i_1, _), (i_2, _), (i_3, _), (i_4, _), (j_1, l), (j_1, r), (j_2, l), (j_2, r), (j_3, l), (d_1, _)\},\\
&\ \{\{(i_1, _), (i_2, _), (i_3, _), (i_4, _), (i_5, _), (j_1, l), (j_1, r), (j_2, l), (j_2, r), (j_3, l), (d_1, _)\}\},\\
&\ [\{(i_1, _), (i_2, _), (i_3, _), (i_4, _), (i_5, _), (j_1, l), (j_1, r), (j_2, l), (j_2, r), (j_3, l), (d_1, _)\} \mapsto\\
&\quad \{((g_2, \mathit{bronze}), (i_2, _) \wedge (i_3, _) \wedge (j_1, l) \wedge (j_1, r) \wedge (d_1, _)),\\
&\quad\ ((g_1, \mathit{silver}), (i_1, _) \wedge (i_2, _) \wedge (i_3, _) \wedge (j_2, l) \wedge (j_2, r) \wedge (j_3, l) \wedge (d_1, _))\}]).
\end{aligned} \tag{5.1}$$

U, the result of the $\texttt{read}_c()$ operation from the representation (5.1), is $\{(g_2, \mathit{bronze})\}$ only because the value (g_1, silver) is not fully-computed. The following shows the naive representation of $S'(r).s(v)$:

$$\begin{aligned}
&(\{(i_1, _), (i_2, _), (i_3, _), (i_4, _), (i_5, _), (j_1, l), (j_1, r), (j_2, l), (j_2, r), (j_3, l), (j_3, r), (d_1, _)\},\\
&\ \{(i_1, _), (i_2, _), (i_3, _), (i_4, _), (j_1, l), (j_1, r), (j_2, l), (j_2, r), (j_3, l), (j_3, r), (d_1, _)\},\\
&\ \{\{(i_1, _), (i_2, _), (i_3, _), (i_4, _), (i_5, _), (j_1, l), (j_1, r), (j_2, l), (j_2, r), (j_3, l), (j_3, r), (d_1, _)\}\},\\
&\ [\{(i_1, _), (i_2, _), (i_3, _), (i_4, _), (i_5, _), (j_1, l), (j_1, r), (j_2, l), (j_2, r), (j_3, l), (j_3, r), (d_1, _)\} \mapsto\\
&\quad \{((g_2, \mathit{bronze}), (i_2, _) \wedge (i_3, _) \wedge (j_1, l) \wedge (j_1, r) \wedge (d_1, _)),\\
&\quad\ ((g_1, \mathit{silver}), (i_1, _) \wedge (i_2, _) \wedge (i_3, _) \wedge (j_2, l) \wedge (j_2, r) \wedge (j_3, l) \wedge (j_3, r) \wedge (d_1, _))\}]).
\end{aligned} \tag{5.2}$$

U', the result of the $\texttt{read}_c()$ operation from the representation (5.2), is $\{(g_2, \mathit{bronze}), (g_1, \mathit{silver})\}$. This satisfies $U \subseteq U'$. In order to satisfy $U = U'$, the system configuration of τ should receive the computation result for (j_3, r), and it will receive that result when it is at the quiescent state.

5.6 Proof of the Consistent Read Theorem

This section provides the formal proof of the Consistent Read theorem. The theorem claims that the results of the $\texttt{read}_c()$ operation also exist in the results of the $\texttt{read}_c()$ operation on the corresponding consistent execution. In order to prove the theorem, we introduce two important concepts: the shadow state and the complete-dot set. An execution with the shadow states stores every information about the inserted value because the `delete`() and `filter`() operations for the shadow state do not remove the information about the deleted or filtered values. A complete-dot set is a set of dots that are identical to the ones stored with the fully-computed values. Here, a fully-computed value means that the value is the result of the completion of every dataflow computation without any glitches.

The proof starts to prove that for every execution, there is the corresponding execution with the shadow states (described in Lemma 5.6.2 in Section 5.6.1). Then, Lemma 5.6.11 shows that the complete-dot sets from the results of the $\texttt{read}_c()$ operation also exist in the results of the $\texttt{read}_c()$ operation on the corresponding execution with the shadow states. The existence of the values with such complete-dot sets is proved in Lemma 5.6.9. Lastly, Lemma 5.6.13 proves that there is the corresponding consistent execution that has those values with complete-dots.

5.6.1 Shadow State

In order to provide the formal proof for Theorem 5.5.1, we first consider a slightly different execution of P using what we call shadow replica states.

Definition 5.6.1. A *shadow replica state* or *shadow state* ss is a map from $G.V$ to $\mathcal{X}$.

Note that a shadow replica state is nothing but a replica state that we defined previously. We introduce this new terminology because some operations for OR_{PC} sets such as `delete`() and `filter`() might lose the information about some inserted values. In order to store the information about those values, we employ a different operational semantics for shadow replica states. Specifically, the operational semantics for the shadow states consists of all the tradition relations in Section 5.3 except that the $\textsc{Input}_{del}$ and $\text{F}(v)_{filter}$ rules are changed as follows:

$$\frac{v \in G.V_I}{((v, \texttt{delete}, u)\alpha, ss, n) \Longrightarrow_r (\alpha, ss, n)}\ \textsc{Input}_{del} \text{ for } ss$$

$$\frac{(X, (v, \mathit{self}, \texttt{undef})) \Longrightarrow_{t_h} X'}{(v, \texttt{filter}(g), [w \mapsto X]) \Longrightarrow_s X'}\ \text{F}(v)_{filter} \text{ for } ss$$

The definitions of the $\textsc{Input}_{del}$ and $\text{F}(v)_{filter}$ rules for shadow states imply that the computations with the shadow states ignore any `delete`() and `filter`() operations. Recall

that the corresponding operations for usual replica states remove the information about inserted values from the replica states. As a result of these changes, the shadow states keep all information about the `insert`() operations propagated through the dataflow graph at dataflow nodes.

The computation of a dataflow program $P = (F, G)$ in Section 5.3 and that of the program with shadow replica states (and corresponding shadow replica/system configurations) are closely related. The key behind this relationship is the following correspondence between replica states s and shadow replica states ss: for every node v in $G.V$, if $s(v) = (A, L, \mathcal{C}, K)$ and $ss(v) = (A_s, L_s, \mathcal{C}_s, K_s)$, then

$$A = A_s, \quad A = L_s, \quad \mathcal{C} = \mathcal{C}_s, \quad \text{and} \quad K(C_i)(u) \subseteq K_s(C_i)(u) \text{ for all } C_i \in \mathcal{C} \text{ and } u \in \mathcal{U}.$$

We extend this relation to replica and system configurations by insisting that two related replica or system configurations are identical except that every shadow replica state in one configuration are related to the corresponding replica state in the other configuration. We write $ss \sim s$, $R_s \sim R$, and $S_s \sim S$ to denote these relations, where R_s and S_s are replica and system configurations involving shadow replica states and R and S are the ones that use normal replica states, not shadow ones.

Lemma 5.6.2. *Let S_I be defined the following system configuration:*

$$S_I(r) = (\alpha_r, s_I, 0), \qquad s_I(v) = (\{\}, \{\}, \{\}, []).$$

Then, $S_I \sim S_I$. Also, for all shadow replica configurations R_s and normal replica configurations R, if $R_s \sim R$, then

$$\forall R'_s.\ (R_s \Longrightarrow_r R'_s) \ \text{ implies } \ \exists R'.\ (R \Longrightarrow_r R') \wedge R'_s \sim R'$$

and

$$\forall R'.\ (R \Longrightarrow_r R') \ \text{ implies } \ \exists R'_s.\ (R_s \Longrightarrow_r R'_s) \wedge R'_s \sim R'.$$

Similarly, for all shadow system configurations S_s and normal system configurations S, if $S_s \sim S$, then

$$\forall S'_s.\ (S_s \Longrightarrow S'_s) \ \text{ implies } \ \exists S'.\ (S \Longrightarrow S') \wedge S'_s \sim S'$$

and

$$\forall S'.\ (S \Longrightarrow S') \ \text{ implies } \ \exists S'_s.\ (S_s \Longrightarrow S'_s) \wedge S'_s \sim S'.$$

By all of these results, shadow and normal system configurations remain related by $\sim$ when they are run by corresponding transitions from the same initial configuration S_I.

Proof. Note that S_I is viewed as a shadow system configuration as well as a normal system configuration. The claimed relationship $S_I \sim S_I$ is immediate from the definition of $\sim$.

Consider shadow replica and system configurations R_s and S_s, and normal replica and system configurations R and S. Assume that

$$R_s \sim R \quad \text{and} \quad S_s \sim S.$$

We will prove the claimed preservation of the $\sim$ relation by case analysis. The proof of each case consists of i) identifying what is changed by the rule for the transformation, ii) constructing the corresponding configuration by applying the same rule, and iii) showing that the output of the rule and the constructed corresponding configuration satisfy the required $\sim$ relation. For example, if $R = ((v', \texttt{insert}, u)\alpha, s, n)$ is transformed by the $\textsc{Input}_{ins}$ rule, and $R' = (\alpha', s', n')$ is the output replica configuration of the transformation, then i) we first note that s' is of the form $s[v' \mapsto X']$ for some X', where X' is built by the processing of the input tuple $(v', \texttt{insert}, u)$ with $s(v')$, ii) we next transform $R_s = ((v', \texttt{insert}, u)\alpha_s, ss, n_s)$ by the same rule and get $R'_s = (\alpha'_s, ss[v' \mapsto X'_s], n'_s)$ for some X'_s, where X'_s is built by the processing of the input tuple $(v', \texttt{insert}, u)$ with $ss(v')$, and iii) finally we show that R'_s with X'_s and R' with X' satisfy the $\sim$ relation.

There are three rules for transforming replica configurations. We assume that the transformation happens on a replica r' in $\mathcal{R}$. Firstly, we handle the $\textsc{Input}_{ins}$ rule. The rule updates the $\mathrm{OR}_{\mathrm{PC}}$ set of a node v' from the input replica configuration of the rule (either R or R_s) by processing the input tuple for the insertion. We will prove that a constructed corresponding configuration by applied the same $\textsc{Input}_{ins}$ rule contains the updated $\mathrm{OR}_{\mathrm{PC}}$ set of v' satisfying the $\sim$ relation. Suppose that

$$R = ((v', \texttt{insert}, u)\alpha, s, n) \quad \text{and} \quad R \Longrightarrow_r (\alpha', s', n') \text{ by } \textsc{Input}_{ins}.$$

Let $(A, L, \mathcal{C}, K) = s(v')$, $e = (v', r', n)$, and $h = (e, (v', \texttt{undef}, \texttt{undef}))$. Then, by the definition of $\textsc{Input}_{ins}$,

$$\mathcal{C} = \{A\}, \qquad \alpha' = \alpha, \qquad n' = n + 1, \qquad \text{and} \qquad s' = s[v' \mapsto X'],$$

where $X' = (A \cup \{h\}, L \cup \{h\}, \{A \cup \{h\}\}, [A \cup \{h\} \mapsto K(A)[u \mapsto K(A)(u) \oplus \{\{h\}\}]])$. Meanwhile, since $R_s \sim R$, there exists a shadow replica state ss such that

$$R_s = ((v', \texttt{insert}, u)\alpha, ss, n) \quad \text{and} \quad ss \sim s.$$

The condition $ss \sim s$ here implies that for some K_s,

$$ss(v) = (A, A, \{A\}, K_s) \quad \text{and} \quad K(A)(u) \subseteq K_s(A)(u) \text{ for all } u.$$

Then, by the $\textsc{Input}_{ins}$ rule,

$$R_s \Longrightarrow_r (\alpha, ss[v' \mapsto X'_s], n+1)$$

where $X'_s = (A \cup \{h\}, A \cup \{h\}, \{A \cup \{h\}\}, [A \cup \{h\} \mapsto K_s(A)[u \mapsto K_s(A)(u) \oplus \{\{h\}\}]])$. Furthermore, we can claim that

$$ss[v' \mapsto X'_s] \sim s[v' \mapsto X'] \quad \text{and} \quad (\alpha, ss[v' \mapsto X'_s], n{+}1) \sim (\alpha, s[v' \mapsto X'], n{+}1) = (\alpha', s', n').$$

This is because $K(A)(u) \subseteq K_s(A)(u)$ implies $K(A)(u) \oplus \{\{h\}\} \subseteq K_s(A)(u) \oplus \{\{h\}\}$.

We can similarly prove that if $R_s \Longrightarrow_r R'_s$ by the $\textsc{Input}_{ins}$ rule for shadow replica configurations, there exists R' such that $R'_s \sim R'$ and $R \Longrightarrow_r R'$ by the $\textsc{Input}_{ins}$ rule for normal replica configurations.

Secondly, we handle the $\textsc{Input}_{del}$ rule. For the normal replica configuration, the rule updates the $\mathrm{OR}_{\mathrm{PC}}$ set of a node v' by processing the input tuple for the deletion. However, for the shadow replica configuration, the rule does not change the input replica configuration. We will prove that a constructed corresponding configuration by applied the same $\textsc{Input}_{del}$ rule contains the updated $\mathrm{OR}_{\mathrm{PC}}$ set of v' satisfying the $\sim$ relation. Suppose that

$$R = ((v', \texttt{delete}, u)\alpha, s, n) \quad \text{and} \quad R \Longrightarrow_r (\alpha', s', n') \text{ by } \textsc{Input}_{del}.$$

Let $(A, L, \mathcal{C}, K) = s(v')$. Then, by the definition of $\textsc{Input}_{del}$,

$$\mathcal{C} = \{A\}, \qquad \alpha' = \alpha, \qquad n' = n, \qquad \text{and} \qquad s' = s[v' \mapsto X'],$$

where $X' = (A, L \setminus \bigcup\{d \mid d \in K(A)(u)\}, \{A\}, [A \mapsto K(A)[u \mapsto \{\}]])$. Meanwhile, since $R_s \sim R$, there exists a shadow replica state ss such that

$$R_s = ((v', \texttt{delete}, u)\alpha, ss, n) \quad \text{and} \quad ss \sim s.$$

The condition $ss \sim s$ here implies that for some K_s,

$$ss(v') = (A, A, \{A\}, K_s) \quad \text{and} \quad K(A)(u) \subseteq K_s(A)(u) \ \text{ for all } u.$$

Then, by the $\textsc{Input}_{del}$ rule for a shadow replica state,

$$R_s \Longrightarrow_r (\alpha, ss[v' \mapsto X'_s], n)$$

where $X'_s = (A, A, \{A\}, K_s)$. Furthermore, we can claim that

$$ss[v' \mapsto X'_s] \sim s[v' \mapsto X'] \quad \text{and} \quad (\alpha, ss[v' \mapsto X'_s], n) \sim (\alpha, s[v' \mapsto X'], n) = (\alpha', s', n').$$

This is because i) $\{\} \subseteq K_s(A)(u)$ for the deleted value u and ii) $K(A)(u') \subseteq K_s(A)(u')$ for all u' in $\mathcal{U}$ implies $K(A)(u'') \subseteq K_s(A)(u'')$ for all u'' in $\mathcal{U} \setminus \{u\}$.

We can similarly prove that if $R_s \Longrightarrow_r R'_s$ by the INPUT_{del} rule for shadow replica configurations, there exists R' such that $R'_s \sim R'$ and $R \Longrightarrow_r R'$ by the INPUT_{del} rule for normal replica configurations.

Lastly for transforming replica configurations, we now handle the DFLOW rule. The rule updates the OR_{PC} set of a node v' by i) computing the dataflow operation using the OR_{PC} sets of its previous nodes from the input replica configuration (either R or R_s) and ii) combining the result of the dataflow operation with the existing OR_{PC} set of v' from the input replica configuration. We will prove that a constructed corresponding configuration by applied the same DFLOW rule contains the updated OR_{PC} set of v' satisfying the $\sim$ relation. Suppose that

$$R = (\alpha, s, n) \quad \text{and} \quad R \Longrightarrow_r (\alpha', s', n') \text{ by DFLOW.}$$

Let $(A, L, \mathcal{C}, K) = s(v')$. Then, by the definition of DFLOW,

$$\alpha' = \alpha, \quad n' = n, \quad (v', F(v'), s|_{pred(v')}) \Longrightarrow_s (A', L', \mathcal{C}', K'), \quad \text{and} \quad s' = s[v' \mapsto X''],$$

where $X'' = (A, L, \mathcal{C}, K) \sqcup (A', L', \mathcal{C}', K')$.

From the definition of the DFLOW rule, we could consider that the DFLOW rule has two steps: i) executing a dataflow operation through $\Longrightarrow_s$ rules, and ii) combining the output of a $\Longrightarrow_s$ rule with the existing OR_{PC} set of the node using the $\sqcup$ operation. We assume that, if we let R' be the output replica configuration of the DFLOW rule, then there is a temporary replica configuration R^* between the two steps such that

$$R \Longrightarrow_{step} R^* \Longrightarrow_{step} R'.$$

Similarly for shadow replica configurations,

$$R_s \Longrightarrow_{step} R^*_s \Longrightarrow_{step} R'_s.$$

We will prove the claimed preservation of the $\sim$ relation for each step: $R^*_s \sim R^*$ and $R'_s \sim R'$. We assume that updating dataflow paths is successfully performed for these $\Longrightarrow_s$ rules. Therefore, we can ignore the comparison of dataflow paths here.

We handle the first step of the DFLOW rule: the $\Longrightarrow_s$ rule. There are four sub-rules for the $\Longrightarrow_s$ rule. The $\text{F}(v)_{map}$ rule executes the `map()` operation on the OR_{PC} set of a node w from either R or R_s, if we let $pred(v') = \{w\}$. We will prove that a constructed corresponding configuration by applied the same $\text{F}(v)_{map}$ rule contains the updated OR_{PC} set of v' satisfying the $\sim$ relation. Suppose that

$$(v', \texttt{map}(g), s|_{\{w\}}) \Longrightarrow_s X' \text{ by } \text{F}(v)_{map}.$$

Let $(A, L, \mathcal{C}, K) = s(w)$. Then, by the definition of $\text{F}(v)_{map}$,

$$X' = (A', L', \mathcal{C}', K')$$

where

$$A' = A, \quad L' = L, \quad \mathcal{C}' = \mathcal{C}, \quad \text{and} \quad \forall C_i \in \mathcal{C}, u \in \mathcal{U}.\ K'(C_i)(u) = \bigoplus\{K(C_i)(u') \mid u' \in g^{-1}(u)\}.$$

Meanwhile, since $R_s \sim R$, there exists a shadow replica state ss such that

$$R_s = (\alpha, ss, n) \quad \text{and} \quad ss \sim s.$$

The condition $ss \sim s$ here implies that for some K_s,

$$ss(w) = (A, A, \mathcal{C}, K_s) \quad \text{and} \quad K(C_i)(u) \subseteq K_s(C_i)(u) \ \text{ for all } C_i \text{ in } \mathcal{C} \text{ and } u \text{ in } \mathcal{U}.$$

Then, by the $\mathrm{F}(v)_{map}$ rule for a shadow replica state,

$$pred(v') = \{w\} \quad \text{and} \quad (v', \texttt{map}(g), ss|_{\{w\}}) \Longrightarrow_s X'_s$$

where $X'_s = (A, A, \mathcal{C}, K'_s)$ and $\forall C_i \in \mathcal{C}, u \in \mathcal{U}.\ K'_s(C_i)(u) = \bigoplus\{K_s(C_i)(u') \mid u' \in g^{-1}(u)\}$. Furthermore, we can claim that

$$ss[v' \mapsto X'_s] \sim s[v' \mapsto X'].$$

This is because

$$\begin{aligned}
&\forall C_i \in \mathcal{C}, u \in \mathcal{U}.\ K(C_i)(u) \subseteq K_s(C_i)(u) \\
\Longrightarrow\ &\forall C_i \in \mathcal{C}, u \in \mathcal{U}.\ \bigcup\{K(C_i)(u') \mid u' \in g^{-1}(u)\} \subseteq \bigcup\{K_s(C_i)(u') \mid u' \in g^{-1}(u)\} \\
\Longleftrightarrow\ &\forall C_i \in \mathcal{C}, u \in \mathcal{U}.\ \bigoplus\{K(C_i)(u') \mid u' \in g^{-1}(u)\} \subseteq \bigoplus\{K_s(C_i)(u') \mid u' \in g^{-1}(u)\}.
\end{aligned}$$

We can similarly prove that if $(v', \texttt{map}(g), ss|_{pred(v')}) \Longrightarrow_s X'_s$ by the $\mathrm{F}(v)_{map}$ rule for shadow replica configurations, there exists X' such that

$$ss[v' \mapsto X'_s] \sim s[v' \mapsto X']$$

and

$$(v', \texttt{map}(g), s|_{pred(v')}) \Longrightarrow_s X'$$

by the $\mathrm{F}(v)_{map}$ rule for normal replica configurations.

We now handle the $\mathrm{F}(v)_{filter}$ rule. For the normal replica configuration, the rule executes the `filter`() operation on the $\mathrm{OR}_{\mathrm{PC}}$ set of a node w from the input replica configuration, if we let $pred(v') = \{w\}$. However, for the shadow replica configuration, the rule only updates the dataflow paths of the input replica configuration. We will prove that a constructed corresponding configuration by applied the same $\mathrm{F}(v)_{filter}$ rule contains the updated $\mathrm{OR}_{\mathrm{PC}}$ set of v' satisfying the $\sim$ relation. Suppose that

$$(v', \texttt{filter}(g), s|_{\{w\}}) \Longrightarrow_s X' \text{ by } \mathrm{F}(v)_{filter}.$$

Let $(A, L, \mathcal{C}, K) = s(w)$. Then, by the definition of $\mathrm{F}(v)_{\mathit{filter}}$,

$$A' = A, \qquad L' = L, \qquad \mathcal{C}' = \mathcal{C}, \qquad \text{and}$$
$$\forall C_i \in \mathcal{C}, u \in \mathcal{U}.\ K'(C_i)(u) = (\text{if } (g(u) = \texttt{true}) \text{ then } K(C_i)(u) \text{ else } \{\}),$$

where $X' = (A', L', \mathcal{C}', K')$. Meanwhile, since $R_s \sim R$, there exists a shadow replica state ss such that

$$R_s = (\alpha, ss, n) \quad \text{and} \quad ss \sim s.$$

The condition $ss \sim s$ here implies that for some K_s,

$$ss(w) = (A, A, \mathcal{C}, K_s) \quad \text{and} \quad K(C_i)(u) \subseteq K_s(C_i)(u) \ \text{ for all } C_i \text{ in } \mathcal{C} \text{ and } u \text{ in } \mathcal{U}.$$

Then, by the $\mathrm{F}(v)_{\mathit{filter}}$ rule for a shadow replica state,

$$\mathit{pred}(v') = \{w\} \quad \text{and} \quad (v', \texttt{filter}(g), ss|_{\{w\}}) \Longrightarrow_s X'_s$$

where

$$X'_s = (A, A, \mathcal{C}, K_s).$$

Furthermore, we can claim that

$$ss[v' \mapsto X'_s] \sim s[v' \mapsto X'].$$

This is because for every C_i in $\mathcal{C}$, i) $\{\} \subseteq K_s(C_i)(u)$ for all u in $\{w \in \mathcal{U} \mid g(u) = \texttt{true}\}$ and ii) $K(C_i)(u') \subseteq K_s(C_i)(u')$ for all u' in $\mathcal{U}$ implies $K(C_i)(w') \subseteq K_s(C_i)(w')$ for all w' in $\{w \in \mathcal{U} \mid g(u) = \texttt{false}\}$.

We can similarly prove that if $(v', \texttt{filter}(g), ss|_{\mathit{pred}(v)}) \Longrightarrow_s X'_s$ by the $\mathrm{F}(v)_{\mathit{filter}}$ rule for shadow replica configurations, there exists X' such that

$$ss[v' \mapsto X'_s] \sim s[v' \mapsto X']$$

and

$$(v', \texttt{filter}(g), s|_{\mathit{pred}(v')}) \Longrightarrow_s X'$$

by the $\mathrm{F}(v)_{\mathit{filter}}$ rule for normal replica configurations.

The $\mathrm{F}(v)_{\cup}$ rule executes the $\cup$ operation on the $\mathrm{OR}_{\mathrm{PC}}$ sets of nodes w, w' from either R or R_s, if we let $\mathit{pred}(v') = \{w, w'\}$. We will prove that a constructed corresponding configuration by applied the same $\mathrm{F}(v)_{\cup}$ rule contains the updated $\mathrm{OR}_{\mathrm{PC}}$ set of v' satisfying the $\sim$ relation. Suppose that

$$(v', \cup, s|_{\{w,w'\}}) \Longrightarrow_s X'' \text{ by } \mathrm{F}(v)_{\cup}.$$

Let $(A, L, \mathcal{C}, K) = s(w)$ and $(A', L', \mathcal{C}', K') = s(w')$. Then, by the definition of $\mathrm{F}(v)_{\cup}$,

$$X'' = (A'', L'', \mathcal{C}'', K'')$$

where

$$
\begin{aligned}
&A'' = A \cup A',\\
&L'' = (L \cap L') \cup (L \setminus A') \cup (L' \setminus A),\\
&\mathcal{C}'' = max(\{C_i \cup C_j \mid C_i \in \mathcal{C} \wedge C_j \in \mathcal{C}'\}),\\
&\forall C_k \in \mathcal{C}'', u \in \mathcal{U}.\ K''(C_k)(u) = \big\{d \in (K(C_i)(u) \oplus K'(C_j)(u)) \mid\\
&\qquad\qquad\qquad\qquad d \subseteq L'' \wedge (C_i, C_j) \in (\mathcal{C} \times \mathcal{C}') \wedge C_k = C_i \cup C_j\big\}.
\end{aligned}
$$

Meanwhile, since $R_s \sim R$, there exists a shadow replica state ss such that

$$R_s = (\alpha, ss, n) \quad \text{and} \quad ss \sim s.$$

The condition $ss \sim s$ here implies that for some K_s and K'_s,

$$
\begin{aligned}
ss(w) = (A, A, \mathcal{C}, K_s) \quad &\text{and} \quad K(C_i)(u) \subseteq K_s(C_i)(u) \ \text{ for all } C_i \text{ in } \mathcal{C} \text{ and } u \text{ in } \mathcal{U},\\
ss(w') = (A', A', \mathcal{C}', K'_s) \quad &\text{and} \quad K'(C_j)(u) \subseteq K'_s(C_j)(u) \ \text{ for all } C_j \text{ in } \mathcal{C}' \text{ and } u \text{ in } \mathcal{U}.
\end{aligned}
$$

Then, by the $\mathrm{F}(v)_{\cup}$ rule for a shadow replica state,

$$pred(v') = \{w, w'\} \quad \text{and} \quad (v', \cup, ss|_{\{w,w'\}}) \Longrightarrow_s X''_s$$

where

$$X''_s = (A'', A'', \mathcal{C}'', K''_s)$$

and

$$
\begin{aligned}
&\forall C_k \in \mathcal{C}'', u \in \mathcal{U}.\ K''_s(C_k)(u) = \big\{d \in (K_s(C_i)(u) \oplus K'_s(C_j)(u)) \mid\\
&\qquad\qquad\qquad\qquad d \subseteq L'' \wedge (C_i, C_j) \in (\mathcal{C} \times \mathcal{C}') \wedge C_k = C_i \cup C_j\big\}.
\end{aligned}
$$

Furthermore, we can claim that

$$ss[v' \mapsto X''_s] \sim s[v' \mapsto X''].$$

This is because

$$
\begin{aligned}
&\forall C_k \in \mathcal{C}'', u \in \mathcal{U}.\ C_k = C_i \cup C_j \wedge K(C_i)(u) \subseteq K_s(C_i)(u) \wedge K'(C_j)(u) \subseteq K'_s(C_j)(u)\\
\Longrightarrow\ &\forall C_k \in \mathcal{C}'', u \in \mathcal{U}.\ C_k = C_i \cup C_j \wedge K(C_i)(u) \cup K'(C_j)(u) \subseteq K_s(C_i)(u) \cup K'_s(C_j)(u)\\
\Longrightarrow\ &\forall C_k \in \mathcal{C}'', u \in \mathcal{U}.\ C_k = C_i \cup C_j \wedge K(C_i)(u) \oplus K'(C_j)(u) \subseteq K_s(C_i)(u) \oplus K'_s(C_j)(u)\\
\Longrightarrow\ &\forall C_k \in \mathcal{C}'', u \in \mathcal{U}.\\
&\qquad \big\{d \in (K(C_i)(u) \oplus K'(C_j)(u)) \mid d \subseteq L'' \wedge (C_i, C_j) \in (\mathcal{C} \times \mathcal{C}') \wedge C_k = C_i \cup C_j\big\}\\
&\qquad \subseteq\\
&\qquad \big\{d \in (K_s(C_i)(u) \oplus K'_s(C_j)(u)) \mid d \subseteq L'' \wedge (C_i, C_j) \in (\mathcal{C} \times \mathcal{C}') \wedge C_k = C_i \cup C_j\big\}.
\end{aligned}
$$

We can similarly prove that if $(v', \cup, ss|_{pred(v')}) \Longrightarrow_s X''_s$ by the $\mathrm{F}(v)_\cup$ rule for shadow replica configurations, there exists X'' such that

$$ss[v' \mapsto X''_s] \sim s[v' \mapsto X'']$$

and

$$(v', \cup, s|_{pred(v')}) \Longrightarrow_s X''$$

by the $\mathrm{F}(v)_\cup$ rule for normal replica configurations.

The $\mathrm{F}(v)_\times$ rule executes the $\times$ operation on the $\mathrm{OR_{PC}}$ sets of nodes w, w' from either R or R_s, if we let $pred(v') = \{w, w'\}$. We will prove that a constructed corresponding configuration by applied the same $\mathrm{F}(v)_\times$ rule contains the updated $\mathrm{OR_{PC}}$ set of v' satisfying the $\sim$ relation. Suppose that

$$(v', \times, s|_{\{w,w'\}}) \Longrightarrow_s X'' \text{ by } \mathrm{F}(v)_\times.$$

Let $(A, L, \mathcal{C}, K) = s(w)$ and $(A', L', \mathcal{C}', K') = s(w')$. Then, by the definition of $\mathrm{F}(v)_\times$,

$$X'' = (A'', L'', \mathcal{C}'', K'')$$

where

$$\begin{aligned}
&A'' = A \cup A',\\
&L'' = (L \cap L') \cup (L \setminus A') \cup (L' \setminus A),\\
&\mathcal{C}'' = max(\{C_i \cup C_j \mid C_i \in \mathcal{C} \wedge C_j \in \mathcal{C}'\}),\\
&\forall C_k \in \mathcal{C}'', u \in \mathcal{U}.\ K''(C_k)(u) = \big\{d \in (K(C_i)(u_1) \otimes K'(C_j)(u_2)) \mid\\
&\qquad\qquad d \subseteq L'' \wedge (C_i, C_j) \in (\mathcal{C} \times \mathcal{C}') \wedge C_k = C_i \cup C_j \wedge u = (u_1, u_2)\big\}.
\end{aligned}$$

Meanwhile, since $R_s \sim R$, there exists a shadow replica state ss such that

$$R_s = (\alpha, ss, n) \quad \text{and} \quad ss \sim s.$$

The condition $ss \sim s$ here implies that for some K_s and K'_s,

$$\begin{aligned}
ss(w) = (A, A, \mathcal{C}, K_s) \quad &\text{and} \quad K(C_i)(u) \subseteq K_s(C_i)(u) \ \text{ for all } C_i \text{ in } \mathcal{C} \text{ and } u \text{ in } \mathcal{U},\\
ss(w') = (A', A', \mathcal{C}', K'_s) \quad &\text{and} \quad K'(C_j)(u) \subseteq K'_s(C_j)(u) \ \text{ for all } C_j \text{ in } \mathcal{C}' \text{ and } u \text{ in } \mathcal{U}.
\end{aligned}$$

Then, by the $\mathrm{F}(v)_\times$ rule for a shadow replica state,

$$pred(v') = \{w, w'\} \quad \text{and} \quad (v', \times, ss|_{\{w,w'\}}) \Longrightarrow_s X''_s$$

where

$$X''_s = (A'', A'', \mathcal{C}'', K''_s)$$

and

$$\forall C_k \in \mathcal{C}'', u \in \mathcal{U}.\ K''_s(C_k)(u) = \big\{ d \in (K_s(C_i)(u) \otimes K'_s(C_j)(u)) \mid \\ d \subseteq L'' \wedge (C_i, C_j) \in (\mathcal{C} \times \mathcal{C}') \wedge C_k = C_i \cup C_j \wedge u = (u_1, u_2) \big\}.$$

Furthermore, we can claim that

$$ss[v' \mapsto X''_s] \sim s[v' \mapsto X''].$$

This is because

$$\begin{aligned}
&\forall C_i \in \mathcal{C}, C_j \in \mathcal{C}', u_1, u_2 \in \mathcal{U}.\ K(C_i)(u_1) \subseteq K_s(C_i)(u_1) \wedge K'(C_j)(u_2) \subseteq K'_s(C_j)(u_2) \\
\Longrightarrow\ &\forall C_k \in \mathcal{C}'', u \in \mathcal{U}. \\
&\quad C_k = C_i \cup C_j \wedge u = (u_1, u_2) \wedge K(C_i)(u_1) \subseteq K_s(C_i)(u_1) \wedge K'(C_j)(u_2) \subseteq K'_s(C_j)(u_2) \\
\Longrightarrow\ &\forall C_k \in \mathcal{C}'', u \in \mathcal{U}. \\
&\quad C_k = C_i \cup C_j \wedge u = (u_1, u_2) \\
&\quad \wedge \\
&\quad \{d \cup d' \mid d \in K(C_i)(u_1) \wedge d' \in K'(C_j)(u_2)\} \\
&\qquad \subseteq \{d \cup d' \mid d \in K_s(C_i)(u_1) \wedge d' \in K'_s(C_j)(u_2)\} \\
\Longrightarrow\ &\forall C_k \in \mathcal{C}'', u \in \mathcal{U}. \\
&\quad C_k = C_i \cup C_j \wedge u = (u_1, u_2) \wedge K(C_i)(u_1) \otimes K'(C_j)(u_2) \subseteq K_s(C_i)(u_1) \otimes K'_s(C_j)(u_2) \\
\Longrightarrow\ &\forall C_k \in \mathcal{C}'', u \in \mathcal{U}. \\
&\quad \big\{ d \in (K(C_i)(u_1) \otimes K'(C_j)(u_2)) \\
&\qquad \mid d \subseteq L'' \wedge (C_i, C_j) \in (\mathcal{C} \times \mathcal{C}') \wedge C_k = C_i \cup C_j \wedge u = (u_1, u_2) \big\} \\
&\quad \subseteq \\
&\quad \big\{ d \in (K_s(C_i)(u_1) \otimes K'_s(C_j)(u_2)) \\
&\qquad \mid d \subseteq L'' \wedge (C_i, C_j) \in (\mathcal{C} \times \mathcal{C}') \wedge C_k = C_i \cup C_j \wedge u = (u_1, u_2) \big\}.
\end{aligned}$$

We can similarly prove that if $(v', \times, ss|_{pred(v')}) \Longrightarrow_s X''_s$ by the $\mathrm{F}(v)_\times$ rule for shadow replica configurations, there exists X'' such that

$$ss[v' \mapsto X''_s] \sim s[v' \mapsto X'']$$

and

$$(v', \times, s|_{pred(v')}) \Longrightarrow_s X''$$

by the $\mathrm{F}(v)_\times$ rule for normal replica configurations.

We do not prove the claimed preservation of the $\sim$ relation for the $\sqcup$ operation (the second step of the DFlow rule) here since the behaviors of the $\sqcup$ operation are identical to the ones of the $\textsc{Merge}_{dflow}$ rule for $S_s \sim S$.

We have proved the claimed preservation of the $\sim$ relation for replica configurations ($R_s \sim R$) that is a part of the STEP rule for system configurations. We now prove the claimed preservation of the $\sim$ relation for two remaining rules of system configurations ($S_s \sim S$): the MERGE$_{input}$ and MERGE$_{dflow}$ rules.

We firstly handle the MERGE$_{input}$ rule. The rule updates an OR$_{\mathrm{PC}}$ set of a node v' from a replica r_1's state by combining it with an OR$_{\mathrm{PC}}$ set of a node v' from a replica r_2's state with the operation $\sqcup_s$ for either S or S_s. We will prove that a constructed corresponding configuration by applied the same MERGE$_{input}$ rule contains the updated OR$_{\mathrm{PC}}$ sets of v' satisfying the $\sim$ relation. Suppose that

$$s = S(r_1).s, \quad s' = S(r_2).s \quad \text{and} \quad S \Longrightarrow S' \text{ by } \textsc{Merge}_{input}.$$

Let $(A, L, \mathcal{C}, K) = s(v')$ and $(A', L', \mathcal{C}', K') = s'(v')$. Then, by the definition of MERGE$_{input}$,

$$\mathcal{C} = \{A\}, \quad \mathcal{C}' = \{A'\} \quad \text{and} \quad S' = S[r_1.s \mapsto s[v' \mapsto X'']]$$

where $X'' = (A, L, \{A\}, K) \sqcup_s (A', L', \{A'\}, K')$. Meanwhile, since $S_s \sim S$, there exist shadow replica state ss and ss' such that

$$ss = S_s(r_1).ss, \quad ss' = S_s(r_2).ss, \quad ss \sim s. \quad \text{and} \quad ss' \sim s'.$$

The conditions $ss \sim s$ and $ss' \sim s'$ here imply that for some K_s and K'_s,

$$\begin{aligned} ss(v') = (A, A, \{A\}, K_s) \quad &\text{and} \quad K(A)(u) \subseteq K_s(A)(u) \ \text{ for all } u, \\ ss'(v') = (A', A', \{A'\}, K'_s) \quad &\text{and} \quad K'(A')(u) \subseteq K'_s(A')(u) \ \text{ for all } u. \end{aligned}$$

Then, by the MERGE$_{input}$ rule,

$$S_s \Longrightarrow S_s[r_1.ss \mapsto ss[v' \mapsto X''_s]]$$

where $X''_s = (A, A, \{A\}, K_s) \sqcup (A', A', \{A'\}, K'_s)$. Furthermore, we can claim that

$$\begin{aligned} &ss[v' \mapsto X''_s] \sim s[v' \mapsto X''], \quad ss'[v' \mapsto X''_s] \sim s'[v' \mapsto X''] \quad \text{and} \\ &S[r_1.s \mapsto s[v' \mapsto X'']] \sim S_s[r_1.ss \mapsto ss[v' \mapsto X''_s]]. \end{aligned}$$

This is because

$$\begin{aligned} &\forall u \in \mathcal{U}.\ K(A)(u) \subseteq K_s(A)(u) \wedge K'(A')(u) \subseteq K'_s(A')(u) \\ \Longrightarrow\ &\forall u \in \mathcal{U}.\ K(A)(u) \cup K'(A')(u) \subseteq K_s(A)(u) \cup K'_s(A')(u) \\ \Longleftrightarrow\ &\forall u \in \mathcal{U}.\ K(A)(u) \sqcup K'(A')(u) \subseteq K_s(A)(u) \sqcup K'_s(A')(u) \\ \Longrightarrow\ &\forall u \in \mathcal{U}.\ \{d \in K(A)(u) \cup K'(A')(u) \mid d \subseteq A''\} \subseteq \{d \in K_s(A)(u) \cup K'_s(A')(u) \mid d \subseteq A''\} \\ \Longrightarrow\ &\forall u \in \mathcal{U}.\ K''(A'')(u) \subseteq K''_s(A'')(u). \end{aligned}$$

We can similarly prove that if $S_s \Longrightarrow S'_s$ by the $\textsc{Merge}_{input}$ rule for shadow system configurations, there exists S' such that $S'_s \sim S'$ and $S \Longrightarrow S'$ by the $\textsc{Merge}_{input}$ rule for normal system configurations.

We now handle the $\textsc{Merge}_{dflow}$ rule. The rule updates an $\mathrm{OR}_{\mathrm{PC}}$ set of a node v' from a replica r_1's state by combining it with an $\mathrm{OR}_{\mathrm{PC}}$ set of a node v' from a replica r_2's state with the operation $\sqcup$ for either S or S_s. We will prove that a constructed corresponding configuration by applied the same $\textsc{Merge}_{dflow}$ rule contains the updated $\mathrm{OR}_{\mathrm{PC}}$ sets of v' satisfying the $\sim$ relation. Suppose that

$$s = S(r_1).s, \quad s' = S(r_2).s \quad \text{and} \quad S \Longrightarrow S' \text{ by } \textsc{Merge}_{dflow}.$$

Let $(A, L, \mathcal{C}, K) = s(v')$ and $(A', L', \mathcal{C}', K') = s'(v')$. Then, by the definition of $\textsc{Merge}_{dflow}$,

$$S' = S[r_1.s \mapsto s[v' \mapsto X'']]$$

where $X'' = (A'', L'', \mathcal{C}'', K'')$ and $(A'', L'', \mathcal{C}'', K'') = (A, L, \mathcal{C}, K) \sqcup (A', L', \mathcal{C}', K')$. Meanwhile, since $S_s \sim S$, there exist shadow replica state ss and ss' such that

$$ss = S_s(r_1).ss, \quad ss' = S_s(r_2).ss, \quad ss \sim s. \quad \text{and} \quad ss' \sim s'.$$

The conditions $ss \sim s$ and $ss' \sim s'$ here imply that for some K_s and K'_s,

$$\begin{aligned} ss(v') = (A, A, \mathcal{C}, K_s) \quad &\text{and} \quad K(C_i)(u) \subseteq K_s(C_i)(u) \ \text{ for all } C_i \text{ in } \mathcal{C} \text{ and } u \text{ in } \mathcal{U}, \\ ss'(v') = (A', A', \mathcal{C}', K'_s) \quad &\text{and} \quad K'(C_j)(u) \subseteq K'_s(C_j)(u) \ \text{ for all } C_j \text{ in } \mathcal{C}' \text{ and } u \text{ in } \mathcal{U}. \end{aligned}$$

Then, by the $\textsc{Merge}_{dflow}$ rule,

$$S_s \Longrightarrow S_s[r_1.ss \mapsto ss[v' \mapsto X''_s]]$$

where $X''_s = (A'', A'', \mathcal{C}'', K''_s)$ and $(A'', A'', \mathcal{C}'', K''_s) = (A, A, \mathcal{C}, K_s) \sqcup (A', A', \mathcal{C}', K'_s)$. Furthermore, we can claim that

$$\begin{aligned} &ss[v' \mapsto X''_s] \sim s[v' \mapsto X''], \quad ss'[v' \mapsto X''_s] \sim s'[v' \mapsto X''] \quad \text{and} \\ &S[r_1.s \mapsto s[v' \mapsto X'']] \sim S_s[r_1.ss \mapsto ss[v' \mapsto X''_s]]. \end{aligned}$$

This is because

$$\begin{aligned} &\forall C_k \in \mathcal{C}'', u \in \mathcal{U}.\ C_k = C_i \cup C_j \wedge K(C_i)(u) \subseteq K_s(C_i)(u) \wedge K'(C_j)(u) \subseteq K'_s(C_j)(u) \\ \Longrightarrow\ &\forall C_k \in \mathcal{C}'', u \in \mathcal{U}.\ C_k = C_i \cup C_j \wedge K(C_i)(u) \cup K'(C_j)(u) \subseteq K_s(C_i)(u) \cup K'_s(C_j)(u) \\ \Longrightarrow\ &\forall C_k \in \mathcal{C}'', u \in \mathcal{U}.\ C_k = C_i \cup C_j \wedge K(C_i)(u) \oplus K'(C_j)(u) \subseteq K_s(C_i)(u) \oplus K'_s(C_j)(u) \\ \Longrightarrow\ &\forall C_k \in \mathcal{C}'', u \in \mathcal{U}. \\ &\qquad \big\{ d \in (K(C_i)(u) \oplus K'(C_j)(u)) \mid d \subseteq L'' \wedge (C_i, C_j) \in (\mathcal{C} \times \mathcal{C}') \wedge C_k = C_i \cup C_j \big\} \\ &\qquad \subseteq \\ &\qquad \big\{ d \in (K_s(C_i)(u) \oplus K'_s(C_j)(u)) \mid d \subseteq L'' \wedge (C_i, C_j) \in (\mathcal{C} \times \mathcal{C}') \wedge C_k = C_i \cup C_j \big\}. \end{aligned}$$

This implies $\forall C_k \in \mathcal{C}'', \forall u \in \mathcal{U}.\ K''(C_k)(u) \subseteq K''_s(C_k)(u)$ because

$$
\begin{aligned}
&\forall C_k \in \mathcal{C}'', \forall u \in \mathcal{U}. \\
&\quad join(L'', C_k, \{K(C_i) \mid C_i \in \mathcal{C} \wedge C_i \subseteq C_k\} \cup \{K'(C_j) \mid C_j \in \mathcal{C}' \wedge C_j \subseteq C''_i\})(u) \\
&\quad \subseteq \\
&\quad join(A'', C_k, \{K_s(C_i) \mid C_i \in \mathcal{C} \wedge C_i \subseteq C_k\} \cup \{K'_s(C_j) \mid C_j \in \mathcal{C}' \wedge C_j \subseteq C_k\})(u) \\
\iff &\forall C_k \in \mathcal{C}'', \forall u \in \mathcal{U}. \\
&\quad \{d \in \left(\forall P_S(u) \in \mathcal{P}_S(u) \cup \mathcal{P}'_S(u).\ \bigcup P_S(u)\right) \mid d \subseteq L''\} \\
&\quad \subseteq \\
&\quad \{d' \in \left(\forall P_{S_s}(u) \in \mathcal{P}_{S_s}(u) \cup \mathcal{P}_{S'_s}(u).\ \bigcup P_{S_s}(u)\right) \mid d' \subseteq A''\} \\
&\qquad \text{where} \\
&\qquad \mathcal{P}_S(u) = \{K(C_i)(u) \mid C_i \in \mathcal{C} \wedge C_i \subseteq C_k\}, \\
&\qquad \mathcal{P}'_S(u) = \{K'(C_j)(u) \mid C_j \in \mathcal{C}' \wedge C_j \subseteq C_k\}, \\
&\qquad \mathcal{P}_{S_s}(u) = \{K_s(C_i)(u) \mid C_i \in \mathcal{C} \wedge C_i \subseteq C_k\}, \\
&\qquad \mathcal{P}_{S'_s}(u) = \{K'_s(C_j)(u) \mid C_j \in \mathcal{C}' \wedge C_j \subseteq C_k\} \\
&\qquad \text{(from the definition of } join()\text{and the definition of } \sqcup).
\end{aligned}
$$

We now have

$$
\begin{aligned}
&\forall C_k \in \mathcal{C}'', \forall u \in \mathcal{U}. \\
&\quad \{d \in \left(\forall P_S(u) \in \mathcal{P}_S(u) \cup \mathcal{P}'_S(u).\ \bigcup P_S(u)\right) \mid d \subseteq L''\} \\
&\quad \subseteq \\
&\quad \{d' \in \left(\forall P_{S_s}(u) \in \mathcal{P}_{S_s}(u) \cup \mathcal{P}_{S'_s}(u).\ \bigcup P_{S_s}(u)\right) \mid d' \subseteq A''\}
\end{aligned}
$$

because

$$
\begin{aligned}
\mathcal{P}_S(u) \subseteq \mathcal{P}_{S_s}(u) \qquad & \text{(from } \forall C_i \in \mathcal{C}, u \in \mathcal{U}.\ K(C_i)(u) \subseteq K_s(C_i)(u)), \\
\mathcal{P}'_S(u) \subseteq \mathcal{P}_{S'_s}(u) \qquad & \text{(from } \forall C_j \in \mathcal{C}', u \in \mathcal{U}.\ K'(C_j)(u) \subseteq K'_s(C_j)(u)), \\
L'' \subseteq A'' \qquad & \text{(from the definition of OR}_{\text{PC}} \text{ sets)}.
\end{aligned}
$$

We can similarly prove that if $S_s \Longrightarrow S'_s$ by the $\textsc{Merge}_{dflow}$ rule for shadow system configurations, there exists S' such that $S'_s \sim S'$ and $S \Longrightarrow S'$ by the $\textsc{Merge}_{dflow}$ rule for normal system configurations.

□

Lemma 5.6.2 shows that alongside the normal execution of a dataflow program from the initial system configuration, we can perform the corresponding computation with operational rules for shadow system configurations. This means that for every reachable

system configuration S, there is a corresponding shadow system configuration S_s (i.e. $S_s \sim S$). Since the `delete`() and `filter`() operations are ignored by the operational rules for shadow system configurations, S_s records all the values and event histories that have ever appeared in the original normal execution that led to S.

5.6.2 Dataflow Tree

In this subsection, we introduce dataflow trees to describe the structure of dataflow programs. The dataflow trees help to understand the characteristics of dataflow paths in dots.

Definition 5.6.3. A *dataflow tree* t for a node v is one of the following inductively-defined tuples:

$$
\begin{aligned}
&(v, \texttt{undef}, \texttt{undef}) \mid \\
&(v, t', \texttt{undef}) \mid (v, \texttt{undef}, t') \mid \\
&(v, t', \texttt{unk}) \mid (v, \texttt{unk}, t'') \mid (v, t', t'')
\end{aligned}
$$

where t' and t'' themselves are inductively-defined dataflow trees for some nodes. If a tree t does not contain `unk` in any of its nested tuple, it is called *complete*.

Note that dataflow paths defined previously are just dataflow trees that do not use the last (v_0, t'_0, t''_0) case anywhere inside. Intuitively, each dataflow tree t represents a part of a dataflow graph. The `undef` represents an unused argument of the $\cup$ operation or the absence of arguments at the input nodes or those with `filter`() and `map`() operations in the dataflow graph. Also, `unk` is used to express the presence of some but unspecified argument in the $\times$ operation in the graph. For dataflow trees t, t', we write $t \in t'$ to mean that t appears somewhere in t' when we view t' as nested tuples. In that case, we write the notation $C[t] = t'$ to express that C is the rest of t' when removing a particular occurrence of t in t'. Now, we could describe the dataflow paths with dataflow trees and the C notation for a dataflow program $P = (G, F)$: a dataflow path t_h is a dataflow tree such that there do not exist v, t'_1, t'_2 and C satisfying $C[(v, t'_1, t'_2)] = t_h$ for some node v in $G.V$ and dataflow trees t'_1, t'_2.

Definition 5.6.4. A dataflow tree t is *compatible* for a dataflow program $P = (G, F)$ if it belongs to the set T that Algorithm 1 with a node v in $G.V$ computes for P. We use the phrase *"a v-oriented dataflow tree"* to refer a P-compatible dataflow tree computed by Algorithm 1 with the node v.

We define the special dataflow tree $\perp$. The tree $\perp$ is *complete* and *compatible* for any dataflow program. Furthermore, the tree $\perp$ is *oriented* for any node in $G.V$ of a dataflow program.

Algorithm 1 Construction of compatible dataflow trees for the dataflow program $P = (G, F)$

procedure CompatibleTree(v)
  if $pred(v) = \emptyset$ **then**
    return $\{(v, \texttt{undef}, \texttt{undef})\}$
  else if $F(v)$ is is $\texttt{map}(g)$ or $\texttt{filter}(g)$ **then**
    return $\{(v, t, \texttt{undef}) \mid t \in \text{CompatibleTrees}(\texttt{fst}(pred(v)))\}$
  else if $F(v)$ is $\cup$ **then**
    $T_1 \leftarrow \text{CompatibleTrees}(\texttt{fst}(pred(v)))$
    $T_2 \leftarrow \text{CompatibleTrees}(\texttt{snd}(pred(v)))$
    return $\{(v, t_1, \texttt{undef}) \mid t_1 \in T_1\} \cup \{(v, \texttt{undef}, t_2) \mid t_2 \in T_2\}$
  else if $F(v)$ is $\times$ **then**
    $T_1 \leftarrow \text{CompatibleTrees}(\texttt{fst}(pred(v)))$
    $T_2 \leftarrow \text{CompatibleTrees}(\texttt{snd}(pred(v)))$
    $T'_1 \leftarrow \{(v, t_1, \texttt{unk}) \mid t_1 \in T_1\}$
    $T'_2 \leftarrow \{(v, \texttt{unk}, t_2) \mid t_2 \in T_2\}$
    $T' \leftarrow \{(v, t_1, t_2) \mid t_1 \in T_1 \wedge t_2 \in T_2\}$
    return $T' \cup T'_1 \cup T'_2$
  end if
end procedure

When dataflow trees t_1 and t_2 are compatible for a dataflow program $P = (G, F)$, sometimes they can be combined by the following `merge`() operation:

$$\texttt{merge}(t_1, t_2) = \begin{cases} t_3, & \text{if there exist } v, t'_1, t'_2, C \text{ such that } C[(v, t'_1, \texttt{unk})] = t_1 \\ & \qquad \wedge\, C[(v, \texttt{unk}, t'_2)] = t_2 \wedge C[(v, t'_1, t'_2)] = t_3; \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Intuitively, the `unk` expresses the absence of information about a dataflow tree. The `merge`() operation removes such incomplete knowledge by combining information from two compatible dataflow trees which are oriented towards the same node.

Definition 5.6.5. A set of dataflow paths T is *unanimous* if

1. T is either the empty set $\{\}$ or a finite set with cardinality n; and
2. when T is finite with cardinality n, there is a sequence of sets of dataflow trees $[T_n, T_{n-1}, \ldots, T_1]$ such that
   (a) $T_n = T$,
   (b) for all $i \in \{2, \ldots, n\}$,
   $$\exists t, t' \in T_i.\, t'' = \texttt{merge}(t, t') \wedge (T_i \setminus \{t, t'\}) \cup \{t''\} = T_{i-1},$$
   (c) T_1 is a singleton set that contains a dataflow tree t_u:
   $$T_1 = \{t_u\}.$$

We define t_u as a *unanimous result*. When T is the empty set $\{\}$, the unanimous result constructed from T is the $\perp$ tree.

The interesting point of a unanimous set of dataflow paths is that we can retrieve a unanimous result after applying the `merge`() operation for the set sequentially.

Lemma 5.6.6 (Dot-as-Unanimous-Set)**.** *Let S be the last system configuration of τ, r be a replica id, and v be a node in $G.V$. Then, for every dot d in $S(r).s(v)$, all dataflow paths in d are v-oriented, and they are also compatible with P. Furthermore, the set of these dataflow paths is* unanimous*, and the unanimous result constructed from this set is complete and also v-oriented.*

Proof. Let τ be $[S_1, S_2, \ldots, S_n]$. We prove that every S_k in τ satisfies the property described for the last S_n in the lemma. Our proof is by induction on k, the index of S_k in τ. For the base case $k = 1$, the system configuration S_1 is S_I, and the claimed property holds simply because for every r in $\mathcal{R}$ and every v in $G.V$, $S_I(r).s(v) = \perp$ and so the only d in $S_I(r).s(v)$ is the empty set $\{\}$. Therefore, if we let T be the set of all the dataflow paths in d, T is the empty set $\{\}$ such that T is unanimous, and the unanimous result constructed from T is $\perp$ that is complete and v-oriented.

We next prove the inductive case. Assume that $k \geq 1$ and S_k satisfies the claimed property. We have to show that S_{k+1} also satisfies the property.

Since τ is an execution, we have

$$S_k \Longrightarrow S_{k+1}.$$

We do the case analysis on the rule used for this transformation. During our case analysis, we use the predicate $\Phi(d, v, s)$ on tuples of a dot, a node and a replica state. The predicate holds if

- d is a dot in the $\mathrm{OR}_{\mathrm{PC}}$ set of the node v from s;
- when T is the set of all the dataflow paths in d,
  - every dataflow path in T is v-oriented and compatible with P, and
  - T is *unanimous*, and the unanimous result constructed from T is complete and v-oriented.

There are three rules to consider for the transformation of system configurations. The $\textsc{Merge}_{input}$ and $\textsc{Merge}_{dflow}$ rules neither add a new dot nor modify the existing dots to/from the input system configurations. But S_k satisfies the claimed property by induction hypothesis. Thus, it is obvious that S_{k+1} also satisfies the claimed property for these two rules.

We now handle the last rule for the transformation of system configurations: the STEP rule. The STEP rule updates the replica configuration of a replica r' in $\mathcal{R}$ by executing a rule for transforming replica configurations. Suppose that

$$S_k(r') = (\alpha, s, n) \quad \text{and} \quad S_k \Longrightarrow S_{k+1} \text{ by STEP}.$$

By the definition of the STEP rule,

$$(\alpha, s, n) \Longrightarrow_r (\alpha', s', n') \qquad \text{and} \qquad S_{k+1} = S_k[r' \mapsto (\alpha', s', n')].$$

In order to prove that S_{k+1} satisfies the claimed property, we need to show that, for every node v' in $G.V$, every dot d' in $s'(v')$ satisfies the predicate $\Phi(d', v', s')$. We will do so using the induction hypothesis that, for every node v in $G.V$, every dot d in $s(v)$ satisfies the predicate $\Phi(d, v, s)$.

Since the STEP rule is defined using

$$(\alpha, s, n) \Longrightarrow_r (\alpha', s', n'),$$

we also do the case analysis on the rule used for this transformation. There are three rules for the transformation of replica configurations. Firstly, we handle the INPUT_{ins} rule. The rule updates the OR_{PC} set of a node v' by processing the input tuple for the insertion. We will prove that every dot d' in $s'(v')$ satisfies the predicate $\Phi(d', v', s')$. Suppose that

$$R = ((v', \texttt{insert}, u)\alpha, s, n) \quad \text{and} \quad R \Longrightarrow_r (\alpha', s', n') \text{ by } \text{INPUT}_{ins}.$$

Let $(A, L, \mathcal{C}, K) = s(v')$, $e = (v', r', n)$, and $h = (e, (v', \texttt{undef}, \texttt{undef}))$. Then, by the definition of the INPUT_{ins} rule,

$$\mathcal{C} = \{A\}, \qquad \alpha' = \alpha, \qquad n' = n + 1, \qquad \text{and} \qquad s' = s[v' \mapsto X'],$$

where $X' = (A \cup \{h\}, L \cup \{h\}, \{A \cup \{h\}\}, [A \cup \{h\} \mapsto K(A)[u \mapsto K(A)(u) \oplus \{\{h\}\}]])$. Meanwhile, since S_k satisfies the claimed property by induction hypothesis, every d in $s(v')$ satisfies the predicate $\Phi(d, v', s)$. Then, we can claim that every dot d' in $s'(v')$ also satisfies the predicate $\Phi(d', v', s')$. This is because

1. the set of all dataflow paths in the newly added dot $\{h\}$ is $\{(v', \texttt{undef}, \texttt{undef})\}$;
2. $(v', \texttt{undef}, \texttt{undef})$, the only member of $\{(v', \texttt{undef}, \texttt{undef})\}$, is v'-oriented and compatible with P;
3. $\{(v', \texttt{undef}, \texttt{undef})\}$ is *unanimous*, and $(v', \texttt{undef}, \texttt{undef})$, the unanimous result constructed from $\{(v', \texttt{undef}, \texttt{undef})\}$, is complete and v'-oriented.

Secondly, we handle the $\textsc{Input}_{del}$ rule. The $\textsc{Input}_{del}$ rule neither adds a new dot nor modifies the existing dots. Therefore, for every node v in $G.V$, every dot d' in $s'(v)$ satisfies the predicate $\Phi(d', v, s')$. This is because every d in $s(v)$ satisfies the predicate $\Phi(d, v, s)$ by induction hypothesis.

Lastly for the transformation of replica configurations, we now handle the DFlow rule. The rule updates the $\mathrm{OR}_{\mathrm{PC}}$ set of a node v' in $G.V$ by i) computing the dataflow operation using the $\mathrm{OR}_{\mathrm{PC}}$ sets of its previous nodes from the input replica configuration of the rule (α, s, n) and ii) combining the result of the dataflow operation with the existing $\mathrm{OR}_{\mathrm{PC}}$ set of v' from the input replica configuration. We will prove that, if we let (α', s', n') be the output replica configuration of the rule, then every dot d' in $s'(v')$ satisfies the predicate $\Phi(d', v', s')$. Suppose that

$$R = (\alpha, s, n) \quad \text{and} \quad R \Longrightarrow_r (\alpha', s', n') \text{ by DFlow.}$$

Let $(A, L, \mathcal{C}, K) = s(v')$. Then, by the definition of the DFlow rule,

$$\alpha' = \alpha, \quad n' = n, \quad (v', F(v'), s|_{pred(v')}) \Longrightarrow_s (A', L', \mathcal{C}', K'), \quad \text{and} \quad s' = s[v' \mapsto X''],$$

where $X'' = (A, L, \mathcal{C}, K) \sqcup (A', L', \mathcal{C}', K')$.

From the definition of the DFlow rule, we could consider that the DFlow rule has two steps: executing a dataflow operation through $\Longrightarrow_s$ rules, and combining the output of a $\Longrightarrow_s$ rule with the existing $\mathrm{OR}_{\mathrm{PC}}$ set of the node using the $\sqcup$ operation. We assume that, if we let R' be the output replica configuration of the DFlow rule, then there is a temporary replica configuration R^* between the two steps such that

$$R \Longrightarrow_{step} R^* \Longrightarrow_{step} R'.$$

We will prove that, if we let $R^* = (\alpha^*, s^*, n^*)$ and $R' = (\alpha', s', n')$, then every dot d^* in $s^*(v')$ and every dot d' in $s'(v')$ satisfy the predicate $\Phi(d^*, v', s^*)$ and the predicate $\Phi(d', v', s')$, respectively.

We handle the first step of the DFlow rule: the $\Longrightarrow_s$ rules. The $\mathrm{F}(v)_{map}$ rule executes the `map()` operation on the $\mathrm{OR}_{\mathrm{PC}}$ set of a node w from R, if we let $pred(v') = \{w\}$. We will prove that, if we let $X'' = s^*(v')$, then every dot d'' in X'' satisfies the predicate $\Phi(d'', v', s^*)$. Suppose that

$$(v', \texttt{map}(g), s|_{\{w\}}) \Longrightarrow_s X'' \text{ by } \mathrm{F}(v)_{map}.$$

Let $(A, L, \mathcal{C}, K) = s(w)$. Then, by the definition of $\mathrm{F}(v)_{map}$,

$$((A, L, \mathcal{C}, K), (v', self, \texttt{undef})) \Longrightarrow_{t_h} (A', L', \mathcal{C}', K') \quad \text{and} \quad X'' = (A'', L'', \mathcal{C}'', K'')$$

where

$$A'' = A', \qquad L'' = L', \qquad \mathcal{C}'' = \mathcal{C}', \qquad \text{and}$$
$$\forall C_j \in \mathcal{C}', u \in \mathcal{U}.\ K''(C_j)(u) = \bigoplus\{K'(C_j)(u') \mid u' \in g^{-1}(u)\}.$$

Meanwhile, since S_k satisfies the claimed property by induction hypothesis, for every C_i in $\mathcal{C}$ and u in $\mathcal{U}$, every dot d in $K(C_i)(u)$ satisfies the predicate $\Phi(d, w, s)$. Then, we can claim that, for every C_j in $\mathcal{C}'$ and u in $\mathcal{U}$, every dot d'' in $K''(C_j)(u)$ satisfies the predicate $\Phi(d'', v', s^*)$. This is because

1. by the definition of the $\mathrm{F}(v)_{map}$ rule, every C_j in $\mathcal{C}'$ is computed from C_i in $\mathcal{C}$ such that $C_j = \{(e, (v', t_h, \texttt{undef})) \mid (e, t_h) \in C_i\}$;

2. by the definition of the $\mathrm{F}(v)_{map}$ rule, for every u in $\mathcal{U}$, every dot d'' in $K''(C_j)(u)$ is computed from d in $K(C_i)(u)$ such that $d'' = \{(e, (v', t_h, \texttt{undef})) \mid (e, t_h) \in d\}$ where C_j is computed from C_i;

3. if we let T be the set of all dataflow paths in d and T'' be the set of all dataflow paths in d'', then
$$T'' = \{(v', t, \texttt{undef}) \mid t \in T\};$$

4. every dataflow path in T'' is v'-oriented and compatible with P;

5. if we let t_u be the unanimous result constructed from T, then T'' is *unanimous*, and $(v', t_u, \texttt{undef})$, the unanimous result constructed from T'', is complete and v'-oriented.

The $\mathrm{F}(v)_{filter}$ rule executes the `filter()` operation on the $\mathrm{OR}_{\mathrm{PC}}$ set of a node w from R, if we let $pred(v') = \{w\}$. We will prove that, if we let $X'' = s^*(v')$, then every dot d'' in X'' satisfies the predicate $\Phi(d'', v', s^*)$. Suppose that

$$(v', \texttt{filter}(g), s|_{\{w\}}) \Longrightarrow_s X' \text{ by } \mathrm{F}(v)_{filter}.$$

Let $(A, L, \mathcal{C}, K) = s(w)$. Then, by the definition of the $\mathrm{F}(v)_{filter}$ rule,

$$((A, L, \mathcal{C}, K), (v', self, \texttt{undef})) \Longrightarrow_{t_h} (A', L', \mathcal{C}', K') \quad \text{and} \quad X'' = (A'', L'', \mathcal{C}'', K'')$$

where

$$A'' = A', \qquad L'' = L', \qquad \mathcal{C}'' = \mathcal{C}', \qquad \text{and}$$
$$\forall C_j \in \mathcal{C}', u \in \mathcal{U}.\ K''(C_j)(u) = (\text{if } (g(u) = \texttt{true}) \text{ then } K'(C_j)(u) \text{ else } \{\}),$$

Meanwhile, since S_k satisfies the claimed property by induction hypothesis, for every C_i in $\mathcal{C}$ and u in $\mathcal{U}$, every dot d in $K(C_i)(u)$ satisfies the predicate $\Phi(d, w, s)$. Then, we can claim that, for every C_j in $\mathcal{C}'$ and u in $\mathcal{U}$, every dot d'' in $K''(C_j)(u)$ satisfies the predicate $\Phi(d'', v', s^*)$. This is because

1. by the definition of the $\mathrm{F}(v)_{\mathit{filter}}$ rule, every C_j in $\mathcal{C}'$ is computed from C_i in $\mathcal{C}$ such that $C_j = \{(e, (v', t_h, \texttt{undef})) \mid (e, t_h) \in C_i\}$;

2. by the definition of the $\mathrm{F}(v)_{\mathit{filter}}$ rule, for every u in $\mathcal{U}$, every dot d'' in $K''(C_j)(u)$ is

   - $d'' = \{(e, (v', t_h, \texttt{undef})) \mid (e, t_h) \in d\}$ where $g(u) = \texttt{true}$, C_j is computed from C_i, and d is in $K(C_i)(u)$,
   - $d'' = \emptyset$ where $g(u) = \texttt{false}$;

3. when $g(u) = \texttt{false}$, $d''(= \emptyset)$ immediately satisfies the predicate $\Phi(d'', v', s^*)$;

4. when $g(u) = \texttt{true}$,

   - if we let T be the set of all dataflow paths in d and T'' be the set of all dataflow paths in d'', then

     $$T'' = \{(v', t, \texttt{undef}) \mid t \in T\},$$

   - every dataflow path in T'' is v'-oriented and compatible with P,
   - if we let t_u be the unanimous result constructed from T, then T'' is *unanimous*, and $(v', t_u, \texttt{undef})$, the unanimous result constructed from T'', is complete and v'-oriented.

The $\mathrm{F}(v)_{\cup}$ rule executes the $\cup$ operation on the $\mathrm{OR}_{\mathrm{PC}}$ sets of nodes w, w' from R, if we let $\mathit{pred}(v') = \{w, w'\}$. We will prove that, if we let $X'' = s^*(v')$, then every dot d'' in X'' satisfies the predicate $\Phi(d'', v', s^*)$. Suppose that

$$(v', \cup, s|_{\{w,w'\}}) \Longrightarrow_s X'' \text{ by } \mathrm{F}(v)_{\cup}.$$

Let $(A, L, \mathcal{C}, K) = s(w)$ and $(A', L', \mathcal{C}', K') = s(w')$. Then, by the definition of the $\mathrm{F}(v)_{\cup}$ rule,

$$((A, L, \mathcal{C}, K), (v', \mathit{self}, \texttt{undef})) \Longrightarrow_{t_h} (A_w, L_w, \mathcal{C}_w, K_w),$$
$$((A', L', \mathcal{C}', K'), (v', \texttt{undef}, \mathit{self})) \Longrightarrow_{t_h} (A_{w'}, L_{w'}, \mathcal{C}_{w'}, K_{w'}) \quad \text{and} \quad X'' = (A'', L'', \mathcal{C}'', K'')$$

where

$$A'' = A_w \cup A_{w'},$$
$$L'' = (L_w \cap L_{w'}) \cup (L_w \setminus A_{w'}) \cup (L_{w'} \setminus A_w),$$
$$\mathcal{C}'' = \mathit{max}(\{C_i \cup C_j \mid C_i \in \mathcal{C}_w \wedge C_j \in \mathcal{C}_{w'}\}),$$
$$\forall C_k \in \mathcal{C}'', u \in \mathcal{U}.\ K''(C_k)(u) = \big\{d \in (K_w(C_i)(u) \oplus K_{w'}(C_j)(u)) \mid$$
$$d \subseteq L'' \wedge (C_i, C_j) \in (\mathcal{C}_w \times \mathcal{C}_{w'}) \wedge C_k = C_i \cup C_j\big\}.$$

Meanwhile, since S_k satisfies the claimed property by induction hypothesis, for every C_i in $\mathcal{C}_w$, C_j in $\mathcal{C}_{w'}$ and u in $\mathcal{U}$, every dot d_1 in $K_w(C_i)(u)$ and every dot d_2 in $K_{w'}(C_j)(u)$ satisfy the predicate $\Phi(d_1, w, s)$ and the predicate $\Phi(d_2, w', s)$, respectively. Then, we can claim that, for every C_k in $\mathcal{C}''$ and u in $\mathcal{U}$, every dot d'' in $K''(C_j)(u)$ satisfies the predicate $\Phi(d'', v', s^*)$. This is because

1. by the definition of the $\mathrm{F}(v)_{\cup}$ rule, every C_k in $C_{A''}$ is computed from C_i in $\mathcal{C}_w$ and C_j in $\mathcal{C}_{w'}$ such that

$$C_k = \{(e, (v', t_h, \texttt{undef})) \mid (e, t_h) \in C_i\} \cup \{(e', (v', \texttt{undef}, t'_h)) \mid (e', t'_h) \in C_j\};$$

2. by the definition of the $\mathrm{F}(v)_{\cup}$ rule, for every u in $\mathcal{U}$, every dot d'' in $K''(C_k)(u)$ is computed from either d in $K(C_i)(u)$ or d' in $K(C_j)(u)$ such that

$$d'' = \begin{cases} \{(e, (v', t_h, \texttt{undef})) \mid (e, t_h) \in d_1\} & \text{if } d'' \text{ is from } d_1 \\ \{(e', (v', \texttt{undef}, t'_h)) \mid (e', t'_h) \in d_2\} & \text{if } d'' \text{ is from } d_2 \end{cases}$$

   where C_k is computed from C_i and C_j;

3. if we let T_1 be the set of all dataflow paths in d, T_2 be the set of all dataflow paths in d', and T'' be the set of all dataflow paths in d'', then

$$T'' = \begin{cases} \{(v', t_1, \texttt{undef}) \mid t_1 \in T_1\} & \text{if } d'' \text{ is from } d_1 \\ \{(v', \texttt{undef}, t_2) \mid t_2 \in T_2\} & \text{if } d'' \text{ is from } d_2 \end{cases};$$

4. every dataflow path in T'' is v'-oriented and compatible with P;

5. if we let t_{u_1} be the unanimous result of T_1 and t_{u_2} be the unanimous result of T_2, then T'' is *unanimous* in both cases, and either $(v', t_{u_1}, \texttt{undef})$ or $(v', \texttt{undef}, t_{u_2})$, the unanimous result constructed from T'', is complete and v'-oriented.

The $\mathrm{F}(v)_{\times}$ rule executes the $\times$ operation on the $\mathrm{OR}_{\mathrm{PC}}$ sets of nodes w, w' from R, if we let $pred(v') = \{w, w'\}$. We will prove that, if we let $X'' = s^*(v')$, then every dot d'' in X'' satisfies the predicate $\Phi(d'', v', s^*)$. Suppose that

$$(v', \times, s|_{\{w,w'\}}) \Longrightarrow_s X'' \text{ by } \mathrm{F}(v)_{\times}.$$

Let $(A, L, \mathcal{C}, K) = s(w)$ and $(A', L', \mathcal{C}', K') = s(w')$. Then, by the definition of the $\mathrm{F}(v)_{\times}$ rule,

$$((A, L, \mathcal{C}, K), (v, \mathit{self}, \texttt{unk})) \Longrightarrow_{t_h} (A_w, L_w, \mathcal{C}_w, K_w)$$
$$((A', L', \mathcal{C}', K'), (v, \texttt{unk}, \mathit{self})) \Longrightarrow_{t_h} (A_{w'}, L_{w'}, \mathcal{C}_{w'}, K_{w'}) \quad \text{and} \quad X'' = (A'', L'', \mathcal{C}'', K'')$$

where

$$
\begin{aligned}
&A'' = A_w \cup A_{w'},\\
&L'' = (L_w \cap L_{w'}) \cup (L_w \setminus A_{w'}) \cup (L_{w'} \setminus A_w),\\
&\mathcal{C}'' = max(\{C_i \cup C_j \mid C_i \in \mathcal{C}_w \wedge C_j \in \mathcal{C}_{w'}\}),\\
&\forall C_k \in \mathcal{C}'', u \in \mathcal{U}.\\
&\quad K''(C_k)(u) = \big\{d \in (K_w(C_i)(u_1) \otimes K_{w'}(C_j)(u_2)) \mid\\
&\qquad d \subseteq L'' \wedge (C_i, C_j) \in (\mathcal{C}_w \times \mathcal{C}_{w'}) \wedge C_k = C_i \cup C_j \wedge u = (u_1, u_2)\big\}.
\end{aligned}
$$

Meanwhile, since S_k satisfies the claimed property by induction hypothesis, for every C_i in $\mathcal{C}_w$, C_j in $\mathcal{C}_{w'}$ and u in $\mathcal{U}$, every dot d_1 in $K_w(C_i)(u)$ and every dot d_2 in $K_{w'}(C_j)(u)$ satisfy the predicate $\Phi(d_1, w, s)$ and the predicate $\Phi(d_2, w', s)$, respectively. Then, we can claim that, for every C_k in $\mathcal{C}''$ and u in $\mathcal{U}$, every dot d'' in $K''(C_j)(u)$ satisfies the predicate $\Phi(d'', v', s^*)$. This is because

1. by the definition of the $\mathrm{F}(v)_\times$ rule, every C_k in $\mathcal{C}''$ is computed from C_i in $\mathcal{C}_w$ and C_j in $\mathcal{C}_{w'}$ such that

$$C_k = \{(e, (v', t_h, \texttt{unk})) \mid (e, t_h) \in C_i\} \cup \{(e', (v', \texttt{unk}, t'_h)) \mid (e', t'_h) \in C_j\};$$

2. by the definition of the $\mathrm{F}(v)_\times$ rule, for every u in $\mathcal{U}$, every dot d'' in $K''(C_k)(u)$ is computed from d in $K(C_i)(u_1)$ and d' in $K(C_j)(u_2)$ such that

$$d'' = \{(e, (v', t_h, \texttt{unk})) \mid (e, t_h) \in d_1\} \cup \{(e', (v', \texttt{unk}, t'_h)) \mid (e', t'_h) \in d_2\}$$

   where $u = (u_1, u_2)$ and C_k is computed from C_i and C_j;

3. if we let T_1 be the set of all dataflow paths in d, T_2 be the set of all dataflow paths in d', and T'' be the set of all dataflow paths in d'', then

$$T'' = \{(v, t_1, \texttt{unk}) \mid t_1 \in T_1\} \cup \{(v, \texttt{unk}, t_2) \mid t_2 \in T_2\};$$

4. every dataflow path in T'' is v'-oriented and compatible with P;

5. if we let t_{u_1} be the unanimous result of T_1 and t_{u_2} be the unanimous result of T_2, then T'' is *unanimous*, and (v', t_{u_1}, t_{u_2}), the unanimous result constructed from T'', is complete and v'-oriented.

For the second step of the the DFlow rule, we do not prove the step for the $\sqcup$ operation here since the behaviors of the $\sqcup$ operation are identical to the ones of the $\textsc{Merge}_{dflow}$ rule that has already been proved for the transformation of system configurations.

We have completed the case analysis for the transformation of replica configurations. This means that the case analysis for the Step rule also has been completed. $\square$

Lemma 5.6.6 has proven that a set of all dataflow paths in a dot is unanimous, and it shows the meaning of the unanimous result of the set: it should consist of all the paths of some complete dataflow tree for v. Based on this information, we could calculate the set of all dataflow paths in a dot stored in $S(r).s(v)$ once we have some complete dataflow trees for v.

5.6.3 Definitions and examples for Complete-Dot Set

We will now define some utility operations for computing the contents of dots from a set of event histories in consistent dataflow executions. Our definitions will be illustrated with examples based on the dataflow program in Figure 3.1.

Let gm be the node for `GroupMedal`, gs be the node for `GroupScores`, rd be the node for `RankDivider`, ud be the node for `UserData`, gd be the node for `GroupData`, and d be the node for `Divider`. Define t_e to be the following complete dataflow tree of the node gm:

$$\begin{aligned} t_e = (gm,\ & \\ & (gs, (ud, \texttt{undef}, \texttt{undef}), (gd, \texttt{undef}, \texttt{undef})), \\ & (rd, (gd, \texttt{undef}, \texttt{undef}), (d, \texttt{undef}, \texttt{undef}))) \end{aligned}$$

Also, define the set of event histories H_e in the $\mathrm{OR}_{\mathrm{PC}}$ set of gm as follows:

$$\begin{aligned} H_e = \{&((ud, r_1, 1), (gm, (gs, (ud, \texttt{undef}, \texttt{undef}), \texttt{unk}), \texttt{unk})), \\ &((gd, r_2, 1), (gm, (gs, \texttt{unk}, (gd, \texttt{undef}, \texttt{undef})), \texttt{unk})), \\ &((gd, r_2, 1), (gm, \texttt{unk}, (rd, (gd, \texttt{undef}, \texttt{undef}), \texttt{unk}))), \\ &((gd, r_2, 2), (gm, (gs, \texttt{unk}, (gd, \texttt{undef}, \texttt{undef})), \texttt{unk})), \\ &((d, r_1, 1), (gm, \texttt{unk}, (rd, \texttt{unk}, (d, \texttt{undef}, \texttt{undef}))))\}. \end{aligned}$$

The first operation is *path_set*() that generates a unanimous set of dataflow paths from a complete dataflow tree for P (the outcome of the operation is based on Lemma 5.6.6): for each complete and P-compatible dataflow tree t,

$path_set(t) = T$

[where T is the unanimous set of P-compatible dataflow paths whose unanimous result of T is t]

For example, the outcome of $path_set(t_e)$ is as follows:

$$\begin{aligned} path_set(t_e) = \{&(gm, (gs, (ud, \texttt{undef}, \texttt{undef}), \texttt{unk}), \texttt{unk}), \\ &(gm, (gs, \texttt{unk}, (gd, \texttt{undef}, \texttt{undef})), \texttt{unk}), \\ &(gm, \texttt{unk}, (rd, (gd, \texttt{undef}, \texttt{undef}), \texttt{unk})), \\ &(gm, \texttt{unk}, (rd, \texttt{unk}, (d, \texttt{undef}, \texttt{undef})))\}. \end{aligned}$$

The *input_node_set*() operation returns a set of all input nodes included in a complete dataflow tree:

$$input_node_set(t) = \{v \mid C[(v, \texttt{undef}, \texttt{undef})] = t \text{ for some } C\}$$

$$[\ \text{where}\quad t = \text{a complete and } P\text{-compatible dataflow tree.}\ ]$$

For t_e, we have

$$input_node_set(t_e) = \{ud, gd, d\}.$$

The *inserted_event_set*() operation finds the set of all events inserted at a node v among a given set of event histories H:

$$inserted_event_set(v, H) = \{e \mid e.v = v \wedge (e, t_h) \in H \text{ for some } t_h\}.$$

There are three input nodes (ud, gd and d) in t_e, and we have

$$\begin{aligned} inserted_event_set(ud, H_e) &= \{(ud, r_1, 1)\}, \\ inserted_event_set(gd, H_e) &= \{(gd, r_2, 1), (gd, r_2, 2)\}, \\ inserted_event_set(d, H_e) &= \{(d, r_1, 1)\} \end{aligned}$$

for each input node in t_e. Let $\mathcal{P}(X)$ denote the collection of subsets of X. The next operation is:

$$inserted_event_collection(t, H) = \begin{cases} \prod\limits_{v \in input_node_set(t)} (\mathcal{P}(inserted_event_set(v, H)) \setminus \{\emptyset\}), \\ \qquad \text{if } \forall v \in input_node_set(t).\ inserted_event_set(v, H) \neq \emptyset; \\ \text{undefined}, \\ \qquad \text{otherwise}, \end{cases}$$

$$\left[\text{ where } \quad t = \text{a complete and } P\text{-compatible dataflow tree. } \right]$$

The combinations of inserted events for every input node in a complete dataflow tree among a set of event histories can be found by the *inserted_event_collection*() operation sometimes. Each tuple of inserted event sets for every input node, a member of the output value of the operation, implies an observable combination of inserted event sets in consistent executions. For t_e and H_e in our example, the power sets (except the empty set) of each inserted-event set are as follows:

$$\begin{aligned} &\mathcal{P}(inserted_event_set(ud, H_e)) \setminus \{\emptyset\} = \{\{(ud, r_1, 1)\}\}, \\ &\mathcal{P}(inserted_event_set(gd, H_e)) \setminus \{\emptyset\} = \{\{(gd, r_2, 1)\}, \{(gd, r_2, 2)\}, \{(gd, r_2, 1), (gd, r_2, 2)\}\}, \\ &\mathcal{P}(inserted_event_set(d, H_e)) \setminus \{\emptyset\} = \{\{(d, r_1, 1)\}\}. \end{aligned}$$

Thus, the outcome of the *inserted_event_collection*(t_e, H_e) operation is

$$\begin{aligned} inserted_event_collection(t_e, H_e) = \{&(\{(ud, r_1, 1)\}, \{(gd, r_2, 1)\}, \{(d, r_1, 1)\}), \\ &(\{(ud, r_1, 1)\}, \{(gd, r_2, 2)\}, \{(d, r_1, 1)\}), \\ &(\{(ud, r_1, 1)\}, \{(gd, r_2, 1), (gd, r_2, 2)\}, \{(d, r_1, 1)\})\}. \end{aligned}$$

Definition 5.6.7. The *complete-dot set collection* $\mathcal{D}_c$ for a complete dataflow tree t and a set of event histories H is defined by

$$\mathcal{D}_c = \{\{\{(e, t_h) \mid e \in E_j \wedge t_h \in \mathit{path_set}(t) \wedge C[(e.v, \mathtt{undef}, \mathtt{undef})] = t_h \text{ for some } C\} \mid E \in E_1 \times \ldots \times E_n\} \mid (E_1, \ldots, E_n) \in \mathit{inserted_event_collection}(t, H)\}$$

where $e \in E$ means that e is some component of the tuple E.

By using the $\mathit{path_set}(t)$ operation, we can retrieve a set of dataflow paths from a complete dataflow tree t. We can also retrieve the tuples of inserted event sets, the outcome of the $\mathit{inserted_event_collection}(t, H)$ operation, from a set of event histories H and a complete dataflow tree t. By combining these two types of information (a set of inserted events and a set of dataflow paths), we can construct a dot. The complete-dot set collection is a collection of those dots which are identical to the completely-computed dots through a dataflow with a complete dataflow tree t using a set of inserted events in H.

Now, we describe how a complete-dot set collection can be retrieved from t_e and H_e. The outcomes of the $\mathit{path_set}(t_e)$ operation and $\mathit{inserted_event_collection}(t_e, H_e)$ operation are already computed in the previous examples. Firstly, we use the first tuple in $\mathit{inserted_event_collection}(t_e, H_e)$: $(\{(ud, r_1, 1)\}, \{(gd, r_2, 1), (gd, r_2, 2)\}, \{(d, r_1, 1)\})$. From the tuple, we have

$$\{(ud, r_1, 1)\} \times \{(gd, r_2, 1), (gd, r_2, 2)\} \times \{(d, r_1, 1)\} = \{((ud, r_1, 1), (gd, r_2, 1), (d, r_1, 1)), ((ud, r_1, 1), (gd, r_2, 2), (d, r_1, 1))\}.$$

The first complete-dot set D_1 from the tuple $(\{(ud, r_1, 1)\}, \{(gd, r_2, 1), (gd, r_2, 2)\}, \{(d, r_1, 1)\})$ is

$$\begin{aligned} D_1 = \{\{&((ud, r_1, 1), (gm, (gs, (ud, \mathtt{undef}, \mathtt{undef}), \mathtt{unk}), \mathtt{unk})),\\ &((gd, r_2, 1), (gm, (gs, \mathtt{unk}, (gd, \mathtt{undef}, \mathtt{undef})), \mathtt{unk})),\\ &((gd, r_2, 1), (gm, \mathtt{unk}, (rd, (gd, \mathtt{undef}, \mathtt{undef}), \mathtt{unk}))),\\ &((d, r_1, 1), (gm, \mathtt{unk}, (rd, \mathtt{unk}, (d, \mathtt{undef}, \mathtt{undef}))))\},\\ \{&((ud, r_1, 1), (gm, (gs, (ud, \mathtt{undef}, \mathtt{undef}), \mathtt{unk}), \mathtt{unk})),\\ &((gd, r_2, 2), (gm, (gs, \mathtt{unk}, (gd, \mathtt{undef}, \mathtt{undef})), \mathtt{unk})),\\ &((gd, r_2, 2), (gm, \mathtt{unk}, (rd, (gd, \mathtt{undef}, \mathtt{undef}), \mathtt{unk}))),\\ &((d, r_1, 1), (gm, \mathtt{unk}, (rd, \mathtt{unk}, (d, \mathtt{undef}, \mathtt{undef}))))\}\}. \end{aligned}$$

For remaining tuples, we have

$$\begin{aligned} D_2 = \{\{&((ud, r_1, 1), (gm, (gs, (ud, \mathtt{undef}, \mathtt{undef}), \mathtt{unk}), \mathtt{unk})),\\ &((gd, r_2, 1), (gm, (gs, \mathtt{unk}, (gd, \mathtt{undef}, \mathtt{undef})), \mathtt{unk})),\\ &((gd, r_2, 1), (gm, \mathtt{unk}, (rd, (gd, \mathtt{undef}, \mathtt{undef}), \mathtt{unk}))),\\ &((d, r_1, 1), (gm, \mathtt{unk}, (rd, \mathtt{unk}, (d, \mathtt{undef}, \mathtt{undef}))))\}\}\\ &\text{(from the tuple } (\{(ud, r_1, 1)\}, \{(gd, r_2, 1)\}, \{(d, r_1, 1)\})),\\ D_3 = \{\{&((ud, r_1, 1), (gm, (gs, (ud, \mathtt{undef}, \mathtt{undef}), \mathtt{unk}), \mathtt{unk})),\\ &((gd, r_2, 2), (gm, (gs, \mathtt{unk}, (gd, \mathtt{undef}, \mathtt{undef})), \mathtt{unk})),\\ &((gd, r_2, 2), (gm, \mathtt{unk}, (rd, (gd, \mathtt{undef}, \mathtt{undef}), \mathtt{unk}))),\\ &((d, r_1, 1), (gm, \mathtt{unk}, (rd, \mathtt{unk}, (d, \mathtt{undef}, \mathtt{undef}))))\}\}\\ &\text{(from the tuple } (\{(ud, r_1, 1)\}, \{(gd, r_2, 2)\}, \{(d, r_1, 1)\})). \end{aligned}$$

Therefore, the complete-dot set collection $\mathcal{D}_c$ for t_e and H_e is

$$\mathcal{D}_c = \{D_1, D_2, D_3\}.$$

The complete-dot set D_1 for the complete dataflow tree t_e and the set of event histories H_e implies that if all input tuples for the events $(\{(ud, r_1, 1)\}, \{(gd, r_2, 1), (gd, r_2, 2)\}, \{(d, r_1, 1)\})$ in H_e are completely computed and merged at the node $\texttt{fst}(t_e)$, the node state for $\texttt{fst}(t_e)$ must have the values whose provenances contain the dots in D_1. The interesting fact relating to the complete-dot set collection is that the given event history set might not contain all event histories in its complete-dot set collection. For example, the event-history set H_e contains the event histories for the complete-dot set D_2, but not for D_1 or D_3. This is because some information related to the event $(gd, r_2, 2)$ is missing: $((gd, r_2, 2), (gm, \texttt{unk}, (rd, (gd, \texttt{undef}, \texttt{undef}), \texttt{unk})))$. This means that H_e does not have the complete information about the computation of the event $(gd, r_2, 2)$.

Definition 5.6.8. Let t be a complete dataflow tree. A complete-dot set D is *valid* for a set of event histories H if

1. D is a member of the complete-dot set collection for t and H; and
2. every dot in D is included in H, that is,

$$\forall d \in D.\ d \subseteq H.$$

We now introduce a utility operation for compute a collection of valid complete-dot sets for a node v, a complete dataflow tree t for v, and a set of event histories H in the OR_{PC} set of v as follows:

$$\mathit{valid_cds_collection}(v, t, H) = \mathcal{D}$$

[where $\mathcal{D}$ is a collection of valid complete-dot sets for v, t and H.]

From the example we described above, a collection of valid complete-dot sets for gm, t_e and H_e is:

$$\mathit{valid_cds_collection}(gm, t_e, H_e) = \{D_2\}.$$

5.6.4 Lemma for Complete-Dot Set

We now fix an S_I-starting execution τ_s for shadow system configurations such that S is $\sim$-related to the last shadow system configuration S_s of τ_s. (For the existence of such τ_s, see Lemma 5.6.2). Let $(A', L', \mathcal{C}', K')$ be the components of $S_s(r).ss(v)$ and T_v be a set of complete and v-oriented dataflow trees. Then, it is not difficult to see that for every event-history set C_j in $\mathcal{C}'$, we can compute the complete-dot set collection $\mathcal{D}_c$ for T_v and C_j. Define $\mathcal{D}$ to be the collection of valid complete-dot sets for C_j. The $\mathcal{D}$ then satisfies the following property with respect to K' and C_j.

Lemma 5.6.9 (Complete-Dot-Sets)**.** *For every dot set D in $\mathcal{D}$ and every dot d in D, there exists a value u in $\mathcal{U}$ such that $d \in K'(C_j)(u)$.*

Proof. Let τ_s be $[S_1, S_2, \ldots, S_n]$. We prove that every S_k in τ_s satisfies the following property: for every v in $G.V$ and r in $\mathcal{R}$, if we let $(A_s, A_s, \mathcal{C}_s, K_s)$ be the components of $S_k(r).ss(v)$, C_i be an element of $\mathcal{C}_s$ and $\mathcal{D}$ be the collection of valid complete-dot sets for v and C_i, then

$$\forall D \in \mathcal{D}, d \in D.\ \exists u \in \mathcal{U}.\ d \in K_s(C_i)(u).$$

Our proof is by induction on k, the index of S_k in τ_s. For the base case $k = 1$, the system configuration S_1 is S_I, and the claimed property above holds simply because, for every v in $G.V$ and r in $\mathcal{R}$, $S_I(r).ss(v) = \perp$ and so the only d in $S_I(r).ss(v)$ is the empty set $\{\}$.

We next prove the inductive case. Assume that $k \geq 1$ and S_k satisfies the claimed property. We have to show that S_{k+1} also satisfies the property.

Since τ_s is an execution, we have

$$S_k \Longrightarrow S_{k+1}.$$

We do the case analysis on the rule used for this transformation. During our case analysis, we use the predicate $\Phi(H, v, ss)$ on tuples of a set of event histories, a node and a shadow replica state. The predicate holds if

- when $(A_s, A_s, \mathcal{C}_s, K_s)$ is the components of $ss(v)$,

$$H \in \mathcal{C}_s;$$

- when $\mathcal{D}$ is the collection of valid complete-dot sets for v and H,

$$\forall D \in \mathcal{D}, d \in D.\ \exists u \in \mathcal{U}.\ d \in K_s(H)(u).$$

In our case analysis, we compute the collection of valid complete-dot sets for the output of a rule in order to check whether the predicate $\Phi()$ holds for the output or not. In particular, we compute the collection by combining the collections of valid complete-dot sets for the inputs of the rule, instead of using our original method of applying the *valid_cds_collection*() operation defined above. The benefit of this alternative method is that the collections for the inputs themselves are computed from the ones satisfying the predicate $\Phi()$ by the induction hypothesis. However, there are a couple of things to consider in order to use this alternative method, which will be described below. Before we discuss them, we need to recall how the collection of valid complete-dot sets is computed. Let v be a node, t be a complete dataflow tree for v, and H be a set of event histories in the $\mathrm{OR}_{\mathrm{PC}}$ set of v. When we compute the collection of valid complete-dot sets using the *valid_cds_collection*(v, t, H) operation, we use the following properties:

- t has a set of input nodes. Each input node generates an insertion event for every `insert`() operation.
- For each input node of t, we can compute a set of insertion events from H (the events that are generated at the node).
- For each input node, we construct the power set of this insertion-event set for the node. Then, we compute all the possible combinations of the elements of these power sets of all the input nodes.
- Each such combination gives rise to a complete-dot set. In fact, there is a one-to-one relationship between those combinations and complete dot sets.
- Among those complete-dot sets, the operation only returns the complete-dot sets that are valid for H.

We now describe the two things to consider when the alternative method (combining the collections of valid complete-dot sets for the inputs) is used for computing the collection of valid complete-dot sets for the output. First, the complete dataflow trees that are used to compute the collections for the inputs should be subtrees of the tree t for the output. It is obvious that if the complete dataflow trees t' and t'' for the inputs are not subtrees of t, the collections for the inputs computed by using those trees have no relation to the collection for the output. Second, among the members of collections for the inputs, we only use the valid complete-dot sets satisfying the following condition: the insertion-event set used to compute such a set should contain the same members for an input node whenever the node is shared by the subtrees t' and t''. The remaining valid complete-dot sets for the inputs (computed using the combinations of different insertion-event sets for such shared input nodes) are excluded because they are not valid for the output. This is because the collections for the inputs should have the same information about insertion events if those events happen at nodes in both t' and t''.

There are three rules to consider for the transformation of system configurations. Firstly, we handle the STEP rule. The STEP rule updates the replica configuration of a replica r' in $\mathcal{R}$ by executing a rule for transforming replica configurations. Suppose that

$$S_k(r') = (\alpha, ss, n) \quad \text{and} \quad S_k \Longrightarrow S_{k+1} \text{ by STEP}.$$

By the definition of the STEP rule,

$$(\alpha, ss, n) \Longrightarrow_r (\alpha', ss', n') \qquad \text{and} \qquad S_{k+1} = S_k[r' \mapsto (\alpha', ss', n')].$$

In order to prove that S_{k+1} satisfies the claimed property, we need to show that, for every node v in $G.V$, if we let $(A'_s, A'_s, \mathcal{C}'_s, K'_s)$ be the components of $ss'(v)$, then every C_j

in $\mathcal{C}'_s$ satisfies the predicate $\Phi(C_j, v, ss')$. We will do so using the induction hypothesis that, if we let $(A_s, A_s, \mathcal{C}_s, K_s)$ be the components of $ss(v)$, then every C_i in $\mathcal{C}_s$ satisfies the predicate $\Phi(C_i, v, ss)$.

Since the STEP rule is defined using the transformation of replica configurations:

$$(\alpha, ss, n) \Longrightarrow_r (\alpha', ss', n'),$$

we also do the case analysis on the rule used for this transformation. There are three rules for transforming replica configurations. We firstly handle the INPUT_{ins} rule. The rule updates the OR_{PC} set of a node v' by processing the input tuple for the insertion. We will prove that, if we let $(A'_s, A'_s, \mathcal{C}'_s, K'_s)$ be the components of $ss'(v')$, then every C_j in $\mathcal{C}'_s$ satisfies the predicate $\Phi(C_j, v', ss')$. Suppose that

$$R = ((v', \texttt{insert}, u)\alpha, ss, n) \quad \text{and} \quad R \Longrightarrow_r (\alpha', ss', n') \text{ by } \text{INPUT}_{ins}.$$

Let $(A_s, A_s, \mathcal{C}_s, K_s) = ss(v')$, $e = (v', r', n)$, and $h = (e, (v', \texttt{undef}, \texttt{undef}))$. Then, by the definition of the INPUT_{ins} rule,

$$\mathcal{C}_s = \{A_s\}, \qquad \alpha' = \alpha, \qquad n' = n + 1, \qquad \text{and} \qquad ss' = ss[v' \mapsto X'],$$

where $X' = (A_s \cup \{h\}, L_s \cup \{h\}, \{A_s \cup \{h\}\}, [A_s \cup \{h\} \mapsto K_s(A_s)[u \mapsto K_s(A_s)(u) \oplus \{\{h\}\}]])$. Meanwhile, since S_k satisfies the claimed property by induction hypothesis, if we let $(A_s, A_s, \{A_s\}, K_s)$ be the components of $ss(v')$, then A_s satisfies the predicate $\Phi(A_s, v', ss)$. Now, we can claim that, if we let $(A'_s, A'_s, \{A'_s\}, K'_s)$ be the components of X', then A'_s satisfies the predicate $\Phi(A'_s, v', ss')$. This is because, if we let $\mathcal{D}$ be the collection of valid complete-dot sets for v' and A_s, and $\mathcal{D}'$ be the collection of valid complete-dot sets for v' and A'_s, then

$$\begin{aligned}
&\mathcal{D}' = \{\{\{h\}\}\} \cup \{D \cup \{\{h\}\} \mid D \in \mathcal{D}\} \\
&\wedge K'_s(A'_s)(u) = K_s(A_s)(u) \oplus \{\{h\}\} \\
&\wedge \{h\} \in K'_s(A'_s)(u) \\
&\wedge \forall D \in \mathcal{D}, d \in D.\ \exists u' \in \mathcal{U}.\ d \in K_s(A_s)(u') \qquad (\text{from } \Phi(A_s, v', ss)) \\
\Longrightarrow& \\
&\quad \forall D' \in \mathcal{D}', d' \in D'.\ \exists u' \in \mathcal{U}.\ d' \in K'_s(A'_s)(u').
\end{aligned}$$

We secondly handle the INPUT_{del} rule. The INPUT_{del} rule for shadow states does not change the input replica configuration: $(\alpha', ss', n') = (\alpha, ss, n)$. Therefore, the following property holds for input and output replica configurations: for every v in $G.V$, if we let $(A_s, A_s, \{A_s\}, K_s)$ be the components of $ss(v)$, then A_s satisfies the predicate $\Phi(A_s, v', ss)$.

We now handle the last rule for transforming replica configurations: the DFlow rule. The rule updates the $\mathrm{OR}_{\mathrm{PC}}$ set of a node v' in $G.V$ by i) computing the dataflow operation using the $\mathrm{OR}_{\mathrm{PC}}$ sets of its previous nodes from the input replica configuration of the rule (α, s, n) and ii) combining the result of the dataflow operation with the existing $\mathrm{OR}_{\mathrm{PC}}$ set of v' from the input replica configuration. We will prove that, if we let $(A'_s, A'_s, \mathcal{C}'_s, K'_s)$ be the components of $ss'(v')$, then every C_j in $\mathcal{C}'_s$ satisfies the predicate $\Phi(C_j, v', ss')$. Suppose that

$$R = (\alpha, ss, n) \quad \text{and} \quad R \Longrightarrow_r (\alpha', ss', n') \text{ by DFlow.}$$

Let $(A_s, A_s, \mathcal{C}_s, K_s) = ss(v')$. Then, by the definition of the DFlow rule,

$$\alpha' = \alpha, \quad n' = n, \quad (v', F(v'), ss|_{pred(v')}) \Longrightarrow_s (A'_s, A'_s, \mathcal{C}'_s, K'_s), \quad \text{and} \quad ss' = ss[v' \mapsto X''],$$

where $X'' = (A_s, A_s, \mathcal{C}_s, K_s) \sqcup (A'_s, A'_s, \mathcal{C}'_s, K'_s)$. From the definition of the DFlow rule, we could consider that the DFlow rule has two steps: executing a dataflow operation through $\Longrightarrow_s$ rules, and combining the output of a $\Longrightarrow_s$ rule with the existing $\mathrm{OR}_{\mathrm{PC}}$ set of the node using the $\sqcup$ operation. We assume that, if we let R' be the output replica configuration of the DFlow rule, then there is a temporary replica configuration R^* between the two steps such that

$$R \Longrightarrow_{step} R^* \Longrightarrow_{step} R'.$$

We will prove that, if we let $R^* = (\alpha^*, ss^*, n^*)$, $(A^*_s, A^*_s, \mathcal{C}^*_s, K^*_s)$ be the components of $ss^*(v')$, $R' = (\alpha', ss', n')$ and $(A'_s, A'_s, \mathcal{C}'_s, K'_s)$ be the components of $ss'(v')$, then every C_j in $\mathcal{C}^*_s$ and every C_k in $\mathcal{C}'_s$ satisfy the predicate $\Phi(C_j, v', ss^*)$ and the predicate $\Phi(C_k, v', ss')$, respectively.

We handle the first step of the DFlow rule: the $\Longrightarrow_s$ rules. The $\mathrm{F}(v)_{map}$ rule executes the `map()` operation on the $\mathrm{OR}_{\mathrm{PC}}$ set of a node w from R, if we let $pred(v') = \{w\}$. We will prove that, if we let $X'' = ss^*(v')$ and $(A''_s, A''_s, \mathcal{C}''_s, K''_s)$ be the components of X'', then every C_k in $\mathcal{C}''_s$ satisfies the predicate $\Phi(C_k, v', ss^*)$. Suppose that

$$(v', \texttt{map}(g), ss|_{\{w\}}) \Longrightarrow_s X'' \text{ by } \mathrm{F}(v)_{map}.$$

Let $(A_s, A_s, \mathcal{C}_s, K_s) = ss(w)$. Then, by the definition of the $\mathrm{F}(v)_{map}$ rule,

$$((A_s, A_s, \mathcal{C}_s, K_s), (v, \mathit{self}, \texttt{undef})) \Longrightarrow_{t_h} (A'_s, A'_s, \mathcal{C}'_s, K'_s) \quad \text{and} \quad X'' = (A''_s, A''_s, \mathcal{C}''_s, K''_s)$$

where

$$A''_s = A'_s, \quad \mathcal{C}''_s = \mathcal{C}'_s, \quad \text{and} \quad \forall C_j \in \mathcal{C}'_s, u \in \mathcal{U}.\ K''_s(C_j)(u) = \bigoplus \{K'_s(C_j)(u') \mid u' \in g^{-1}(u)\}.$$

Meanwhile, since S_k satisfies the claimed property by induction hypothesis, every C_i in $\mathcal{C}_s$ satisfies the predicate $\Phi(C_i, w, ss)$. Then, we can claim that every C_k in $\mathcal{C}''_s$ satisfies

the predicate $\Phi(C_k, v', ss^*)$. This is because, if we let $\mathcal{D}$ be the collection of valid complete-dot sets for w and C_i, $\mathcal{D}''$ be the collection of valid complete-dot sets for v' and C_k and C_k is computed from C_i, then

$$
\begin{aligned}
& C_k = \{(e, (v', t, \texttt{undef})) \mid (e, t) \in C_i\} \\
\wedge\, & \mathcal{D}'' = \{\{\{(e, (v', t, \texttt{undef})) \mid (e, t) \in d\} \mid d \in D\} \mid D \in \mathcal{D}\} \\
\wedge\, & \forall u \in \mathcal{U}.\ K''_s(C_k)(u) = \bigoplus \{\{\{(e, (v', t, \texttt{undef})) \mid (e, t) \in d\} \mid d \in K_s(C_i)(u')\} \mid u' \in g^{-1}(u)\} \\
\wedge\, & \forall D \in \mathcal{D}, d \in D.\ \exists u' \in \mathcal{U}.\ d \in K_s(C_i)(u') \qquad (\text{from } \Phi(C_i, w, ss)) \\
\Longrightarrow\ & \\
& \forall D'' \in \mathcal{D}'', d'' \in D''.\ \exists u'' \in \mathcal{U}.\ d'' \in K''_s(C_k)(u'').
\end{aligned}
$$

The $\mathrm{F}(v)_{\mathit{filter}}$ rule only updates dataflow paths of event histories in the $\mathrm{OR}_{\mathrm{PC}}$ set of a node w from R, if we let $\mathit{pred}(v') = \{w\}$. We will prove that, if we let $X'' = ss^*(v')$ and $(A''_s, A''_s, \mathcal{C}''_s, K''_s)$ be the components of X'', then every C_k in $\mathcal{C}''_s$ satisfies the predicate $\Phi(C_k, v', ss^*)$. Suppose that

$$(v', \texttt{filter}(g), ss|_{\{w\}}) \Longrightarrow_s X'' \text{ by } \mathrm{F}(v)_{\mathit{filter}}.$$

Let $(A_s, A_s, \mathcal{C}_s, K_s) = ss(w)$. Then, by the definition of the $\mathrm{F}(v)_{\mathit{filter}}$ rule,

$$((A_s, A_s, \mathcal{C}_s, K_s), (v', \mathit{self}, \texttt{undef})) \Longrightarrow_{th} (A'_s, A'_s, \mathcal{C}'_s, K'_s) \quad \text{and} \quad X'' = (A''_s, A''_s, \mathcal{C}''_s, K''_s)$$

where

$$A''_s = A'_s, \qquad \mathcal{C}''_s = \mathcal{C}'_s, \qquad \text{and} \qquad K''_s = K'_s.$$

Meanwhile, since S_k satisfies the claimed property by induction hypothesis, every C_i in $\mathcal{C}_s$ satisfies the predicate $\Phi(C_i, w, ss)$. Then, we can claim that every C_k in $\mathcal{C}''_s$ satisfies the predicate $\Phi(C_k, v', ss^*)$. This is because, if we let $\mathcal{D}$ be the collection of valid complete-dot sets for w and C_i, $\mathcal{D}''$ be the collection of valid complete-dot sets for v' and C_k, and C_k is computed from C_i, then

$$
\begin{aligned}
& C_k = \{(e, (v', t, \texttt{undef})) \mid (e, t) \in C_i\} \\
\wedge\, & \mathcal{D}'' = \{\{\{(e, (v', t, \texttt{undef})) \mid (e, t) \in d\} \mid d \in D\} \mid D \in \mathcal{D}\} \\
\wedge\, & \forall u \in \mathcal{U}.\ K''_s(C_k)(u) = \{\{(e, (v', t, \texttt{undef})) \mid (e, t) \in d\} \mid d \in K_s(C_i)(u)\} \\
\wedge\, & \forall D \in \mathcal{D}, d \in D.\ \exists u' \in \mathcal{U}.\ d \in K_s(C_i)(u') \qquad (\text{from } \Phi(C_i, w, ss)) \\
\Longrightarrow\ & \\
& \forall D'' \in \mathcal{D}'', d'' \in D''.\ \exists u'' \in \mathcal{U}.\ d'' \in K''_s(C_k)(u'').
\end{aligned}
$$

Before we do case analysis for the rules for binary operations, such as the $\mathrm{F}(v)_{\cup}$ rule and the $\mathrm{F}(v)_{\times}$ rule, we want to discuss the following extra condition regarding input

nodes shared by complete dataflow trees for the two input parameters of the binary operations. The condition is used to ensure the validity of the collection of complete-dot sets for the results of the binary operations.

Let v' be a node for the output of the operation, C_k be a set of event histories in $ss(v')$, $\{w, w'\}$ be $pred(v')$, and C_i and C_j be sets of event histories in $ss(w)$ and $ss(w')$, respectively, such that C_k is computed from C_i and C_j. Then, the condition says the following:

> Whenever an input node v is shared by two complete dataflow trees, one for w and the other for w',
>
> $$inserted_event_set(v, C_i) = inserted_event_set(v, C_j).$$
>
> The equality here means that C_i and C_j have the same set of insertion events at v.

This condition ensures that if we let t be a complete dataflow tree for v', then

$$inserted_event_set(v', C_k) = inserted_event_set(v', C_i) = inserted_event_set(v', C_j).$$

This equality in turn implies that every complete-dot-set collection for C_k is valid; what we mean by valid will be explained in the following part of the proof.

ensures the property that if we let t be a complete dataflow tree for v', then

$$inserted_event_set(v', C_k) = inserted_event_set(v', C_i) = inserted_event_set(v', C_j).$$

This property is used crucially in our proof for the binary operations.

In this case, the collection is constructed by combining the two collections of complete-dot sets computed from C_i and C_j. The property mentioned above must be satisfied by every collection of complete-dot sets and it can be described as follows: if we let t be the complete dataflow tree for v' which is used to compute the collection, then invoking the *inserted_event_set*() operation for an input node v_i of t should return a single result. If the condition described above is not satisfied, there is a clear violation of the property of the collection of complete-dot sets. Not satisfying the condition means that two different sets of event histories (C_i and C_j) have different results of the *inserted_event_set*() operation against the shared input node ($inserted_event_set(v, C_i) \neq inserted_event_set(v, C_j)$). This implies that $inserted_event_set(v, C_k)$ leads to two different results since C_k is computed from C_i and C_j.

The $\mathrm{F}(v)_{\cup}$ rule executes the $\cup$ operation on the $\mathrm{OR}_{\mathrm{PC}}$ sets of nodes w, w' from R, if we let $pred(v') = \{w, w'\}$. We will prove that, if we let $X'' = ss^*(v')$ and $(A''_s, A''_s, \mathcal{C}''_s, K''_s)$ be the components of X'', then every C_k in $\mathcal{C}''_s$ satisfies the predicate $\Phi(C_k, v', ss^*)$. Suppose that

$$(v', \cup, ss|_{\{w,w'\}}) \Longrightarrow_s X'' \text{ by } \mathrm{F}(v)_{\cup}.$$

Let $(A_s, A_s, \mathcal{C}_s, K_s) = ss(w)$ and $(A'_s, A'_s, \mathcal{C}'_s, K'_s) = ss(w')$. Then, by the definition of the $\mathrm{F}(v)_{\cup}$ rule,

$$((A_s, A_s, \mathcal{C}_s, K_s), (v', \mathit{self}, \texttt{undef})) \Longrightarrow_{t_h} (A_w, L_w, \mathcal{C}_w, K_w),$$
$$((A'_s, A'_s, \mathcal{C}'_s, K'_s), (v', \texttt{undef}, \mathit{self})) \Longrightarrow_{t_h} (A_{w'}, L_{w'}, \mathcal{C}_{w'}, K_{w'}) \quad \text{and} \quad X'' = (A''_s, A''_s, \mathcal{C}''_s, K''_s)$$

where

$$A''_s = A_w \cup A_{w'},$$
$$\mathcal{C}''_s = max(\{C_i \cup C_j \mid C_i \in \mathcal{C}_w \wedge C_j \in \mathcal{C}_{w'}\}),$$
$$\forall C_k \in \mathcal{C}''_s, u \in \mathcal{U}.\ K''_s(C_k)(u) = \big\{d \in (K_w(C_i)(u) \oplus K_{w'}(C_j)(u)) \mid d \subseteq A''_s \wedge (C_i, C_j) \in (\mathcal{C}_w \times \mathcal{C}_{w'}) \wedge C_k = C_i \cup C_j\big\}.$$

Meanwhile, since S_k satisfies the claimed property by induction hypothesis, i) every C_i in $\mathcal{C}_s$ satisfies the predicate $\Phi(C_i, w, ss)$ and ii) every C_j in $\mathcal{C}'_s$ satisfies the predicate $\Phi(C_j, w', ss)$. We claim that every C_k in $\mathcal{C}''_s$ satisfies the predicate $\Phi(C_k, v', ss^*)$. This is because, if we let

- $\mathcal{D}$ be the collection of valid complete-dot sets for w and C_i,
- $\mathcal{D}'$ be the collection of valid complete-dot sets for w' and C_j,
- $\mathcal{D}''$ be the collection of valid complete-dot sets for v' and C_k,
- C_k is computed from C_i and C_j, and
- C_i and C_j satisfy the extra condition for the binary operations,

then

$$\begin{aligned}
& C_k = \{(e, (v, t, \texttt{undef})) \mid (e, t) \in C_i\} \cup \{(e, (v, \texttt{undef}, t)) \mid (e, t) \in C_j\} \\
\wedge\ & \mathcal{D}'' = \{\{\{(e, (v, t, \texttt{undef})) \mid (e, t) \in d\} \mid d \in D\} \mid D \in \mathcal{D}\} \\
& \qquad \cup \{\{\{(e, (v, \texttt{undef}, t)) \mid (e, t) \in d'\} \mid d' \in D'\} \mid D' \in \mathcal{D}'\} \\
\wedge\ & \forall u \in \mathcal{U}.\ K''_s(C_k)(u) = \{\{(e, (v, t, \texttt{undef})) \mid (e, t) \in d\} \mid d \in K_s(C_i)(u)\} \\
& \qquad \cup \{\{(e, (v, \texttt{undef}, t)) \mid (e, t) \in d'\} \mid d' \in K'_s(C_j)(u)\} \\
\wedge\ & \forall D \in \mathcal{D}, d \in D.\ \exists u' \in \mathcal{U}.\ d \in K_s(C_i)(u') \qquad (\text{from } \Phi(C_i, w, ss)) \\
\wedge\ & \forall D' \in \mathcal{D}', d' \in D'.\ \exists u' \in \mathcal{U}.\ d' \in K'_s(C_j)(u') \qquad (\text{from } \Phi(C_j, w', ss)) \\
\Longrightarrow\ & \\
& \forall D'' \in \mathcal{D}'', d'' \in D''.\ \exists u'' \in \mathcal{U}.\ d'' \in K''_s(C_k)(u'').
\end{aligned}$$

The $\mathrm{F}(v)_{\times}$ rule executes the $\times$ operation on the $\mathrm{OR}_{\mathrm{PC}}$ sets of nodes w, w' from R, if we let $\mathit{pred}(v') = \{w, w'\}$. We will prove that, if we let $X'' = ss^*(v')$ and $(A''_s, A''_s, \mathcal{C}''_s, K''_s)$

be the components of X'', then every C_k in $\mathcal{C}''_s$ satisfies the predicate $\Phi(C_k, v', ss^*)$. Suppose that

$$(v', \times, ss|_{\{w,w'\}}) \Longrightarrow_s X'' \text{ by } \mathrm{F}(v)_\times.$$

Let $(A_s, A_s, \mathcal{C}_s, K_s) = ss(w)$ and $(A'_s, A'_s, \mathcal{C}'_s, K'_s) = ss(w')$. Then, by the definition of the $\mathrm{F}(v)_\times$ rule,

$$((A_s, A_s, \mathcal{C}_s, K_s), (v', \mathit{self}, \mathtt{unk})) \Longrightarrow_{t_h} (A_w, L_w, \mathcal{C}_w, K_w),$$
$$((A'_s, A'_s, \mathcal{C}'_s, K'_s), (v', \mathtt{unk}, \mathit{self})) \Longrightarrow_{t_h} (A_{w'}, L_{w'}, \mathcal{C}_{w'}, K_{w'}) \quad \text{and} \quad X'' = (A''_s, A''_s, \mathcal{C}''_s, K''_s)$$

where

$$A''_s = A_w \cup A_{w'},$$
$$\mathcal{C}''_s = max(\{C_i \cup C_j \mid C_i \in \mathcal{C}_w \wedge C_j \in \mathcal{C}_{w'}\}),$$
$$\forall C_k \in \mathcal{C}''_s, u \in \mathcal{U}.\ K''_s(C_k)(u) = \big\{d \in (K_w(C_i)(u_1) \otimes K_{w'}(C_j)(u_2)) \mid d \subseteq A''_s \wedge (C_i, C_j) \in (\mathcal{C}_w \times \mathcal{C}_{w'}) \wedge C_k = C_i \cup C_j \wedge u = (u_1, u_2)\big\}.$$

Meanwhile, since S_k satisfies the claimed property by induction hypothesis, i) every C_i in $\mathcal{C}_s$ satisfies the predicate $\Phi(C_i, w, ss)$ and ii) every C_j in $\mathcal{C}'_s$ satisfies the predicate $\Phi(C_j, w', ss)$. We claim that every C_k in $\mathcal{C}''_s$ satisfies the predicate $\Phi(C_k, v', ss^*)$. This is because, if we let

- $\mathcal{D}$ be the collection of valid complete-dot sets for w and C_i,
- $\mathcal{D}'$ be the collection of valid complete-dot sets for w' and C_j,
- $\mathcal{D}''$ be the collection of valid complete-dot sets for v' and C_k,
- C_k is computed from C_i and C_j, and
- C_i and C_j satisfy the extra condition for the binary operations,

then

$$\begin{aligned}
&C_k = \{(e, (v, t, \mathtt{unk})) \mid (e, t) \in C_i\} \cup \{(e, (v, \mathtt{unk}, t)) \mid (e, t) \in C_j\} \\
&\wedge \mathcal{D}'' = \{\{\{(e, (v, t, \mathtt{unk})) \mid (e, t) \in d\} \cup \{(e', (v, \mathtt{unk}, t')) \mid (e', t') \in d'\} \mid d \in D \wedge d' \in D'\} \\
&\qquad\qquad \mid D \in \mathcal{D} \wedge D' \in \mathcal{D}'\} \\
&\wedge \forall u \in \mathcal{U}.\ K''_s(C_k)(u) = \{\{(e, (v, t, \mathtt{unk})) \mid (e, t) \in d\} \cup \{(e', (v, \mathtt{unk}, t')) \mid (e', t') \in d'\} \\
&\qquad\qquad \mid d \in K_s(C_i)(u_1) \wedge d' \in K'_s(C_j)(u_2)\} \\
&\wedge \forall D \in \mathcal{D}, d \in D.\ \exists u' \in \mathcal{U}.\ d \in K_s(C_i)(u') \qquad (\text{from } \Phi(C_i, w, ss)) \\
&\wedge \forall D' \in \mathcal{D}', d' \in D'.\ \exists u' \in \mathcal{U}.\ d' \in K'_s(C_j)(u') \qquad (\text{from } \Phi(C_j, w', ss)) \\
\Longrightarrow& \\
&\forall D'' \in \mathcal{D}'', d'' \in D''.\ \exists u'' \in \mathcal{U}.\ d'' \in K''_s(C_k)(u'')
\end{aligned}$$

where $u = (u_1, u_2)$.

For the second step of the the DFlow rule, we do not prove the step for the $\sqcup$ operation here since the behaviors of the $\sqcup$ operation are identical to the ones of the $\textsc{Merge}_{dflow}$ rule that will be proved later.

We have completed the case analysis for the transformation of replica configurations. This means that the case analysis for the Step rule also has been completed.

We now handle two remaining rules for the transformation of system configurations: $\textsc{Merge}_{input}$ and $\textsc{Merge}_{dflow}$ rules.

We firstly handle the $\textsc{Merge}_{input}$ rule. The rule updates an OR_{PC} set of a node v' from a replica r_1's state by combining it with an OR_{PC} set of a node v' from a replica r_2's state with the $\sqcup_s$ operation. We will prove that, if we let S_{k+1} be the output system configuration of the rule, $X'' = S_{k+1}(r_1).ss(v')(= S_{k+1}(r_2).ss(v'))$, and $(A''_s, A''_s, \mathcal{C}''_s, K''_s)$ be the components of X'', then every C_k in $\mathcal{C}''_s$ satisfies the predicate $\Phi(C_k, v', S_{k+1}(r_1).ss)$. Suppose that

$$ss = S_k(r_1).ss, \quad ss' = S_k(r_2).ss \quad \text{and} \quad S_k \Longrightarrow S_{k+1} \text{ by } \textsc{Merge}_{input}.$$

Let $(A_s, A_s, \mathcal{C}_s, K_s) = ss(v')$ and $(A'_s, A'_s, \mathcal{C}'_s, K'_s) = ss'(v')$. Then, by the definition of the $\textsc{Merge}_{input}$ rule,

$$\mathcal{C}_s = \{A_s\}, \quad \mathcal{C}'_s = \{A'_s\} \quad \text{and} \quad S_{k+1} = S_k[r_1.ss \mapsto ss[v' \mapsto X'']]$$

where $X'' = (A_s, A_s, \{A_s\}, K_s) \sqcup_s (A'_s, A'_s, \{A'_s\}, K'_s)$. Meanwhile, since S_k satisfies the claimed property by induction hypothesis, i) A_s satisfies the predicate $\Phi(A_s, v', ss)$ and ii) A'_s satisfies the predicate $\Phi(A'_s, v', ss)$. Then, we can claim that, if we let $(A''_s, A''_s, \{A''_s\}, K''_s)$ be the components of X'', then A''_s satisfies the predicate $\Phi(A''_s, v', S_{k+1}(r_1).ss)$. This is because, if we let

- $\mathcal{D}$ be the collection of valid complete-dot sets for v' and A_s,
- $\mathcal{D}'$ be the collection of valid complete-dot sets for v' and A'_s,
- $\mathcal{D}''$ be the collection of valid complete-dot sets for v' and A''_s,

then

$$\begin{aligned}
&A''_s = A_s \cup A'_s \\
\wedge\ &\mathcal{D}'' = \mathcal{D} \cup \mathcal{D}' \cup \{D \cup D' \mid D \in \mathcal{D} \wedge D' \in \mathcal{D}'\} \\
\wedge\ &\forall u \in \mathcal{U}.\ K''_s(A''_s)(u) = K_s(A_s)(u) \oplus K'_s(A'_s)(u) \\
\wedge\ &\forall D \in \mathcal{D}, d \in D.\ \exists u \in U.\ d \in K_s(A_s)(u) \qquad (\text{from } \Phi(A_s, v', ss)) \\
\wedge\ &\forall D' \in \mathcal{D}', d' \in D'.\ \exists u' \in U.\ d' \in K'_s(A'_s)(u') \qquad (\text{from } \Phi(A'_s, v', ss)) \\
\Longrightarrow\ &\\
&\forall D'' \in \mathcal{D}'', d'' \in D''.\ \exists u'' \in U.\ d'' \in K''_s(A''_s)(u'').
\end{aligned}$$

We now handle the MERGE_{dflow} rule. The rule updates an OR_{PC} set of a node v' from a replica r_1's state by combining it with an OR_{PC} set of a node v' from a replica r_2's state with the $\sqcup$ operation. We will prove that, if we let S_{k+1} be the output system configuration of the rule, $X'' = S_{k+1}(r_1).ss(v')(= S_{k+1}(r_2).ss(v'))$, and $(A''_s, A''_s, \mathcal{C}''_s, K''_s)$ be the components of X'', then every C_k in $\mathcal{C}''_s$ satisfies the predicate $\Phi(C_k, v', S_{k+1}(r_1).ss)$. Suppose that

$$ss = S_k(r_1).ss, \quad ss' = S_k(r_2).ss \quad \text{and} \quad S_k \Longrightarrow S_{k+1} \text{ by } \text{MERGE}_{dflow}.$$

Let $(A_s, A_s, \mathcal{C}_s, K_s) = ss(v')$ and $(A'_s, A'_s, \mathcal{C}'_s, K'_s) = ss'(v')$. Then, by the definition of the MERGE_{dflow} rule,

$$S_{k+1} = S_k[r_1.ss \mapsto ss[v' \mapsto X'']]$$

where $X'' = (A_s, A_s, \mathcal{C}_s, K_s) \sqcup (A'_s, A'_s, \mathcal{C}'_s, K'_s)$. Meanwhile, since S_k satisfies the claimed property by induction hypothesis, i) every C_i in $\mathcal{C}_s$ satisfies the predicate $\Phi(C_i, v', ss)$ and ii) every C_j in $\mathcal{C}'_s$ satisfies the predicate $\Phi(C_j, v', ss)$. Then, we can claim that, if we let $(A''_s, A''_s, \mathcal{C}'', K''_s)$ be the components of X'', then every C_k in $\mathcal{C}''_s$ satisfies the predicate $\Phi(C_k, v', S_{k+1}(r_1).ss)$. This is because, if we let

- $\mathcal{D}$ be the collection of valid complete-dot sets for v' and C_i,
- $\mathcal{D}'$ be the collection of valid complete-dot sets for v' and C_j,
- $\mathcal{D}''$ be the collection of valid complete-dot sets for v' and C_k, and
- C_k is either C_i or C_j,

then

$$\mathcal{D}'' = \begin{cases} \mathcal{D} & \text{if } C_k = C_i \wedge \neg\exists C_j \in \mathcal{C}'_s.\ C_i \subset C_j \\ \mathcal{D}' & \text{if } C_k = C_j \wedge \neg\exists C_i \in \mathcal{C}_s.\ C_j \subseteq C_i \end{cases}$$

$$\wedge\, K''_s(C_k)(u) = \begin{cases} K_s(C_i)(u) \cup (\bigcup\{K'_s(C_j)(u) \mid C_j \in \mathcal{C}'_s \wedge C_j \subseteq C_i\}) \\ \qquad \text{if } C_k = C_i \wedge \neg\exists C_j \in \mathcal{C}'_s.\ C_i \subset C_j \\ (\bigcup\{K_s(C_i)(u) \mid C_i \in \mathcal{C}_s \wedge C_i \subseteq C_j\}) \cup K'_s(C_j)(u) \\ \qquad \text{if } C_k = C_j \wedge \neg\exists C_i \in \mathcal{C}_s.\ C_j \subseteq C_i \end{cases}$$

$\wedge\, \forall D \in \mathcal{D}, d \in D.\ \exists u \in U.\ d \in K_s(C_i)(u)$ where $\mathcal{D}$ is computed for v and C_i in $\mathcal{C}_s$

$\wedge\, \forall D' \in \mathcal{D}', d' \in D'.\ \exists u' \in U.\ d' \in K'_s(C_j)(u')$ where $\mathcal{D}'$ is computed for v and C_j in $\mathcal{C}'_s$

$\Longrightarrow$

$\forall D'' \in \mathcal{D}'', d'' \in D''.\ \exists u'' \in U.\ d'' \in K''_s(C_k)(u'')$

where $\mathcal{D}''$ is computed for v and C_k in $\mathcal{C}''_s$, and C_k is either C_i or C_j.

□

Lemma 5.6.9 says that retrieving the collection of valid complete-dot sets from a node v and a subset C_j in the $\mathcal{C}'$ component guarantees the existence of values with dots from the collection in $K'(C_j)$ on every shadow node state for v.

5.6.5 Properties between Executions

This subsection explores some properties that are related between two different executions, which include executions with normal and shadow states or the execution with shadow states and the complete execution.

The following lemma shows the property of events stored in two different executions:

Lemma 5.6.10. *For every input tuple for insertion $(v, \texttt{insert}, u)$ in $S_I(r).\alpha$, events e', e'', and S_I-starting executions τ', τ'', if the tuple generates e' in τ' and e'' in τ'' through the* $\textsc{Input}_{ins}$ *rule performed on the replica r, then $e' = e''$.*

Proof. Let $(v, \texttt{insert}, u)$ be an input tuple in $S_I(r).\alpha$, and τ', τ'' be S_I-starting executions. Also, let e', e'' be the events generated by $(v, \texttt{insert}, u)$ in τ', τ'' respectively. For the sake of contradiction, suppose that $e' \neq e''$, that is, $e' = (v, r, k')$, $e'' = (v, r, k'')$ and $k' \neq k''$ for some k' and k''. By the definition of the $\textsc{Input}_{ins}$ rule, (v, r, k') is generated by the k'th insertion of $S_I(r).\alpha$, and (v, r, k'') is generated by the k''th insertion of $S_I(r).\alpha$. The former means that $(v, \texttt{insert}, u)$ is the k'th input tuple in $S_I(r).\alpha$, but the latter means that the same tuple is the k''th input tuple in the same $S_I(r).\alpha$. Since $k' \neq k''$, this cannot happen, which gives the desired contradiction. □

Based on Lemma 5.6.10, we can easily test the equality of two dots in different executions whenever those execution start from the same system configuration.

Recall that we fixed a complete dataflow tree t for v and a shadow system configuration S_s for S satisfying $S \sim S_s$ and $(A', L', \mathcal{C}', K')$ for $S_s(r).ss(v)$. Let $(A, L, \mathcal{C}, K)$ be the components of $S(r).s(v)$ and (C_i', D', U) be the outcome of the $\texttt{read}_c(C_i, D, (A, L, \mathcal{C}, K))$ operation.

Lemma 5.6.11 (A Complete-Dot-Set For The Consistent Read)**.** *For every complete-dot set D' in the outcome of the $\texttt{read}_c()$ operation, there exist $C_j \in \mathcal{C}'$ and a valid complete-dot set collection for v and C_j (in a corresponding shadow system configuration S_s) such that*

$$D' \in \mathit{valid_cds_collection}(v, t, C_j).$$

Proof. By the definition of the $\sim$ relation, we have

$$\mathcal{C} = \mathcal{C}'.$$

From the definition of the $\texttt{read}_c()$ operation, we also have

$$D' \in \mathit{valid_cds_collection}(v, t, C_i)$$

where C_i is the first parameter of the operation and C_i is in $\mathcal{C}$. It is immediate that there exists C_j in $\mathcal{C}'$ such that $C_j = C_i$. For this C_j, we have

$$\mathit{valid_cds_collection}(v, t, C_i) = \mathit{valid_cds_collection}(v, t, C_j).$$

□

Definition 5.6.12. The *computation tree* b for a value-dot pair (u, d) is one of the following inductively-defined tuples:

$$((u, d), \texttt{undef}, \texttt{undef}) \mid$$
$$((u, d), b', \texttt{undef}) \mid ((u, d), \texttt{undef}, b') \mid$$
$$((u, d), b', b'')$$

where b' and b'' themselves are inductively-defined computation trees for some value-dot pairs. We define a special computation tree $\perp$.

Recall that S_s is the last shadow system configuration of τ_s. The computation tree b for a value-dot pair (u, d) in $S_s(r).ss(v)$ can be computed as follows:

1. If $S_s = S_I$, i.e, $\tau_s = [S_I]$, then we return $\perp$ for any (u, d).

2. If v is an input node ($pred(v) = \emptyset$), then we return $((u, d), \texttt{undef}, \texttt{undef})$.

3. If v is not an input node and the node state transition function $F(v)$ for v is either `map()` or `filter()`, then we return $((u, d), b', \texttt{undef})$, where b' is computed recursively for the value-dot pair (u', d') in the previous system configuration of S_s in τ_s that executes either the $\mathrm{F}(v)_{map}$ rule or the $\mathrm{F}(v)_{filter}$ rule and results in (u, d).

4. If v is not an input node and the node state transition function $F(v)$ for v is $\cup$, then we return either $((u, d), b', \texttt{undef})$ or $((u, d), \texttt{undef}, b')$ depending on whether (u, d) comes from the left argument of $\cup$ or the right argument. Here b' is computed recursively for the value-dot pair (u', d') in the previous system configuration of S_s in τ_s that executes the $\mathrm{F}(v)_{\cup}$ rule and results in (u, d).

5. If v is not an input node and the node state transition function $F(v)$ for v is $\times$, then we return $((u, d), b', b'')$, where b' and b'' are computed recursively for the value-dot pairs (u', d') and (u'', d'') respectively in the previous system configuration of S_s in τ_s that executes the $\mathrm{F}(v)_{\times}$ rule and results in (u, d).

The computation tree of a value-dot pair implies that the value-dot pair can inductively be computed by the program of the $\textsc{Input}_{ins}$ rule or the DFlow rule on some other similarly-constructed value-dot pairs from its sub-tree tuples.

For a shadow execution τ'_s, its last configuration S'_s, a replica id r' and a node v', we define an operation for computing the computation tree for each value-dot pair (u, d) in $S'_s(r').ss(v')$:

$$b = computation_tree((u, d), \tau'_s),$$

which computes b in the same way as described by the five steps from the above.

Recall the notion of a consistent execution defined in Definition 5.4.1. In a consistent execution, replicas fully process all input streams first, and then execute the rules related to dataflow operations on each replica without communicating one another.

Let $(_, _, \mathcal{C}', K')$ be the components of $S_s(r).ss(v)$, C_j be an element of $\mathcal{C}'$, and $\mathcal{D}$ be the collection of valid complete-dot sets for v and C_j. Define a set of values U_c by

$$U_c = \{u \mid \forall D \in \mathcal{D}.\ K'(C_j)(u) \cap D \neq \emptyset\}.$$

We now consider the property of value-dot pairs (u, d) such that $u \in U_c$ and $d \in K'(C_j)(u) \cap D$ for some $D \in \mathcal{D}$. For such a pair (u, d) and its computation tree b, the next lemma says that there exists an S_I-starting consistent execution τ' with the following property:

Value-Dot If S_F is the last system configuration of τ', then, for every r' in $\mathcal{R}$, the pair (u, d) is in $S_F(r').ss(v)$, and has b as its computation tree under τ'.

Lemma 5.6.13 (Strong Consistency in Weakly Consistent Distributed Systems)**.** *For every value-dot pair (u, d) in $U_c \times (K'(C_j)(u) \cap D)$ and its computation tree b, there exists an S_I-starting consistent execution τ' that satisfies the* ***Value-Dot*** *property.*

Proof. Let T_s be a sequence $[\tau_1, \tau_2, \ldots, \tau_n]$ of S_I-starting executions for shadow system configurations such that

1. $\tau_1 = [S_1]$ and $S_1 = S_I$;
2. $\tau_{k+1} = \tau_k^\frown [S_{k+1}]$ for every $k \in \{1, \ldots, n-1\}$;
3. S_n, the last system configuration of τ_n, is identical to S_s in the lemma.

We prove that every τ_k in T_s satisfies the following property: for every v in $G.V$ and r in $\mathcal{R}$, if we let

- $(A_s, A_s, \mathcal{C}_s, K_s)$ be the components of $S_k(r).ss(v)$,
- C_i be an element of $\mathcal{C}_s$,
- $\mathcal{D}$ be the collection of valid complete-dot sets for v and C_i,
- D be a valid complete-dot set in $\mathcal{D}$, and
- U_c be a set of values such that $K_s(C_i)(u) \cap D \neq \emptyset$ for every $u \in U_c$,

then for every value-dot pair (u, d) in $U_c \times (K_s(C_i)(u) \cap D)$ and its computation tree b, there exists an S_I-starting consistent execution τ'_k that satisfies the **Value-Dot** property.

Our proof is by induction on k, the index of τ_k in T_s. For the base case $k = 1$, the last and only system configuration of τ_1 is S_I and for any value-dot pair $(u, \{\})$ in $S_I(r).ss(v)$, we have

$$\mathit{computation_tree}((u, \{\}), \tau_1) = \bot.$$

We set τ' to be $[S_I]$. This τ' is an S_I-starting consistent execution for τ_1, as required. It is also immediate that, for every r' in $\mathcal{R}$, $(u, \{\})$ is in $S_I(r').ss(v)$. Furthermore, we have

$$\mathit{computation_tree}((u, \{\}), \tau') = \bot.$$

We next prove the inductive case. Assume that $k \geq 1$ and τ_k satisfies the claimed property. We have to show that τ_{k+1} also satisfies the property.

Since S_k and S_{k+1} are the last system configurations of τ_k and τ_{k+1} respectively, we have

$$S_k \Longrightarrow S_{k+1}.$$

We do the case analysis on the rule used for this transformation. The proof of each case consists of i) identifying what is changed by the rule for the transformation, ii) constructing an S_I-starting consistent execution for S_{k+1} by using the execution for S_k, and iii) showing that the constructed consistent execution satisfies the **Value-Dot** property.

For example, if S_k is transformed by the Step rule, and S_{k+1} is the output system configuration of the transformation, then the Step rule updates the configuration of some replica r' in S_k by a rule for replica transformation ($\Longrightarrow_r$). This means that S_{k+1} is of the form $S_k[r' \mapsto R']$ for some R', where $S_k(r') \Longrightarrow_r R'$. The Input_{ins} rule is one of the rules for $\Longrightarrow_r$. If this rule is used, $S_k(r')$ has the form $((v', \texttt{insert}, u_i)\alpha, ss, n)$ for some v', u_i, α, ss, n, and R' is $(\alpha, ss', n + 1)$ for some ss'. In this case, we first note that the Input_{ins} rule updates the OR_{PC} set of v' in $S_k(r').ss$. This means that $ss' = ss[v' \mapsto X']$ for some X', where X' is built by incorporating $(v', \texttt{insert}, u_i)$ to $ss(v')$. Next, if we let S_F be the last system configuration of the S_I-starting consistent execution τ_k for S_k, then we can construct a required execution τ'_k for S_{k+1} simply by setting $\tau'_k = \tau_k$. The reason that τ_k can be used as the execution for S_{k+1} is that all input tuples have already been processed in τ_k including $(v', \texttt{insert}, u_i)$ by the definition of a consistent execution. Finally, we show that τ'_k satisfies the **Value-Dot** property.

There are three rules to consider. Firstly, we handle the Step rule. Suppose that

$$S_k(r') = (\alpha, ss, n) \quad \text{and} \quad S_k \Longrightarrow S_{k+1} \text{ by Step}.$$

By the definition of STEP,

$$(\alpha, ss, n) \Longrightarrow_r (\alpha', ss', n') \qquad \text{and} \qquad S_{k+1} = S_k[r' \mapsto (\alpha', ss', n')].$$

In order to prove that τ_{k+1} satisfies the claimed property, we only consider what are related to the rule: the replica states for r'. Other unchanged replica states in S_{k+1} of τ_{k+1} satisfy the claimed property by induction hypothesis. Therefore, we need to show that, for every v in $G.V$, if we let

- $(A'_s, A'_s, \mathcal{C}'_s, K'_s)$ be the components of $ss'(v)$,
- C_j be an element of $\mathcal{C}'_s$,
- $\mathcal{D}'$ be the collection of valid complete-dot sets for v and C_j,
- D' be a valid complete-dot set in $\mathcal{D}'$, and
- U'_c be a set of values such that $K'_s(C_j)(u') \cap D' \neq \emptyset$ for every $u' \in U'_c$,

then for every value-dot pair (u', d') in $U'_c \times (K'_s(C_j)(u') \cap D')$ and its computation tree b', there exists an S_I-starting consistent execution τ'_{k+1} that satisfies the **Value-Dot** property. By induction hypothesis, for every v in $G.V$, if we let

- $(A_s, A_s, \mathcal{C}_s, K_s)$ be the components of $ss(v)$,
- C_i be an element of $\mathcal{C}_s$,
- $\mathcal{D}$ be the collection of valid complete-dot sets for v and C_i,
- D be a valid complete-dot set in $\mathcal{D}$, and
- U_c be a set of values such that $K_s(C_i)(u) \cap D \neq \emptyset$ for every $u \in U_c$,

then for every value-dot pair (u, d) in $U_c \times (K_s(C_i)(u) \cap D)$ and its computation tree b, there exists an S_I-starting consistent execution τ'_k that satisfies the **Value-Dot** property.

Since we need to prove the existence of an S_I-starting consistent execution satisfying the **Value-Dot** property for the following transformation

$$(\alpha, ss, n) \Longrightarrow_r (\alpha', ss', n'),$$

we also do the case analysis on the rule used for this transformation. There are three rules for transforming replica configurations. We firstly handle the INPUT_{ins} rule. Suppose that

$$R = ((v', \texttt{insert}, u_i)\alpha, ss, n) \quad \text{and} \quad R \Longrightarrow_r (\alpha', ss', n') \text{ by } \text{INPUT}_{ins}.$$

Let $(A_s, A_s, \mathcal{C}_s, K_s) = ss(v')$, $e = (v', r', n)$, and $h = (e, (v', \texttt{undef}, \texttt{undef}))$. Then, by the definition of $\textsc{Input}_{ins}$,

$$\mathcal{C}_s = \{A_s\}, \qquad \alpha' = \alpha, \qquad n' = n + 1, \qquad \text{and} \qquad ss' = ss[v' \mapsto X'],$$

where $X' = (A_s \cup \{h\}, L_s \cup \{h\}, \{A_s \cup \{h\}\}, [A_s \cup \{h\} \mapsto K_s(A_s)[u_i \mapsto K_s(A_s)(u_i) \oplus \{\{h\}\}]])$. Similar to the transformation of system configurations, we only consider what are related to the rule: $ss(v')$ and $X'(= ss'(v'))$. Meanwhile, since τ_k satisfies the claimed property by induction hypothesis, if we let

- $\mathcal{D}$ be the collection of valid complete-dot sets for v' and A_s,
- D be a valid complete-dot set in $\mathcal{D}$, and
- U_c be a set of values such that $K_s(A_s)(u) \cap D \neq \emptyset$ for every $u \in U_c$,

then for every value-dot pair (u, d) in $U_c \times (K_s(A_s)(u) \cap D)$ and its computation tree b, there exists an S_I-starting consistent execution τ'_k that satisfies the **Value-Dot** property. Now, we claim that, if we let

- $(A'_s, A'_s, \{A'_s\}, K'_s)$ be the components of X',
- $\mathcal{D}'$ be the collection of valid complete-dot sets for v' and A'_s,
- D' be a valid complete-dot set in $\mathcal{D}'$, and
- U'_c be a set of values such that $K'_s(A'_s)(u') \cap D' \neq \emptyset$ for every $u' \in U'_c$,

then for every value-dot pair (u', d') in $U'_c \times (K'_s(A'_s)(u') \cap D')$ and its computation tree b', there exists an S_I-starting consistent execution τ'_{k+1} that satisfies the **Value-Dot** property. This is because

$$\begin{aligned}
&\mathcal{D}' = \{\{\{h\}\}\} \cup \{D \cup \{\{h\}\} \mid D \in \mathcal{D}\} \\
&\wedge U'_c = U_c \cup \{u_i\} \\
&\wedge U'_c \times (K'_s(A'_s)(u') \cap D') = \big(U_c \times (K_s(A_s)(u) \cap D)\big) \cup \{(u_i, \{h\})\} \\
&\wedge \tau_{k+1} = \tau_k^\frown[S_{k+1}] \\
&\wedge \mathit{computation_tree}((u_i, \{h\}), \tau_{k+1}) = ((u_i, \{h\}), \texttt{undef}, \texttt{undef}) \\
&\wedge \tau'_{k+1} = \tau'_k \\
\Longrightarrow& \\
&\forall r'' \in \mathcal{R}.\, (u_i, \{h\}) \in S'_F(r'').s(v') \text{ where } S'_F \text{ is the last system configuration of } \tau'_{k+1} \\
&\wedge \mathit{computation_tree}((u_i, \{h\}), \tau'_{k+1}) = ((u_i, \{h\}), \texttt{undef}, \texttt{undef}).
\end{aligned}$$

We secondly handle the $\textsc{Input}_{del}$ rule. The $\textsc{Input}_{del}$ rule for shadow states does not change the state of the input of the rule: $ss' = ss$. Therefore, the following property holds for the output of the rule ss': for every v in $G.V$, if we let

- $(A'_s, A'_s, \{A'_s\}, K'_s)$ be the components of $ss'(v)$,
- $\mathcal{D}'$ be the collection of valid complete-dot sets for v and A'_s,
- D' be a valid complete-dot set in $\mathcal{D}'$, and
- U'_c be a set of values such that $K'_s(A'_s)(u') \cap D' \neq \emptyset$ for every $u' \in U'_c$,

then for every value-dot pair (u', d') in $U'_c \times (K'_s(A'_s)(u') \cap D')$ and its computation tree b', there exists an S_I-starting consistent execution τ'_{k+1} that satisfies the **Value-Dot** property since ss is from τ_k that satisfies the claimed property by induction hypothesis.

We now handle the last rule for transforming replica configurations: the DFlow rule. Suppose that

$$R = (\alpha, ss, n) \quad \text{and} \quad R \Longrightarrow_r (\alpha', ss', n') \text{ by DFlow.}$$

Let $(A_s, A_s, \mathcal{C}_s, K_s) = ss(v')$. Then, by the definition of DFlow,

$$\alpha' = \alpha, \quad n' = n, \quad (v', F(v), ss|_{pred(v')}) \Longrightarrow_s (A'_s, A'_s, \mathcal{C}'_s, K'_s), \quad \text{and} \quad ss' = ss[v' \mapsto X''],$$

where $X'' = (A_s, A_s, \mathcal{C}_s, K_s) \sqcup (A'_s, A'_s, \mathcal{C}'_s, K'_s)$. Similar to the transformation of system configurations, we only consider what are related to the rule: $ss|_{pred(v')}$, $ss(v')$ and $ss'(v')$. From the definition of the DFlow rule, we could consider that the DFlow rule has two steps: i) executing a dataflow operation through $\Longrightarrow_s$ rules, and ii) combining the outcome of a $\Longrightarrow_s$ rule with the existing node state using the operation $\sqcup$. We assume that there is a temporary execution $\tau_{k+1/2}$ between the two steps, if we let S_k be the last system configuration of τ_k, such that

$$S_k \Longrightarrow_{step} S_{k+1/2} \Longrightarrow_{step} S_{k+1}.$$

We will prove that the outputs of the two steps satisfy the following property: if we let

- $(A'_s, A'_s, \mathcal{C}'_s, K'_s)$ be the components of either $S_{k+1/2}(r').ss(v')$ or $S_{k+1}(r').ss(v')$,
- C_j be an element of $\mathcal{C}'_s$,
- $\mathcal{D}'$ be the collection of valid complete-dot sets for v' and C_j,
- D' be a valid complete-dot set in $\mathcal{D}'$, and
- U'_c be a set of values such that $K'_s(C_j)(u') \cap D' \neq \emptyset$ for every $u' \in U'_c$,

then for every value-dot pair (u', d') in $U'_c \times (K'_s(C_j)(u') \cap D')$ and its computation tree b', there exists an S_I-starting consistent execution τ'_{k+1} that satisfies the **Value-Dot** property. We will also do so for the inputs of the two steps using the induction hypothesis that, if we let

- $(A_s, A_s, \mathcal{C}_s, K_s)$ be the components of either $S_k(r').ss(v')$ or $S_{k+1/2}(r').ss(v')$,
- C_i be an element of $\mathcal{C}_s$,
- $\mathcal{D}$ be the collection of valid complete-dot sets for v' and C_i,
- D be a valid complete-dot set in $\mathcal{D}$, and
- U_c be a set of values such that $K_s(C_i)(u) \cap D \neq \emptyset$ for every $u \in U_c$,

then for every value-dot pair (u, d) in $U_c \times (K_s(C_i)(u) \cap D)$ and its computation tree b, there exists an S_I-starting consistent execution τ'_k that satisfies the **Value-Dot** property.

We handle the first step of the DFlow rule: $\Longrightarrow_s$. For the $\mathrm{F}(v)_{map}$ rule, suppose that

$$pred(v') = \{w\} \quad \text{and} \quad (v', \texttt{map}(g), ss|_{pred(v')}) \Longrightarrow_s X'' \text{ by } \mathrm{F}(v)_{map}.$$

Let $(A_s, A_s, \mathcal{C}_s, K_s) = ss(w)$. Then, by the definition of $\mathrm{F}(v)_{map}$,

$$((A_s, A_s, \mathcal{C}_s, K_s), (v', \mathit{self}, \texttt{undef})) \Longrightarrow_{t_h} (A'_s, A'_s, \mathcal{C}'_s, K'_s) \quad \text{and} \quad X'' = (A''_s, A''_s, \mathcal{C}''_s, K''_s)$$

where

$$A''_s = A'_s, \quad \mathcal{C}''_s = \mathcal{C}'_s, \quad \text{and} \quad \forall C_j \in \mathcal{C}'_s, u \in \mathcal{U}.\ K''_s(C_j)(u) = \bigoplus \{K'_s(C_j)(u') \mid u' \in g^{-1}(u)\}.$$

Similar to other transformation, we only consider what are related to the $\mathrm{F}(v)_{map}$ rule: $ss(w)$ and $X''(= S_{k+1/2}(r').ss(v'))$. Meanwhile, since $ss(w)$ is from τ_k that satisfies the claimed property by induction hypothesis, if we let

- C_i be an element of $\mathcal{C}_s$,
- $\mathcal{D}$ be the collection of valid complete-dot sets for w and C_i,
- D be a valid complete-dot set in $\mathcal{D}$, and
- U_c be a set of values such that $K_s(C_i)(u) \cap D \neq \emptyset$ for every $u \in U_c$,

then for every value-dot pair (u, d) in $U_c \times (K_s(C_i)(u) \cap D)$ and its computation tree b, there exists an S_I-starting consistent execution τ'_k that satisfies the **Value-Dot** property. Now, we can claim that, if we let

- C_k be an element of $\mathcal{C}''_s$,
- $\mathcal{D}''$ be the collection of valid complete-dot sets for v' and C_k,
- D'' be a valid complete-dot set in $\mathcal{D}''$, and
- U''_c be a set of values such that $K''_s(C_k)(u'') \cap D'' \neq \emptyset$ for every $u'' \in U''_c$,

then for every value-dot pair (u'', d'') in $U''_c \times (K''_s(C_k)(u'') \cap D'')$ and its computation tree b'', there exists an S_I-starting completed consistent execution τ''_{k+1} that satisfies the **Value-Dot** property. This is because

$$
\begin{aligned}
&\mathcal{D}'' = \{\{\{(e, (v', t, \texttt{undef})) \mid (e,t) \in d\} \mid d \in D\} \mid D \in \mathcal{D}\} \\
&\wedge U''_c = \{g(u) \mid u \in U_c\} \\
&\wedge C_k = \{(e, (v', t, \texttt{undef})) \mid (e,t) \in C_i\} \\
&\wedge K''_s(C_k)(u) = \bigoplus \{\{\{(e, (v', t, \texttt{undef})) \mid (e,t) \in d\} \mid d \in K_s(C_i)(u')\} \mid u' \in g^{-1}(u)\} \\
&\wedge \tau_{k+1} = \tau_k^\frown[S_{k+1}] \\
&\wedge \mathit{computation_tree}((g(u), \{(e, (v', t, \texttt{undef})) \mid (e,t) \in d\}), \tau_{k+1}) \\
&\qquad = ((g(u), \{(e, (v', t, \texttt{undef})) \mid (e,t) \in d\}), \mathit{computation_tree}((u,d), \tau_k), \texttt{undef}) \\
&\wedge \tau''_{k+1} = \text{processing the } \Longrightarrow_{\mathit{dflow}} \text{ rules including the } \mathrm{F}(v)_{\mathit{map}} \text{ rule from } \tau'_k \\
\Longrightarrow& \\
&\quad (g(u), \{(e, (v', t, \texttt{undef})) \mid (e,t) \in d\}) \in S''_F(r').s(v') \\
&\wedge \mathit{computation_tree}((g(u), \{(e, (v', t, \texttt{undef})) \mid (e,t) \in d\}), \tau''_{k+1}) \\
&\qquad = ((g(u), \{(e, (v', t, \texttt{undef})) \mid (e,t) \in d\}), \mathit{computation_tree}((u,d), \tau'_k), \texttt{undef}).
\end{aligned}
$$

For the $\mathrm{F}(v)_{\mathit{filter}}$ rule, suppose that

$$\mathit{pred}(v') = \{w\} \quad \text{and} \quad (v', \texttt{filter}(g), ss|_{\mathit{pred}(v')}) \Longrightarrow_s X'' \text{ by } \mathrm{F}(v)_{\mathit{filter}}.$$

Let $(A_s, A_s, \mathcal{C}_s, K_s) = ss(w)$. Then, by the definition of $\mathrm{F}(v)_{\mathit{filter}}$,

$$((A_s, A_s, \mathcal{C}_s, K_s), (v', \mathit{self}, \texttt{undef})) \Longrightarrow_{t_h} (A'_s, A'_s, \mathcal{C}'_s, K'_s) \quad \text{and} \quad X'' = (A''_s, A''_s, \mathcal{C}''_s, K''_s)$$

where

$$A''_s = A'_s, \qquad \mathcal{C}''_s = \mathcal{C}'_s, \qquad \text{and} \qquad K''_s = K'_s.$$

Similar to other transformation, we only consider what are related to the $\mathrm{F}(v)_{\mathit{filter}}$ rule: $ss(w)$ and $X''(= S_{k+1/2}(r').ss(v'))$. Meanwhile, $ss(w)$ is from τ_k that satisfies the claimed property by induction hypothesis, if we let

- C_i be an element of $\mathcal{C}_s$,
- $\mathcal{D}$ be the collection of valid complete-dot sets for w and C_i,
- D be a valid complete-dot set in $\mathcal{D}$, and
- U_c be a set of values such that $K_s(C_i)(u) \cap D \neq \emptyset$ for every $u \in U_c$,

then for every value-dot pair (u, d) in $U_c \times (K_s(C_i)(u) \cap D)$ and its computation tree b, there exists an S_I-starting consistent execution τ'_k that satisfies the **Value-Dot** property. Now, we can claim that, if we let

- C_k be an element of $\mathcal{C}''_s$,
- $\mathcal{D}''$ be the collection of valid complete-dot sets for v' and C_k,
- D'' be a valid complete-dot set in $\mathcal{D}''$, and
- U''_c be a set of values such that $K''_s(C_k)(u'') \cap D'' \neq \emptyset$ for every $u'' \in U''_c$,

then for every value-dot pair (u'', d'') in $U''_c \times (K''_s(C_k)(u'') \cap D'')$ and its computation tree b'', there exists an S_I-starting consistent execution τ''_{k+1} that satisfies the **Value-Dot** property. This is because

$$
\begin{aligned}
&\mathcal{D}'' = \{\{\{(e, (v', t, \texttt{undef})) \mid (e, t) \in d\} \mid d \in D\} \mid D \in \mathcal{D}\} \\
&\wedge U''_c = U_c \\
&\wedge C_k = \{(e, (v', t, \texttt{undef})) \mid (e, t) \in C_i\} \\
&\wedge K''_s(C_k)(u) = \{\{(e, (v', t, \texttt{undef})) \mid (e, t) \in d\} \mid d \in K_s(C_i)(u)\} \\
&\wedge \tau_{k+1} = \tau_k^\frown[S_{k+1}] \\
&\wedge \mathit{computation_tree}((u, \{(e, (v', t, \texttt{undef})) \mid (e, t) \in d\}), \tau_{k+1}) \\
&\qquad = ((u, \{(e, (v', t, \texttt{undef})) \mid (e, t) \in d\}), \mathit{computation_tree}((u, d), \tau_k), \texttt{undef}) \\
&\wedge \tau''_{k+1} = \text{processing the } \Longrightarrow_{\mathit{dflow}} \text{ rules with the } \mathrm{F}(v)_{\mathit{filter}} \text{ rule from } \tau'_k \\
\Longrightarrow \\
&(u, \{(e, (v', t, \texttt{undef})) \mid (e, t) \in d\}) \in S''_F(r').s(v') \\
&\wedge \mathit{computation_tree}((u, \{(e, (v', t, \texttt{undef})) \mid (e, t) \in d\}), \tau''_{k+1}) \\
&\qquad = ((u, \{(e, (v', t, \texttt{undef})) \mid (e, t) \in d\}), \mathit{computation_tree}((u, d), \tau'_k), \texttt{undef}).
\end{aligned}
$$

For the $\mathrm{F}(v)_\cup$ rule, suppose that

$$\mathit{pred}(v') = \{w, w'\} \quad \text{and} \quad (v', \cup, ss|_{\mathit{pred}(v')}) \Longrightarrow_s X'' \text{ by } \mathrm{F}(v)_\cup.$$

Let $(A_s, A_s, \mathcal{C}_s, K_s) = ss(w)$ and $(A'_s, A'_s, \mathcal{C}'_s, K'_s) = ss(w')$. Then, by the definition of $\mathrm{F}(v)_\cup$,

$$
\begin{aligned}
&((A_s, A_s, \mathcal{C}_s, K_s), (v', \mathit{self}, \texttt{undef})) \Longrightarrow_{t_h} (A_w, L_w, \mathcal{C}_w, K_w), \\
&((A'_s, A'_s, \mathcal{C}'_s, K'_s), (v', \texttt{undef}, \mathit{self})) \Longrightarrow_{t_h} (A_{w'}, L_{w'}, \mathcal{C}_{w'}, K_{w'}) \quad \text{and} \quad X'' = (A''_s, A''_s, \mathcal{C}''_s, K''_s)
\end{aligned}
$$

where

$$A''_s = A_w \cup A_{w'},$$
$$\mathcal{C}''_s = max(\{C_i \cup C_j \mid C_i \in \mathcal{C}_w \wedge C_j \in \mathcal{C}_{w'}\}),$$
$$\forall C_k \in \mathcal{C}''_s, u \in \mathcal{U}.\ K''_s(C_k)(u) = \big\{d \in (K_w(C_i)(u) \oplus K_{w'}(C_j)(u)) \mid d \subseteq A''_s \wedge (C_i, C_j) \in (\mathcal{C}_w \times \mathcal{C}_{w'}) \wedge C_k = C_i \cup C_j\big\}.$$

Similar to other transformation, we only consider what are related to the $\mathrm{F}(v)_{\cup}$ rule: $ss(w)$, $ss(w')$ and $X''(= S_{k+1/2}(r').ss(v'))$. Meanwhile, $ss(w)$ and $ss(w')$ are from τ_k that satisfies the claimed property by induction hypothesis, i) if we let

- C_i be an element of $\mathcal{C}_s$,
- $\mathcal{D}$ be the collection of valid complete-dot sets for w and C_i,
- D be a valid complete-dot set in $\mathcal{D}$, and
- U_c be a set of values such that $K_s(C_i)(u) \cap D \neq \emptyset$ for every $u \in U_c$,

then for every value-dot pair (u, d) in $U_c \times (K_s(C_i)(u) \cap D)$ and its computation tree b, there exists an S_I-starting consistent execution τ'_k that satisfies the **Value-Dot** property and ii) if we let

- C_j be an element of $\mathcal{C}'_s$,
- $\mathcal{D}'$ be the collection of valid complete-dot sets for w' and C_j,
- D' be a valid complete-dot set in $\mathcal{D}'$, and
- U'_c be a set of values such that $K'_s(C_j)(u') \cap D' \neq \emptyset$ for every $u' \in U'_c$,

then for every value-dot pair (u', d') in $U'_c \times (K'_s(C_j)(u') \cap D')$ and its computation tree b', there exists an S_I-starting consistent execution τ''_k that satisfies the **Value-Dot** property. Now, we can claim that if we let

- C_k be an element of $\mathcal{C}''_s$,
- $\mathcal{D}''$ be the collection of valid complete-dot sets for v' and C_k,
- D'' be a valid complete-dot set in $\mathcal{D}''$, and
- U''_c be a set of values such that $K''_s(C_k)(u'') \cap D'' \neq \emptyset$ for every $u'' \in U''_c$,

then for every value-dot pair (u'', d'') in $U''_c \times (K''_s(C_k)(u'') \cap D'')$ and its computation tree b'', there exists an S_I-starting consistent execution τ''_{k+1} that satisfies the **Value-Dot** property. This is because

$$
\begin{aligned}
&\mathcal{D}'' = \{\{\{(e, (v', t, \texttt{undef})) \mid (e,t) \in d\} \mid d \in D\} \mid D \in \mathcal{D}\} \\
&\qquad \cup \{\{\{(e, (v', \texttt{undef}, t)) \mid (e,t) \in d'\} \mid d' \in D'\} \mid D' \in \mathcal{D}'\} \\
&\wedge U''_c = U_c \cup U'_c \\
&\wedge C_k = \{(e, (v', t, \texttt{undef})) \mid (e,t) \in C_i\} \cup \{(e, (v', \texttt{undef}, t)) \mid (e,t) \in C_j\} \\
&\wedge K''_s(C_k)(u) = \{\{(e, (v', t, \texttt{undef})) \mid (e,t) \in d\} \mid d \in K_s(C_i)(u)\} \\
&\qquad\qquad \cup \{\{(e, (v', \texttt{undef}, t)) \mid (e,t) \in d'\} \mid d' \in K'_s(C_j)(u)\} \\
&\wedge \tau_{k+1} = \tau_k^\frown[S_{k+1}] \\
&\wedge \mathit{computation_tree}((u, \{(e, (v', t, \texttt{undef})) \mid (e,t) \in d\}), \tau_{k+1}) \\
&\qquad = ((u, \{(e, (v', t, \texttt{undef})) \mid (e,t) \in d\}), \mathit{computation_tree}((u,d), \tau_k), \texttt{undef}) \\
&\wedge \mathit{computation_tree}((u', \{(e', (v', \texttt{undef}, t')) \mid (e',t') \in d'\}), \tau_{k+1}) \\
&\qquad = ((u', \{(e', (v', \texttt{undef}, t')) \mid (e',t') \in d'\}), \texttt{undef}, \mathit{computation_tree}((u',d'), \tau_k)) \\
&\wedge \tau''_{k+1} = \text{processing the } \Longrightarrow_{\mathit{dflow}} \text{ rules with the } \mathrm{F}(v)_\cup \text{ rule from } \tau'_k \text{ and } \tau''_k \\
\Longrightarrow& \\
&\quad (u, \{(e, (v', t, \texttt{undef})) \mid (e,t) \in d\}), (u', \{(e', (v', \texttt{undef}, t')) \mid (e',t') \in d'\}) \in S''_F(r').s(v') \\
&\wedge \mathit{computation_tree}((u, \{(e, (v', t, \texttt{undef})) \mid (e,t) \in d\}), \tau''_{k+1}) \\
&\qquad = ((u, \{(e, (v', t, \texttt{undef})) \mid (e,t) \in d\}), \mathit{computation_tree}((u,d), \tau'_k), \texttt{undef}) \\
&\wedge \mathit{computation_tree}((u', \{(e', (v', \texttt{undef}, t')) \mid (e',t') \in d'\}), \tau''_{k+1}) \\
&\qquad = ((u', \{(e', (v', \texttt{undef}, t')) \mid (e',t') \in d'\}), \texttt{undef}, \mathit{computation_tree}((u',d'), \tau''_k)).
\end{aligned}
$$

For the $\mathrm{F}(v)_\times$ rule, suppose that

$$\mathit{pred}(v') = \{w, w'\} \quad \text{and} \quad (v', \times, ss|_{\mathit{pred}(v')}) \Longrightarrow_s X'' \text{ by } \mathrm{F}(v)_\times.$$

Let $(A_s, A_s, \mathcal{C}_s, K_s) = ss(w)$ and $(A'_s, A'_s, \mathcal{C}'_s, K'_s) = ss(w')$. Then, by the definition of $\mathrm{F}(v)_\times$,

$$
\begin{aligned}
&((A_s, A_s, \mathcal{C}_s, K_s), (v', \mathit{self}, \texttt{unk})) \Longrightarrow_{t_h} (A_w, L_w, \mathcal{C}_w, K_w), \\
&((A'_s, A'_s, \mathcal{C}'_s, K'_s), (v', \texttt{unk}, \mathit{self})) \Longrightarrow_{t_h} (A_{w'}, L_{w'}, \mathcal{C}_{w'}, K_{w'}) \quad \text{and} \quad X'' = (A''_s, A''_s, \mathcal{C}''_s, K''_s)
\end{aligned}
$$

where

$$
\begin{aligned}
&A''_s = A_w \cup A_{w'},\\
&\mathcal{C}''_s = max(\{C_i \cup C_j \mid C_i \in \mathcal{C}_w \wedge C_j \in \mathcal{C}_{w'}\}),\\
&\forall C_k \in \mathcal{C}''_s, u \in \mathcal{U}.\\
&\quad K''_s(C_k)(u) = \big\{d \in (K_w(C_i)(u_1) \otimes K_{w'}(C_j)(u_2)) \mid\\
&\qquad d \subseteq A''_s \wedge (C_i, C_j) \in (\mathcal{C}_w \times \mathcal{C}_{w'}) \wedge C_k = C_i \cup C_j \wedge u = (u_1, u_2)\big\}.
\end{aligned}
$$

Similar to other transformation, we only consider what are related to the $\mathrm{F}(v)_{\times}$ rule: $ss(w)$, $ss(w')$ and $X''(= S_{k+1/2}(r').ss(v'))$. Meanwhile, $ss(w)$ and $ss(w')$ are from τ_k that satisfies the claimed property by induction hypothesis, i) if we let

- C_i be an element of $\mathcal{C}_s$,
- $\mathcal{D}$ be the collection of valid complete-dot sets for w and C_i,
- D be a valid complete-dot set in $\mathcal{D}$, and
- U_c be a set of values such that $K_s(C_i)(u) \cap D \neq \emptyset$ for every $u \in U_c$,

then for every value-dot pair (u, d) in $U_c \times (K_s(C_i)(u) \cap D)$ and its computation tree b, there exists an S_I-starting consistent execution τ'_k that satisfies the **Value-Dot** property and ii) if we let

- C_j be an element of $\mathcal{C}'_s$,
- $\mathcal{D}'$ be the collection of valid complete-dot sets for w' and C_j,
- D' be a valid complete-dot set in $\mathcal{D}'$, and
- U'_c be a set of values such that $K'_s(C_j)(u') \cap D' \neq \emptyset$ for every $u' \in U'_c$,

then for every value-dot pair (u', d') in $U'_c \times (K'_s(C_j)(u') \cap D')$ and its computation tree b', there exists an S_I-starting consistent execution τ''_k that satisfies the **Value-Dot** property. Now, we can claim that if we let

- C_k be an element of $\mathcal{C}''_s$,
- $\mathcal{D}''$ be the collection of valid complete-dot sets for v' and C_k,
- D'' be a valid complete-dot set in $\mathcal{D}''$, and
- U''_c be a set of values such that $K''_s(C_k)(u'') \cap D'' \neq \emptyset$ for every $u'' \in U''_c$,

then for every value-dot pair (u'', d'') in $U''_c \times (K''_s(C_k)(u'') \cap D'')$ and its computation tree b'', there exists an S_I-starting consistent execution τ''_{k+1} that satisfies the **Value-Dot** property. This is because

$$
\begin{aligned}
&\mathcal{D}'' = \{\{\{(e, (v', t, \texttt{unk})) \mid (e,t) \in d\} \cup \{(e', (v', \texttt{unk}, t')) \mid (e', t') \in d'\} \mid d \in D \wedge d' \in D'\} \\
&\qquad\qquad \mid D \in \mathcal{D} \wedge D' \in \mathcal{D}'\} \\
&\wedge U''_c = U_c \times U'_c \\
&\wedge C_k = \{(e, (v', t, \texttt{unk})) \mid (e,t) \in C_i\} \cup \{(e, (v', \texttt{unk}, t)) \mid (e,t) \in C_j\} \\
&\wedge K''_s(C_k)(u) = \{\{(e, (v', t, \texttt{unk})) \mid (e,t) \in d\} \cup \{(e', (v', \texttt{unk}, t')) \mid (e', t') \in d'\} \\
&\qquad\qquad \mid d \in K_s(C_i)(u_1) \wedge d' \in K'_s(C_j)(u_2)\} \\
&\wedge \tau_{k+1} = \tau_k^\frown[S_{k+1}] \\
&\wedge \mathit{computation_tree}((u, \{(e, (v', t, \texttt{unk})) \mid (e,t) \in d\} \cup \{(e', (v', \texttt{unk}, t')) \mid (e', t') \in d'\}), \tau_{k+1}) \\
&\quad = ((u, \{(e, (v', t, \texttt{unk})) \mid (e,t) \in d\} \cup \{(e', (v', \texttt{unk}, t')) \mid (e', t') \in d'\}), \\
&\qquad \mathit{computation_tree}((u_1, d), \tau_k), \mathit{computation_tree}((u_2, d'), \tau_k)) \\
&\wedge \tau''_{k+1} = \text{processing the } \Longrightarrow_{\mathit{dflow}} \text{ rules with the } \mathrm{F}(v)_\times \text{ rule from } \tau'_k \text{ and } \tau''_k \\
\Longrightarrow \\
&\quad (u, \{(e, (v, t, \texttt{unk})) \mid (e,t) \in d\} \cup \{(e', (v, \texttt{unk}, t')) \mid (e', t') \in d'\}) \in S''_F(r').s(v') \\
&\wedge \mathit{computation_tree}((u, \{(e, (v', t, \texttt{unk})) \mid (e,t) \in d\} \cup \{(e', (v', \texttt{unk}, t')) \mid (e', t') \in d'\}), \tau''_{k+1}) \\
&\quad = ((u, \{(e, (v', t, \texttt{unk})) \mid (e,t) \in d\} \cup \{(e', (v', \texttt{unk}, t')) \mid (e', t') \in d'\}), \\
&\qquad \mathit{computation_tree}((u_1, d), \tau'_k), \mathit{computation_tree}((u_2, d'), \tau''_k))
\end{aligned}
$$

where $u = (u_1, u_2)$.

For the second step of the the DFlow rule, we do not prove the step for the $\sqcup$ operation here since the behaviors of the $\sqcup$ operation are the part of the Merge$_{dflow}$ rule that will be proved later for the claimed property.

We have handled the last rule for transforming replica state, and this means that the Step rule for system configurations has been proved.

We now handle two remaining rules for transforming system configurations: Merge$_{input}$ and Merge$_{dflow}$ rules.

For the Merge$_{input}$ rule, suppose that

$$ss = S_k.r_1.ss, \quad ss' = S_k.r_2.ss \quad \text{and} \quad S_k \Longrightarrow S_{k+1} \text{ by } \textsc{Merge}_{input}.$$

Let $(A_s, A_s, \mathcal{C}_s, K_s) = ss(v')$ and $(A'_s, A'_s, \mathcal{C}'_s, K'_s) = ss'(v')$. Then, by the definition of Merge$_{input}$,

$$\mathcal{C}_s = \{A_s\}, \quad \mathcal{C}'_s = \{A'_s\} \quad \text{and} \quad S_{k+1} = S_k[r_1.ss \mapsto ss[v' \mapsto X'']]$$

where $X'' = (A_s, A_s, \{A_s\}, K_s) \sqcup (A'_s, A'_s, \{A'_s\}, K'_s)$. Similar to other transformation, we only consider what are related to the $\textsc{Merge}_{input}$ rule: $ss(v')(= S_k(r_1).ss(v'))$, $ss'(v')(= S_k(r_2).ss(v'))$, and $X''(= S_{k+1}(r_1).ss(v'))$. Meanwhile, since $ss(v')$ and $ss'(v')$ are from τ_k that satisfies the claimed property by induction hypothesis, i) if we let

- $\mathcal{D}$ be the collection of valid complete-dot sets for v' and A_s,
- D be a valid complete-dot set in $\mathcal{D}$, and
- U_c be a set of values such that $K_s(A_s)(u) \cap D \neq \emptyset$ for every $u \in U_c$,

then for every value-dot pair (u, d) in $U_c \times (K_s(A_s)(u) \cap D)$ and its computation tree b, there exists an S_I-starting consistent execution τ'_k that satisfies the **Value-Dot** property and ii) if we let

- $\mathcal{D}'$ be the collection of valid complete-dot sets for v' and A'_s,
- D' be a valid complete-dot set in $\mathcal{D}'$, and
- U'_c be a set of values such that $K'_s(A'_s)(u') \cap D' \neq \emptyset$ for every $u' \in U'_c$,

then for every value-dot pair (u', d') in $U'_c \times (K'_s(A'_s)(u') \cap D')$ and its computation tree b', there exists an S_I-starting consistent execution τ''_k that satisfies the **Value-Dot** property. Now, we can claim that if we let

- $\mathcal{D}''$ be the collection of valid complete-dot sets for v' and A''_s,
- D'' be a valid complete-dot set in $\mathcal{D}''$, and
- U''_c be a set of values such that $K''_s(A''_s)(u'') \cap D'' \neq \emptyset$ for every $u'' \in U''_c$,

then for every value-dot pair (u'', d'') in $U''_c \times (K''_s(A''_s)(u'') \cap D'')$ and its computation tree b'', there exists an S_I-starting consistent execution τ''_{k+1} that satisfies the **Value-Dot**

property. This is because

$$
\begin{aligned}
&\mathcal{D}'' = \mathcal{D} \cup \mathcal{D}' \cup \{D \cup D' \mid D \in \mathcal{D} \wedge D' \in \mathcal{D}'\} \\
&\wedge U_c'' = U_c \cup U_c' \\
&\wedge A_s'' = A_s \cup A_s' \\
&\wedge K_s''(A_s'')(u) = K_s(A_s)(u) \oplus K_s'(A_s')(u) \\
&\wedge \tau_{k+1} = \tau_k^\frown[S_{k+1}] \\
&\wedge \mathit{computation_tree}((u,d),\tau_{k+1}) = \mathit{computation_tree}((u,d),\tau_k) \\
&\wedge \mathit{computation_tree}((u',d'),\tau_{k+1}) = \mathit{computation_tree}((u',d'),\tau_k) \\
&\wedge \tau_{k+1}'' = \text{processing the } \Longrightarrow_{\mathit{input}} \text{ rules with the } \textsc{Merge}_{\mathit{input}} \text{ rule from } \tau_k' \text{ and } \tau_k'' \\
\Longrightarrow \\
&\quad (u,d),(u',d') \in S_F''(r_1).s(v') \wedge (u,d),(u',d') \in S_F''(r_2).s(v') \\
&\wedge \mathit{computation_tree}((u,d),\tau_{k+1}'') = \mathit{computation_tree}((u,d),\tau_k') \\
&\wedge \mathit{computation_tree}((u',d'),\tau_{k+1}'') = \mathit{computation_tree}((u',d'),\tau_k'').
\end{aligned}
$$

For the $\textsc{Merge}_{\mathit{dflow}}$ rule, suppose that

$$ss = S_k.r_1.ss, \quad ss' = S_k.r_2.ss \quad \text{and} \quad S_k \Longrightarrow S_{k+1} \text{ by } \textsc{Merge}_{\mathit{dflow}}.$$

Let $(A_s, A_s, \mathcal{C}_s, K_s) = ss(v')$ and $(A_s', A_s', \mathcal{C}_s', K_s') = ss'(v')$. Then, by the definition of $\textsc{Merge}_{\mathit{dflow}}$,

$$S_{k+1} = S_k[r_1.ss \mapsto ss[v' \mapsto X'']]$$

where $X'' = (A_s, A_s, \mathcal{C}_s, K_s) \sqcup (A_s', A_s', \mathcal{C}_s', K_s')$. Similar to other transformation, we only consider what are related to the $\textsc{Merge}_{\mathit{dflow}}$ rule: $ss(v')(= S_k(r_1).ss(v'))$, $ss'(v')(= S_k(r_2).ss(v'))$, and $X''(= S_{k+1}(r_1).ss(v'))$. Meanwhile, since $ss(v')$ and $ss'(v')$ are from τ_k that satisfies the claimed property by induction hypothesis, i) if we let

- C_i be an element of $\mathcal{C}_s$,
- $\mathcal{D}$ be the collection of valid complete-dot sets for v' and C_i,
- D be a valid complete-dot set in $\mathcal{D}$, and
- U_c be a set of values such that $K_s(C_i)(u) \cap D \neq \emptyset$ for every $u \in U_c$,

then for every value-dot pair (u, d) in $U_c \times (K_s(C_i)(u) \cap D)$ and its computation tree b, there exists an S_I-starting consistent execution τ_k' that satisfies the **Value-Dot** property and ii) if we let

- C_j be an element of $\mathcal{C}_s'$,

- $\mathcal{D}'$ be the collection of valid complete-dot sets for v' and C_j,
- D' be a valid complete-dot set in $\mathcal{D}'$, and
- U'_c be a set of values such that $K'_s(C_j)(u') \cap D' \neq \emptyset$ for every $u' \in U'_c$,

then for every value-dot pair (u', d') in $U'_c \times (K'_s(C_j)(u') \cap D')$ and its computation tree b', there exists an S_I-starting consistent execution τ''_k that satisfies the **Value-Dot** property. Now, we can claim that if we let

- C_k be an element of $\mathcal{C}''_s$,
- $\mathcal{D}''$ be the collection of valid complete-dot sets for v' and C_k,
- D'' be a valid complete-dot set in $\mathcal{D}''$, and
- U''_c be a set of values such that $K''_s(C_k)(u'') \cap D'' \neq \emptyset$ for every $u'' \in U''_c$,

then for every value-dot pair (u'', d'') in $U''_c \times (K''_s(C_k)(u'') \cap D'')$ and its computation tree b'', there exists an S_I-starting consistent execution τ''_{k+1} that satisfies the **Value-Dot** property. This is because

$$\mathcal{D}'' = \begin{cases} \mathcal{D} & \text{if } C_k = C_i \wedge \neg\exists C_j \in \mathcal{C}'_s.\ C_i \subset C_j \\ \mathcal{D}' & \text{if } C_k = C_j \wedge \neg\exists C_i \in \mathcal{C}_s.\ C_j \subseteq C_i \end{cases}$$

$$\wedge\, U''_c = \begin{cases} U_c & \text{if } C_k = C_i \wedge \neg\exists C_j \in \mathcal{C}'_s.\ C_i \subset C_j \\ U'_c & \text{if } C_k = C_j \wedge \neg\exists C_i \in \mathcal{C}_s.\ C_j \subseteq C_i \end{cases}$$

$$\wedge\, K''_s(A''_s)(u) = \begin{cases} K_s(C_i)(u) \cup (\bigcup\{K'_s(C_j)(u) \mid C_j \in \mathcal{C}'_s \wedge C_j \subseteq C_i\}) \\ \qquad \text{if } C_k = C_i \wedge \neg\exists C_j \in \mathcal{C}'_s.\ C_i \subset C_j \\ (\bigcup\{K_s(C_i)(u) \mid C_i \in \mathcal{C}_s \wedge C_i \subseteq C_j\}) \cup K'_s(C_j)(u) \\ \qquad \text{if } C_k = C_j \wedge \neg\exists C_i \in \mathcal{C}_s.\ C_j \subseteq C_i \end{cases}$$

$$\wedge\, \tau_{k+1} = \tau_k^\frown[S_{k+1}]$$

$$\wedge\, \mathit{computation_tree}((u,d), \tau_{k+1}) = \mathit{computation_tree}((u,d), \tau_k)$$

$$\wedge\, \mathit{computation_tree}((u',d'), \tau_{k+1}) = \mathit{computation_tree}((u',d'), \tau_k)$$

$$\wedge\, \tau''_{k+1} = \text{processing the } \Longrightarrow_{\mathit{dflow}} \text{ rules with the } \textsc{Merge}_{\mathit{dflow}} \text{ rule from } \tau'_k \text{ and } \tau''_k$$

$$\Longrightarrow$$

$$(u,d) \text{ or } (u',d') \in S''_F(r_1).s(v') \wedge (u,d) \text{ or } (u',d') \in S''_F(r_2).s(v')$$

$$\wedge\, \mathit{computation_tree}((u,d), \tau''_{k+1}) = \mathit{computation_tree}((u,d), \tau'_k)$$

$$\wedge\, \mathit{computation_tree}((u',d'), \tau''_{k+1}) = \mathit{computation_tree}((u',d'), \tau''_k).$$

□

Lemma 5.6.13 shows that for every value whose dot is in the valid complete-dot sets on shadow system configurations, there is a consistent execution whose last system

configuration contains the identical value and dot. The existence of the identical value in the consistent execution means that each value with the valid complete-dot can provide a *strong* form of guarantee about the sameness of the replicated values.

5.6.6 Proof of the Theorem

We now provide the formal proof for the theorem that provides a formal guarantee on what we get by performing the $\texttt{read}_c()$ operation.

Theorem 5.6.14 (Consistent Read)**.** *Let S be the last system configuration of τ, and (C'_i, D', U') be the result of performing $\texttt{read}_c()$ on $S(r).s(v)$. Then, there exists a consistent execution τ' such that*

1. *τ' starts at S_I; and*
2. *when S' is the last system configuration of τ' and (C'_i, D', U'_c) is the result of executing $\texttt{read}_c()$ on $S'(r).ss(v)$, we have that $U' \subseteq U'_c$.*

Proof. Let τ be $[S_1, S_2, \ldots, S_n]$. We prove that every S_k in τ satisfies the following property: for every v in $G.V$ and every replica id r, if we let (C'_i, D', U') be the result of executing $\texttt{read}_c(C_i, D, S_k(r).s(v))$ where

- C_i and D are $\{\}$ and $\{\}$ if $k = 1$, and
- if not, they are the first two components of the result of executing $\texttt{read}_c()$ on $S_j(r).s(v)$ for some $j < k$,

then there exists a consistent execution τ' such that

1. τ' starts at S_I; and
2. when S' is the last system configuration of τ', executing $\texttt{read}_c()$ on $S'(r).ss(v)$ produces (C'_i, D', U'_c) for some $U'_c \supseteq U'$.

The important part here is the superset relationship $U'_c \supseteq U'$.

By Lemma 5.6.2, every S_k has a corresponding shadow system configuration S_s satisfying $S_s \sim S_k$. Lemma 5.6.11 implies that, if we let $(A_s, A_s, \mathcal{C}_s, K_s) = S_s(r).ss(v)$, then we can find C_j, an element of $\mathcal{C}_s$, that satisfies $C_j = C'_i$, and also $\mathcal{D}$, the valid complete-dot set collection for v and C_j, that satisfies $D' \in \mathcal{D}$. Lemma 5.6.9 shows that for these C_j and $\mathcal{D}$, there is a set of values U'_c such that every value u'_c in U'_c satisfies $K_s(C_j)(u'_c) \cap D' \neq \emptyset$. Lastly, by Lemma 5.6.13, there exists a consistent execution τ' such that, if we let S' be the last system configuration of τ' and $(A', A', \mathcal{C}', K') = S'(r).ss(v)$, then $S'(r).ss(v)$ contains every value u'_c in U'_c and its dot d in $K'(C_j)(u') \cap D'$.

From the definition of the $\texttt{read}_c()$ operation, we have

$$\forall u' \in U'.\, K(C_i')(u') \cap D' \neq \emptyset \quad \text{and} \quad U_c' = \{u_c' \mid K'(C_i')(u_c') \cap D' \neq \emptyset\}.$$

Furthermore, we also have

$$\forall u' \in U'.\, K(C_i')(u') \subseteq K_s(C_i')(u')$$

from the definition of the $\sim$ relation. This implies

$$\forall u' \in U'.\, K_s(C_i')(u') \cap D' \neq \emptyset.$$

By Lemma 5.6.13, every u' in U' and its dot d' in $K_s(C_i')(u') \cap D'$ exists in $S'(r).ss(v)$. This means

$$\forall u' \in U'.\, K'(C_i')(u') \cap D' \neq \emptyset.$$

Now, we can claim that

$$U' \subseteq U_c'.$$

□

Theorem 5.6.14 implies that the result of the $\texttt{read}_c()$ operation guarantees the consistency of the outcome by proving the existence of a consistent execution that produces the identical result. The behaviors of consistent executions are similar to the ones of executions on a single node, which can provide strong consistency of the replicated data among weakly consistent distributed systems.

Chapter 6

Data Types Built Upon $\mathrm{OR}_{\mathrm{PC}}$ Sets

In this chapter, we introduce client-level data types for our programming language. There are three client-level data types: the Last-Writer-Wins (LWW) register, the Grow-only (G) counter, and the Observed-Remove (OR) bag. The LWW register is used to store a *single* value and read the stored value. The G counter is a data type that *counts* how many increments have happened. The OR bag stores various kinds of values with *duplication*. These client-level data types are built upon the $\mathrm{OR}_{\mathrm{PC}}$ set, which was formulated in Chapter 4 and acts as the system-level data type that our implemented system internally uses to express dataflow programs. As a result, the programs programmed with our client-level data types inherit the key theoretical properties of dataflow programs described and proved in the previous chapter. Also, these programs can use the operations of the $\mathrm{OR}_{\mathrm{PC}}$ set with the client-level data types, such as $\cup$ and $\times$, in order to implement complicated functionalities.

We first explain the constraints of the client-level data types, which are designed to implement the intended semantics of these data types. Then, we show how to express these constraints using partial orders for events. We also illustrate how the $\mathrm{OR}_{\mathrm{PC}}$ set operations can be used with those client-level data types. Finally, we provide the description of the details of each data type.

6.1 Data Type Constraints, Partially Ordered Events and Mixed Data Type Operations

Every data type for our programming language, both client-level and system-level, has its own constraints. Those constraints are either from the semantics of the data type itself or from the fact that the data type is used in distributed systems. For example, the removal of a value from the $\mathrm{OR}_{\mathrm{PC}}$ set assumes that the $\mathrm{OR}_{\mathrm{PC}}$ set has already *observed* the `insert`() operation for the value. In order to determine whether the `insert`() operation for a value is observed or not, our system stores a value with an event that serves as the identifier of the `insert`() operation for the value. Even though the same

values are inserted multiple times at different replicas, each value is stored with the different event so that they can be differentiated. This means that a removed value from an $\mathrm{OR}_{\mathrm{PC}}$ set may reappear due to the unobserved `insert`() operation (inserted with the different event) that happens at another replica. The events are used not only for differentiating `insert`() operations but also for ensuring the per-object causality. In order to guarantee the per-object causality, the $\mathrm{OR}_{\mathrm{PC}}$ set stores a set of all event histories (an event history consists of an event and its dataflow path) as a component and this component should be a downward-closed set.

Each client-level data type also has its own constraints. Here, we focus on explaining what kinds of the constraints each type has and describing how those constraints are expressed in our system. The details of how each data type is formulated will be described in the following sections. The important things about those data types are i) each data type has the same structure as the $\mathrm{OR}_{\mathrm{PC}}$ set (a tuple of the components: A, L, $\mathcal{C}$, and K), ii) the constraints of each data type include the condition that the component A of an $\mathrm{OR}_{\mathrm{PC}}$ set which stores all event histories should be a downward-closed set because every data type is built upon the $\mathrm{OR}_{\mathrm{PC}}$ set, and iii) the other constraints of the data type are expressed by partial orders on the events.

Here are details of the overview that we have just given. Recall that an event (v, r, n) is a tuple of node id, replica id, and non-negative integer, and also that the following operation is used to ensure the downward closure of the component A of an $\mathrm{OR}_{\mathrm{PC}}$ set that stores all event histories:

$$next(A) = \{((v, r, n+1), (v, \texttt{undef}, \texttt{undef})) \mid ((v, r, n), (v, \texttt{undef}, \texttt{undef})) \in A \\ \wedge ((v, r, n+1), (v, \texttt{undef}, \texttt{undef})) \notin A\}.$$

Firstly, the Last-Writer-Wins (LWW) register supports two operations: `write`() and `read`(). The `write`() operation stores a value to the register and the `read`() operation returns the stored value from the register. The LWW register has two constraints: i) the register can store only one value and ii) if there are the multiple invocations of the `write`() operation with different values, the result of the `read`() operation should be the most recently written value. The following example shows how the LWW register is used with two replicas (*replica a* and *replica b*):

Listing 6.1: LWW register in *replica a*

```
1 // reg:LWWRegister[Integer]
2
3 write(45, reg);
4
5 write(21, reg);
6
7 x := read(reg);
```

Listing 6.2: LWW register in *replica b*

```
1 // reg:LWWRegister[Integer]
2
3 write(70, reg);
4
5 y := read(reg);
```

We assume that two `read`() operations in Listing 6.1 and Listing 6.2 are invoked after the two replicas exchange the information about the written values for the register. Recall that two replicas in our system should have the identical state after exchanging their states in order to guarantee the strong convergence. Therefore, the results of the `read`() operations above (x and y) should be identical. Based on the constraints of the LWW register, the result in this case would be either 21 or 70 because 21 is written later than 45 according to the program order. In order to choose a value between 21 and 70, there should be a way to determine which value is written later than the other.

Before we explain our solution to determine which value is written later than the other for the LWW register, we first check whether the existing data type (the OR_{PC} set) can be used to express the semantics of the LWW register. The following example shows an attempt to implement the functionality that is identical to the example above, using the OR_{PC} set:

Listing 6.3: OR_{PC} set in *replica a*

```
1  // register:ORSet[Integer]
2
3  x' := read(register);
4  x'?delete(x', register):NULL;
5  insert(45, register);
6
7  x' := read(register);
8  x'?delete(x', register):NULL;
9  insert(21, register);
10
11 x := read(register);
```

Listing 6.4: OR_{PC} set in *replica b*

```
1 // register:ORSet[Integer]
2
3 y' := read(register);
4 y'?delete(y', register):NULL;
5 insert(70, register);
6
7 y := read(register);
```

The code line 3-5 in Listing 6.3 is designed to behave like the `write`() operation of the LWW register. These three lines read the content of the OR_{PC} set, then delete the value if exists, and finally insert a new value. This delete-then-insert behaviour is to ensure that the OR_{PC} set has a single value as in the case of the LWW register, but the OR_{PC}

set might end up with multiple values if the delete-then-insert behaviour is not *atomic*. For instance, assume that the delete-then-insert behaviour is not atomic and the $\mathrm{OR}_{\mathrm{PC}}$ set is empty when *replica a* invokes the `read`() operation (described in the code line 3 in Listing 6.3). Then, *replica b* completes its delete-then-insert behaviour, and this means that the $\mathrm{OR}_{\mathrm{PC}}$ set now contains 70. When *replica a* continues its delete-then-insert behaviour presented in the code line 4-5 in Listing 6.3, it will insert 45 without delete anything since the $\mathrm{OR}_{\mathrm{PC}}$ set was empty. In this case, the $\mathrm{OR}_{\mathrm{PC}}$ set ends up with two values 70 and 45. Even if the atomicity assumption met, storing a single value can also be violated. Assume that the two replicas exchange the information about their insertions located in the code line 9 in Listing 6.3 and the code line 5 in Listing 6.4 before invoking their `read`() operations. After this information exchange, the $\mathrm{OR}_{\mathrm{PC}}$ sets in both replicas have two values 21 and 70 because the two insertions are valid for the $\mathrm{OR}_{\mathrm{PC}}$ set.

We now describe how the two constraints of the LWW register are expressed in our language. The key idea of the constraints is that there should be a way to determine which value is written later than the other when two values are written at different replicas. Some implementation of the LWW register stores a value with a global timestamp, which intuitively specifies when the value is written. We use the following partial order relation to determine the order of the values:

Definition 6.1.1. The *node-dependent order* $\sqsubseteq_n$ is the following partial order on the events:

$$e \sqsubseteq_n e' \iff e.v = e'.v \wedge (e.n < e'.n \vee (e.n = e'.n \wedge e.r < e'.r)).$$

Two events are comparable by the node-dependent order only when their node ids are identical. Which of these two is larger is determined by their replica-local timestamps (i.e., $e.n$ and $e'.n$) together with their replica ids (i.e., $e.r$ and $e'.r$). Note that all events with the same node id are totally ordered by our node-dependent order. Since all insertion events for a LWW register have the same node id, this order can determine which written value to the register is more recent than another written value.

The LWW register has the same structure as the $\mathrm{OR}_{\mathrm{PC}}$ set. As a result, the component A of a LWW register should be downward closed. In order to ensure the downward closure of the component A of a LWW register for the node-dependent order, we extend the $\mathit{next}(A)$ operation as follows:

$$\begin{aligned}
&\mathit{next_with_order}(A, \sqsubseteq_n) \\
&\quad = \{((v, r, n+1), (v, \texttt{undef}, \texttt{undef})) \mid \\
&\qquad\quad (\exists r' \in \mathcal{R} \mid ((v, r', n), (v, \texttt{undef}, \texttt{undef})) \in A) \\
&\qquad\quad \wedge (\forall r'' \in \mathcal{R} \mid ((v, r'', k), (v, \texttt{undef}, \texttt{undef})) \notin A) \\
&\qquad\quad \wedge k > n \wedge r \in \mathcal{R}\}.
\end{aligned}$$

The *next_with_order*$(A, \sqsubseteq_n)$ operation guarantees that only the events which satisfy the following property can be inserted: the n value of a newly inserted event should be $k+1$ where k is the maximum of the n values of the events in A. This implies that any event history which contains an event in *next_with_order*$(A, \sqsubseteq_n)$ dominates every event history in A. However, the operation cannot prevent some events from having the same n value. This is the reason that the node-dependent order is designed to guarantee that two comparable events having the same n value are compared based on their r values (two different comparable events cannot have the same r value when the v and n values are identical). This functionality is similar to Lamport's logical clock [29] in distributed systems.

The important thing of the extension of the *next*(A) operation is that the extended operations should maintain the component A as downward-closed. For example, any event history h in the result of the *next_with_order*$(A, \sqsubseteq_n)$ operation dominates all event history in A and this means that $A \cup \{h\}$ is downward-closed.

We now explain how the node-dependent order is used in the example code in Listing 6.1 and Listing 6.2. Let v be the node id of the LWW register in the example. Then, the value 45 is stored with the event $(v, a, 1)$ after the code line 3 in Listing 6.1 is invoked. For the second `write`() operation described in the code line 5 in Listing 6.1, the value 21 is stored with the event $(v, a, 2)$. Here, we can determine that the value 21 is written later than the value 45 based on $(v, a, 1) \sqsubseteq_n (v, a, 2)$. On the other hand, due to the `write`() operation in Listing 6.2, the value 70 is stored with the event $(v, b, 1)$. After the two replicas exchange the information about the written values, they compare the events of the written values based on the node-dependent order. The two replicas then choose the value 21 as the last written value because $(v, b, 1) \sqsubseteq_n (v, a, 2)$.

Secondly, the Grow-only (G) counter supports two operations: `increase`() and `read`(). The former increases the current counter value by 1, and the latter returns the current counter value. The G counter has two constraints: i) the stored counter value should be a single and non-negative integer, and ii) the current counter value should be identical to the number of the observed invocations of the `increase`() operation. The following shows the example code that describes how the G counter is used with two replicas (*replica a* and *replica b*):

Listing 6.5: G counter in *replica a*

```
1 // counter : GCounter
2
3 increase(counter);
4
5 increase(counter);
6
7 x := read(counter);
```

Listing 6.6: G counter in *replica b*

```
1 // counter : GCounter
2
3 increase(counter);
4
5 y := read(counter);
```

We assume that two `read`() operations in Listing 6.5 and Listing 6.6 are invoked after the two replicas exchange the information about the increments for the counter. In order to guarantee the strong convergence, the results of the `read`() operations above (x and y) should be identical. Based on the constraints of the G counter, the result in this case should be 3.

The important property of the constraints of the G counter is that the `increase`() operation should increase the counter value of the input by 1. For the G counter in our language, an event (v, r, n) acts as the identifier of the n-th `increase`() operation invoked on the node v at the replica r. The following partial order relation is used to compare those events in G counter based on how many `increase`() operations were invoked:

Definition 6.1.2. The *replica-dependent order* $\sqsubseteq_r$ is the partial order on the events defined by

$$e \sqsubseteq_r e' \iff e.v = e'.v \wedge e.r = e'.r \wedge e.n \leq e'.n.$$

The events with the replica-dependent order can only be compared when those events have the identical node id and replica id. Having the larger value of n indicates that the event is stored after more `increase`() operations are invoked. For instance, the counter value 5 with the event $(v, r, 5)$ is stored when the fifth `increase`() operations is invoked at the replica r. On the other hand, the counter value 3 with the event $(v, r, 3)$ is stored when the third `increase`() operations is invoked at the replica r. Based on $(v, r, 3) \sqsubseteq_r (v, r, 5)$, it is clear that the counter value 5 already contains all increments for the counter value 3.

Similar to the LWW register, the component A of a G counter should be downward closed. In order to ensure the downward closure of the component A of a G counter for the replica-dependent order, we can simply re-use the $next(A)$ operation of the $\mathrm{OR}_{\mathrm{PC}}$ set because the n-th `increase`() operation of a G counter can be treated as the n-th `insert`() operation of an $\mathrm{OR}_{\mathrm{PC}}$ set:

$$next_with_order(A, \sqsubseteq_r) = next(A).$$

We now explore how the replica-dependent order is used in our language with the example code described in Listing 6.5 and Listing 6.6. Let v be the node id of the G counter in the example. Then, the first counter value 1 is stored with the event $(v, a, 1)$ after the code line 3 in Listing 6.5 is invoked. For the second `increase`() operation described in the code line 5 in Listing 6.5, the counter value 2 is stored with the event $(v, a, 2)$. Here, we can imply that the counter value 2 includes the increment for the counter value 1 with $(v, a, 1)$ based on $(v, a, 1) \sqsubseteq_r (v, a, 2)$. On the other hand, from the `increase`() operation in Listing 6.6, the counter value 1 is stored with the event $(v, b, 1)$. After the two replicas exchange the information about their counter values, they compare the events of the values based on the replica-dependent order. The two replicas then accumulate the two counter values 2 and 1 because the two events $(v, a, 2)$ and $(v, b, 1)$, respectively, are not comparable by the replica-dependent order.

Lastly, the Observed-Remove (OR) bag supports the `add`(), `remove`() and `read`() operations similar to the OR_{PC} set, but it can implement a multiset, instead of a set, which may contain duplicated values. Handling the duplicated values for the OR bag is not related to the partial order of the events, so it will be explained when the data type is formulated. Except the fact that the OR bag can store the duplicated values, the OR bag has the same constraint as the OR_{PC} set that is related to how the stored values are removed. The following example shows how the OR bag is used with two replicas (*replica a* and *replica b*):

Listing 6.7: OR bag in *replica a*

```
1 // bag : ORBag[Integer]
2
3 add(45, bag);
4
5 x := read(bag);
```

Listing 6.8: OR bag in *replica b*

```
1 // bag : ORBag[Integer]
2
3 add(45, bag);
4
5 remove(45, bag);
6
7 y := read(bag);
```

We assume that two `read`() operations in Listing 6.7 and Listing 6.8 are invoked after the two replicas exchange the information about the additions and removals for the bag. In order to guarantee the strong convergence, the results of the `read`() operations above (x and y) should be identical. Based on the constrains of the OR bag, the identical result depends on what `add`() operations are observed before the `remove`() operation in Listing 6.8 is invoked. For example, if the `remove`() operation is invoked without observing the `add`() operation in Listing 6.7, the result should be $\{(45, 1)\}$ where 45 is added by the code line 3 in Listing 6.7. On the other hand, if the `remove`() operation is invoked after observing both `add`() operations, the result should be $\{\}$.

In order to express the constraint regarding the removal of a value, the OR bag uses the following partial order relation:

Definition 6.1.3. The *independent order* $\sqsubseteq_i$ is the discrete partial order on the events. That is,

$$e \sqsubseteq_i e' \iff e = e'.$$

The definition of the independent order implies that there is no relation between two different events. This means that the event with the independent order acts as the identifier of an `add`() operation and this functionality is identical to the event in the OR_{PC} set. Similar to the G counter, the OR bag also re-uses the $next(A)$ operation of the OR_{PC} set for the downward closure of the component A of a OR bag:

$$next_with_order(A, \sqsubseteq_i) = next(A).$$

We now introduce two utility operations, *maximal*() and *maximal_orset*(), which use the partial order relations described above to make their inputs Pareto-optimal. The first operation takes two parameters: a set of event-histories and a partial order of events. The result of the operation is a set of event-histories such that the event of a event-history in the set is not dominated by that of another event-history in the set:

$$maximal(H, \sqsubseteq) = \{(e,t) \in H \mid \forall (e',t) \in H.\ e \sqsubseteq e' \implies e = e'\}.$$

If we let H is from a LWW register that uses the node-dependent order $\sqsubseteq_n$, the result of $maximal(H, \sqsubseteq_n)$ should contain the only event-history stored with the last written value of the register. It is because the event stored with the last written value dominates all the events stored with the previously written values and all event-histories having those events are removed by the *maximal*() operation.

The second utility operation is defined as follows:

$$maximal_orset((A, L, \mathcal{C}, K), \sqsubseteq) = (A', L', \mathcal{C}', K')$$
$$\left[\begin{array}{ll} \text{where} & A' = maximal(A, \sqsubseteq), \\ & L' = maximal(L, \sqsubseteq), \\ & \mathcal{C}' = \{maximal(C_i, \sqsubseteq) \mid C_i \in \mathcal{C}\}, \\ & K' = \lambda C_i \in \mathcal{C}.\ [maximal(C_i, \sqsubseteq) \mapsto (\lambda u.\ \{d \mid d \in K(C_i)(u) \ \wedge d \subseteq L'\})]. \end{array}\right]$$

The *maximal_orset*() operation applies the *maximal*() operation to every set of event-histories in the OR_{PC} set $(A, L, \mathcal{C}, K)$ so that the OR_{PC} set can be Pareto-optimal with respect to the order $\sqsubseteq$.

Every client-level data type is built upon the OR_{PC} set. In fact, it can be viewed as an OR_{PC} set with additional properties. Therefore, it is possible that the client-level data types can be used as the parameters of the operations of the OR_{PC} set such as `map`(), `filter`(), $\cup$, and $\times$. For example, the following dataflow program shows how the `filter`() operation can be used with the LWW register:

Listing 6.9: A LWW register with `filter()`

```
1  // grade : LWWRegister[Integer]
2  // pass_grade : LWWRegister[Integer]
3
4  // f = (fun value -> value > 50)
5
6  filter(f, grade, pass_grade);
7
8  write(45, grade);
9  x := read(pass_grade);
10
11 write(70, grade);
12 y := read(pass_grade);
```

Due to the `filter()` operation with the function **f** (line 6), the LWW register **pass_grade** can only have the value that is greater than 50. This means that **x** is the empty set (which indicates that the register has no value) and **y** is 70 in the program above.

The binary operations ($\cup$ and $\times$) of the OR_{PC} set provide more flexibility of the usage for the client-level data types as parameters. The following example describes that the OR bag object and the LWW register object can be used with the $\times$ operation as parameters:

Listing 6.10: LWW register and OR bag with $\times$

```
1  // interest_rate : LWWRegister[Floating-point]
2  // deposits : ORBag[Integer]
3  // deposit_and_rate : ORBag[Integer × Floating-point]
4
5  deposit_and_rate := deposits × interest_rate;
6
7  write(0.1, interest_rate);
8  add(1000, deposits);
9  add(2000, deposits);
10
11 z := read(deposit_and_rate);
```

Line 5 shows that the input parameters of the $\times$ operations are the objects of the OR bag and the LWW register data types, and the data type of the output is the OR bag. The successful result of the `read()` operation, **z**, is $\{((1000, 0.1), 1), ((2000, 0.1), 1)\}$.

6.2 Last-Writer-Wins Register

The registers support two operations: the one for writing a value to the register and the other for reading the previously written value from the register. When these registers are used in distributed systems, it is possible that multiple values are simultaneously written

to the same register. The Last-Writer-Wins (LWW) register stores the last written value when the multiple values are written. There are various ways to determine the order of the written values for the LWW register and our programming language uses the partial order $\sqsubseteq_n$ of the events, defined in the previous section, for determining that order.

As we briefly explained in the previous section, every client-level data type including the LWW register is built upon the OR_{PC} set. To be more precise, the LWW register is a simple OR_{PC} set such that it supports two operations: the `write`() and `read`() operations. In order to satisfy the constraints of the LWW register, it is important that the simple OR_{PC} set (the system-level data type of the LWW register) should always have a single value that are written the most recently. The `write`() operation is defined as follows in order to make sure that the LWW register is maintained as a singleton set:

$$\begin{aligned}
&\texttt{write} : \mathcal{E} \times \mathcal{U} \times \mathcal{X}_s \to \mathcal{X}_s \\
&\texttt{write}(e, u, (A, L, \mathcal{C}, K)) = \mathit{maximal_orset}(\texttt{insert}(e, u, (A, L, \mathcal{C}, K)), \sqsubseteq_n).
\end{aligned}$$

Recall that the `insert`() operation uses the *next*() operation to ensure the downward closure of the component A of an OR_{PC} set. For the client-level data types including the LWW register, the `insert`() operation uses the extended version of the *next*() operation: *next_with_order*(). The *next_with_order*() operation with $\sqsubseteq_n$ (the partial order of the events that are used for the LWW register) is defined in the previous section as follows:

$$\begin{aligned}
&\mathit{next_with_order}(A, \sqsubseteq_n) \\
&\quad = \{((v, r, n+1), (v, \texttt{undef}, \texttt{undef})) \mid \\
&\qquad\qquad (\exists r' \in \mathcal{R} \mid ((v, r', n), (v, \texttt{undef}, \texttt{undef})) \in A) \\
&\qquad\qquad \wedge (\forall r'' \in \mathcal{R} \mid ((v, r'', k), (v, \texttt{undef}, \texttt{undef})) \notin A) \\
&\qquad\qquad \wedge k > n \wedge r \in \mathcal{R}\}.
\end{aligned}$$

The definition of the *next_with_order*$(A, \sqsubseteq_n)$ operation implies that the event-history contains the successfully added event e is the most dominant event-history in the component A of the result of the `write`() operation. This means that the component A of the result of the operation will have only the newly added event-history after applying the *maximal_orset*() operation and the previously written value will also be removed by becoming its provenance as an empty set due to *maximal_orset*().

The `read`() operation for the OR_{PC} set can be re-used when we read a written value from the LWW register. The interesting fact about the `read`() operation for the LWW register is that once a value is successfully written, the successful result of the operation should always be a singleton set. This is because the event-history that is inserted with the successfully written value is a member of *next_with_order*$(A, \sqsubseteq_n)$ and it is the most

dominant event-history in A. The operation is defined as follows:

$$\begin{aligned}
&\texttt{read} : 2^{\mathcal{H}} \times \mathcal{X} \rightarrow 2^{\mathcal{H}} \times 2^{\mathcal{U}} \cup \{\texttt{undef}\} \\
&\texttt{read}(C_i, (A, L, \mathcal{C}, K)) = \text{ if (there is } C_i' \in \mathcal{C} \text{ with } C_i \subseteq C_i') \text{ then} \\
&\qquad\qquad\qquad\qquad \text{let } C_i' \text{ be such an element in } \mathcal{C} \text{ in} \\
&\qquad\qquad\qquad\qquad (C_i', \{u \in \mathcal{U} \mid K(C_i')(u) \neq \{\}\}) \\
&\qquad\qquad\qquad \text{else } \texttt{undef}
\end{aligned}$$

Similar to the `read`() operation of the OR_{PC} set, the selection of C_i' in $\mathcal{C}$ that satisfies $C_i \subseteq C_i'$ is non-deterministic. For practical purposes, our implementation selects the one that can return the maximal number of values.

6.3 Grow-only Counter

The Grow-only (G) counter is a simple OR_{PC} set that supports two operations: the `increase`() operation for increasing the counter value by one and the `read`() operation for reading the current counter value. In the case of the simple OR_{PC} set, an event (v, r, n) acts as the identifier of the n-th `insert`() operation on the replica r, but for the G counter, (v, r, n) can be treated as the identifier of the n-th `increase`() operation on r. The G counter can re-use the *next*() operation of the OR_{PC} set for ensuring the downward closure of the component A because the events have the same structure but the different meaning. The $\mathit{next_with_order}(A, \sqsubseteq_r)$ operation for the G counter is defined as follows:

$$\begin{aligned}
\mathit{next_with_order}(A, \sqsubseteq_r) = \{((v, r, n+1), (v, \texttt{undef}, \texttt{undef})) \mid &((v, r, n), (v, \texttt{undef}, \texttt{undef})) \in A \\
&\wedge ((v, r, n+1), (v, \texttt{undef}, \texttt{undef})) \notin A\}.
\end{aligned}$$

The definition of the $\mathit{next_with_order}(A, \sqsubseteq_r)$ operation implies that the event which can be added to the G counter should be more dominant than any other events in A compared by the partial order $\sqsubseteq_r$. For example, if an event (v, r, n) is successfully added to the component A of the G counter (i.e. $(v, r, n) \in \mathit{next_with_order}(A, \sqsubseteq_r)$ and $A' = A \cup \{(v, r, n)\}$) and the operation $\mathit{maximal}(A', \sqsubseteq_r)$ is invoked, then the result of the $\mathit{maximal}(A', \sqsubseteq_r)$ should not have any events that contain v and r except the newly added (v, r, n).

The following shows the definition of the two operations for the G counter:

$\texttt{increase} : \mathcal{E} \times \mathcal{X}_s \to \mathcal{X}_s$
$\texttt{increase}(e, (A, L, \mathcal{C}, K)) = \mathit{maximal_orset}(\texttt{insert}(e, (\bot, e.v), (A, L, \mathcal{C}, K)), \sqsubseteq_r)$

$\texttt{read} : 2^{\mathcal{H}} \times \mathcal{X} \to 2^{\mathcal{H}} \times 2^{\mathcal{U}} \cup \{\texttt{undef}\}$
$\texttt{read}(C_i, (A, L, \mathcal{C}, K)) =$
  if (there is $C_i' \in \mathcal{C}$ with $C_i \subseteq C_i'$) then
    let C_i' be such an element in $\mathcal{C}$ in
    $(C_i', \{\mathit{counter_evaluation}(v, K(C_i')((\bot, v))) \mid K(C_i')((\bot, v)) \neq \{\}\})$
  else undef

Instead of storing the counter value directly into the OR_{PC} set, the G counter stores the node-specific value (see $(\bot, e.v)$ from the definition of `increase`()) to the internal set. This node-specific value makes sure that every replica in the system will insert the same value for each `increase`() operation, and it guarantees that the internal OR_{PC} set of the G counter is a singleton set. The real counter value (how many `increase`() operations were invoked) is stored inside of the provenance of the node-specific value as an event (the identifier of the n-th `increase`() operation). The partial order $\sqsubseteq_r$ of the events, defined in the previous section, guarantees that there is no event-history which contains the event inserted at the same replica (no duplicated count). In order to calculate the real counter value, the `read`() operation uses the following operation:

$$\mathit{counter_evaluation}(v, p) = \sum_{(v, _, n) \in \{e \mid \exists d \in p.\ (e, _) \in d\}} n.$$

The $\mathit{counter_evaluation}(v, p)$ operation finds all events that are inserted at v from the event-histories in p and accumulates every n value from them. The provenance p here does not contain the duplicated counter due to the partial order $\sqsubseteq_r$ and the $\mathit{maximal_orset}()$ operation from `increase`().

6.4 Observed-Remove Bag

The Observed-Remove (OR) bag is a similar data type with the OR_{PC} set, except it can store duplicated values (as a multiset). One of the benefits for being a multiset is that it is easy to get the correct results for aggregate operations. For those operations, it is vital to know the information about how many duplicated values are inserted to the input.

Similar to other client-level data types, the OR bag is a simple OR_{PC} set such that it supports the following two operations:

$$\texttt{add} : \mathcal{E} \times \mathcal{U} \times \mathcal{X}_s \to \mathcal{X}_s$$
$$\texttt{add}(e, u, (A, L, \mathcal{C}, K)) = \texttt{insert}(e, u, (A, L, \mathcal{C}, K))$$

$$\texttt{remove} : \mathcal{U} \times \mathcal{X}_s \to \mathcal{X}_s$$
$$\texttt{remove}(u, (A, L, \mathcal{C}, K)) = \texttt{delete}(u, (A, L, \mathcal{C}, K))$$

$$\textit{next_with_order}(A, \sqsubseteq_i) = \{((v, r, n+1), (v, \texttt{undef}, \texttt{undef})) \mid ((v, r, n), (v, \texttt{undef}, \texttt{undef})) \in A \wedge ((v, r, n+1), (v, \texttt{undef}, \texttt{undef})) \notin A\}.$$

Since the OR bag has the same functionality as the OR_{PC} set except handling the duplicated values, the definition of the `add`() and `remove`() operations are designed to invoke the `insert`() and `delete`() operations of the OR_{PC} set, respectively. The definition of the $\textit{next_with_order}(A, \sqsubseteq_i)$ operations is also identical to the one of the *next*() operation of the OR_{PC} set.

$$\texttt{read} : 2^{\mathcal{H}} \times \mathcal{X} \to 2^{\mathcal{H}} \times 2^{(\mathcal{U} \times \mathbb{N})} \cup \{\texttt{undef}\}$$
$$\begin{aligned}\texttt{read}(C_i, (A, L, \mathcal{C}, K)) = \ & \text{if (there is } C'_i \in \mathcal{C} \text{ with } C_i \subseteq C'_i) \text{ then} \\ & \quad \text{let } C'_i \text{ be such an element in } \mathcal{C} \text{ in} \\ & \quad (C'_i, \{(u, n) \mid K(C'_i)(u) \neq \{\} \wedge u \in \mathcal{U} \wedge n = |K(C'_i)(u)|\}) \\ & \text{else } \texttt{undef}\end{aligned}$$

The result of the `read`() operation of the OR bag is a set of pairs, and each pair consists of two element: a value and a positive integer number. In order to avoid storing duplicated value multiple times, our implementation of the OR bag stores the pair of the added value and the number that indicates how many times it has been added.

Chapter 7

Operations for Aggregation

In this chapter, we extend our programming language with operations for aggregation. There are two operations for aggregation in our language: `agg_sum`() and `agg_cnt`(), both of which take an OR bag as their input. The `agg_sum`() operation adds all the elements in the input OR bag, and the `agg_cnt`() operation counts the number of elements in the bag. For example, assume that an OR bag B has the following elements:

$$\{(5, 2), (7, 1), (11, 3)\}.$$

Recall that an element in an OR bag is a pair of a value and its multiplicity. The result of `agg_sum`(B) is 50 ($5+5+7+11+11+11$) and the result of `agg_cnt`(B) is 6 ($2+1+3$).

Implementing aggregation operations for weakly consistent distributed systems is challenging because different replicas in the system might store the different values in the corresponding input OR bag at the same time. As a result, the aggregation operations at these replicas produce different results. However, in order to ensure strong convergence, these replicas should have the identical state when they exchange their states and merge them. For instance, assume that an OR bag B has the following elements in *replica a*:

$$\{(5, 1), (7, 1), (11, 3)\},$$

and B has the following elements in *replica b*:

$$\{(5, 2), (7, 1), (11, 2)\}.$$

The results of `agg_sum`(B) at *replica a* and *replica b* are 45 and 39, respectively. When *replica b* receives the result of `agg_sum`(B) from *replica a* (45), it is difficult to merge the received 45 with the existing 39, without knowing how 45 is calculated.

Our solution is to make the aggregation operations use the only values that are stored with the complete-dot set of the input OR bag, not the entire values of the OR bag. Recall that every value stored with the complete-dot set also exists in the state of an OR_{PC} corresponding to some consistent execution (defined and proved in Section 5.5).

We first explain the details of aggregation operations supported by our programming language, including their formal definitions. Then, we formally prove that dataflow programs with those operations still satisfy the theoretical properties described and proved in Chapter 5.

7.1 Details of Aggregation Operations

The main challenge of implementing aggregation operations in our programming language is that two replicas should have the identical result of the operations after exchanging their results and merging them. However, due to the characteristics of weakly consistent distributed systems, the input parameters of the operations in two replicas might have different values and this can make the replicas have different results of the operations. This implies that the results of the operations should contain the information about how they are computed including what values are used for the computation and this information is used for getting the identical result when two different results are merged.

In order to provide all the necessary information about how the results of the operations are computed, we need to resolve the following three problems:

1. The results should contain the information about input values.

2. Two different results should be comparable in order to decide which result contains more values than the other.

3. Instead of using the entire content of the input OR bag, the operations should use a consistent part of the content. When we consider the content of an OR bag, we can assume that there is an execution such that its last system configuration has the bag with the identical content. In Section 5.5, we proved that such execution has the corresponding consistent execution that is reproduced by the same operation history of the execution but performed in a single node. The consistency here means that the part exists in such corresponding consistent execution.

Before we introduce the definitions of the aggregation operations, we explain how those three problems mentioned above are resolved. Firstly, in order to express what values are involved in the result of the operation, we define the following event:

Definition 7.1.1. An *group event* is a tuple $e = (v, r, H) \in \mathcal{V} \times \mathcal{R} \times \mathcal{H}$ such that H is a set of event-histories. We use $\mathcal{G}$ to denote the set of all group events.

Recall that an event (v, r, n) is a tuple of node id, replica id, and non-negative integer, and it acts as the identifier of the n-th `add`() operation invoked on the node v at the

replica r. Instead of the integer n, a group event has a set of event-histories H and this set indicates which event-histories are involved to compute the result of the aggregation. Furthermore, since every event-history has the corresponding value inserted at an OR bag, H can also provide the information about what values are involved in the result.

Secondly, the following partial order relation is used to compare two group events:

Definition 7.1.2. The *group-event order* $\sqsubseteq_g$ is the following partial order on the group events:

$$e \sqsubseteq_g e' \iff e.v = e'.v \wedge e.H \subseteq e'.H.$$

Two group events are comparable when their node ids are identical. The order of these two is determined by their event-history sets (i.e., $e.H$ and $e'.H$). This order can determine which result of the operation has already included another result. Note that this order can also be used with the two utility operations, *maximal*() and *maximal_orset*(), defined in the previous chapter.

Finally, the last problem is related to the input parameters of the operations. Using some consistent contents of the inputs can reduce the difficulty of generating the identical result by merging two different results. Recall that the following operation is defined in Section 5.5:

$$valid_cds_collection(v, t, H) = \mathcal{D}.$$

This operation is used by the $\texttt{read}_c()$ operation in order to return a set of values such that each value has a dot in its provenance identical to the one from the corresponding consistent execution. The main idea of the $\texttt{read}_c()$ operation is that each valid complete-dot set in the result of *valid_cds_collection*() operation exists in such corresponding consistent execution. In order to retrieve the maximum complete-dot set, we define the following operation:

$$fn_{cds_max}(v, H) = D$$
$$\text{where } D = max(\mathcal{D}) \wedge \mathcal{D} = valid_cds_collection(v, t, H).$$

We define the following operation in order to generate a provenance contains the group-event that includes all event-histories in a given provenance:

$$gen_ge_prove(v, r, p) = \{\{((v, r, \bigcup p), \bot)\}\}$$

Note that every event-history that contains a group-event can be generated by only the *gen_ge_prove*() operation and it has $\bot$ as its dataflow path. When the aggregation operations update the dataflow path of those event-histories, they update the dataflow paths of all event-histories in $e.H$ instead of $\bot$.

Assume that v is a node id and r is a replica id where operations are invoked. We define operations for aggregation as follows:

$$\texttt{agg_sum} : \mathcal{X} \to \mathcal{X}$$

$$\texttt{agg_sum}((A, L, \mathcal{C}, K)) = \Big(A,\ L,\ \mathcal{C},\ (\lambda C_i \in \mathcal{C}.\ K'(C_i))\Big)$$

$$\left[\begin{array}{ll} \text{where} & D(C_i) = \mathit{fn}_{\mathit{cds_max}}(v, C_i), \\ & U'(C_i) = \{u \in \mathcal{U} \mid K(C_i)(u) \cap D(C_i) \neq \emptyset\}, \\ & p(C_i) = \bigotimes_{u' \in U'(C_i)} \{K(C_i)(u') \cap D(C_i)\}, \\ & K'(C_i) = \left[\sum_{u' \in U'(C_i)} u' \mapsto p(C_i) \otimes \mathit{gen_ge_prove}(v, r, p(C_i))\right] \end{array}\right]$$

$$\texttt{agg_cnt} : \mathcal{X} \to \mathcal{X}$$

$$\texttt{agg_cnt}((A, L, \mathcal{C}, K)) = \Big(A,\ L,\ \mathcal{C},\ (\lambda C_i \in \mathcal{C}.\ K'(C_i))\Big)$$

$$\left[\begin{array}{ll} \text{where} & D(C_i) = \mathit{fn}_{\mathit{cds_max}}(v, C_i), \\ & U'(C_i) = \{u \in \mathcal{U} \mid K(C_i)(u) \cap D(C_i) \neq \emptyset\}, \\ & p(C_i) = \bigotimes_{u' \in U'(C_i)} \{K(C_i)(u') \cap D(C_i)\}, \\ & K'(C_i) = [|U'(C_i)| \mapsto p(C_i) \otimes \mathit{gen_ge_prove}(v, r, p(C_i))] \end{array}\right]$$

The `agg_sum`() and `agg_cnt`() operations have the similar computation steps and the only difference is how aggregated values are computed; the former adds the elements in the input and the latter counts the number of elements in the input. The computation steps of the operations are applied for every set of event-histories C_i in $\mathcal{C}$ and those steps can be described as follows:

- Firstly, the operation computes a complete-dot set for v and C_i by using the $\mathit{fn}_{\mathit{cds_max}}()$ operation. The complete-dot set here contains information about the dots in the corresponding consistent execution.
- Secondly, among the values in the input OR bag, the operation selects a set of values such that each value has a dot in its provenance identical to the one from the complete-dot set computed from the first step.
- Thirdly, the operation generates a provenance by performing the $\otimes$ operation repeatedly over a set of provenances such that those provenances are from the values in the set from the second step.
- Finally, the operation updates C_i's provenance store $K'(C_i)$ having a single relation such that the aggregated value (either a summation or a count) is associated to the provenance computed by performing the $\otimes$ operation with the provenance from the third step $p(C_i)$ and its group-event provenance $\mathit{gen_ge_prove}(v, r, p(C_i))$.

7.2 Proof of Theorem 5.5.1 for Aggregation Operations

In this section, we prove that including the newly added aggregation operations does not break Theorem 5.5.1. The original proof of the theorem without aggregate operations uses six lemmas (Lemma 5.6.2, 5.6.6, 5.6.9, 5.6.10, 5.6.13, and 5.6.11) and each lemma, except 5.6.10 and 5.6.11, contains the case analysis on the rules used for transformations, including the ones for the dataflow operations. We reuse all these reasoning steps when we prove the extended version of the theorem with aggregation operations in this section. The only thing that we will have to do is to extend the case analysis in the four lemmas so that it also covers newly added rules for aggregation operations.

Before we introduce the rules for the aggregation operations, we define two utility operations:

$$
\begin{aligned}
&\mathit{fn}_{agg_sum}(S, H, v) = S' \\
&\qquad \text{where} \quad S' = [u'' \mapsto p(H) \otimes \mathit{gen_ge_prove}(v, r, p(H))] \\
&\qquad\qquad\quad u'' = \sum_{u' \in U'(H)} u' \\
&\qquad\qquad\quad p(H) = \bigotimes_{u' \in U'(H)} \{S(u') \cap D(H)\} \\
&\qquad\qquad\quad U'(H) = \{u \in \mathcal{U} \mid S(u) \cap D(H) \neq \emptyset\} \\
&\qquad\qquad\quad D(H) = \mathit{fn}_{cds_max}(v, H),
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{fn}_{agg_cnt}(S, H, v) = S' \\
&\qquad \text{where} \quad S' = [u'' \mapsto p(H) \otimes \mathit{gen_ge_prove}(v, r, p(H))] \\
&\qquad\qquad\quad u'' = |U'(H)| \\
&\qquad\qquad\quad p(H) = \bigotimes_{u' \in U'(H)} \{S(u') \cap D(H)\} \\
&\qquad\qquad\quad U'(H) = \{u \in \mathcal{U} \mid S(u) \cap D(H) \neq \emptyset\} \\
&\qquad\qquad\quad D(H) = \mathit{fn}_{cds_max}(v, H).
\end{aligned}
$$

We define the following rule for the $\Longrightarrow_s$ relation that transforms node states through the aggregation operations:

$$
\frac{\begin{array}{l}(X, (v, self, \texttt{undef})) \Longrightarrow_{t_h} X' \\ X''.A = X'.A \qquad X''.L = X'.L \qquad X''.\mathcal{C} = X'.\mathcal{C} \\ X''.K = \lambda C_i \in X'.\mathcal{C}.\ X''.K(C_i) = \mathit{fn}_{agg_*}(X'.K(C_i), C_i, v)\end{array}}{(v, \texttt{agg_}*, [w \mapsto X]) \Longrightarrow_s X''} \ \mathrm{F}(v)_{agg_*}
$$

Every $*$ here indicates that this rule can be used for either `agg_sum`() or `agg_cnt`(), and this rule can also be applied for transforming shadow node states.

Firstly, Lemma 5.6.2 proves that every reachable system configuration S has a corresponding shadow system configuration S_s (i.e. $S_s \sim S$) and every reachable replica configuration R also has a corresponding shadow replica configuration R_s (i.e. $R_s \sim R$). Recall that the following correspondence between replica states s and shadow replica states ss is the key behind the $\sim$ relation: for every node v in $G.V$, if $s(v) = (A, L, \mathcal{C}, K)$ and $ss(v) = (A_s, L_s, \mathcal{C}_s, K_s)$, then

$$A = A_s, \quad A = L_s, \quad \mathcal{C} = \mathcal{C}_s, \quad \text{and} \quad K(C_i)(u) \subseteq K_s(C_i)(u) \ \text{ for all } C_i \in \mathcal{C} \text{ and } u \in \mathcal{U}.$$

The last component $K(C_i)(u) \subseteq K_s(C_i)(u)$ implies that the corresponding shadow replica state contains every value stored in a normal replica state. It holds because the `delete`() and `filter`() operations are ignored for shadow replica states. However, this comparison cannot be applied for the results of the aggregation operations directly. The reason is that the values used for the operations are removed and only their events are stored with the aggregated value. Therefore, the $\sim$ relation should have a way to check whether the aggregated value in the shadow state includes all the input values that are used to compute the aggregated value in the normal state. The partial order $\sqsubseteq_g$ can be used for this kind of comparison. Now, we modify the last component of the $\sim$ relation for the results of aggregation operations as follows: for all $C_i \in \mathcal{C}$ and $u \in \mathcal{U}$

$$\forall d \in K(C_i)(u).\ \begin{cases} d \in K_s(C_i)(u) & \text{if } d \cap \mathcal{G} = \emptyset \\ \forall e \in d \cap \mathcal{G}.\ (\exists u' \in \mathcal{U}.\ \exists d' \in K_s(C_i)(u').\ \exists e' \in d' \cap \mathcal{G}.\ e \sqsubseteq_g e') & \text{otherwise} \end{cases}$$

The definition of this new $\sim$ relation indicates that for every aggregated value in a normal execution, there exists an aggregated value in its corresponding shadow execution such that the aggregated value in the shadow execution includes all the input values of the aggregation in the normal execution. The proof of Lemma 5.6.2 for the aggregation operations uses this $\sim$ relation described above.

Proof of Lemma 5.6.2. Consider shadow replica configuration R_s, and normal replica configuration R. Assume that

$$R_s \sim R.$$

Suppose that

$$R = (\alpha, s, n) \quad \text{and} \quad R \Longrightarrow_r (\alpha', s', n') \text{ by DFlow}.$$

Let v' be a node to store the result of the DFlow rule and $(A, L, \mathcal{C}, K) = s(v')$. Then, by the definition of DFlow,

$$\alpha' = \alpha, \quad n' = n, \quad (v', F(v'), s|_{pred(v')}) \Longrightarrow_s (A', L', \mathcal{C}', K'), \quad \text{and} \quad s' = s[v' \mapsto X''],$$

where $X'' = (A, L, \mathcal{C}, K) \sqcup (A', L', \mathcal{C}', K')$.

By the definition of the DFlow rule, the DFlow rule has two steps: i) executing a dataflow operation using a rule for the $\Longrightarrow_s$ relation, and ii) combining the output

of the rule with the existing $\mathrm{OR}_{\mathrm{PC}}$ of the node using the $\sqcup$ operation. Thus, if we let R' be the output replica configuration of the DFlow rule, then there is a temporary replica configuration R^* between the two steps such that

$$R \Longrightarrow_{step1} R^* \Longrightarrow_{step2} R'.$$

where R^* is constructed from R by the executed rule for $\Longrightarrow_s$. Let R^* and R'_s be shadow replica configurations generated by executing the shadow versions of the two steps mentioned above on R_s:

$$R_s \Longrightarrow_{step1} R^*_s \Longrightarrow_{step2} R'_s.$$

Updating dataflow paths is a simple and automatic process for the rules for $\Longrightarrow_s$. Therefore, we skip the comparison of dataflow paths that appears when we check whether two event-histories have an identical dataflow path.

We need to show $R^*_s \sim R^*$ and $R'_s \sim R'$. We will only explain the former, because the proof of the latter has been proved in Chapter 5 and is not related to the aggregation operations. The proof of $R^*_s \sim R^*$ is by the case analysis on the rule for $\Longrightarrow_s$ used in the first step. In this proof, we handle only the case for aggregation. Assume that the $\mathrm{F}(v)_{agg_*}$ rule is applied to R in the first of the two steps mentioned above. Recall that this rule executes either `agg_sum`() or `agg_cnt`() operation on the OR bag of a node w, if we let $pred(v') = \{w\}$. The rule updates the node state of v' only as a result. We will prove $R^*_s \sim R^*$ by showing that R^*_s, constructed by applying the $\mathrm{F}(v)_{agg_*}$ rule to R_s, contains the updated OR bag of v' satisfying the $\sim$ relation with the updated OR bag of v' in R^* (i.e., $ss(v') \sim s(v')$). This is because for every node v in $G.V \setminus \{v'\}$, $ss(v) \sim s(v)$ is already satisfied by the assumption $R_s \sim R$.

Suppose that

$$(v', \texttt{agg_*}, s|_{\{w\}}) \Longrightarrow_s X' \text{ by } \mathrm{F}(v)_{agg_*}.$$

Let $(A, L, \mathcal{C}, K) = s(w)$. Then, by the definition of $\mathrm{F}(v)_{agg_*}$,

$$X' = (A', L', \mathcal{C}', K')$$

where

$$A' = A, \quad L' = L, \quad \mathcal{C}' = \mathcal{C}, \quad \text{and} \quad \forall C_i \in \mathcal{C}.\ K'(C_i) = fn_{agg_*}(K(C_i), C_i, v').$$

Meanwhile, since $R_s \sim R$, there exists a shadow replica state ss such that

$$R_s = (\alpha, ss, n) \quad \text{and} \quad ss \sim s.$$

The condition $ss \sim s$ here implies that for some K_s,

$$ss(w) = (A, A, \mathcal{C}, K_s) \quad \text{and} \quad K(C_i)(u) \subseteq K_s(C_i)(u) \ \text{ for all } C_i \text{ in } \mathcal{C} \text{ and } u \text{ in } \mathcal{U}.$$

Then, by the $\mathrm{F}(v)_{agg_*}$ rule for a shadow replica state,

$$pred(v') = \{w\} \quad \text{and} \quad (v', \mathtt{agg_*}, ss|_{\{w\}}) \Longrightarrow_s X'_s$$

where $X'_s = (A, A, \mathcal{C}, K'_s)$ and $\forall C_i \in \mathcal{C}.\ K'_s(C_i) = fn_{agg_*}(K_s(C_i), C_i, v')$. Furthermore, we can claim that

$$R^*_s \sim R^* \quad \text{and} \quad ss[v' \mapsto X'_s] \sim s[v' \mapsto X'].$$

This is because

$$\begin{aligned}
&\forall C_i \in \mathcal{C}, u \in \mathcal{U}.\ K(C_i)(u) \subseteq K_s(C_i)(u) \\
\Longrightarrow\ &\forall C_i \in \mathcal{C}.\ \forall d' \in fn_{agg_*}(K(C_i), C_i, v')(u').\ \forall d'' \in fn_{agg_*}(K_s(C_i), C_i, v')(u'').\ e' \sqsubseteq_g e'' \\
&\quad \text{for some } e' \text{ and } e'' \text{ such that} \\
&\qquad \{e'\} = d' \cap \mathcal{G}, \{e''\} = d'' \cap \mathcal{G}, \\
&\qquad u' \text{ is the aggregated value from } fn_{agg_*}(K(C_i), C_i, v'), \text{ and} \\
&\qquad u'' \text{ is the aggregated value from } fn_{agg_*}(K_s(C_i), C_i, v') \\
\Longrightarrow\ &\forall C_i \in \mathcal{C}, \forall d \in K'(C_i)(u).\ \forall e \in d \cap \mathcal{G}.\ (\exists u' \in \mathcal{U}.\ \exists d' \in K'_s(C_i)(u').\ \exists e' \in d' \cap \mathcal{G}.\ e \sqsubseteq_g e').
\end{aligned}$$

We can similarly prove that if $(v', \mathtt{agg_*}, ss|_{pred(v')}) \Longrightarrow_s X'_s$ by the $\mathrm{F}(v)_{agg_*}$ rule for shadow replica configurations, there exists X' such that

$$ss[v' \mapsto X'_s] \sim s[v' \mapsto X']$$

and

$$(v', \mathtt{agg_*}, s|_{\{w\}}) \Longrightarrow_s X'$$

by the $\mathrm{F}(v)_{agg_*}$ rule for normal replica configurations. □

Secondly, Lemma 5.6.6 proves that a set of all dataflow paths in a dot is unanimous, and it shows the meaning of the unanimous result of the set: it should consist of all the paths of some complete dataflow tree for v'.

Proof of Lemma 5.6.6. Suppose that

$$R = (\alpha, s, n) \quad \text{and} \quad R \Longrightarrow_r (\alpha', s', n') \text{ by DFlow.}$$

Let $(A, L, \mathcal{C}, K) = s(v')$. Then, by the definition of the DFlow rule,

$$\alpha' = \alpha, \quad n' = n, \quad (v', F(v'), s|_{pred(v')}) \Longrightarrow_s (A', L', \mathcal{C}', K'), \quad \text{and} \quad s' = s[v' \mapsto X''],$$

where $X'' = (A, L, \mathcal{C}, K) \sqcup (A', L', \mathcal{C}', K')$.

From the definition of the DFlow rule, we could consider that the DFlow rule has two steps: executing a dataflow operation through $\Longrightarrow_s$ rules, and combining the

output of a $\Longrightarrow_s$ rule with the existing $\mathrm{OR}_{\mathrm{PC}}$ of the node using the $\sqcup$ operation. We assume that, if we let R' be the output replica configuration of the DFlow rule, then there is a temporary replica configuration R^* between the two steps such that

$$R \Longrightarrow_{step1} R^* \Longrightarrow_{step2} R'.$$

We need to show that, if we let $R^* = (\alpha^*, s^*, n^*)$ and $R' = (\alpha', s', n')$, then every dot d^* in $s^*(v')$ and every dot d' in $s'(v')$ satisfy the predicate $\Phi(d^*, v', s^*)$ and the predicate $\Phi(d', v', s')$, respectively. We will only explain the former (related to R^*), because the proof of the latter (related to R') has been proved in Chapter 5 and is not related to the aggregation operations.

The proof related to R^* is by the case analysis on the rule for $\Longrightarrow_s$ used in the first step. In this proof, we handle only the case for aggregation. Assume that the $\mathrm{F}(v)_{agg_*}$ rule is applied to R in the first of the two steps mentioned above. Recall that this rule executes the `agg_*()` operation on the OR bag of a node w from R, if we let v' be a node to store the result of the operation and $pred(v') = \{w\}$. The rule updates the node state of v' only as a result. We will prove that, if we let $X'' = s^*(v')$, then every dot d'' in X'' satisfies the predicate $\Phi(d'', v', s^*)$. This is because for every node v in $G.V \setminus \{v'\}$, if we let $X = s(v)$, then every dot d in X satisfies the predicate $\Phi(d, v, s)$ by the induction hypothesis. Suppose that

$$(v', \texttt{agg_*}, s|_{\{w\}}) \Longrightarrow_s X'' \text{ by } \mathrm{F}(v)_{agg_*}.$$

Let $(A, L, \mathcal{C}, K) = s(w)$. Then, by the definition of $\mathrm{F}(v)_{agg_*}$,

$$((A, L, \mathcal{C}, K), (v', self, \texttt{undef})) \Longrightarrow_{t_h} (A', L', \mathcal{C}', K') \quad \text{and} \quad X'' = (A'', L'', \mathcal{C}'', K'')$$

where

$$A'' = A', \quad L'' = L', \quad \mathcal{C}'' = \mathcal{C}', \quad \text{and} \quad \forall C_j \in \mathcal{C}', u \in \mathcal{U}.\ K''(C_j)(u) = fn_{agg_*}(K'_s(C_j), C_j, v').$$

Meanwhile, since S_k satisfies the claimed property by induction hypothesis, for every C_i in $\mathcal{C}$ and u in $\mathcal{U}$, every dot d in $K(C_i)(u)$ satisfies the predicate $\Phi(d, w, s)$. Then, we can claim that, for every C_j in $\mathcal{C}'$ and u in $\mathcal{U}$, every dot d'' in $K''(C_j)(u)$ satisfies the predicate $\Phi(d'', v', s^*)$. This is because

1. by the definition of the $\mathrm{F}(v)_{agg_*}$ rule, every C_j in $\mathcal{C}'$ is computed from C_i in $\mathcal{C}$ such that $C_j = \{(e, (v', t_h, \texttt{undef})) \mid (e, t_h) \in C_i\}$;

2. by the definition of the $\mathrm{F}(v)_{agg_*}$ rule, for every u in $\mathcal{U}$, every dot d'' in $K''(C_j)(u)$ is computed from d in $K(C_i)(u)$ such that $d'' = \{(e, (v', t_h, \texttt{undef})) \mid (e, t_h) \in d\}$ where C_j is computed from C_i;

3. if we let T be the set of all dataflow paths in d and T'' be the set of all dataflow paths in d'', then
$$T'' = \{(v', t, \texttt{undef}) \mid t \in T\};$$

4. every dataflow path in T'' is v'-oriented and compatible with P;

5. if we let t_u be the unanimous result constructed from T, then T'' is *unanimous*, and $(v', t_u, \texttt{undef})$, the unanimous result constructed from T'', is complete and v'-oriented.

□

Thirdly, Lemma 5.6.9 proves that computing the collection of valid complete-dot sets guarantees the existence of values with dots from the collection on every shadow node state.

Proof. The $\mathrm{F}(v)_{agg_*}$ rule executes the $\texttt{agg_*}()$ operation on the OR bag of a node w from R, if we let $pred(v') = \{w\}$. We will prove that, if we let $X'' = ss^*(v')$ and $(A''_s, A''_s, \mathcal{C}''_s, K''_s)$ be the components of X'', then every C_k in $\mathcal{C}''_s$ satisfies the predicate $\Phi(C_k, v', ss^*)$. Suppose that

$$(v', \texttt{agg_*}, ss|_{\{w\}}) \Longrightarrow_s X'' \text{ by } \mathrm{F}(v)_{agg_*}.$$

Let $(A_s, A_s, \mathcal{C}_s, K_s) = ss(w)$. Then, by the definition of the $\mathrm{F}(v)_{agg_*}$ rule,

$$((A_s, A_s, \mathcal{C}_s, K_s), (v, \mathit{self}, \texttt{undef})) \Longrightarrow_{t_h} (A'_s, A'_s, \mathcal{C}'_s, K'_s) \quad \text{and} \quad X'' = (A''_s, A''_s, \mathcal{C}''_s, K''_s)$$

where

$$A''_s = A'_s, \qquad \mathcal{C}''_s = \mathcal{C}'_s, \qquad \text{and} \qquad \forall C_j \in \mathcal{C}'_s.\ K''_s(C_j) = \mathit{fn}_{agg_*}(K'_s(C_j), C_j, v').$$

Meanwhile, since S_k satisfies the claimed property by induction hypothesis, every C_i in $\mathcal{C}_s$ satisfies the predicate $\Phi(C_i, w, ss)$. Then, we can claim that every C_k in $\mathcal{C}''_s$ satisfies the predicate $\Phi(C_k, v', ss^*)$. This is because, if we let $\mathcal{D}$ be the collection of valid complete-dot sets for w and C_i, $\mathcal{D}''$ be the collection of valid complete-dot sets for v'

and C_k and C_k is computed from C_i, then

$$
\begin{aligned}
& C_k = \{(e, (v', t, \texttt{undef})) \mid (e, t) \in C_i\} \\
\wedge\, & \mathcal{D}'' = \{\{\{(e, (v', t, \texttt{undef})) \mid (e, t) \in d\} \mid d \in D\} \mid D \in \mathcal{D}\} \\
\wedge\, & K''_s(C_k) = [u'' \mapsto p''] \\
& \quad \text{where} \\
& \quad u'' \text{ is the aggregation of the values in } ss(w) \text{ that have their dots as members of } max(\mathcal{D}) \\
& \quad \text{and } p'' \text{ is computed by invoking } \otimes \text{ operations repeatedly for those values' provenances} \\
\wedge\, & \forall D \in \mathcal{D}, d \in D.\ \exists u' \in \mathcal{U}.\ d \in K_s(C_i)(u') \qquad (\text{from } \Phi(C_i, w, ss)) \\
\Longrightarrow & \\
& \forall D'' \in \mathcal{D}'', d'' \in D''.\ d'' \in K''_s(C_k)(u'').
\end{aligned}
$$

□

Lastly, Lemma 5.6.13 proves that for every value whose dot is in the valid complete-dot sets on shadow system configurations, there is a completed and consistent execution whose last system configuration contains the identical value and dot.

Proof. For the $\mathrm{F}(v)_{agg_*}$ rule, suppose that

$$pred(v') = \{w\} \quad \text{and} \quad (v', \texttt{agg_*}, ss|_{pred(v')}) \Longrightarrow_s X'' \text{ by } \mathrm{F}(v)_{agg_*}.$$

Let $(A_s, A_s, \mathcal{C}_s, K_s) = ss(w)$. Then, by the definition of $\mathrm{F}(v)_{agg_*}$,

$$((A_s, A_s, \mathcal{C}_s, K_s), (v', \mathit{self}, \texttt{undef})) \Longrightarrow_{t_h} (A'_s, A'_s, \mathcal{C}'_s, K'_s) \quad \text{and} \quad X'' = (A''_s, A''_s, \mathcal{C}''_s, K''_s)$$

where

$$A''_s = A'_s, \qquad \mathcal{C}''_s = \mathcal{C}'_s, \qquad \text{and} \qquad \forall C_j \in \mathcal{C}'_s.\ K''_s(C_j) = \mathit{fn}_{agg_*}(K'_s(C_j), C_j, v').$$

Similar to other transformation, we only consider what are related to the $\mathrm{F}(v)_{agg_*}$ rule: $ss(w)$ and $X''(= S_{k+1/2}(r').ss(v'))$. Meanwhile, since $ss(w)$ is from τ_k that satisfies the claimed property by induction hypothesis, if we let

- C_i be an element of $\mathcal{C}_s$,
- $\mathcal{D}$ be the collection of valid complete-dot sets for w and C_i,
- D be a valid complete-dot set in $\mathcal{D}$, and
- U_c be a set of values such that $K_s(C_i)(u) \cap D \neq \emptyset$ for every $u \in U_c$,

then for every value-dot pair (u, d) in $U_c \times (K_s(C_i)(u) \cap D)$ and its computation tree b, there exists an S_I-starting consistent execution τ'_k that satisfies **Value-Dot**. Now, we can claim that, if we let

- C_k be an element of $\mathcal{C}''_s$,
- $\mathcal{D}''$ be the collection of valid complete-dot sets for v' and C_k,
- D'' be a valid complete-dot set in $\mathcal{D}''$, and
- U''_c be a set of values such that $K''_s(C_k)(u'') \cap D'' \neq \emptyset$ for every $u'' \in U''_c$,

then for every value-dot pair (u'', d'') in $U''_c \times (K''_s(C_k)(u'') \cap D'')$ and its computation tree b'', there exists an S_I-starting completed consistent execution τ''_{k+1} that satisfies **Value-Dot**. This is because

$$
\begin{aligned}
&\mathcal{D}'' = \{\{\{(e, (v', t, \texttt{undef})) \mid (e, t) \in d\} \mid d \in D\} \mid D \in \mathcal{D}\} \\
&\wedge C_k = \{(e, (v', t, \texttt{undef})) \mid (e, t) \in C_i\} \\
&\wedge K''_s(C_k) = [u'' \mapsto p''] \\
&\qquad \text{where} \\
&\qquad u'' \text{ is the aggregation of the values in } ss(w) \text{ that have their dots as members of } max(\mathcal{D}) \\
&\qquad \text{and } p'' \text{ is computed by invoking } \otimes \text{ operations repeatedly for those values' provenances} \\
&\wedge U''_c = \{u''\} \\
&\wedge \tau_{k+1} = \tau_k^\frown[S_{k+1}] \\
&\wedge \mathit{computation_tree}((u'', \{(e, (v', t, \texttt{undef})) \mid (e, t) \in d\}), \tau_{k+1}) \\
&\qquad = ((u'', \{(e, (v', t, \texttt{undef})) \mid (e, t) \in d\}), \mathit{computation_tree}((u, d), \tau_k), \texttt{undef}) \\
&\wedge \tau''_{k+1} = \text{processing the } \Longrightarrow_{\mathit{dflow}} \text{ rules including the } \mathrm{F}(v)_{\mathit{agg_*}} \text{ rule from } \tau'_k \\
\Longrightarrow& \\
&(u'', \{(e, (v', t, \texttt{undef})) \mid (e, t) \in d\}) \in S''_F(r').s(v') \\
&\wedge \mathit{computation_tree}((u'', \{(e, (v', t, \texttt{undef})) \mid (e, t) \in d\}), \tau''_{k+1}) \\
&\qquad = ((u'', \{(e, (v', t, \texttt{undef})) \mid (e, t) \in d\}), \mathit{computation_tree}((u, d), \tau'_k), \texttt{undef}).
\end{aligned}
$$

□

7.3 The Example Program for Aggregation Operations

In this section, we introduce an example program for aggregation operations: the average calculator. The input node, the OR bag data type, stores values to get the average. Two intermediate nodes, the OR bag data types, store two aggregation results: one for the number of values in the input bag (the `agg_cnt`() operation) and the other for the sum of values in the input bag (the `agg_sum`() operation). The other intermediate node,

the OR bag data type, combines the two aggregate results. The final node, the OR bag data type, calculates the division for the average.

The following example codes demonstrate how the average calculator is implemented:

Listing 7.1: The average calculator

```
1  // InputData : ORBag[Integer]
2  // SumOfValues : ORBag[Integer]
3  // NumOfValues : ORBag[Integer]
4  // Intermediate : ORBag[(SumOfValues,NumOfValues)]
5  // Avg : ORBag[Float]
6
7  SumOfValues := agg_sum(InputData);
8
9  NumOfValues := agg_cnt(InputData);
10
11 Intermediate := SumOfValues × NumOfValues;
12
13 // f1 =
14 //      fun ((SumOfValues,NumOfValues)) ->
15 //          SumOfValues / NumOfValues
16 map(f1, Intermediate, Avg);
17
18 read_c(Avg);
```

The code line 18 in Listing 7.1 ensures that the final result is computed using the contents of the node *SumOfValues* and the node *NumOfValues* from the same input data. This glitch-free outcome is guaranteed by the `read`$_c$() operation.

Note that our example program can be generalised for other statistical measures such as averages of sums of squares, cubes, etc. by applying the `map`() operation before the `agg_sum`() operation. The following example codes show how we can calculate the average of sums of squares:

Listing 7.2: The generalised average calculator

```
1  // InputData : ORBag[Integer]
2  // IntermediateSquare : ORBag[Integer]
3  // SumOfValues : ORBag[Integer]
4  // NumOfValues : ORBag[Integer]
5  // Intermediate : ORBag[(SumOfValues,NumOfValues)]
6  // Avg : ORBag[Float]
7
8  // f1 =
9  //      fun (InputData) ->
10 //          InputData * InputData
11 map(f1, InputData, IntermediateSquare);
12
13 SumOfValues := agg_sum(IntermediateSquare);
```

```
14
15 NumOfValues := agg_cnt(InputData);
16
17 Intermediate := SumOfValues × NumOfValues;
18
19 // f2 =
20 //       fun ((SumOfValues,NumOfValues)) ->
21 //             SumOfValues / NumOfValues
22 map(f2, Intermediate, Avg);
23
24 read_c(Avg);
```

The `map`() operation, described in the code lines 8-11 in Listing 7.2, updates input values to their squares. We can change this program for the average of sums of cubes with the provision of the map function that returns cubes of the input.

Chapter 8

Experiments

This chapter explains the implementation details of the newly implemented runtime system of our programming language described in the previous chapters and the details of our experiments. The explanation includes i) details of how the language and its runtime system are implemented, ii) details of example programs written in our programming language and executed on our runtime system, iii) the list of optimisation techniques we used to improve the performance of the example programs, and iv) the presentation of the results of our experiments using the example programs. The complete source code of the experiments can be found in Appendix A.

8.1 Implementation Details

Our programming language Lasp_{PC} is an extension of Lasp [35]. The system-level data type, the OR_{PC} set, and the client-level data types, the LWW register, the G counter, and the OR bag, are added to Lasp and the utility functions to maintain provenance and cover are also added. The implementation of provenance and cover has the $\mathcal{O}(n^2)$ complexity since they are defined as a collection of sets of event-histories in Chapter 4. The existing dataflow operations in Lasp such as `map`(), `filter`(), $\cup$ and $\times$ operations are extended to support those data types of Lasp_{PC}. The aggregation operations, `agg_sum`() and `agg_cnt`(), and their utility functions including the ones to compute the complete-dot set are also added to Lasp. The aggregation operations consist of two computational processes: the one to compute the complete-dot sets from cover and the other to compute the aggregation of the values. The complexity of the first process is $\mathcal{O}(n^2)$ because the complete-dot sets are computed from cover and the complexity of the second process is $\mathcal{O}(n^3)$ because it requires the manipulation of provenance ($\mathcal{O}(n^2)$) of the n values.

The following shows the complete definition of the Lasp_{PC} language:

Listing 8.1: Complete definition of Lasp_{PC}

```
1 % Data types:
```

```
2  LWW_Register
3  G_Counter
4  OR_Bag
5
6  % Operations for specific data types:
7  write(value, lww_register)
8
9  increase(g_counter)
10
11 add(value, or_bag)
12 remove(value, or_bag)
13 agg_sum(or_bag)
14 agg_cnt(or_bag)
15
16 % Operations for any data types:
17 read(subset_in_cover, data_type)
18 read_c(subset_in_cover, complete_dot_set, data_type)
19 map(function, data_type_input, data_type_output)
20 filter(function, data_type_input, data_type_output)
21 data_type ∪ data_type
22 data_type × data_type
```

In order to implement the optimisation techniques described in Section 8.3, it is required to add more functionalities to Lasp or to modify the existing functionalities in Laps. For example, the exclusion set for the synchronisation filter mode and the filter functionality is added and the cache calculation logic for dataflow operations is modified to support the incremental computation mode.

8.2 Example Programs

Every program written in our programming language consists of two parts: dataflow nodes and dataflow operations. Dataflow nodes, input, intermediate and final nodes, store input data, intermediate data and final results, respectively, based on their data types. Then, dataflow operations compute the results of the operations using data from their input dataflow nodes and store the results on their output dataflow nodes. Thus, the discussion of example programs in this chapter focuses on the details of the following two parts: what kind of data is stored on dataflow nodes and how this data is computed by dataflow operations. The discussion also includes the general description of each example program.

8.2.1 The Replicated Leader Board

The first example program to explain is the replicated leader board for an online mobile game introduced in Chapter 3. Each mobile device (acts as a user) adds its scores after

a single round game play is completed and the total scores of all users in a group, a set of users, are calculated. The replicated leader board is stored at each device and it shows the rank of groups based on the total scores of the individual group. Then, a medal (gold, silver or bronze) is assigned to each group based on the rank of the group. In a typical Client/Server architecture design, the leader board is computed and stored on the server only and each device (client) requests the status of the leader board whenever it is required. However, in our runtime system, the leader board is computed and stored on each device (replicated) and updates are periodically exchanged among all devices in the system.

Figure 3.1 shows the key dataflow nodes in a replicated leader board example program: three input nodes, two intermediate nodes and one final node. The three input nodes are declared as follows:

- *UserData* (the OR bag data type) - It stores pairs of a user identifier and scores of the user.
- *GroupData* (the OR bag data type) - It stores pairs of a group identifier and a set of group members (users).
- *Divider* (the LWW register data type) - It stores the threshold information for medals in the percentage format.

Available dataflow operations for input nodes depend on their data types: the `add`() or `remove`() operations for the OR bags and the `write`() operation for the LWW registers. For *UserData* and *GroupData*, the `add`() or `remove`() operations insert or delete their data (either a pair of a user identifier and scores of the user or a pair of a group identifier and a set of group members, respectively). The threshold information for medals is updated by the `write`() operation for *Divider*. Due to the constraints of the LWW register data type, only a single value for the threshold can be stored in the register.

Not all intermediate nodes for the program are illustrated in Figure 3.1. It only shows the key nodes for the computation. For example, the following example codes describe how the node *GroupScores* is computed from two input nodes, *UserData* and *GroupData*:

Listing 8.2: The *GroupScores* node from *UserData* and *GroupData*

```
1 // UserData : ORBag[(UserID, UserScores)]
2 // GroupData : ORBag[(GroupID, MembersSet)]
3 // GroupScores : ORBag[(GroupID, GroupScores)]
4 // Intermediate1 :
5 //     ORBag[((GroupID, MembersSet), (UserID, UserScores))]
6 // Intermediate2 :
7 //     ORBag[((GroupID, MembersSet), (UserID, UserScores))]
```

```
8  // Intermediate3 : ORBag[(GroupID, UserScores)]
9
10 Intermediate1 := GroupData × UserData;
11
12 // f1 =
13 //      fun (((GroupID, MembersSet), (UserID, UserScores))) ->
14 //          UserID ∈ MembersSet
15 filter(f1, Intermediate1, Intermediate2);
16
17 // f2 =
18 //      fun (((GroupID, MembersSet), (UserID, UserScores))) ->
19 //          (GroupID, UserScores)
20 map(f2, Intermediate2, Intermediate3);
21
22 GroupScores := agg_sum(Intermediate3);
```

The node *GroupScores* stores pairs of a group identifier and its total scores. In order to compute the result for the node *GroupScores*, the × operation, which is described in the code line 10 in Listing 8.2, finds all possible combinations of *GroupData* and *UserData*. Then, the `filter()` operation in the code line 15 selects the combinations such that 'UserID' from *UserData* is the member of 'MembersSet' from *GroupData*. The code line 20 shows that the `map()` operation updates the format of the data for the next operation `agg_sum()`. Finally, the `agg_sum()` operation in the code line 22 sums up total scores for each group and stores the result in the node *GroupScores*.

Similar to the node *GroupScores*, the node *RankDivider* is also computed using multiple intermediate nodes:

Listing 8.3: The *RankDivider* node from *GroupData* and *Divider*

```
1  // GroupData : ORBag[(GroupID, MembersSet)]
2  // Divider : LWWRegister[Threshold]
3  // RankDivider : ORBag[(GroupCount, Threshold)]
4  // Intermediate1 : ORBag[(GroupCount)]
5  // Intermediate2 : ORBag[(GroupCount, Threshold)]
6
7  Intermediate1 := agg_cnt(GroupData);
8
9  Intermediate2 := Intermediate1 × Divider;
10
11 // f1 =
12 //      fun ((GroupCount, Threshold)) ->
13 //          //compute ThresholdForRank from
14 //          //     GroupCount and Threshold
15 //          (ThresholdForRank)
16 map(f1, Intermediate2, RankDivider);
```

The node *RankDivider* stores the threshold information for assigning medals in the rank-based format. The threshold information stored in the node *Divider* is in the percentage format. In order to compute the result in the rank-based format from the original percentage format, it is essential to know the total number of groups, which is computed by the `agg_cnt`() operation described in the code line 7 in Listing 8.3. Then, the $\times$ operation in the code line 9 combines the total number and the percentage-based threshold. Last, the code line 16 shows that the `map`() operation converts the combination into the rank-based threshold value and stores the result on the node *RankDivider*.

In order to compute the result for the final node *GroupMedal*, a single intermediate node is used. The following example codes demonstrate how *GroupMedal* is computed:

Listing 8.4: The *GroupMedal* node from *GroupScores* and *RankDivider*

```
1  // GroupScores : ORBag[(GroupID, GroupScores)]
2  // RankDivider : ORBag[(GroupCount, Threshold)]
3  // GroupMedal : ORBag[(GroupID, Medal)]
4  // Intermediate :
5  //     ORBag[((GroupID, GroupScores), (GroupCount, Threshold))]
6
7  Intermediate := GroupScores × RankDivider;
8
9  // f1 =
10 //     fun (((GroupID, GroupScores), (GroupCount, Threshold))) ->
11 //         //assign a medal to each group based on
12 //         //its scores and threshold
13 //         (GroupID, Medal)
14 map(f1, Intermediate, GroupMedal);
```

8.2.2 The MapReduce Program

MapReduce is a programming model for processing large data sets on multiple devices using a parallel and distributed algorithm. The MapReduce program has two key procedures: the map procedure and the reduce procedure. The map procedure processes the input data and generates the result that is computed using only its local input data. The reduce procedure combines the results of map procedures that are computed from multiple devices and computes the final result. Each procedure can be executed on multiple devices in parallel. For example, assume that there is the MapReduce program that counts the number of words in various documents and those documents are stored and processed in multiple devices. The map procedure for this program reads its input data which is stored in its device in the form of multiple documents and counts each word in the documents. The result of the map procedure contains pairs of a word and its count in each local document. Then, the reduce procedure receives all the results of

the map procedure and combines them for the final result, the total pairs of a word and its count.

Our example program for MapReduce originates from one of the queries of BigBench [21] which will be explained in the next subsection. The goal of the program is to provide a unique session id for each user computed from streams of click information of users based on the items they viewed (clicked). The following example codes show how the MapReduce program is implemented:

Listing 8.5: The example program for MapReduce

```
1  // WebClickStreams:ORBag[{rows for the web_clickstreams table}]
2  // all other variables : ORBag
3
4  // f1 =
5  //      fun ({a row for the web_clickstreams table}) ->
6  //          wcs_item_id ≠ NULL && wcs_user_id ≠ NULL
7  filter(f1, WebClickStreams, InterWebClickStreams1);
8  // f2 =
9  //      fun ({a row for the web_clickstreams table}) ->
10 //          //select only necessary columns
11 //          //wcs_virtual_timestamp=ClickDate*24*60*60+ClickTime
12 //          (wcs_user_id,wcs_item_id,wcs_virtual_timestamp)
13 map(f2, InterWebClickStreams1, InterWebClickStreams2);
14
15 // f3 =
16 //      fun (left_value, right_value) ->
17 //          left_value < right_value
18 InterWebClickStreams3 = agg_sort(f3, InterWebClickStreams2);
19 // f4 =
20 //      fun ([(wcs_user_id,wcs_item_id,wcs_virtual_timestamp)]) ->
21 //          [(wcs_item_id,(wcs_user_id,wcs_virtual_timestamp))]
22 map(f4, InterWebClickStreams3, FinalResult);
```

The input data of the program, *WebClickStreams*, consists of streams of click information including when the click happened (date and time), what kind of the item was clicked and which user did the click. The map procedure of our program, described in the code lines 4-13 in Listing 8.5, processes each piece of the click information and generates a tuple of a user id, an item id and a virtual timestamp. The virtual timestamps here are monotonically increasing in the order of time and date (see the code line 11). The code lines 15-22 show the reduce procedure of our program. For the reduce procedure, we introduce a new aggregation operation, `agg_sort`(), which aggregates every values from the input and generates a single sorted list as a result. The first parameter of the operation is a function for the comparison of two values. The `agg_sort`() operation combines all the tuples from the map procedure. Then, the `map`() operation generates a unique session id per user by pairing a user id and a virtual timestamp.

8.2.3 BigBench Queries

BigBench [21], an end-to-end big data benchmark, was introduced in order to evaluate the performance of big data storages and processing systems. It has been adopted by Transaction Processing Performance Council (TPC) as TPCx-BB. The input data of the benchmark is generated based on a fictitious retail business that sells product via online and offline stores. The benchmark consists of thirty analytical queries expressed in SQL and Machine Learning (ML) algorithms.

Our example programs for BigBench [21] are the ones that implement three analytical queries expressed in SQL (Query 1, Query 24 and Query 29) on our runtime system. Since our system cannot interpret and execute the SQL queries directly, those queries need to be manually converted into our programming language. During the conversion, we used three optimisation techniques to improve the performance of the execution.

First, we executed the *SELECT* statements in SQL queries at the earliest possible point. The *SELECT* statement is converted to the `map`() operation in our programming language, and it reduces the number of columns in the rows of input tables. Therefore, the sooner the operation is executed, the smaller the size of the row becomes.

Second, we introduce a new dataflow operation for the *JOIN* statement in the SQL queries. The *JOIN* statement requires two dataflow operations in our programming language: the $\times$ operation followed by the `filter`() operation. The $\times$ operation finds all possible combinations of two tables, then the `filter`() operation selects the only combinations that have the identical column for *JOIN*. The preliminary experiments show that the execution of the $\times$ operation is very slow when the number of rows in input tables significantly increases. Furthermore, most of the combinations (from the results of the $\times$ operation) are removed by the following `filter`() operation. For the solution to the issue related to the *JOIN* statement, the new $\times_C$ operation, the $\times$ operation with condition, is added. The $\times_C$ operation maintains the input tables sorted by the joining columns and generates the combinations that have the identical value for the joining columns. This implies that the `filter`() operation is no longer required for the *JOIN* statement.

Last, when a SQL query contains the *WHERE* and *JOIN* statements at the same time, we execute the *WHERE* statement before the *JOIN* statement. The *WHERE* statement is converted to the `filter`() operation and the *JOIN* statement is converted to the the $\times_C$ operation. Despite the application of the second optimisation technique, the $\times_C$ operation is slow when the number of rows is large. Therefore, it is important to reduce the number of rows for the *JOIN* statements as much as possible. Applying the `filter`() operation first can avoid the generation of unnecessary combinations for the $\times_C$ operation.

Note that our BigBench [21] example programs are not just interpretations of SQL queries. Our implementation is close to a dynamic database program: it contains the information about the relation and the outcome is produced incrementally. The outcome is computed only for newly added updates and merged with existing results. It can be treated as a incremental implementation of SQL.

The input nodes in each example program for BigBench [21] are declared as the OR bag data types and each input node represents a table in a relational database management system. The `add`() operation for an input node is identical to the *INSERT INTO* statement in SQL queries and the operation will add the values for the rows in the table. The intermediate nodes and dataflow operations execute SQL queries as each statement is converted to dataflow operations. For example, the *WHERE* statement is converted to the `filter`() operation, the *SELECT* statement to the `map`() operation, the *GROUP BY* statement to the `agg_sum`() operation and the *COUNT* function to the `agg_cnt`() operation. The results of each query are stored in the final nodes.

We now describe the details of each SQL query and how these queries are converted to example programs in our runtime system. Query 1 is to find top 100 product pairs that are sold together frequently in given stores. The SQL query for Query 1 is defined as follows:

Listing 8.6: The SQL query for Query 1

```
1  SELECT item_sk_1,item_sk_2,COUNT(*) AS cnt
2  FROM
3  (
4    SELECT makePairs(sort_array(itemArray),false)
5      AS (item_sk_1,item_sk_2)
6    FROM
7    (
8      SELECT collect_set(ss_item_sk) AS itemArray
9      FROM store_sales s,item i
10     WHERE s.ss_item_sk = i.i_item_sk
11     AND i.i_category_id IN {1,2,3}
12     AND s.ss_store_sk IN {10,20,33,40,50}
13     GROUP BY ss_ticket_number
14   ) soldItemsPerTicket
15 ) soldTogetherPairs
16 GROUP BY item_sk_1,item_sk_2
17 HAVING cnt > 50
18 ORDER BY cnt DESC,item_sk_1,item_sk_2
19 LIMIT 100
```

Query 1 uses two input tables: *store_sales* and *item*. It joins the two input tables with the *item_sk* value, described in the code line 10 in Listing 8.6, and groups rows that contain the identical *ss_ticket_number* value. The identical *ss_ticket_number* value here

means that they were sold together. Then, the *makePairs()* function generates pairs of two items that are sold together. Finally, it counts the number of pairs and finds the pairs with the count value larger than 50. The following example codes show how the SQL query for Query 1 is converted to our example program:

Listing 8.7: The example program for Query 1

```
1  // StoreSales : ORBag[{rows for the store_sales table}]
2  // Item : ORBag[{rows for the item table}]
3  // all other variables : ORBag
4
5  // f1 =
6  //     fun ({a row for the store_sales table}) ->
7  //         //select only necessary columns
8  //         (ss_ticket_number,ss_item_sk,ss_store_sk)
9  map(f1, StoreSales, InterStoreSales1);
10 // f2 =
11 //     fun ({a row for the item table}) ->
12 //         //select only necessary columns
13 //         (i_item_sk,i_category_id)
14 map(f2, Item, InterItem1);
15
16 // f3 =
17 //     fun ((ss_ticket_number,ss_item_sk,ss_store_sk)) ->
18 //         ss_store_sk ∈ {10,20,33,40,50}
19 filter(f3, InterStoreSales1, InterStoreSales2);
20 // f4 =
21 //     fun ((i_item_sk,i_category_id)) ->
22 //         i_category_id ∈ {1,2,3}
23 filter(f4, InterItem1, InterItem2);
24
25 // ss_item_sk = i_item_sk
26 InterJoin1 := InterStoreSales2 ×_C InterItem2;
27
28 // f5 =
29 //     fun (((ss_ticket_number,ss_item_sk,ss_store_sk),
30 //           (i_item_sk,i_category_id)) ->
31 //         //select only necessary columns
32 //         (ss_ticket_number,i_item_sk)
33 map(f5, InterJoin1, InterJoin2);
34
35 InterJoin3 := agg_sum(InterJoin2);
36
37 // f6 =
38 //     fun ((ss_ticket_number,i_item_sk_SET)) ->
39 //         //makePairs() and count the cnt value
40 //         (i_item_sk_1,i_item_sk_2,cnt)
41 map(f6, InterJoin3, InterJoin4);
```

```
42
43 InterJoin5 := agg_sum(InterJoin4);
44
45 // f7 =
46 //      fun ((i_item_sk_1,i_item_sk_2,cnt)) ->
47 //          cnt > 50
48 filter(f7, InterJoin5, FinalResult);
```

In this example program, the three optimisation techniques explained above are used. The code lines 5-14 in Listing 8.7 describe how the first optimisation technique is used. The `map()` operations remove the unnecessary columns at the earliest possible opportunity in order to reduce the size of individual rows. The code line 26 in Listing 8.7 shows the second optimisation technique, the $\times_C$ operation. The third one, the `filter()` operation before the $\times_C$ operation, is in the code lines 16-23 in Listing 8.7. The two aggregation operations in the code lines 35 and 43 in Listing 8.7 are converted from the two *GROUP BY* statements in the code lines 13 and 16 in Listing 8.6, respectively.

Query 24 is to compute the cross-price elasticity of demand for a given product. The detailed explanation of what the cross-price elasticity of demand means is beyond the scope of this thesis. Hence, we focus on how the given SQL query is converted to our example program. The SQL query for Query 24 is defined as follows:

Listing 8.8: The SQL query for Query 24

```
1  CREATE TABLE temp_table AS
2  SELECT
3    i_item_sk,
4    imp_sk,
5    (imp_competitor_price - i_current_price)/i_current_price
6      AS price_change,
7    imp_start_date,
8    (imp_end_date - imp_start_date) AS no_days_comp_price
9  FROM item i,item_marketprices imp
10 WHERE i.i_item_sk = imp.imp_item_sk
11 AND i.i_item_sk = 10000
12 ORDER BY i_item_sk,imp_sk,imp_start_date
13
14 SELECT ws_item_sk,
15        avg(
16          (current_ss_quant + current_ws_quant
17            - prev_ss_quant - prev_ws_quant)
18          / ((prev_ss_quant + prev_ws_quant) * ws.price_change))
19          AS cross_price_elasticity
20 FROM
21     (
22       SELECT
23         ws_item_sk,
```

```
24          imp_sk,
25          price_change,
26          SUM(
27            CASE WHEN ((ws_sold_date_sk >= c.imp_start_date)
28              AND (ws_sold_date_sk
29                < (c.imp_start_date + c.no_days_comp_price)))
30            THEN ws_quantity ELSE 0 END) AS current_ws_quant,
31          SUM(
32            CASE WHEN ((ws_sold_date_sk
33                >= (c.imp_start_date - c.no_days_comp_price))
34              AND (ws_sold_date_sk < c.imp_start_date))
35            THEN ws_quantity ELSE 0 END) AS prev_ws_quant
36        FROM web_sales ws
37        JOIN temp_table c ON ws.ws_item_sk = c.i_item_sk
38        GROUP BY ws_item_sk,imp_sk,price_change
39      ) ws
40  JOIN
41      (
42        SELECT
43          ss_item_sk,
44          imp_sk,
45          price_change,
46          SUM(
47            CASE WHEN ((ss_sold_date_sk >= c.imp_start_date)
48              AND (ss_sold_date_sk
49                < (c.imp_start_date + c.no_days_comp_price)))
50            THEN ss_quantity ELSE 0 END) AS current_ss_quant,
51          SUM(
52            CASE WHEN ((ss_sold_date_sk
53                >= (c.imp_start_date - c.no_days_comp_price))
54              AND (ss_sold_date_sk < c.imp_start_date))
55            THEN ss_quantity ELSE 0 END) AS prev_ss_quant
56        FROM store_sales ss
57        JOIN temp_table c ON c.i_item_sk = ss.ss_item_sk
58        GROUP BY ss_item_sk,imp_sk,price_change
59      ) ss
60   ON (ws.ws_item_sk = ss.ss_item_sk and ws.imp_sk = ss.imp_sk)
61  GROUP BY  ws.ws_item_sk
```

The query computes *temp_table* first which contains the information related to the price of the given product. The code lines 1-12 in Listing 8.8 show how *temp_table* is computed using two input tables: *item* and *item_marketprices*. The *temp_table* is used to compute the *JOIN* statement using the *web_sales* table (the code line 37) and the *store_sales* table (the code line 57). The results of these two *JOIN* statements are combined by another *JOIN* statement (the code line 40). The result of this last *JOIN* statement contains all the information necessary for computing the cross-price elasticity of demand. The

following example codes show how the SQL query for Query 24 is converted to our example program:

Listing 8.9: The example program for Query 24

```
1  // Item : ORBag[{rows for the item table}]
2  // ItemMarketPrices :
3  //     ORBag[{rows for the item_marketprices table}]
4  // WebSales : ORBag[{rows for the web_sales table}]
5  // StoreSales : ORBag[{rows for the store_sales table}]
6  // all other variables : ORBag
7
8  // f1 =
9  //     fun ({a row for the item table}) ->
10 //         i_item_sk = 10000
11 filter(f1, Item, InterItem1);
12 // f2 =
13 //     fun ({a row for the item_marketprices table}) ->
14 //         imp_item_sk = 10000
15 filter(f2, ItemMarketPrices, InterItemMarketPrices1);
16 // f3 =
17 //     fun ({a row for the item table}) ->
18 //         //select only necessary columns
19 //         (i_item_sk,i_current_price)
20 map(f3, InterItem1, InterItem2);
21 // f4 =
22 //     fun ({a row for the item_marketprices table}) ->
23 //         //select only necessary columns
24 //         (imp_sk,imp_item_sk,imp_competitor_price,
25 //           imp_start_date_sk,imp_end_date_sk)
26 map(f4, InterItemMarketPrices1, InterItemMarketPrices2);
27 // i_item_sk = imp_item_sk
28 InterJoinTemp1 := InterItem2 × InterItemMarketPrices2;
29 // f5 =
30 //     fun (((i_item_sk,i_current_price),
31 //           (imp_sk,imp_item_sk,imp_competitor_price,
32 //             imp_start_date_sk,imp_end_date_sk)) ->
33 //         //compute the result for temp_table
34 //         (i_item_sk,imp_sk,price_change,
35 //           imp_start_date,no_days_comp_price)
36 map(f5, InterJoinTemp1, TempTable);
37
38 // f6 =
39 //     fun ({a row for the web_sales table}) ->
40 //         ws_item_sk = 10000
41 filter(f6, WebSales, InterWebSales1);
42 // f7 =
43 //     fun ({a row for the web_sales table}) ->
44 //         //select only necessary columns
```

```
45 //          (ws_sold_date_sk, ws_item_sk, ws_quantity)
46 map(f7, InterWebSales1, InterWebSales2);
47 // i_item_sk = ws_item_sk
48 InterJoinWS1 := TempTable × InterWebSales2;
49 // f8 =
50 //      fun (((i_item_sk, imp_sk, price_change,
51 //              imp_start_date, no_days_comp_price),
52 //            (ws_sold_date_sk, ws_item_sk, ws_quantity)) ->
53 //          //compute the result for (temp_table X web_sales)
54 //          ((i_item_sk, imp_sk_ws, price_change),
55 //           (current_ws_quant, prev_ws_quant))
56 map(f8, InterJoinWS1, InterJoinWS2);
57 InterJoinWS3 := agg_sum(InterJoinWS2);
58
59 // f9 =
60 //      fun ({a row for the store_sales table}) ->
61 //          ss_item_sk = 10000
62 filter(f9, StoreSales, InterStoreSales1);
63 // f10 =
64 //      fun ({a row for the store_sales table}) ->
65 //          //select only necessary columns
66 //          (ss_sold_date_sk, ss_item_sk, ss_quantity)
67 map(f10, InterStoreSales1, InterStoreSales2);
68 // i_item_sk = ss_item_sk
69 InterJoinSS1 := TempTable × InterStoreSales2;
70 // f11 =
71 //      fun (((i_item_sk, imp_sk, price_change,
72 //              imp_start_date, no_days_comp_price),
73 //            (ss_sold_date_sk, ss_item_sk, ss_quantity)) ->
74 //          //compute the result for (temp_table X store_sales)
75 //          ((i_item_sk, imp_sk_ss, price_change),
76 //           (current_ss_quant, prev_ss_quant))
77 map(f11, InterJoinSS1, InterJoinSS2);
78 InterJoinSS3 := agg_sum(InterJoinSS2);
79
80 // imp_sk_ws = imp_sk_ss
81 InterJoinResult1 := InterJoinWS3 ×_C InterJoinSS3;
82 // f12 =
83 //      fun (((i_item_sk, imp_sk_ss, price_change),
84 //            (current_ss_quant, prev_ss_quant))) ->
85 //          //compute the cross-price elasticity of demand
86 //              for a single row
87 //          (i_item_sk, (cross_price_elasticity, 1))
88 map(f12, InterJoinResult1, InterJoinResult2);
89 InterJoinResult3 := agg_sum(InterJoinResult2);
90 // f13 =
91 //      fun ((i_item_sk, (cross_price_elasticity_sum, cnt))) ->
```

```
92 //          //compute avg() for
93 //          //    the cross-price elasticity of demand
94 //          (i_item_sk, cross_price_elasticity_avg)
95 map(f13, InterJoinResult3, FinalResult);
```

The code lines 8-36 in Listing 8.9 describe the conversion of the SQL query for *temp_table*. For *temp_table*, the example program uses the third optimisation technique before using the first one. This is because that the *WHERE* statement (the code line 11 in Listing 8.8) selects only one item and, thus, executing this statement first can significantly reduce the number of rows for the computation. Most of the remaining SQL queries are converted in a way that is similar to the one used for Query 1. Notice that normal $\times$ operations are used for the *JOIN* statements instead of $\times_C$: the code lines 48 and 69. This is because that the rows have already been filtered to contain only one item and the condition of *JOIN* is the sameness of that item.

Query 29 is made to perform the category affinity analysis for products purchased together online. Similar to Query 24, we focus on demonstrating how the SQL query is converted instead of explaining what the category affinity analysis means. The SQL query for Query 29 is defined as follows:

Listing 8.10: The SQL query for Query 29

```
1  SELECT category_id_1, category_id_2, COUNT (*) AS cnt
2  FROM (
3    SELECT makePairs(sort_array(itemArray), false)
4      AS (category_id_1, category_id_2)
5    FROM (
6      SELECT collect_set(i_category_id) as itemArray
7      FROM web_sales ws, item i
8      WHERE ws.ws_item_sk = i.i_item_sk
9      AND i.i_category_id IS NOT NULL
10     GROUP BY ws_order_number
11   ) collectedList
12 ) pairs
13 GROUP BY category_id_1, category_id_2
14 ORDER BY cnt DESC, category_id_1, category_id_2
15 LIMIT 100
```

The SQL query for Query 29 demonstrated in Listing 8.10 is similar to the one for Query 1. The two input tables for Query 29 are *web_sales* and *item*. The query combines the two input tables and finds all the possible pairs of categories of items sold together online. Then, it counts the number of each pair and finds top 100 pairs. The example codes below show how the SQL query for Query 29 is converted to our example program:

Listing 8.11: The example program for Query 29

```
1 // WebSales : ORBag[{rows for the web_sales table}]
```

```
2  // Item : ORBag[{rows for the item table}]
3  // all other variables : ORBag
4
5  // f1 =
6  //      fun ({a row for the web_sales table}) ->
7  //          //select only necessary columns
8  //          (ws_order_number, ws_item_sk)
9  map(f1, WebSales, InterWebSales1);
10
11 // f2 =
12 //      fun ({a row for the item table}) ->
13 //          i_category_id ≠ NULL
14 filter(f2, Item, InterItem1);
15 // f3 =
16 //      fun ({a row for the item table}) ->
17 //          //select only necessary columns
18 //          (i_item_sk, i_category_id)
19 map(f3, InterItem1, InterItem2);
20
21 // ws_item_sk = i_item_sk
22 InterJoin1 := InterWebSales1 ×_C InterItem2;
23 // f4 =
24 //      fun (((ws_order_number, ws_item_sk),
25 //            (i_item_sk, i_category_id)) ->
26 //          //select only necessary columns
27 //          (ws_order_number, i_category_id)
28 map(f4, InterJoin1, InterJoin2);
29 InterJoin3 := agg_sum(InterJoin2);
30 // f5 =
31 //      fun ((ws_order_number, i_category_id_SET)) ->
32 //          //makePairs() and count the cnt value
33 //          (i_category_id, i_category_id, cnt)
34 map(f5, InterJoin3, InterJoin4);
35 FinalResult := agg_sum(InterJoin4);
```

The example program demonstrated in Listing 8.11 is converted using the same method employed for other queries. It also uses the three optimisation techniques described above. Additionally, the code lines 11-19 show that the `filter()` operations are used before selecting the necessary columns in order to reduce the number of rows first.

8.3 Three General Optimisations

We use the following three optimisation techniques to improve the performance of our example programs: i) reducing the usage of redundant information, ii) reducing the number of nodes for the periodic synchronisation and iii) using delta state for the OR_{PC}

data type with the incremental computation mode and caches for dataflow operations. Each subsection explains the details of these techniques and describes how the techniques improve the performance of the example programs with figures. All figures in this subsection are based on our first example program, the replicated leader board program.

8.3.1 Removal of Redundant Information

The definition of the OR_{PC} data type in Chapter 4 contains redundant information. Recall that we defined the OR_{PC} data type as follows:

$$X = (A, L, \mathcal{C}, K)$$

The first component A is a set of all event-histories that are added by the `insert`() operations. The second component L is also a set of event-histories that are not cancelled by the `delete`() operations. The third component $\mathcal{C}$ is a cover of the component A. Then, the last component K is a function from the component $\mathcal{C}$ to the set of provenance stores.

There are three pieces of redundant information in the current definition of the OR_{PC} data type. The first piece of redundant information is located in the component A. The component L is defined as a subset of A and this means that every member of L is also in A. Therefore, storing only the difference between A and L can remove the first piece of redundant information. Notice that the difference between A and L is a set of event-histories cancelled by the `delete`() operations. The second piece of redundant information is located in the component L which stores a set of event-histories that are not cancelled by the `delete`() operations. Most of the event-histories stored in L are also stored in the range of the component K, a set of provenance stores, except the ones that are filtered by the `filter`() operations. Therefore, storing only the filtered event-histories can also remove the second piece of redundant information. The component $\mathcal{C}$ that contains the third piece of redundant information can be entirely removed because the component K contains the complete information about $\mathcal{C}$ as the domain of itself.

The definition of the optimised version of the OR_{PC} data type which removes all the three pieces of redundant information described above is as follows:

$$X_O = (R, F, K)$$

The component R is a set of event-histories cancelled by the `delete`() operations. The component F is a set of event-histories that are filtered by the `filter`() operations. The last component K is identical to the one in the definition of the OR_{PC} data type in Chapter 4. The optimised version of the OR_{PC} data type also satisfies the key theoretical properties of dataflow programs described and proved in Chapter 5 because

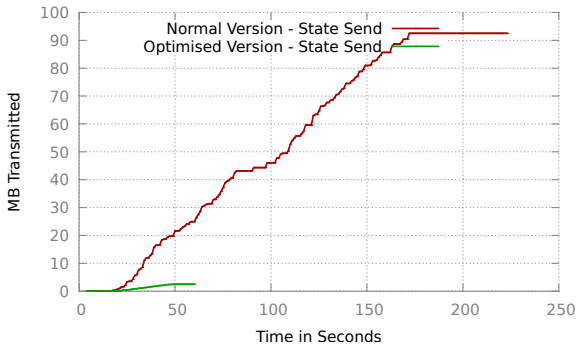


Figure 8.1: Optimisation 1: Removal of redundant information

it only removes the redundant information and it is straightforward to retrieve the normal version of the OR_{PC} data type from the optimised version.

Figure 8.1 contains the performance results of two different execution: the one with the normal version of the OR_{PC} data type, named **Normal Version - State Send**, and the other with the optimised version of the OR_{PC} data type, named **Optimised Version - State Send**. Each execution has ten devices in the system and input nodes in every device process 30 insertions in total. The X-axis represents time in seconds after the execution and Y-axis represents the total number of mega-bytes transmitted among the devices during the execution. The end of the line for each execution indicates that the execution reaches the converged state. Here, the converged state means that every device has the identical result at the final node and there are no more updates. The figure shows that the execution with the optimised version leads to the reduction of a significant amount of transmitted data from 92 mega-bytes to 2 mega-bytes. It reaches the converged state considerably earlier than the execution with the normal version, reducing the time from 223 seconds to 60 seconds.

8.3.2 Synchronisation Filter

By default, every device exchanges the state of the entire nodes in the example program during the periodic synchronisation. This implies that a device will receive all updates for every node, and each node will compute its dataflow operation for all updates even if the output node has already received the results for those updates. In order to reduce this redundant dataflow computation, we introduce the synchronisation filter mode that stores a set of nodes for the exclusion of the periodic synchronisation. The key to using this synchronisation filter mode is that all three key functionalities (decentralized communication, high responsiveness and strong convergence) explained in Chapter 3 are still met after the mode is enabled.

Based on how dataflow operations are structured, we apply three rules to exclude or not to exclude the nodes. First, if there are continuous unary dataflow operations,

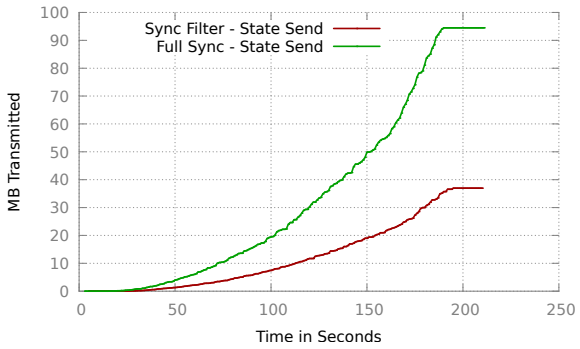


Figure 8.2: Optimisation 2: Synchronisation filter

such as `map`() or `filter`(), we can exclude the intermediate nodes (add them to the exclusion set) for the periodic synchronisation. Assume that there are three nodes in an example program: node A, node B and node C. Node A is an input node of the program and the `map`() operation has node A as an input and node B as an output. Then, the `filter`() operation has node B as an input and node C as an output. Input data flows from node A to node C through node B. In this example, we can safely add node B to the exclusion set because node C can compute the result from the state of node A without the synchronisation of the state of node B. Second, the two inputs of the $\times$ operation should not be added to the exclusion set. Without the synchronisation, the two inputs cannot have all the updates and this means that the result of the $\times$ operation cannot have all the combinations of the inputs. Therefore, the two inputs should perform the periodic synchronisation in order to get all the updates. Third, the input of an aggregation operation, such as `agg_cnt`() or `agg_sum`(), should not be added to the exclusion set. During the `merge`() operation, the output of an aggregation operation compares how many input values are included in each state, and it means that the input of an aggregation operation should receive all the updates and compute them for the correct result. Therefore, the input of an aggregation operation should not be excluded for the periodic synchronisation.

Figure 8.2 contains the performance results of the following two different executions: the one with the disabled synchronisation filter mode (full synchronisation), named **Full Sync - State Send**, and the other with the enabled synchronisation filter mode, named **Sync Filter - State Send**. The exclusion set for the execution with the synchronisation filter contains six nodes among the total of 12 nodes in the program and the members of the exclusion set are selected based on the three rules described above. Each execution has ten devices in the system and the input nodes in every device process 1000 insertions. Similar to the figure for the first optimisation technique, the X-axis represents time in seconds after the execution and Y-axis represents the total number of mega-bytes transmitted among the devices during the execution. The end

of the line for each execution indicates that the execution reaches the converged state. The figure 8.2 shows that the amount of transmitted data of the execution with the synchronisation filter mode is smaller than that of the execution without the mode.

8.3.3 Delta State with the Incremental Computation Mode and Caches

When two nodes exchange the information about updates, they exchange the entire state of each data type similar to the Lasp [35] programming language. The entire state here means that the information exchanged previously is included again for the current exchange. Exchanging the delta states [2] that contain only the newly added updates significantly reduces the size of transmitted data for the state exchange. However, implementing the delta states alone still has a performance issue related to the `read`() operation [11]. The `read`() operation requires the entire state of the data type and every dataflow operation in the system uses the `read`() operation to process its inputs for the computation. Therefore, the benefit of using the delta states applies only to the data transmission, and not for the data computation.

In order to improve the performance of dataflow operations with the delta state, we add the incremental computation mode that allows the dataflow operations taking the delta state instead of the entire state as an input of the operation. We found that the incremental computation mode works well with the following four dataflow operations: `map`(), `filter`(), $\cup$ and $\times$. However, the aggregation operations cannot generate the correct results with the delta states of their inputs because the outputs cannot combine the partially aggregated result (from the delta states) with the existing result. This is because that the output is not aware of how the aggregation result is computed.

Using a cache table is the solution to the issue involved in the use of the delta states for the aggregation operations. If the cache table stores the previous state and its aggregation result, the delta state can compute the aggregation result based on the cached result from the previous state. Since the delta states for the `delete`() operations do not contain the information about the deleted values, the previous state in the cache table can provide the information about those values.

Figure 8.3 contains the performance results of two different executions: the one with the full state exchange, named **State Based - State Send**, and the other with the delta state exchange with the incremental computation mode and caches, named **Delta Based - State Send**. Each execution has ten devices in the system and the input nodes in every device process 1000 insertions in total. Similar to the figures for other optimisation techniques, the X-axis represents time in seconds after the execution and Y-axis represents the total number of mega-bytes transmitted among the devices during the execution. The end of the line for each execution indicates that the execution

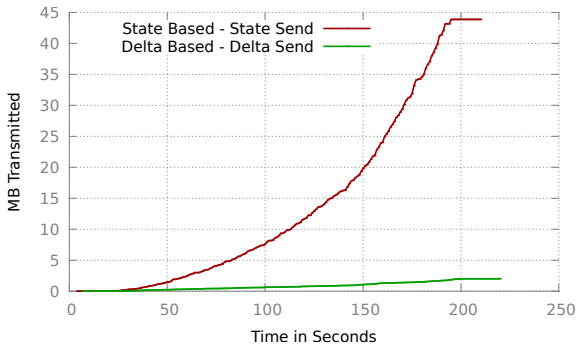


Figure 8.3: Optimisation 3: Delta state with the incremental computation mode and caches

reaches the converged state. The Figure 8.3 shows that the execution with the delta state exchange with the incremental computation mode and caches significantly reduces the amount of data transmitted from 44 mega-bytes to 2 mega-bytes.

8.4 Experiment Results

In this section, we present the results of our experiments. The example programs described in the previous subsections and our runtime system are executed on top of the Kubernetes[1] cluster. Kubernetes is a system for the deployment and management of containerised applications and it allows multiple machines to form a cluster for scaling. Each device for the example programs is deployed on the cluster as a containerised application. The cluster for the experiments consists of the following two machines: one with *CPU: AMD Ryzen 9 5950X (32 cores) @ 3.4GHz, Memory: 64 Gigabytes* and the other with *CPU: AMD Ryzen 7 3700X (16 cores) @ 3.6GHz, Memory: 32 Gigabytes*. Both machines use *Ubuntu 20.04.5 LTS* as their operating systems and the version of Kubernetes is *v1.23.14-2+55c3aa5b608650*. For every figure in this section, the X-axis represents time in seconds after the execution and Y-axis represents the total number of mega-bytes transmitted among the devices during the execution. The yellow dot on the X-axis indicates the time the execution reaches the converged state.

8.4.1 The Replicated Leader Board

The experiment results from the execution of the Replicated Leader Board program are presented in Figure 8.4. The execution was run with 20 devices and it processed 16,000 input data streams. The total amount of data transmitted during the execution was about 30 Megabytes and the time for reaching the converged state was 2,310 seconds.

[1]https://kubernetes.io/

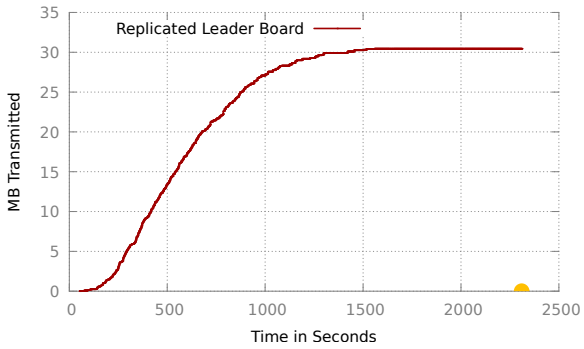


Figure 8.4: Experiment Results for the Replicated Leader Board

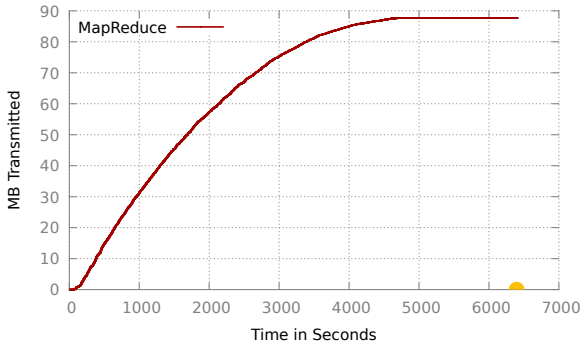


Figure 8.5: Experiment Results for the MapReduce Program

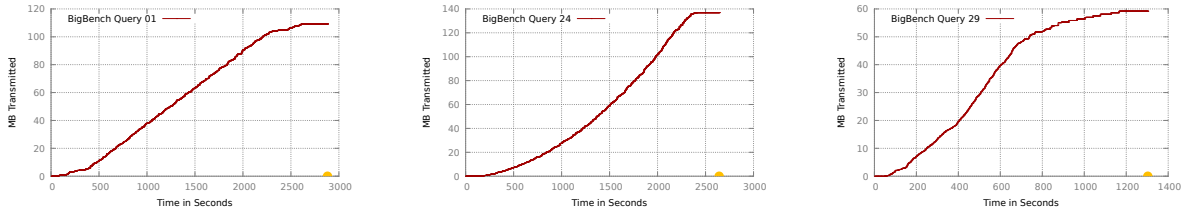
8.4.2 The MapReduce Program

Figure 8.5 shows the experiment results for the execution of the MapReduce program. As we explained in Section 8.2.2, this program uses a part of the computation from BigBench and it means that the input data of the program is a table of relational databases. The execution was run with 20 devices and it processed 200,000 rows for the *web_clickstreams* table. The total amount of data transmitted during the execution was about 90 Megabytes and the time for reaching the converged state was 6,397 seconds.

8.4.3 BigBench Queries

The experiment results for the three example programs of the BigBench queries are presented in Figure 8.6. Each program applied a different query optimisation technique, described in Section 8.2.3. Therefore, the number of devices used for the execution and the number of rows processed for inputs differ between those programs.

Figure 8.6a depicts the performance result of the execution of the example program for BigBench Query 1. As we described in the previous subsections, the BigBench Query 1 uses two input tables: *item* and *store_sales*. The execution was run with 20 devices and it processed 40 per cent of 1 Gigabytes of the input data of BigBench, which equals to 7,120 rows for the *item* table and 267,253 rows for the *store_sales* table. The



(a) Query 1 (b) Query 24 (c) Query 29

Figure 8.6: Experiment Results for BigBench Queries

execution reached the converged state after 2,881 seconds, and about 110 Megabytes of data was transmitted during the execution.

The performance result presented in Figure 8.6b is for the execution of the example program for BigBench Query 24. The BigBench Query 24 uses four input tables, a larger number of tables than the ones for the BigBench Query 1, but this execution was run with 25 devices processing 50 per cent of 1 Gigabytes of the input data of BigBench on the same cluster: 8,900 rows for the *item* table, 44,550 rows for the *item_marketprices* table, 334,104 rows for the *store_sales* table and 334,714 rows for the *web_sales* table. The total amount of data transmitted during the execution was about 140 Megabytes and the time for reaching the converged state was 2,642 seconds.

For the execution of the example program for BigBench Query 29, the performance result is demonstrated in Figure 8.6c. The execution was run with 20 devices and it processed 10 per cent of 1 Gigabytes of the input data of BigBench, which equals to 7,120 rows for the *item* table and 50,000 rows for the *web_sales* table. Considering the similarity between Query 1 and Query 29, the amounts of data processed are significantly different. This is due to the difference in the kinds of the query optimisation techniques applied. In the case of Query 1, the `filter`() operation is used before the $\times_C$ operation for the *store_sales* table and this reduces the large number of rows for the $\times_C$ operation. However, Query 29 does not contain the *WHERE* statement for the *JOIN* statement for the *web_sales* table.

Dippy *et al.* [1] present the results of every BigBench query that was executed on a single machine environment. The direct comparison between our experiment results and the ones from the single machine environment [1] is insignificant as our system does not have any optimisations for handling relational databases and the performance of the machines in the cluster from our experiments is significantly weaker than the one from the single machine environment [1]. However, without any optimisation techniques for relational database management systems (RDBMS) such as the technique for processing large input files or the one for indexing for the values in columns, the number of rows processed by the three example programs suggests that our system guarantees acceptable performance for processing a large number of input data streams.

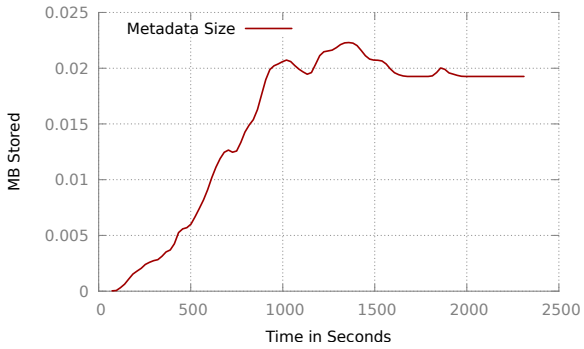


Figure 8.7: Metadata Size of the final node for the Replicated Leader Board

8.4.4 Discussion of the Performance

The experiment results from the execution of the replicated leader board program show that the time for reaching the converged state was 2,310 seconds for 30 Megabytes in Figure 8.4 which is slow for practical use. We speculated that possible reasons for the slow performance would be i) the large size of the metadata needed for provenance and cover and ii) inefficient manipulation of metadata operations. We found that the second possible reason, inefficient manipulation of metadata operations, is actually accountable for the slow performance.

This is confirmed by examining Figure 8.7. As for the large size metadata needed for provenance and cover, Figure 8.7 shows the metadata size of the final node in the replicated leader board program. The size of metadata is not large, as can be seen from the figure - hence not the cause of the slow performance. The largest size of metadata of the final node is about 22 Kilobytes in the figure.

Before we discuss the second reason mentioned above, inefficient manipulation of metadata operations, we need to consider one of the most expensive dataflow operations, the $\times$ operation: its complexity is $\mathcal{O}(n^2)$. Since the execution of our replicated leader board program processed 16,000 input data streams, the number of values in the result of the $\times$ operation would certainly be extremely large. One of the common metadata operations is to check the membership among provenance and cover of those values. Therefore, it would lead to extremely slow performance and resolving this issue would improve the performance of the execution significantly, based on our performance profiling results.

8.5 Further Experiments

8.5.1 Glitch Removal

Table 8.1 and Table 8.2 show the results of two different executions on the same node for the average calculator introduced in Section 7.3. Each execution inserts the value 10

Time (s)	Sum / Count
17	10 / 1
21	20 / 2
21	30 / 3
23	40 / 4
26	50 / 5
32	60 / 6
38	70 / 7
38	80 / 8
39	90 / 9
41	100 / 10

Table 8.1: Results of the consistent read

Time (s)	Sum / Count
17	10 / 1
21	20 / 1
21	20 / 2
24	30 / 2
24	30 / 3
24	30 / 4
24	40 / 4
24	40 / 5
24	50 / 5
32	60 / 6
32	60 / 7
32	70 / 7
34	70 / 8
34	80 / 8
35	80 / 9
35	90 / 9
39	90 / 10
39	100 / 10

Table 8.2: Results of the normal read

to the input 10 times to calculate the average. The cluster for the executions consists of five replicas. The first column shows the time when the `read`$_c$() operation returned the monotonically-advanced outcome and the second column presents the sum and the count of input values when the average was returned.

The results of the first execution presented in Table 8.1 only use the `read`$_c$() operations to retrieve the glitch-free results. This execution yields a smaller number of outcomes than the one with the normal `read`() operations, but, it is clear that it always returns the results without any glitches.

The results of the second execution presented in Table 8.2 only use the `read`() operations that can generate counter-intuitive results. We can get results more often than the one with the consistent read operations but this execution also generates the counter-intuitive results regularly, as highlighted in red in the table (8 out of 18).

This experiment does not use a large number of input data streams but the results show that the performance of the programs is not affected much by the usage of the `read`$_c$() operation.

8.5.2 Convergence

From the execution using the `read`$_c$() operations (as described in Table 8.1), we can also measure how much time it takes for a new insertion of a value to propagate from the input node on a replica to the final nodes on all replicas. The 10th insertion of the

value 10 occurred on the third replica 37 seconds after the execution started. Then, 4 seconds later, the replica that generated the result presented in Table 8.1 received the information about that insertion and it propagated to all other replicas 83 seconds after the insertion. The convergence time can be affected by the number of replicas in the system, the workload of each replica and the status of the network.

8.5.3 Fault Tolerance

The experiments for fault tolerance are carried out using the average calculator introduced in Section 7.3 which is also used in the two other experiments discussed above, glitch removal and convergence. The experiments are designed to terminate the processes of replicas and check how other replicas behave. Since our new language Lasp_{PC} is based on Lasp [35], our expansion is not related to how networks are working. Therefore, the fault tolerance functionality should remain intact. As we expected, the replicas whose processes are not terminated still guarantee our theoretical properties and return the expected results (the correct average value, 10, has been returned).

Chapter 9

Case Study

There are similarities in the way blockchain systems and weakly consistent distributed systems work. To be precise, the *participants* in blockchain systems and the *replicas* in weakly consistent distributed systems behave in a similar way. The similarities are threefold:

- In blockchain systems, each participant can add new blocks or new contents without the consideration of other participants. In weakly consistent distributed systems, each replica can update its local state without the coordination.
- Individual participants' addition of new blocks and contents to blockchain systems, independent of other participants, are highly likely to lead to their having chains of blocks that are different from other participants (unless other participants add the same blocks and contents to the systems). Similarly, local updates to the states of a replica can generate conflicts with the states of other replicas.
- In order to remove the inconsistencies in chains of blocks and contents, the blockchain consensus algorithms ensure that identical chains of blocks and contents are generated between participants after the completion of algorithms. As for weakly consistent distributed systems, conflict resolution methods are used to resolve inconsistencies between the two different local states of any two replicas.

The similarities discussed above provide sound justification for investigating the logic of blockchain consensus protocols for the potential application to future client-level data types of our new language. This chapter presents a detailed study of one of the blockchain consensus protocols, the Stellar Consensus Protocol (SCP) [34], as an example. The purpose of this chapter is to facilitate the implementation of future client-level data types for our new language such as the replicated growable array (RGA) [43] by providing the key ideas of the protocols. The RGA data type is the one that stores and accesses data sequentially and this sequential data is similar to the way multiple blocks are chained in blockchain systems.

Our preliminary experiment shows, however, that it is difficult to apply the logic of the SCP to our current implementation. It is found that, at the early stage of the algorithm execution, the protocol requires the synchronisation of clocks which means some coordination is needed at that stage [34]. This kind of coordination is currently not supported in our implementation. We continued the experiment despite the limitation based on the idea and expectation that there is a possibility that the logic of the SCP can be applied to our new language and its runtime system in the future. It is also thought that this case study can also contribute to the development of new blockchain consensus protocols similar to the SCP.

The main goal of the blockchain consensus protocols, including the SCP, is to maintain the identical single chain of multiple blocks for each participant, and each block stores important information such as transactions of coins or program codes for distributed applications. Similar to the way the weakly consistent distributed systems work, any participant of the blockchain systems can concurrently generate those blocks and add them to the chain, if they fulfil certain criteria. Each protocol has its own criterion. Once the blocks have been generated and added, the blockchain consensus protocols make sure that all participants have the identical chain of blocks after the computation is completed. Notice that this is similar to what the `merge`() operation does during the state exchange in our system.

In order to understand how the SCP works and check the soundness of the protocol, we used the formal modelling and verification with the UPPAAL model checker [5]. Formal modelling and verification are used to verify the accuracy of various properties of a finite-state system, but it has not been widely used for verifying the large size applications due to the state explosion issue. In this chapter, we first explain the details of how the SCP works and how we overcame the issue by using the techniques to reduce the number of states for the SCP. Then, we introduce four design templates that can be used for modelling other blockchain consensus protocols similar to the SCP. Finally, the results of the formal verification are presented at the end of the chapter.

9.1 The Stellar Consensus Protocol

This section describes how the SCP works, including the way its two sub-protocols operate. The SCP is the blockchain consensus protocol for the Stellar cryptocurrency [1] and it introduces the federated Byzantine agreement (FBA) as a core algorithm. The runtime system of the SCP consists of multiple quorums. Each quorum is defined as a set of devices in the system. In order to guarantee the safety and liveness of the FBA, every quorum should have an intersection with another quorum in the system. Each

[1] https://www.stellar.org/

device has a quorum slice configuration that contains a threshold value and a set of trusted devices. These trusted devices are called as *validators*. The set of validators indicates the local view of the quorum that the device belongs to, and the threshold value is used for checking whether the vote is successful or not.

Similar to Practical Byzantine Fault Tolerance (PBFT) [18], the best known variant of Byzantine agreement, the agreement in the FBA is completed by the use of voting messages exchanged between validators. The agreement of a statement α in the FBA requires two steps: i) The statement α has to be *accepted*. Here, *accepted* means that the statement is voted by the number of validators that is larger than the threshold value in the quorum configuration. Then, ii) the accepted statement α has to be *confirmed*. Once a device receives messages that contain the information about α being accepted, and, if the number of messages is larger than the threshold value, the device changes the state of α as *confirmed*. The centralised Byzantine agreement such as PBFT [18] does not require the second step, but for the FBA, the second step is essential because the device in the intersection of two quorums might receive messages that contain the counter-statement $\overline{\alpha}$ from the other quorum and the number of messages might be larger than the threshold value. This can make the two devices in the same quorum, the one inside the intersection and the other outside, to accept two controversial statements.

If we look at the details of the SCP, it consists of two sub-protocols: the Nomination protocol and the Ballot protocol. Each sub-protocol uses the FBA to generate the results for them. For each iteration, the SCP generates a new block and adds it to the chain. The Nomination protocol generates a set of nominated values from the higher-level inputs, a set of transactions for the Stellar cryptocurrency. Those nominated values are the ones that are agreed by the FBA during the execution of the Nomination protocol. The Ballot protocol generates a new block which contains a set of nominated values agreed by the FBA. Both sub-protocols are simultaneously executed so that the Nomination protocol can nominate more values during the execution of the Ballot protocol. The Ballot protocol can combine multiple nominated values for one block and the number of nominated values in the new block depends on how the FBA is agreed.

In order to understand the details of each sub-protocol of the SCP, it is important to know how each device maintains its state and what kind of information is contained in the messages.

9.1.1 The Nomination Protocol

As for the Nomination protocol, each device stores four sets of data for a single block: i) a set of values that the node has voted, ii) a set of values that the node has accepted, iii) a set of values that the node has confirmed and iv) a set of the latest messages received from other nodes. Notice that each device stores only the latest messages because all

the messages from a single node are ordered by their contents. A message for the Nomination protocol contains three kinds of information: sender's id, a set of sender's voted values and a set of sender's accepted values. When a device receives a message, it finds the values that are voted or accepted more times than the threshold value and adds them to the accepted values' set or the confirmed values' set, respectively. The members of the set of values that the node has confirmed are the ones that are nominated by the Nomination protocol.

9.1.2 The Ballot Protocol

Unlike the Nomination protocol which stays in a single state during the computation, the Ballot protocol has three states: PREPARE, COMMIT, EXTERNALIZE. When a value is nominated by the Nomination protocol, the Ballot protocol initiates its computation as the PREPARE state. Until the Ballot protocol reaches the COMMIT state, the Nomination protocol is being executed concurrently, nominating more values. Then, the EXTERNALIZE state means that the content of the new block has been confirmed by the protocol.

A ballot, a base unit of the Ballot protocol, is a pair of a positive integer number and a combined value, and all ballots are totally ordered. The Ballot protocol uses the FBA to agree on following two statements: 'a ballot b is *prepared*' and '*commit* b'. Here, *prepared* means that the opposite statements of all ballots that are less than b are agreed. In order to vote for a statement *commit* b, b has to be *prepared*.

For the Ballot protocol, each device stores five ballots (b, p, p', c, h) and a set of the latest messages received from other nodes: b is a current ballot that are trying to be prepared, p and p' are the two highest (orderly) ballots accepted for being prepared and the meanings of c and h depend on the state of the device. The main goal of the PREPARE state is to confirm ballots to be prepared and to accept the commit statements for those prepared ballots. Before any ballots are voted for the commit statements, h contains the highest ballot confirmed for being prepared if there are any. Once the device has voted for any of the commit statements, c and h store the lowest and highest ballots voted for the commit statements. Then, accepting a commit statement transfers the state of the device from PREPARE to COMMIT. During the COMMIT state, the device tries to confirm the commit statements, and c and h store the lowest and highest ballots accepted for the commit statements. When a commit statement is confirmed, the device moves from the COMMIT state to the EXTERNALIZE state and the combined value in the commit statement will be the content of the newly agreed block.

9.2 Formal Modelling for the SCP

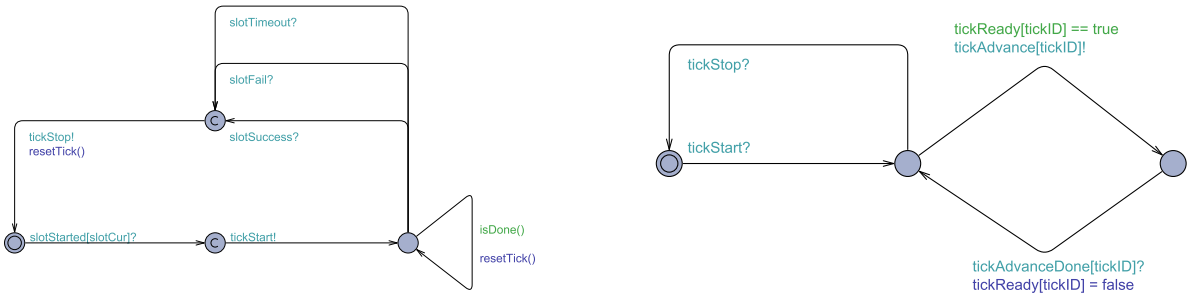
This section explains in detail how the SCP is formally modeled for the UPPAAL model checker program [5]. Before explaining our model for the SCP, it is necessary to introduce our four design templates for modelling other blockchain consensus protocols similar to the SCP. Those protocols are categorised as the voting-based consensus protocols [41].

9.2.1 Four Design Templates for Vote-based Consensus Protocols

This subsection discusses i) what kinds of common functionalities exist for the voting-based consensus protocols and ii) how we devised the templates for modelling those functionalities.

- **Template for the management of messages.** Exchanging the messages is one of the most important functionalities for the voting-based consensus protocols including the SCP. Based on the received messages, each device makes essential decisions such as which device will generate new blocks, what kind of information will be added to new blocks, or which block will be added to the chain of blocks. Furthermore, those messages are commonly stored and managed by each device in the system so that they can be used for the computation of the protocol. However, modelling the message management functionality the same as how it is implemented in the real systems is not a practical decision considering a large number of states it can generate. Therefore, in order to reduce the number of states for the management of messages, we introduce the Global Message Queue (GMQ). When a device generates a new message to send, it stores the new message in the GMQ instead of sending it to the destination. Then, other devices read the message from the GMQ whenever they need them for the computation. The behaviours of the GMQ are identical to the ones of the message management system which cannot fail to deliver messages, without any delays.

- **Template for the management of timers.** Timers are commonly used in various protocols to inform when computations should start, when states should be updated or when computations should stop. Timers in the real systems work based on computer clocks and it is straightforward to use them. However, for the timed automata based model checker programs including UPPAAL [5], clocks work differently from the ones in the real systems [42].



(a) Tick Manager (b) Tick Worker

Figure 9.1: Templates for Tick Manager and Tick Worker

Figure 9.1 shows the templates for Tick Manager and Tick Worker which provide the clock functionality for UPPAAL that is similar to the one in the real system. Here, *tick* indicates a certain amount of clocks that are enough to do some meaningful actions for the device process. Here, *process* means the instance of execution of a given template. The Tick Manager process manages multiple Tick Worker processes, and the number of Tick Worker processes is identical to the total number of device processes in the model. Each Tick Worker process sends the *tickAdvance* synchronisation message to its device process, and the device process consumes the tick to process its computation. After the computation, the device sends *tickAdvanceDone* synchronisation message to its Tick Worker. The Tick Manager ensures that each Tick Worker process sends the *tickAdvance* synchronisation message once in a random order per iteration. This makes sure that ticks are evenly distributed for every device in the system.

- **Template for the management of members.** During the executions of the protocols, members (devices) of the protocols might join the network or leave the network. This functionality is called *open membership*. Most of the blockchain consensus protocols claim that they support open membership, but they are poorly-documented. In order to model the open membership functionality, our template reloads the membership information every time the computation for the new block has started and the total number of members for each block is parameterised. This parameterised total number can control the addition or removal of the members. We capture this by the notion of a *semi-open membership*.

- **Template for the management of faulty members.** The devices in the system can fail during the computation of the protocol. If this happens, the protocol should achieve an agreement with those faulty devices unless the number of faulty devices is larger than the threshold value. The behaviours of the faulty device are too various to be modelled for the model checker programs. The majority of those behaviours is to prevent the device from voting for the correct results. Hence, we

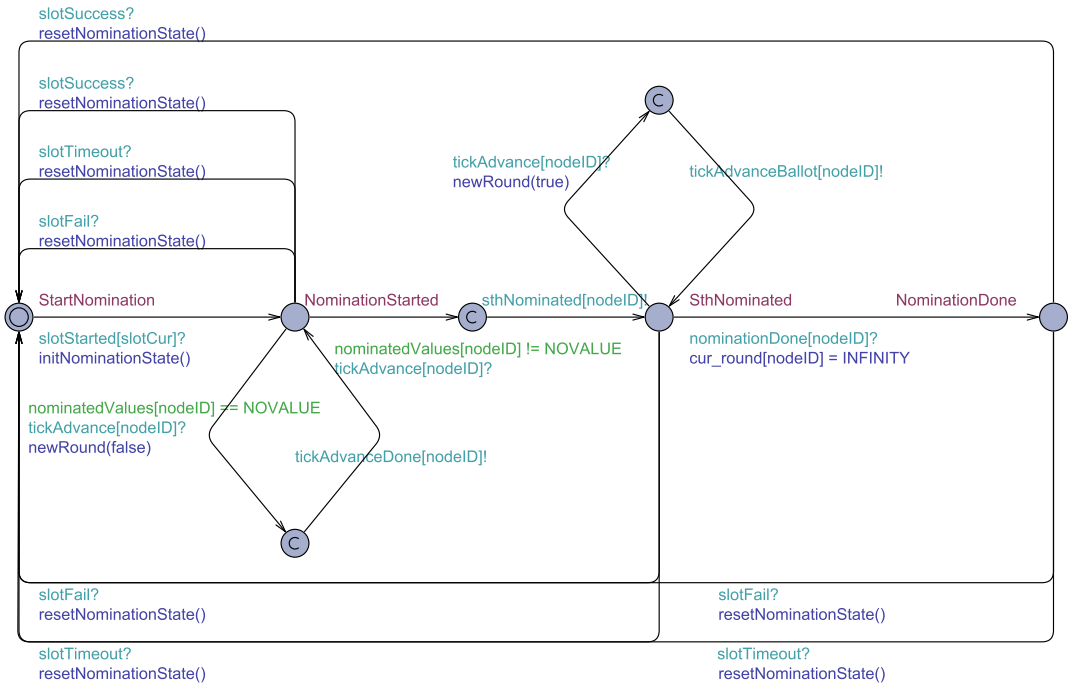


Figure 9.2: Template for the Nomination Protocol

abstract the behaviours of faulty devices as doing nothing. This cannot cover all the behaviours of the faulty devices, but it can significantly reduce the number of states.

9.2.2 Design Templates for the SCP

We now describe the details of the formal models for the SCP. Note that in the SCP, a block of the blockchain is called a slot. Figure 9.2 presents the template for the Nomination protocol of the SCP. The Nomination protocol periodically repeat the same behaviours and each iteration (the repeated behaviours) is called *round*. At the beginning of each round, the protocol sets the timer so that it waits for a certain amount of time. The waiting time is identical to the round number and this round number increases by one once the computation of one round is completed. If the progress of a device is faster than others, the device will have a round number larger than others and, consequently, this will make the device wait longer than others. This can increase the chance for the devices in the protocol to stay in the same round number, and this leads to the increase in the chance to achieve an agreement for the protocol. When the timer has been triggered, the device votes for either a new input value, if the device is the highest priority device, or the values that are voted by the highest priority device. As described in the previous subsection, the calculation for the timer is based on the tick messages (one tick is equal to one round number). In order to reduce the number of states, the template processes the received messages when the timer is triggered in-

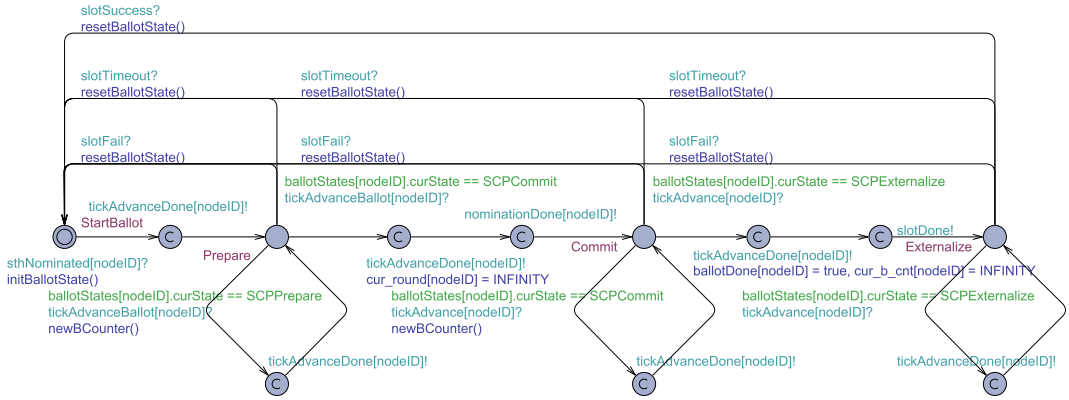


Figure 9.3: Template for the Ballot Protocol

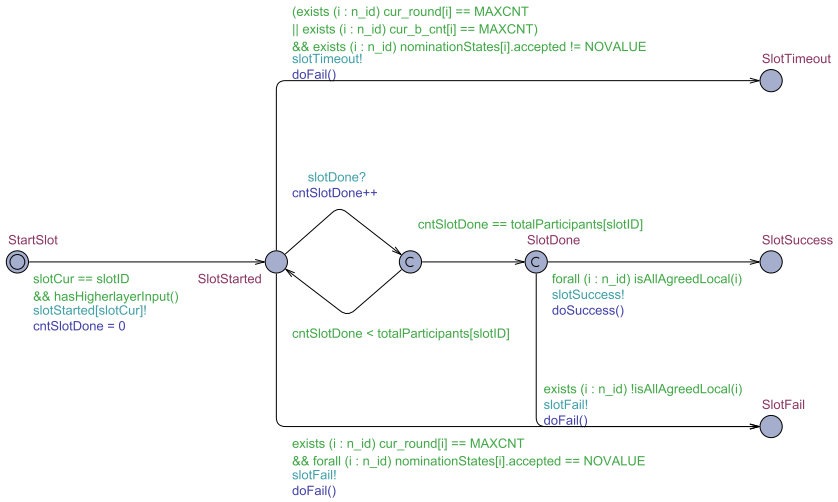


Figure 9.4: Template for the Slot Manager

stead of processing them when they are delivered. This delayed computation can be viewed as the delay in the message delivery. Notice that the template has two locations: *NominationStarted* and *SthNominated*. This is because the Nomination protocol is the one that begins the execution of the Ballot protocol when a value is nominated, and *sthNominated[nodeID]* is the synchronisation massage for that execution.

Figure 9.3 shows the template for the Ballot protocol of the SCP. As stated above, the *sthNominated[nodeID]* message makes the protocol start the execution. Like the way the Ballot protocol is implemented, the template for the Ballot protocol also has three locations: *Prepare*, *Commit* and *Externalize*. The new messages for the protocol are generated either when the new ballots are created by the nomination of new values or when the states of the statements for existing ballots change (accepted or confirmed). Similar to the Nomination protocol, the Ballot protocol also uses timers to process the received messages. The timer value equals to the integer number in the b ballot and that number increases when the combined value is updated. When the location of the template changes from *Prepare* to *Commit*, it sends the *nomination-*

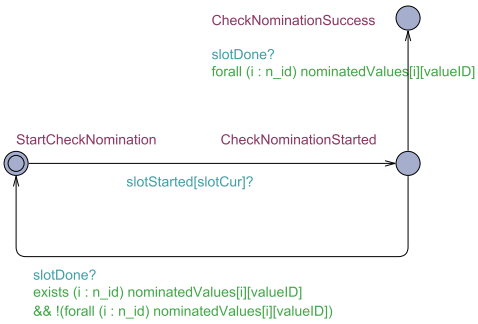


Figure 9.5: Template for the Nomination Protocol Checker

Done[nodeID] synchronisation message to the template for the Nomination protocol to finish the computation of the Nomination protocol - the same message can be found in Figure 9.2. The transition of the location from *Commit* to *Externalize* indicates that the computation for a slot (a block) is completed.

The template for the Slot Manager presented in Figure 9.4 is for the management of the execution of each slot (block) for the SCP. When the execution of a new slot generates the *slotStarted[slotCur]* synchronisation message, the message instantiates every template for the Nomination protocol - the same message can be seen in Figure 9.2. Then, the Slot Manger counts the number of templates that complete the computation of the Ballot protocol and checks whether the slot comes to an agreement.

Each sub-protocol of the SCP has an invariant to satisfy during the execution. For the Nomination protocol, once a value is nominated at a device, the value should be nominated at all the other non-faulty devices until the computation is completed. For the Ballot protocol, if the c ballot has a certain value, the c, h and b ballots should satisfy the following order: $c \sqsubseteq h \sqsubseteq b$. Here, $c \sqsubseteq h$ means that the combined values of c and h are identical and the positive integer value of c is less than or equal to the one of h.

$$
\begin{aligned}
A[\,]\ & forall\ (i : n_id)\ ballotStates[i].c.counter\ !=\ 0\ imply \\
& (ballotStates[i].c.value == ballotStates[i].h.value\ \&\& \\
& \quad ballotStates[i].c.counter <= ballotStates[i].h.counter)\ \&\& \\
& (ballotStates[i].h.value == ballotStates[i].b.value\ \&\& \\
& \quad ballotStates[i].h.counter <= ballotStates[i].b.counter)
\end{aligned}
\tag{9.1}
$$

There are two ways to check invariants in UPPAAL [5]: using a query for the verifier or using a template to observe the invariant. Figure 9.5 shows an observer template for the invariant of the Nomination protocol. It monitors the completion of the computation for each slot and checks whether every nominated value is nominated by all non-faulty devices. The invariant for the Ballot protocol is checked by the query for the verifier. Equation 9.1 above presents a full query to check the invariant of the three ballots.

#	Configuration	Memory Usage Verification Time
1	3 Nodes	13.90 MB 0.52 s
2	4 Nodes	201.64 MB 74.0 s
3	5 Nodes	4.45 GB 3,588.55 s

Table 9.1: Without the Four Design Templates[49]

#	Configuration	Memory Usage Verification Time
1	3 Nodes	60.25 MB 0.35 s
2	4 Nodes	372.69 MB 16.6 s
3	5 Nodes	8.53 GB 458.82 s
4	6 Nodes	206.47 GB 13,861.84 s

Table 9.2: With the Four Design Templates

9.3 Verification Results for the SCP

This section presents the verification results of our model for the SCP. The results are obtained from 'verifier' of the UPPAAL model checker [5]. Table 9.1 and Table 9.2 show the result of the verification executions using the configuration with a single quorum. The verification results in Table 9.1 come from the execution of the formal model of the SCP without using the four design templates described in Section 9.2.1 [49]. The results of the execution that used the four design templates are presented in Table 9.2. It has been expected that the memory usage increases for the execution with the templates because the templates enable more complicated scenarios, such as a scenario with open membership, to be verified. Yet, the correct support for the timer functionality in the four design templates significantly reduces the amount of verification time. For the five node configuration, the time taken reduced from 3,588.55 seconds to 458.82 seconds.

Figure 9.6 depicts the verification result for the scenario with open membership. The execution starts with four nodes and a new node joins the network after the first slot is agreed. Then, the newly joined node leaves the network once the second slot has been agreed. This execution used 358.32 GB memory and it took 23,992.2 seconds.

Our semi-open membership template can also be used to verify the cascade theorem [33]. The theorem indicates that the information about faulty nodes has to be propagated and the quorum configuration also has to be updated accordingly. Figure 9.7 presents the verification result for the cascade theorem. The execution starts with six nodes and one of the six nodes is a faulty node. The agreement of the first slot is expected to fail because the threshold value is 100 per cent which means that all of the six nodes have to be agreed for the successful agreement. Then, the execution for the second slot starts with five nodes and the threshold value remains the same. This implies that the information about one faulty node has been propagated. The

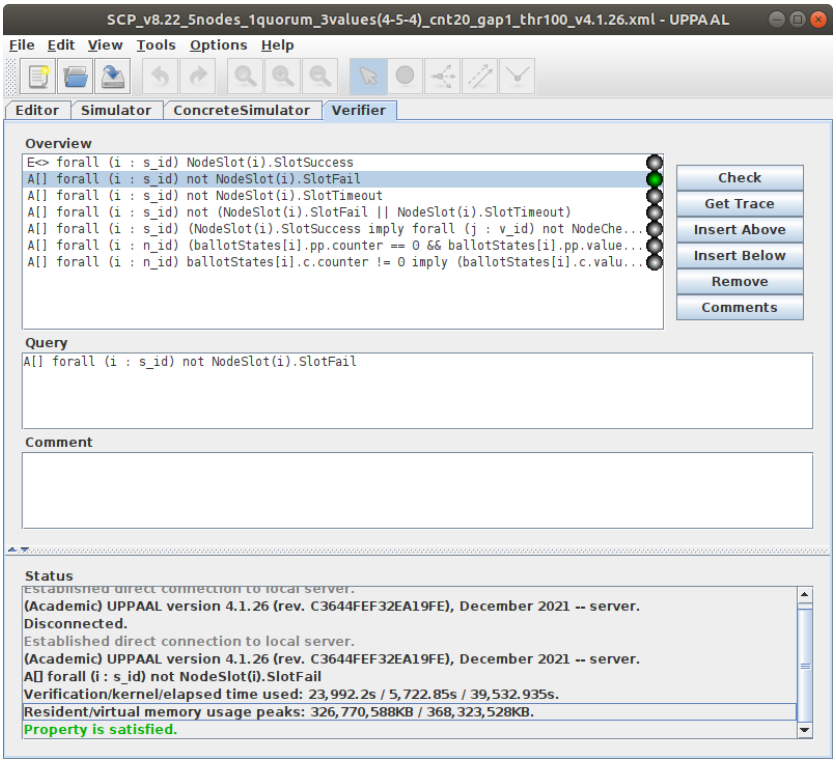


Figure 9.6: Verification Result for Open Membership

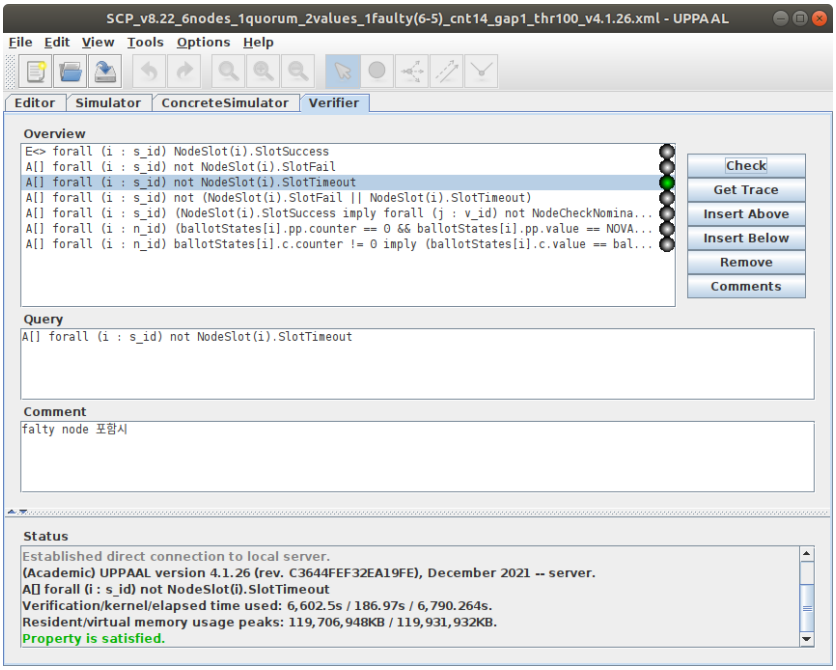


Figure 9.7: Verification Result for the Cascade Theorem

agreement for the second slot is successful because of the propagation. This execution used 119.71 GB memory and the time spent on the verification was 6,602.5 seconds.

Chapter 10

Conclusion

10.1 Summary

This research attempts to provide a solution to a well-known programmability issue in weakly consistent distributed systems, namely, the generation of counter-intuitive results. This is a key challenge relating to developing programs in weakly consistent distributed systems that originates from the lack of key principles, prerequisites and optimisation techniques for designing and supporting programming languages in weakly consistent distributed systems. By tackling this issue, this research shows that it is possible to provide some formal guarantee on the executions of programs run in weakly consistent distributed systems.

The effort we made to fill the research gaps and resolve the programmability issue is fourfold: i) providing a new programming language and its runtime system, ii) defining key properties for the new language and providing proofs for the theoretical properties of the language, iii) extending the new language to support aggregation operations and iv) developing three optimisation techniques for improving the performance of programs written in the new language. We also carried out a case study of a blockchain consensus protocol, the SCP, using formal modelling and verification.

First, we introduced the new programming language for weakly consistent distributed systems and its runtime system. The new language, an extension of Lasp [35], is based on a new language design which is characterised by the more extensive use of *provenance* and the introduction of a new concept, *cover*. By using provenance and cover, it has been possible for us to provide the $\texttt{read}_c()$ operation which extracts the complete computation results from the local state of a replica and, thus, preventing the occurrence of the counter-intuitive results.

Our new language provides two kinds of data types: the system-level data type and the client-level data types. The system-level data type is the OR_{PC} data type. The client-level data types (the LWW register, the G counter and the OR bag data types) are built upon the system-level data type and they are differentiated by the partial orders

for events. Those partial orders for events are used to express the constraints of each data type. For example, one of the constrains of the LWW register data type is that the least written value has to be stored if multiple values are concurrently written. It means that the LWW register data type requires a method to compare the times any two values are written. Totally ordered events behave like timestamp for any written value for the LWW register data type and, therefore, they can be used for the comparison of written times. This two-layered implementation (system-level and client-level) ensures that the addition of new client-level data types can be made without concerns about breaking the theoretical properties satisfied by the system-level data type. Furthermore, our new language supports operations that have the input parameters with different client-level data types because their data types are internally the same (the system-level data type).

Second, we proved the theoretical properties of our new language, the correspondence property and the consistent read property. The correspondence property guarantees the convergence of our runtime system and contains information about how the converging state looks like. The provision of information on the converging state is an advanced feature of our correspondence property when compared to the plain convergence guarantee. Our consistent read property guarantees the obtainment of complete computation results and the results can be retrieved using the new property we introduced, *strong consistency in weakly consistent distributed systems*. We demonstrated that if a result satisfies the strong consistency property, the result contains completely computed values only and this means that the result does not have any counter-intuitive values. The proven theoretical properties of our new language and its runtime system guarantee that the programs written in our new language do not generate any counter-intuitive results. We also proved that the operations for aggregation still satisfy the theoretical properties.

Third, the new design also allows our new language to support operations for aggregation such as `agg_cnt`() and `agg_sum`(), which are considered difficult to be supported in weakly consistent distributed systems. The main difficulty in supporting operations for aggregation arises from the possibility that the local state of a replica can have partially computed results. Since the partially computed results are either incorrect or meaningless, they would only have correct values once the complete computation results have been obtained. We overcame this difficulty by using the method that the $\texttt{read}_c$() operation has used for the extraction of the complete computation results. Consequently, the outcomes of operations for aggregation are the collections of completely-computed values.

Fourth, we developed three optimisation techniques to improve the performance of programs written in our new language: i) the removal of redundant information in the definition of the OR_{PC} data type, ii) the synchronisation filter mode and iii) the delta

state with the incremental computation mode and caches. This thesis clearly shows that the use of the three techniques significantly contributes to the reduction of the amount of transmitted data and the processing time. As for the first optimisation technique, the result shows that the execution with the optimised version of the OR_{PC} data type leads to a considerable reduction in the amount of transmitted data and the processing time to reach the converged state. The result of the second optimisation technique suggests that enabling the synchronisation filter mode reduces the amount of transmitted data as the nodes in the exclusion set do not perform the periodic exchanges. The key to the last optimisation technique is the correct use of the delta state. Using the delta state alone cannot reduce the amount of computation due to the way the `read`() operation works. The operation requires the entire state to generate the outcome. The incremental computation mode ensures that the computation uses only the delta state. The cache table for the third optimisation is required to obtain correct outcomes of the operations for aggregation. The result of this optimisation technique indicates that the use of this technique reduces the amount of transmitted data significantly. Our experiment results show that with these optimisation techniques, our new system shows the acceptable performance. In particular, this thesis shows that the example program for BigBench [21] discussed in Section 8.4.3 can process 722,268 input rows with program domain specific optimisations.

As well as the contributions discussed above, we also studied one of the blockchain consensus protocols, the SCP [34]. In order to check the soundness of the protocol, we have used the UPPAAL model checker [5] for formal modeling and verification. By modeling the SCP, we have provided four design templates that can be used for modeling other blockchain consensus protocols that are similar to the SCP: i) the Global Message Queue template, ii) the timer management template through the Tick Manager and Tick Worker templates, iii) the semi-open membership template and iv) the faulty device template. The verification results show that our model with the four design templates satisfies the invariants of the SCP and it can verify the soundness of the protocol with given quorum configurations. Moreover, the amount of CPU and memory used in the verification of our model was also reasonable, considering the lines of code for the implementation of the SCP.

10.2 Future Work

This section discusses areas for future research. First, more optimisation techniques can be added to improve the performance of programs executed on our runtime system. Among the optimisation techniques can be the reduction of repeat computations and the reduction of the amount of metadata. As for the former, it can be ensured that

input nodes check whether the output nodes have the outcomes of the operations for every new update. In the current implementation, if an input node receives the updates that the output node has already received, the input node still computes the dataflow operation and sends the outcome to the output node. Such redundant computations are removed, if input nodes check the output nodes have the operation outcomes for every new update. As for the latter, one of the ways of reducing the amount of metadata would be the use of garbage collection for completely-computed metadata. The size of the metadata increases continuously in the current implementation and it affects the performance of the computation. The use of garbage collection removes the unnecessary metadata and thus improves the performance of programs.

Second, future research can work on improving the expressiveness of the language. There are three ways of achieving this: i) supporting more client-level data types, ii) supporting more dataflow operations and iii) supporting more functionality of the new programming language.

- **Supporting more client-level data types** As discussed in Chapter 2, many replicated data types are introduced and studied in the distributed systems community. Those data types can be added after their constraints are studied and applied to our partial order relation described in Section 6.1.

- **Supporting more dataflow operations** Supporting more dataflow operations can lead to greater expressiveness of the language. For example, the fold operation is commonly supported by set or bag data types. However, it is particularly challenging to support this operation on weakly consistent distributed systems because it requires the loop functionality for the calculation of results. Our aggregation operation could be a stepping stone to finding a solution to this challenge.

- **Supporting more functionality of the new programming language** In the current implementation, programs with cycles in dataflow graph and loop statement cannot be executed correctly. Supporting loop and cycle would significantly improve the expressiveness of the language. Future research can look into ways of supporting those functionalities.

Third, developing a hybrid system would allow more efficient execution of programs run on weakly consistent distributed systems. Hybrid here means three different levels of consistency being supported on the systems, namely, No-CRDT, Lasp (without provenance and cover) and Lasp_{PC}. The No-CRDT mode can be used in cases where fault tolerance, monotonicity and the consistent read functionality are not needed. If only fault tolerance and monotonicity are required without the need for the consistent read functionality, the Lasp mode can be used. The Lasp_{PC} mode supports all three

functionalities and it can be used for important computations that require consistent results.

Forth, it is useful to have a tool that automatically generates the exclusion set, that satisfies the three rules described in Section 8.3.2 and maximises the improvement. The exclusion set is filled by parameters of the execution when we use the synchronisation filter mode. It means that the content of the set is manually selected. Therefore, it is possible that the selection of the set could violate the three rules. Furthermore, it is difficult to know that the selection maximises the optimisation without the tool.

Last, the four design templates we used for formal modeling and verification of the SCP can also be used for other blockchain consensus protocols similar to the SCP, the voting-based consensus protocols. In particular, Tendermint[1], one of the voting-based consensus protocols based on Byzantine Fault Tolerance (BFT), can be a good candidate for that.

[1] `https://tendermint.com/core/`

Bibliography

[1] Dippy Aggarwal, Shreyas Shekhar, Chris Elford, Umachandar Jayachandran, Sadashivan Krishnamurthy, Jamie Reding, and Brendan Niebruegge. TPCx-BB (BigBench) in a Single-Node Environment. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 64–83. Springer, 2019.

[2] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *Journal of Parallel and Distributed Computing*, 111:162–173, 2018.

[3] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR*, pages 249–260. Citeseer, 2011.

[4] Yael Amsterdamer, Daniel Deutch, and Val Tannen. Provenance for Aggregate Queries. In *Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '11, pages 153–164, New York, NY, USA, 2011. ACM.

[5] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on Uppaal. In *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[6] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. An optimized conflict-free replicated set. Research Report RR-8083, INRIA, October 2012.

[7] Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. On Verifying Causal Consistency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 626–638, New York, NY, USA, 2017. Association for Computing Machinery.

[8] Ahmed Bouajjani, Constantin Enea, and Jad Hamza. Verifying Eventual Consistency of Optimistic Replication Systems. In *Proceedings of the 41st ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, page 285–296, New York, NY, USA, 2014. Association for Computing Machinery.

[9] Manuel Bravo, Luís Rodrigues, and Peter Van Roy. Towards a Scalable, Distributed Metadata Service for Causal Consistency under Partial Geo-Replication. In *Proceedings of the Doctoral Symposium of the 16th International Middleware Conference*, Middleware Doct Symposium '15, New York, NY, USA, 2015. Association for Computing Machinery.

[10] Iwan Briquemont, Manuel Bravo, Zhongmiao Li, and Peter Van Roy. Conflict-Free Partially Replicated Data Types. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 282–289, 2015.

[11] Russell Brown, Zeeshan Lakhani, and Paul Place. Big (ger) sets: decomposed delta CRDT sets in Riak. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*, pages 1–5, 2016.

[12] Sebastian Burckhardt. Principles of Eventual Consistency. *Foundations and Trends® in Programming Languages*, 1(1-2):1–150, 2014.

[13] Sebastian Burckhardt. *Consistency in Distributed Systems*, pages 84–120. Springer International Publishing, Cham, 2015.

[14] Sebastian Burckhardt, Alexey Gotsman, and Hongseok Yang. Understanding eventual consistency. *Tech. Rep. MSR-TR-2013–39*, 2013.

[15] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 271–284, New York, NY, USA, 2014. Association for Computing Machinery.

[16] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[17] Alexandre Carlier, Igor Kopestenski, Dan Martens, and Peter Van Roy. *Lasp on grisp: implementation and evaluation of a general purpose edge computing system for internet of things*. PhD thesis, UCL-Ecole polytechnique de Louvain, 2018.

[18] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[19] Neil Conway, William R Marczak, Peter Alvaro, Joseph M Hellerstein, and David Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 1. ACM, 2012.

[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 205–220, New York, NY, USA, 2007. Association for Computing Machinery.

[21] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. BigBench: Towards an Industry Standard Benchmark for Big Data Analytics. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1197–1208, New York, NY, USA, 2013. ACM.

[22] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59, June 2002.

[23] Wojciech Golab, Muntasir R Rahman, Alvin AuYoung, Kimberly Keeton, and Xiaozhou Li. Eventually consistent: Not what you were expecting? *Communications of the ACM*, 57(3):38–44, 2014.

[24] Victor BF Gomes, Martin Kleppmann, Dominic P Mulligan, and Alastair R Beresford. Verifying strong eventual consistency in distributed systems. In *Proceedings of the ACM on Programming Languages*, volume 1, pages 1–28. ACM New York, NY, USA, 2017.

[25] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause I'm Strong Enough: Reasoning about Consistency Choices in Distributed Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 371–384, New York, NY, USA, 2016. Association for Computing Machinery.

[26] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance Semirings. In *Proceedings of the Twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium*

on Principles of Database Systems, PODS '07, pages 31–40, New York, NY, USA, 2007. ACM.

[27] Radha Jagadeesan and James Riely. Eventual Consistency for CRDTs. In *Programming Languages and Systems*, pages 968–995, Cham, 2018. Springer International Publishing.

[28] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. Mergeable replicated data types. In *Proceedings of the ACM on Programming Languages*, volume 3, pages 1–29. ACM New York, NY, USA, 2019.

[29] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[30] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: Certified Causally Consistent Distributed Key-Value Stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 357–370, New York, NY, USA, 2016. Association for Computing Machinery.

[31] Nicholas V Lewchenko, Arjun Radhakrishna, Akash Gaonkar, and Pavol Černỳ. Sequential programming for replicated data stores. In *Proceedings of the ACM on Programming Languages*, volume 3, pages 1–28. ACM New York, NY, USA, 2019.

[32] Hongjin Liang and Xinyu Feng. Abstraction for Conflict-Free Replicated Data Types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 636–650, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and Secure Global Payments with Stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 80–96, New York, NY, USA, 2019. Association for Computing Machinery.

[34] David Mazieres. The stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation*, 32:1–45, 2015.

[35] Christopher Meiklejohn and Peter Van Roy. Lasp: A Language for Distributed, Coordination-free Programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, PPDP '15, pages 184–195, New York, NY, USA, 2015. ACM.

[36] Christopher Meiklejohn and Peter Van Roy. Selective Hearing: An Approach to Distributed, Eventually Consistent Edge Computation. In *2015 IEEE 34th Symposium on Reliable Distributed Systems Workshop (SRDSW)*, pages 62–67, 2015.

[37] Christopher Meiklejohn and Peter Van Roy. The Implementation and Use of a Generic Dataflow Behaviour in Erlang. In *Proceedings of the 14th ACM SIGPLAN Workshop on Erlang*, Erlang 2015, page 39–45, New York, NY, USA, 2015. Association for Computing Machinery.

[38] Christopher S. Meiklejohn, Vitor Enes, Junghun Yoo, Carlos Baquero, Peter Van Roy, and Annette Bieniusa. Practical Evaluation of the Lasp Programming Model at Large Scale: An Experience Report. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, PPDP '17, page 109–114, New York, NY, USA, 2017. Association for Computing Machinery.

[39] Kartik Nagar and Suresh Jagannathan. Automated Parameterized Verification of CRDTs. In *Computer Aided Verification*, pages 459–477, Cham, 2019. Springer International Publishing.

[40] Sreeja S. Nair, Gustavo Petri, and Marc Shapiro. Proving the Safety of Highly-Available Distributed Objects. In *Programming Languages and Systems*, pages 544–571, Cham, 2020. Springer International Publishing.

[41] Giang-Truong Nguyen and Kyungbaek Kim. A survey about consensus algorithms used in blockchain. *Journal of Information processing systems*, 14(1):101–128, 2018.

[42] Guillermo Rodriguez-Navas and Julián Proenza. Using timed automata for modeling distributed systems with clocks: Challenges and solutions. *IEEE Transactions on Software Engineering*, 39(6):857–868, 2012.

[43] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354–368, 2011.

[44] Ilya Sergey, James R Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. In *Proceedings of the ACM on Programming Languages*, volume 2, pages 1–30. ACM New York, NY, USA, 2017.

[45] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, January 2011.

[46] D.B. Terry, A.J. Demers, K. Petersen, M.J. Spreitzer, M.M. Theimer, and B.B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, pages 140–149, 1994.

[47] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[48] Chao Wang, Constantin Enea, Suha Orhun Mutluergil, and Gustavo Petri. Replication-Aware Linearizability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 980–993, New York, NY, USA, 2019. Association for Computing Machinery.

[49] Junghun Yoo, Youlim Jung, Donghwan Shin, Minhyo Bae, and Eunkyoung Jee. Formal Modeling and Verification of a Federated Byzantine Agreement Algorithm for Blockchain Platforms. In *2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 11–21. IEEE, 2019.

[50] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. Formal Specification and Verification of CRDTs. In *Formal Techniques for Distributed Objects, Components, and Systems*, pages 33–48, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

Appendix A

Source Code of Experiments

A.1 The Replicated Leader Board

```
% input nodes
-define(USER_INFO, {<<"u">>, {?EXT_AWORBAG_INPUT_TYPE, []}}).
-define(GROUP_INFO, {<<"g">>, {?EXT_AWORBAG_INPUT_TYPE, []}}).
-define(DIVIDER, {<<"d">>, {?EXT_LWWREGISTER_INPUT_TYPE, []}}).

% product(<<"g">>, <<"u">>)
-define(GROUP_X_USER, {<<"gXu">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% filter(<<"gXu">>)
-define(GROUP_X_USER_F, {<<"gXuF">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% group_by_sum(<<"gXuF">>)
-define(GROUP_X_USER_F_G, {<<"gXuFG">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% order_by(<<"gXuFG">>)
-define(GROUP_X_USER_F_G_O, {<<"gXuFGO">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).

% set_count(<<"g">>)
-define(GROUP_C, {<<"gC">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% product(<<"gC">>, <<"d">>)
-define(GROUP_C_X_DIVIDER, {<<"gCXd">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% map(<<"gCXd">>)
-define(GROUP_C_X_DIVIDER_M, {<<"gCXdM">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).

% product(<<"gXuFGO">>, <<"gCXdM">>)
-define(
    GROUP_X_USER_F_G_O__X__GROUP_C_X_DIVIDER_M,
    {<<"gXuFGOXgCXdM">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% map(<<"gXuFGOXgCXdM">>)
```

```
-define(
    GROUP_X_USER_F_G_O__X__GROUP_C_X_DIVIDER_M__M,
    {<<"gXuFGOXgCXdMM">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

{InputSetUserId, InputSetUserType} = ?USER_INFO,
{ok, _} = lasp:declare(InputSetUserId, InputSetUserType),
{InputSetGroupId, InputSetGroupType} = ?GROUP_INFO,
{ok, _} = lasp:declare(InputSetGroupId, InputSetGroupType),
{InputRegisterDividerId, InputRegisterDividerType} = ?DIVIDER,
{ok, _} = lasp:declare(InputRegisterDividerId, InputRegisterDividerType),

{InterSetGroupXUserId, InterSetGroupXUserType} = ?GROUP_X_USER,
{ok, _} = lasp:declare(InterSetGroupXUserId, InterSetGroupXUserType),

ok = lasp:product(?GROUP_INFO, ?USER_INFO, ?GROUP_X_USER),

{InterSetGroupXUserMatchId, InterSetGroupXUserMatchType} = ?GROUP_X_USER_F,
{ok, _} =
    lasp:declare(InterSetGroupXUserMatchId, InterSetGroupXUserMatchType),

ok =
    lasp:filter(
        ?GROUP_X_USER,
        fun({{_GroupId, UserIdSet}, {UserId, _UserPoints}}) ->
            ordsets:is_element(UserId, UserIdSet)
        end,
        ?GROUP_X_USER_F),

{GroupBySumGroupXPointsId, GroupBySumGroupXPointsType} = ?GROUP_X_USER_F_G,
{ok, _} =
    lasp:declare(GroupBySumGroupXPointsId, GroupBySumGroupXPointsType),

ok =
    lasp:group_by_sum(
        ?GROUP_X_USER_F,
        fun(undefined, {UserIdR, UserPointsR}) ->
```

```
            {
                ordsets:add_element(UserIdR, ordsets:new()),
                0 + UserPointsR};
            ({UserIdL, UserPointsL}, {UserIdR, UserPointsR}) ->
                {
                    ordsets:add_element(UserIdR, UserIdL),
                    UserPointsL + UserPointsR}
        end,
        ?GROUP_X_USER_F_G),

{
    OrderedGroupBySumGroupXPointsId,
    OrderedGroupBySumGroupXPointsType} = ?GROUP_X_USER_F_G_O,
{ok, _} =
    lasp:declare(
        OrderedGroupBySumGroupXPointsId, OrderedGroupBySumGroupXPointsType),

ok =
    lasp:order_by(
        ?GROUP_X_USER_F_G,
        fun({{_GroupIdL,_GroupUsersL},{_GroupUsersSumedL,GroupPointsL}}=_ElemL,
            {{_GroupIdR,_GroupUsersR},{_GroupUsersSumedR,GroupPointsR}}=_ElemR) ->
            GroupPointsL >= GroupPointsR
        end,
        ?GROUP_X_USER_F_G_O),

{SetCountGroupId, SetCountGroupType} = ?GROUP_C,
{ok, _} = lasp:declare(SetCountGroupId, SetCountGroupType),

ok = lasp:set_count(?GROUP_INFO, ?GROUP_C),

{
    AggResultGroupCountXDividerId,
    AggResultGroupCountXDividerType} = ?GROUP_C_X_DIVIDER,
{ok, _} =
    lasp:declare(
        AggResultGroupCountXDividerId, AggResultGroupCountXDividerType),
```

```
ok = lasp:product(?GROUP_C, ?DIVIDER, ?GROUP_C_X_DIVIDER),

{
    AggGroupCountXDividerRankId,
    AggGroupCountXDividerRankType} = ?GROUP_C_X_DIVIDER_M,
{ok, _} =
    lasp:declare(
        AggGroupCountXDividerRankId, AggGroupCountXDividerRankType),

ok =
    lasp:map(
        ?GROUP_C_X_DIVIDER,
        fun({GroupCount, Divider}) ->
            {RankDivider, 100} =
                lists:foldl(
                    fun(Border, {AccInRankDivider, AccInBorderSum}) ->
                        NewBorderSum = AccInBorderSum + Border,
                        Rank = GroupCount * NewBorderSum div 100,
                        {AccInRankDivider ++ [Rank], NewBorderSum}
                    end,
                    {[], 0},
                    Divider),
            RankDivider
        end,
        ?GROUP_C_X_DIVIDER_M),

{
    AggResultGroupPointsOrderedXRankDividerId,
    AggResultGroupPointsOrderedXRankDividerType
} = ?GROUP_X_USER_F_G_O__X__GROUP_C_X_DIVIDER_M,
{ok, _} =
    lasp:declare(
        AggResultGroupPointsOrderedXRankDividerId,
        AggResultGroupPointsOrderedXRankDividerType),

ok =
    lasp:product(
        ?GROUP_X_USER_F_G_O,
```

```
        ?GROUP_C_X_DIVIDER_M,
        ?GROUP_X_USER_F_G_O__X__GROUP_C_X_DIVIDER_M),

{
    AggGroupRankId,
    AggGroupRankType} = ?GROUP_X_USER_F_G_O__X__GROUP_C_X_DIVIDER_M__M,
{ok, _} = lasp:declare(AggGroupRankId, AggGroupRankType),

ok =
    lasp:map(
        ?GROUP_X_USER_F_G_O__X__GROUP_C_X_DIVIDER_M,
        fun({GroupPointsOrdered, RankDivider}) ->
            {_Index, GroupRanked, _CurRank, _CurRankDivider} =
                lists:foldl(
                    fun
                        (
                            {{GroupId, _UserSet}, {_SumedUsers, _Points}},
                            {
                                AccInIndex,
                                AccInGroupRanked,
                                AccInCurRank,
                                [CurDivider | RestDividers]=AccInRankDivider}) ->
                            {NewCurRank, NewRankDivider} =
                                case AccInIndex == CurDivider of
                                    true ->
                                        {AccInCurRank + 1, RestDividers};
                                    false ->
                                        {AccInCurRank, AccInRankDivider}
                                end,
                            {
                                AccInIndex + 1,
                                ordsets:add_element(
                                    {GroupId, NewCurRank},
                                    AccInGroupRanked),
                                NewCurRank,
                                NewRankDivider};
                        (
                            {{GroupId, _UserSet}, {_SumedUsers, _Points}},
```

```
                                {AccInIndex, AccInGroupRanked, AccInCurRank, []}) ->
                                {
                                    AccInIndex + 1,
                                    ordsets:add_element(
                                        {GroupId, AccInCurRank},
                                        AccInGroupRanked),
                                    AccInCurRank,
                                    []}
                            end,
                            {0, ordsets:new(), 1, RankDivider},
                            GroupPointsOrdered),
                    GroupRanked
                end,
                ?GROUP_X_USER_F_G_O__X__GROUP_C_X_DIVIDER_M__M),

{ok, {NewSubset, NewCDS, ReadResult}} =
    lasp:query_ext_consistent(
        ?GROUP_X_USER_F_G_O__X__GROUP_C_X_DIVIDER_M__M, Subset, CDS),
```

A.2 The MapReduce Program

```
% input node
-define(INPUT_WEB_CLICK_STREAMS, {<<"web_cs">>, {?EXT_AWORBAG_INPUT_TYPE, []}}).
% filter(<<"web_cs">>)
-define(WCSF, {<<"web_csF">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% map(<<"web_csF">>)
-define(WCSFM, {<<"web_csFM">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% order_by(<<"web_csFM">>)
-define(WCSFMO, {<<"web_csFMO">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% map(<<"web_csFMO">>)
-define(WCSFMOM, {<<"web_csFMOM">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).

-define(SESSION_TIMEOUT, 3600).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

{
    InputBagWebClickStreamId,
```

```
    InputBagWebClickStreamType} = ?INPUT_WEB_CLICK_STREAMS,
{ok, _} = lasp:declare(InputBagWebClickStreamId, InputBagWebClickStreamType),

{InterSelectionWCSId, InterSelectionWCSType} = ?WCSF,
{ok, _} = lasp:declare(InterSelectionWCSId, InterSelectionWCSType),

ok =
    lasp:filter(
        ?INPUT_WEB_CLICK_STREAMS,
        fun({_, _, _, WCS_ITEM_SK, _, WCS_USER_SK}) ->
            WCS_ITEM_SK /= undefined andalso WCS_USER_SK /= undefined
        end,
        ?WCSF),

{InterProjectionSWCSId, InterProjectionSWCSType} = ?WCSFM,
{ok, _} = lasp:declare(InterProjectionSWCSId, InterProjectionSWCSType),

ok =
    lasp:map(
        ?WCSF,
        fun(
            {
                WCS_CLICK_DATE_SK, WCS_CLICK_TIME_SK, _,
                WCS_ITEM_SK, _, WCS_USER_SK}) ->
            {ClickDate, _} = string:to_integer(WCS_CLICK_DATE_SK),
            {ClickTime, _} = string:to_integer(WCS_CLICK_TIME_SK),
            {WCS_USER_SK, ClickDate * 24 * 60 * 60 + ClickTime, WCS_ITEM_SK}
        end,
        ?WCSFM),

{OrderedPSWCSId, OrderedPSWCSType} = ?WCSFMO,
{ok, _} = lasp:declare(OrderedPSWCSId, OrderedPSWCSType),

ok =
    lasp:order_by(
        ?WCSFM,
        fun({WCS_USER_SK_L, TimestampL, _WCS_ITEM_SK_L}=_ElemL,
            {WCS_USER_SK_R, TimestampR, _WCS_ITEM_SK_R}=_ElemR) ->
```

```
            WCS_USER_SK_L < WCS_USER_SK_R orelse
                (WCS_USER_SK_L == WCS_USER_SK_R andalso
                    TimestampL =< TimestampR)
        end,
        ?WCSFMO),

{InterMakeSessionIDGPSWCSId, InterMakeSessionIDGPSWCSType} = ?WCSFMOM,
{ok, _} =
    lasp:declare(InterMakeSessionIDGPSWCSId, InterMakeSessionIDGPSWCSType),

ok =
    lasp:map(
        ?WCSFMO,
        fun(ListSessionInfo) ->
            {ResultSet, _, _, _} =
                lists:foldl(
                    fun({WCS_USER_SK, Timestamp, WCS_ITEM_SK},
                        {
                            AccInResultSet, AccInCurUId, AccInLastClickTime,
                            AccInSessionCnt}) ->
                        {NewCurUId, NewLastClickTime, NewSessionCnt0} =
                            case AccInCurUId /= WCS_USER_SK of
                                true ->
                                    {WCS_USER_SK, Timestamp, 1};
                                false ->
                                    {
                                        AccInCurUId,
                                        AccInLastClickTime,
                                        AccInSessionCnt}
                            end,
                        NewSessionCnt1 =
                            case Timestamp - NewLastClickTime >
                                    ?SESSION_TIMEOUT of
                                true ->
                                    NewSessionCnt0 + 1;
                                false ->
                                    NewSessionCnt0
                            end,
```

```
                        NewResultSet =
                            ordsets:add_element(
                                {{NewCurUId, NewSessionCnt1}, WCS_ITEM_SK},
                                AccInResultSet),
                        {
                            NewResultSet, NewCurUId, NewLastClickTime,
                            NewSessionCnt1}
                    end,
                    {ordsets:new(), 0, 0, 1},
                    ListSessionInfo),
            ResultSet
        end,
        ?WCSFMOM),

{ok, {NewSubset, NewCDS, ReadResult}} =
    lasp:query_ext_consistent(?WCSFMOM, Subset, CDS),
```

A.3 BigBench Queries

A.3.1 Query 1

```
-define(INPUT_STORE_SALES, {<<"ss">>, {?EXT_AWORBAG_INPUT_TYPE, []}}).
% map(<<"ss">>)
-define(STORE_SALES_M, {<<"ssM">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% filter(<<"ss">>)
-define(STORE_SALES_M_F, {<<"ssMF">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).

-define(INPUT_ITEM, {<<"i">>, {?EXT_AWORBAG_INPUT_TYPE, []}}).
% map(<<"ss">>)
-define(ITEM_M, {<<"iM">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% filter(<<"ss">>)
-define(ITEM_M_F, {<<"iMF">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).

% product(<<"ssMF">>, <<"iMF">>)
-define(
    STORE_SALES_M_F_X_ITEM_M_F,
    {<<"ssMF_X_iMF">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% filter(<<"ssMF_X_iMF">>)
-define(
```

```
    STORE_SALES_M_F_X_ITEM_M_F__F,
    {<<"ssMF_X_iMF_F">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% map(<<"ssMF_X_iMF_F">>)
-define(
    STORE_SALES_M_F_X_ITEM_M_F__F_M,
    {<<"ssMF_X_iMF_FM">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% group_by_sum(<<"ssMF_X_iMF_FM">>)
-define(
    STORE_SALES_M_F_X_ITEM_M_F__F_M_G,
    {<<"ssMF_X_iMF_FMG">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% map(<<"ssMF_X_iMF_FMG">>)
-define(
    STORE_SALES_M_F_X_ITEM_M_F__F_M_G_M,
    {<<"ssMF_X_iMF_FMGM">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% group_by_sum(<<"ssMF_X_iMF_FMGM">>)
-define(
    STORE_SALES_M_F_X_ITEM_M_F__F_M_G_M_G,
    {<<"ssMF_X_iMF_FMGMG">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, [list]}}).
% map(<<"ssMF_X_iMF_FMGMG">>)
-define(
    STORE_SALES_M_F_X_ITEM_M_F__F_M_G_M_G_M,
    {<<"ssMF_X_iMF_FMGMGM">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).

-define(
    STORE_SALES_STORE_SET,
    ordsets:from_list(["10", "20", "33", "40", "50"])).
-define(ITEM_CATEGORY_SET, ordsets:from_list(["1", "2", "3"])).
-define(NUMBER_OF_PAIR_MINIMUM, 5).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

{InputBagStoreSalesId, InputBagStoreSalesType} = ?INPUT_STORE_SALES,
{ok, _} = lasp:declare(InputBagStoreSalesId, InputBagStoreSalesType),
{
    InterProjectionStoreSalesId,
    InterProjectionStoreSalesType} = ?STORE_SALES_M,
{ok, _} =
    lasp:declare(
```

```
        InterProjectionStoreSalesId, InterProjectionStoreSalesType),
ok =
    lasp:map(
        ?INPUT_STORE_SALES,
        fun(
            {
                _, _, SS_ITEM_SK, _, _, _, _, SS_STORE_SK, _,
                SS_TICKET_NUMBER, _, _, _, _, _, _, _, _, _, _, _, _, _}) ->
            {SS_TICKET_NUMBER, SS_ITEM_SK, SS_STORE_SK}
        end,
        ?STORE_SALES_M),
{
    CheckSelectionStoreSalesId,
    CheckSelectionStoreSalesType} = ?STORE_SALES_M_F,
{ok, _} =
    lasp:declare(CheckSelectionStoreSalesId, CheckSelectionStoreSalesType),
ok =
    lasp:filter(
        ?STORE_SALES_M,
        fun({_SS_TICKET_NUMBER, _SS_ITEM_SK, SS_STORE_SK}) ->
            ordsets:is_element(SS_STORE_SK, ?STORE_SALES_STORE_SET)
        end,
        ?STORE_SALES_M_F),

{InputBagItemId, InputBagItemType} = ?INPUT_ITEM,
{ok, _} = lasp:declare(InputBagItemId, InputBagItemType),
{InterProjectionItemId, InterProjectionItemType} = ?ITEM_M,
{ok, _} = lasp:declare(InterProjectionItemId, InterProjectionItemType),
ok =
    lasp:map(
        ?INPUT_ITEM,
        fun(
            {
                I_ITEM_SK, _, _, _, _, _, _, _, _, _, _, I_CATEGORY_ID,
                _, _, _, _, _, _, _, _, _, _}) ->
            {I_ITEM_SK, I_CATEGORY_ID}
        end,
        ?ITEM_M),
```

```
{CheckSelectionItemId, CheckSelectionItemType} = ?ITEM_M_F,
{ok, _} = lasp:declare(CheckSelectionItemId, CheckSelectionItemType),
ok =
    lasp:filter(
        ?ITEM_M,
        fun({_I_ITEM_SK, I_CATEGORY_ID}) ->
            ordsets:is_element(I_CATEGORY_ID, ?ITEM_CATEGORY_SET)
        end,
        ?ITEM_M_F),

{
    InterJoinStoreSalesItemId,
    InterJoinStoreSalesItemType} = ?STORE_SALES_M_F_X_ITEM_M_F,
{ok, _} =
    lasp:declare(InterJoinStoreSalesItemId, InterJoinStoreSalesItemType),
ok =
    lasp:cond_product(
        ?STORE_SALES_M_F,
        ?ITEM_M_F,
        fun({_SS_TICKET_NUMBER, SS_ITEM_SK, _SS_STORE_SK}) ->
            SS_ITEM_SK
        end,
        fun({I_ITEM_SK, _I_CATEGORY_ID}) ->
            I_ITEM_SK
        end,
        ?STORE_SALES_M_F_X_ITEM_M_F),
{
    InterProjectionJSSIId,
    InterProjectionJSSIType} = ?STORE_SALES_M_F_X_ITEM_M_F__F_M,
{ok, _} = lasp:declare(InterProjectionJSSIId, InterProjectionJSSIType),
ok =
    lasp:map(
        ?STORE_SALES_M_F_X_ITEM_M_F,
        fun(
            {
                {SS_TICKET_NUMBER, SS_ITEM_SK, _SS_STORE_SK},
                {_I_ITEM_SK, _I_CATEGORY_ID}}) ->
            {SS_TICKET_NUMBER, SS_ITEM_SK}
```

```
        end,
        ?STORE_SALES_M_F_X_ITEM_M_F__F_M),
{GroupBySumJSSIId, GroupBySumJSSIType} = ?STORE_SALES_M_F_X_ITEM_M_F__F_M_G,
{ok, _} = lasp:declare(GroupBySumJSSIId, GroupBySumJSSIType),
ok =
    lasp:group_by_sum(
        ?STORE_SALES_M_F_X_ITEM_M_F__F_M,
        fun(undefined, SS_ITEM_SK) ->
            ordsets:add_element(SS_ITEM_SK, ordsets:new());
        (SetSS_ITEM_SK, SS_ITEM_SK) ->
            ordsets:add_element(SS_ITEM_SK, SetSS_ITEM_SK)
        end,
        ?STORE_SALES_M_F_X_ITEM_M_F__F_M_G),
{
    InterMakePairSSItemsId,
    InterMakePairSSItemsType} = ?STORE_SALES_M_F_X_ITEM_M_F__F_M_G_M,
{ok, _} = lasp:declare(InterMakePairSSItemsId, InterMakePairSSItemsType),
ok =
    lasp:map(
        ?STORE_SALES_M_F_X_ITEM_M_F__F_M_G,
        fun({_SS_TICKET_NUMBER, SetSS_ITEM_SK}) ->
            MakePairSSItems = make_pair_sale_stores_items(SetSS_ITEM_SK),
            ordsets:fold(
                fun({SS_ITEM_SK1, SS_ITEM_SK2}, AccInResult) ->
                    AccInResult ++ [{{SS_ITEM_SK1, SS_ITEM_SK2}, 1}]
                end,
                [],
                MakePairSSItems)
        end,
        ?STORE_SALES_M_F_X_ITEM_M_F__F_M_G_M),
{
    GroupBySumPairSSItemsId,
    GroupBySumPairSSItemsType} = ?STORE_SALES_M_F_X_ITEM_M_F__F_M_G_M_G,
{ok, _} = lasp:declare(GroupBySumPairSSItemsId, GroupBySumPairSSItemsType),
ok =
    lasp:group_by_sum(
        ?STORE_SALES_M_F_X_ITEM_M_F__F_M_G_M,
        fun(undefined, PairCnt) ->
```

```
            PairCnt;
        (PairCntSum, PairCnt) ->
            PairCntSum + PairCnt
        end,
        ?STORE_SALES_M_F_X_ITEM_M_F__F_M_G_M_G),
{
    InterSelectionPairSSItemsId,
    InterSelectionPairSSItemsType} = ?STORE_SALES_M_F_X_ITEM_M_F__F_M_G_M_G_M,
{ok, _} =
    lasp:declare(InterSelectionPairSSItemsId, InterSelectionPairSSItemsType),
ok =
    lasp:map(
        ?STORE_SALES_M_F_X_ITEM_M_F__F_M_G_M_G,
        fun(ListElem) ->
            lists:filter(
                fun({{_SS_ITEM_SK1, _SS_ITEM_SK2}, PairCnt}) ->
                    PairCnt > ?NUMBER_OF_PAIR_MINIMUM
                end,
                ListElem)
        end,
        ?STORE_SALES_M_F_X_ITEM_M_F__F_M_G_M_G_M),

{ok, {NewSubset, NewCDS, ReadResult}} =
    lasp:query_ext_consistent(
        ?STORE_SALES_M_F_X_ITEM_M_F__F_M_G_M_G_M, Subset, CDS),
```

A.3.2 Query 24

```
-define(INPUT_ITEM, {<<"i">>, {?EXT_AWORBAG_INPUT_TYPE, []}}).
% filter(<<"i">>)
-define(IF, {<<"iF">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% map(<<"iF">>)
-define(IFM, {<<"iFM">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).

-define(INPUT_ITEM_MARKET_PRICES, {<<"imp">>, {?EXT_AWORBAG_INPUT_TYPE, []}}).
% filter(<<"imp">>)
-define(IMPF, {<<"impF">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% map(<<"impF">>)
-define(IMPFM, {<<"impFM">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
```

```
% product(<<"iFM">>, <<"impFM">>)
-define(
    IFM_IMPFM_X, {<<"iFM_impFM_X">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% map(<<"iFM_impFM_X">>)
-define(
    IFM_IMPFM_XM, {<<"iFM_impFM_XM">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).

-define(INPUT_STORE_SALES, {<<"ss">>, {?EXT_AWORBAG_INPUT_TYPE, []}}).
% filter(<<"ss">>)
-define(SSF, {<<"ssF">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% map(<<"ssF">>)
-define(SSFM, {<<"ssFM">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).

% product(<<"iFM_impFM_XM">>, <<"ssFM">>)
-define(
    IFM_IMPFM_XM_SSFM_X,
    {<<"iFM_impFM_XM_ssFM_X">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% map(<<"iFM_impFM_XM_ssFM_X">>)
-define(
    IFM_IMPFM_XM_SSFM_XM,
    {<<"iFM_impFM_XM_ssFM_XM">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% group_by_sum(<<"iFM_impFM_XM_ssFM_XM">>)
-define(
    IFM_IMPFM_XM_SSFM_XMG,
    {<<"iFM_impFM_XM_ssFM_XMG">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).

-define(INPUT_WEB_SALES, {<<"ws">>, {?EXT_AWORBAG_INPUT_TYPE, []}}).
% filter(<<"ws">>)
-define(WSF, {<<"wsF">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% map(<<"wsF">>)
-define(WSFM, {<<"wsFM">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).

% product(<<"iFM_impFM_XM">>, <<"wsFM">>)
-define(
    IFM_IMPFM_XM_WSFM_X,
    {<<"iFM_impFM_XM_wsFM_X">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% map(<<"iFM_impFM_XM_wsFM_X">>)
```

```
-define(
    IFM_IMPFM_XM_WSFM_XM,
    {<<"iFM_impFM_XM_wsFM_XM">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% group_by_sum(<<"iFM_impFM_XM_wsFM_XM">>)
-define(
    IFM_IMPFM_XM_WSFM_XMG,
    {<<"iFM_impFM_XM_wsFM_XMG">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).

% product(<<"iFM_impFM_XM_ssFM_XMG">>, <<"iFM_impFM_XM_wsFM_XMG">>)
-define(
    IFM_IMPFM_XM_SSFM_XMG_IFM_IMPFM_XM_WSFM_XMG_X,
    {
        <<"iFM_impFM_XM_ssFM_XMG_iFM_impFM_XM_wsFM_XMG_X">>,
        {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% filter(<<"iFM_impFM_XM_ssFM_XMG_iFM_impFM_XM_wsFM_XMG_X">>)
-define(
    IFM_IMPFM_XM_SSFM_XMG_IFM_IMPFM_XM_WSFM_XMG_XF,
    {
        <<"iFM_impFM_XM_ssFM_XMG_iFM_impFM_XM_wsFM_XMG_XF">>,
        {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% map(<<"iFM_impFM_XM_ssFM_XMG_iFM_impFM_XM_wsFM_XMG_XF">>)
-define(
    IFM_IMPFM_XM_SSFM_XMG_IFM_IMPFM_XM_WSFM_XMG_XFM,
    {
        <<"iFM_impFM_XM_ssFM_XMG_iFM_impFM_XM_wsFM_XMG_XFM">>,
        {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% group_by_sum(<<"iFM_impFM_XM_ssFM_XMG_iFM_impFM_XM_wsFM_XMG_XFM">>)
-define(
    IFM_IMPFM_XM_SSFM_XMG_IFM_IMPFM_XM_WSFM_XMG_XFMG,
    {
        <<"iFM_impFM_XM_ssFM_XMG_iFM_impFM_XM_wsFM_XMG_XFMG">>,
        {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% map(<<"iFM_impFM_XM_ssFM_XMG_iFM_impFM_XM_wsFM_XMG_XFMG">>)
-define(
    IFM_IMPFM_XM_SSFM_XMG_IFM_IMPFM_XM_WSFM_XMG_XFMGM,
    {
        <<"iFM_impFM_XM_ssFM_XMG_iFM_impFM_XM_wsFM_XMG_XFMGM">>,
        {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
```

```
-define(ITEM_SK, "10000").

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

{InputBagItemId, InputBagItemType} = ?INPUT_ITEM,
{ok, _} = lasp:declare(InputBagItemId, InputBagItemType),
{
    InputBagItemMarketPricesId,
    InputBagItemMarketPricesType} = ?INPUT_ITEM_MARKET_PRICES,
{ok, _} =
    lasp:declare(InputBagItemMarketPricesId, InputBagItemMarketPricesType),
{InputBagStoreSalesId, InputBagStoreSalesType} = ?INPUT_STORE_SALES,
{ok, _} = lasp:declare(InputBagStoreSalesId, InputBagStoreSalesType),
{InputBagWebSalesId, InputBagWebSalesType} = ?INPUT_WEB_SALES,
{ok, _} = lasp:declare(InputBagWebSalesId, InputBagWebSalesType),

{InterSelectionIId, InterSelectionIType} = ?IF,
{ok, _} = lasp:declare(InterSelectionIId, InterSelectionIType),

ok =
    lasp:filter(
        ?INPUT_ITEM,
        fun(
            {
                I_ITEM_SK, _, _, _, _, _, _, _, _, _, _, _, _,
                _, _, _, _, _, _, _, _, _}) ->
            I_ITEM_SK == ?ITEM_SK
        end,
        ?IF),

{InterSelectionIMPId, InterSelectionIMPType} = ?IMPF,
{ok, _} = lasp:declare(InterSelectionIMPId, InterSelectionIMPType),

ok =
    lasp:filter(
        ?INPUT_ITEM_MARKET_PRICES,
        fun({_, IMP_ITEM_SK, _, _, _, _}) ->
```

```
            IMP_ITEM_SK == ?ITEM_SK
        end,
        ?IMPF),

{InterProjectionSIId, InterProjectionSIType} = ?IFM,
{ok, _} = lasp:declare(InterProjectionSIId, InterProjectionSIType),

ok =
    lasp:map(
        ?IF,
        fun(
            {
                I_ITEM_SK, _, _, _, _, I_CURRENT_PRICE, _, _, _, _,
                _, _, _, _, _, _, _, _, _, _, _, _}) ->
            {CurrentPrice, _} = string:to_float(I_CURRENT_PRICE),
            {I_ITEM_SK, CurrentPrice}
        end,
        ?IFM),

{InterProjectionSIMPId, InterProjectionSIMPType} = ?IMPFM,
{ok, _} = lasp:declare(InterProjectionSIMPId, InterProjectionSIMPType),

ok =
    lasp:map(
        ?IMPF,
        fun(
            {
                IMP_SK, IMP_ITEM_SK, _, IMP_COMPETITOR_PRICE,
                IMP_START_DATE_SK, IMP_END_DATE_SK}) ->
            {CompetitorPrice, _} = string:to_float(IMP_COMPETITOR_PRICE),
            {StartDate, _} = string:to_integer(IMP_START_DATE_SK),
            {EndDate, _} = string:to_integer(IMP_END_DATE_SK),
            {IMP_SK, IMP_ITEM_SK, CompetitorPrice, StartDate, EndDate}
        end,
        ?IMPFM),

{InterJoinPSIPSIMPId, InterJoinPSIPSIMPType} = ?IFM_IMPFM_X,
{ok, _} = lasp:declare(InterJoinPSIPSIMPId, InterJoinPSIPSIMPType),
```

```
ok = lasp:product(?IFM, ?IMPFM, ?IFM_IMPFM_X),

{InterProjectionJPSIPSIMPId, InterProjectionJPSIPSIMPType} = ?IFM_IMPFM_XM,
{ok, _} =
    lasp:declare(InterProjectionJPSIPSIMPId, InterProjectionJPSIPSIMPType),

ok =
    lasp:map(
        ?IFM_IMPFM_X,
        fun(
            {
                {I_ITEM_SK, I_CURRENT_PRICE},
                {
                    IMP_SK, _IMP_ITEM_SK, IMP_COMPETITOR_PRICE,
                    IMP_START_DATE_SK, IMP_END_DATE_SK}}) ->
            PriceChange =
                (IMP_COMPETITOR_PRICE - I_CURRENT_PRICE) / I_CURRENT_PRICE,
            NODaysCompPrice = IMP_END_DATE_SK - IMP_START_DATE_SK,
            {
                I_ITEM_SK, IMP_SK, PriceChange,
                IMP_START_DATE_SK, NODaysCompPrice}
        end,
        ?IFM_IMPFM_XM),

{InterSelectionSSId, InterSelectionSSType} = ?SSF,
{ok, _} = lasp:declare(InterSelectionSSId, InterSelectionSSType),

ok =
    lasp:filter(
        ?INPUT_STORE_SALES,
        fun(
            {
                _, _, SS_ITEM_SK, _, _, _, _, _, _, _, _, _, _, _,
                _, _, _, _, _, _, _, _, _}) ->
            SS_ITEM_SK == ?ITEM_SK
        end,
        ?SSF),
```

```
{InterProjectionSSSId, InterProjectionSSSType} = ?SSFM,
{ok, _} = lasp:declare(InterProjectionSSSId, InterProjectionSSSType),

ok =
    lasp:map(
        ?SSF,
        fun(
            {
                SS_SOLD_DATE_SK, _, SS_ITEM_SK, _, _, _, _, _, _, _,
                SS_QUANTITY, _, _, _, _, _, _, _, _, _, _, _, _}) ->
            {SoldDate, _} = string:to_integer(SS_SOLD_DATE_SK),
            {Quantity, _} = string:to_integer(SS_QUANTITY),
            {SoldDate, SS_ITEM_SK, Quantity}
        end,
        ?SSFM),

{
    InterJoinPJPSIPSIMPPSSSId,
    InterJoinPJPSIPSIMPPSSSType} = ?IFM_IMPFM_XM_SSFM_X,
{ok, _} =
    lasp:declare(InterJoinPJPSIPSIMPPSSSId, InterJoinPJPSIPSIMPPSSSType),

ok = lasp:product(?IFM_IMPFM_XM, ?SSFM, ?IFM_IMPFM_XM_SSFM_X),

{
    InterProjectionJPJPSIPSIMPPSSSId,
    InterProjectionJPJPSIPSIMPPSSSType} = ?IFM_IMPFM_XM_SSFM_XM,
{ok, _} =
    lasp:declare(
        InterProjectionJPJPSIPSIMPPSSSId,
        InterProjectionJPJPSIPSIMPPSSSType),

ok =
    lasp:map(
        ?IFM_IMPFM_XM_SSFM_X,
        fun(
            {
```

```
                {
                    _I_ITEM_SK, IMP_SK, PriceChange,
                    IMP_START_DATE_SK, NODaysCompPrice},
                {SS_SOLD_DATE_SK, SS_ITEM_SK, SS_QUANTITY}}) ->
            CurrentSSQuant =
                case (SS_SOLD_DATE_SK >= IMP_START_DATE_SK) andalso
                    (SS_SOLD_DATE_SK < (IMP_START_DATE_SK + NODaysCompPrice)) of
                    true ->
                        SS_QUANTITY;
                    false ->
                        0
                end,
            PrevSSQuant =
                case
                    (SS_SOLD_DATE_SK >=
                        (IMP_START_DATE_SK - NODaysCompPrice)) andalso
                    (SS_SOLD_DATE_SK < IMP_START_DATE_SK) of
                    true ->
                        SS_QUANTITY;
                    false ->
                        0
                end,
            {{SS_ITEM_SK, IMP_SK, PriceChange}, {CurrentSSQuant, PrevSSQuant}}
        end,
        ?IFM_IMPFM_XM_SSFM_XM),

{
    GroupBySumPJPJPSIPSIMPPSSSId,
    GroupBySumPJPJPSIPSIMPPSSSType} = ?IFM_IMPFM_XM_SSFM_XMG,
{ok, _} =
    lasp:declare(
        GroupBySumPJPJPSIPSIMPPSSSId, GroupBySumPJPJPSIPSIMPPSSSType),

ok =
    lasp:group_by_sum(
        ?IFM_IMPFM_XM_SSFM_XM,
        fun(undefined, {CurrentSSQuant, PrevSSQuant}) ->
            {CurrentSSQuant, PrevSSQuant};
```

```
        (
            {AccCurrentSSQuant, AccPrevSSQuant},
            {CurrentSSQuant, PrevSSQuant}) ->
            {
                AccCurrentSSQuant + CurrentSSQuant,
                AccPrevSSQuant + PrevSSQuant}
        end,
        ?IFM_IMPFM_XM_SSFM_XMG),

{InterSelectionWSId, InterSelectionWSType} = ?WSF,
{ok, _} = lasp:declare(InterSelectionWSId, InterSelectionWSType),

ok =
    lasp:filter(
        ?INPUT_WEB_SALES,
        fun(
            {
                _, _, _, WS_ITEM_SK, _, _, _, _, _, _, _, _, _, _, _, _,
                _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _}) ->
            WS_ITEM_SK == ?ITEM_SK
        end,
        ?WSF),

{InterProjectionSWSId, InterProjectionSWSType} = ?WSFM,
{ok, _} = lasp:declare(InterProjectionSWSId, InterProjectionSWSType),

ok =
    lasp:map(
        ?WSF,
        fun(
            {
                WS_SOLD_DATE_SK, _, _, WS_ITEM_SK, _, _, _, _, _, _, _, _,
                _, _, _, _, _, _, WS_QUANTITY, _, _, _, _, _, _, _, _, _,
                _, _, _, _, _, _}) ->
            {SoldDate, _} = string:to_integer(WS_SOLD_DATE_SK),
            {Quantity, _} = string:to_integer(WS_QUANTITY),
            {SoldDate, WS_ITEM_SK, Quantity}
        end,
```

```
        ?WSFM),

{
    InterJoinPJPSIPSIMPPSWSId,
    InterJoinPJPSIPSIMPPSWSType} = ?IFM_IMPFM_XM_WSFM_X,
{ok, _} =
    lasp:declare(InterJoinPJPSIPSIMPPSWSId, InterJoinPJPSIPSIMPPSWSType),

ok = lasp:product(?IFM_IMPFM_XM, ?WSFM, ?IFM_IMPFM_XM_WSFM_X),

{
    InterProjectionJPJPSIPSIMPPSWSId,
    InterProjectionJPJPSIPSIMPPSWSType} = ?IFM_IMPFM_XM_WSFM_XM,
{ok, _} =
    lasp:declare(
        InterProjectionJPJPSIPSIMPPSWSId, InterProjectionJPJPSIPSIMPPSWSType),

ok =
    lasp:map(
        ?IFM_IMPFM_XM_WSFM_X,
        fun(
            {
                {
                    _I_ITEM_SK, IMP_SK, PriceChange,
                    IMP_START_DATE_SK, NODaysCompPrice},
                {WS_SOLD_DATE_SK, WS_ITEM_SK, WS_QUANTITY}}) ->
            CurrentWSQuant =
                case
                    (WS_SOLD_DATE_SK >= IMP_START_DATE_SK) andalso
                    (WS_SOLD_DATE_SK <
                        (IMP_START_DATE_SK + NODaysCompPrice)) of
                    true ->
                        WS_QUANTITY;
                    false ->
                        0
                end,
            PrevWSQuant =
                case
```

```
                    (WS_SOLD_DATE_SK >=
                        (IMP_START_DATE_SK - NODaysCompPrice)) andalso
                    (WS_SOLD_DATE_SK < IMP_START_DATE_SK) of
                    true ->
                        WS_QUANTITY;
                    false ->
                        0
                end,
            {{WS_ITEM_SK, IMP_SK, PriceChange}, {CurrentWSQuant, PrevWSQuant}}
        end,
        ?IFM_IMPFM_XM_WSFM_XM),

{
    GroupBySumPJPJPSIPSIMPPSWSId,
    GroupBySumPJPJPSIPSIMPPSWSType} = ?IFM_IMPFM_XM_WSFM_XMG,
{ok, _} =
    lasp:declare(
        GroupBySumPJPJPSIPSIMPPSWSId, GroupBySumPJPJPSIPSIMPPSWSType),

ok =
    lasp:group_by_sum(
        ?IFM_IMPFM_XM_WSFM_XM,
        fun(undefined, {CurrentWSQuant, PrevWSQuant}) ->
            {CurrentWSQuant, PrevWSQuant};
        (
            {AccCurrentWSQuant, AccPrevWSQuant},
            {CurrentWSQuant, PrevWSQuant}) ->
            {
                AccCurrentWSQuant + CurrentWSQuant,
                AccPrevWSQuant + PrevWSQuant}
        end,
        ?IFM_IMPFM_XM_WSFM_XMG),

{
    InterJoinGPJPJPSIPSIMPPSSSGPJPJPSIPSIMPPSWSId,
    InterJoinGPJPJPSIPSIMPPSSSGPJPJPSIPSIMPPSWSType} =
    ?IFM_IMPFM_XM_SSFM_XMG_IFM_IMPFM_XM_WSFM_XMG_X,
{ok, _} =
```

```
    lasp:declare(
        InterJoinGPJPJPSIPSIMPPSSSGPJPJPSIPSIMPPSWSId,
        InterJoinGPJPJPSIPSIMPPSSSGPJPJPSIPSIMPPSWSType),

ok =
    lasp:product(
        ?IFM_IMPFM_XM_SSFM_XMG,
        ?IFM_IMPFM_XM_WSFM_XMG,
        ?IFM_IMPFM_XM_SSFM_XMG_IFM_IMPFM_XM_WSFM_XMG_X),

{
    InterSelectionJGPJPJPSIPSIMPPSSSGPJPJPSIPSIMPPSWSId,
    InterSelectionJGPJPJPSIPSIMPPSSSGPJPJPSIPSIMPPSWSType} =
    ?IFM_IMPFM_XM_SSFM_XMG_IFM_IMPFM_XM_WSFM_XMG_XF,
{ok, _} =
    lasp:declare(
        InterSelectionJGPJPJPSIPSIMPPSSSGPJPJPSIPSIMPPSWSId,
        InterSelectionJGPJPJPSIPSIMPPSSSGPJPJPSIPSIMPPSWSType),

ok =
    lasp:filter(
        ?IFM_IMPFM_XM_SSFM_XMG_IFM_IMPFM_XM_WSFM_XMG_X,
        fun({
            {
                {_SS_ITEM_SK, SS_IMP_SK, _SSPriceChange},
                {_CurrentSSQuant, _PrevSSQuant}},
            {
                {_WS_ITEM_SK, WS_IMP_SK, _WSPriceChange},
                {_CurrentWSQuant, _PrevWSQuant}}}) ->
            SS_IMP_SK == WS_IMP_SK
        end,
        ?IFM_IMPFM_XM_SSFM_XMG_IFM_IMPFM_XM_WSFM_XMG_XF),

{
    InterProjectionJGPJPJPSIPSIMPPSSSGPJPJPSIPSIMPPSWSId,
    InterProjectionJGPJPJPSIPSIMPPSSSGPJPJPSIPSIMPPSWSType} =
    ?IFM_IMPFM_XM_SSFM_XMG_IFM_IMPFM_XM_WSFM_XMG_XFM,
{ok, _} =
```

```
    lasp:declare(
        InterProjectionJGPJPJPSIPSIMPPSSSGPJPJPSIPSIMPPSWSId,
        InterProjectionJGPJPJPSIPSIMPPSSSGPJPJPSIPSIMPPSWSType),

ok =
    lasp:map(
        ?IFM_IMPFM_XM_SSFM_XMG_IFM_IMPFM_XM_WSFM_XMG_XF,
        fun({
            {
                {_SS_ITEM_SK, _IMP_SK, _SSPriceChange},
                {CurrentSSQuant, PrevSSQuant}},
            {
                {WS_ITEM_SK, _IMP_SK, WSPriceChange},
                {CurrentWSQuant, PrevWSQuant}}}) ->
            CrossPriceElasticity =
                (CurrentSSQuant + CurrentWSQuant -
                        PrevSSQuant - PrevWSQuant) /
                    ((PrevSSQuant + PrevWSQuant) * WSPriceChange),
            {WS_ITEM_SK, {CrossPriceElasticity, 1}}
        end,
        ?IFM_IMPFM_XM_SSFM_XMG_IFM_IMPFM_XM_WSFM_XMG_XFM),

{
    GroupBySumPJGPJPJPSIPSIMPPSSSGPJPJPSIPSIMPPSWSId,
    GroupBySumPJGPJPJPSIPSIMPPSSSGPJPJPSIPSIMPPSWSType} =
    ?IFM_IMPFM_XM_SSFM_XMG_IFM_IMPFM_XM_WSFM_XMG_XFMG,
{ok, _} =
    lasp:declare(
        GroupBySumPJGPJPJPSIPSIMPPSSSGPJPJPSIPSIMPPSWSId,
        GroupBySumPJGPJPJPSIPSIMPPSSSGPJPJPSIPSIMPPSWSType),

ok =
    lasp:group_by_sum(
        ?IFM_IMPFM_XM_SSFM_XMG_IFM_IMPFM_XM_WSFM_XMG_XFM,
        fun(undefined, {CrossPriceElasticity, Cnt}) ->
            {CrossPriceElasticity, Cnt};
        ({AccCrossPriceElasticity, AccCnt}, {CrossPriceElasticity, Cnt}) ->
            {AccCrossPriceElasticity + CrossPriceElasticity, AccCnt + Cnt}
```

```
        end,
        ?IFM_IMPFM_XM_SSFM_XMG_IFM_IMPFM_XM_WSFM_XMG_XFMG),

{
    InterProjectionGPJGPJPJPSIPSIMPPSSSGPJPJPSIPSIMPPSWSId,
    InterProjectionGPJGPJPJPSIPSIMPPSSSGPJPJPSIPSIMPPSWSType} =
    ?IFM_IMPFM_XM_SSFM_XMG_IFM_IMPFM_XM_WSFM_XMG_XFMGM,
{ok, _} =
    lasp:declare(
        InterProjectionGPJGPJPJPSIPSIMPPSSSGPJPJPSIPSIMPPSWSId,
        InterProjectionGPJGPJPJPSIPSIMPPSSSGPJPJPSIPSIMPPSWSType),

ok =
    lasp:map(
        ?IFM_IMPFM_XM_SSFM_XMG_IFM_IMPFM_XM_WSFM_XMG_XFMG,
        fun({WS_ITEM_SK, {CrossPriceElasticity, Cnt}}) ->
            {WS_ITEM_SK, CrossPriceElasticity / Cnt}
        end,
        ?IFM_IMPFM_XM_SSFM_XMG_IFM_IMPFM_XM_WSFM_XMG_XFMGM),

{ok, {NewSubset, NewCDS, ReadResult}} =
    lasp:query_ext_consistent(
        ?IFM_IMPFM_XM_SSFM_XMG_IFM_IMPFM_XM_WSFM_XMG_XFMGM, Subset, CDS),
```

A.3.3 Query 29

```
-define(INPUT_ITEM, {<<"i">>, {?EXT_AWORBAG_INPUT_TYPE, []}}).
% filter(<<"i">>)
-define(IF, {<<"iF">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% map(<<"iF">>)
-define(IFM, {<<"iFM">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).

-define(INPUT_WEB_SALES, {<<"ws">>, {?EXT_AWORBAG_INPUT_TYPE, []}}).
% map(<<"ws">>)
-define(WSM, {<<"wsM">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).

% product(<<"wsM">>, <<"iFM">>)
-define(
    WSM_IFM_X, {<<"wsM_iFM_X">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
```

```
% map(<<"wsM_iFM_X">>)
-define(
    WSM_IFM_XM,
    {<<"wsM_iFM_XM">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% group_by_sum(<<"wsM_iFM_XM">>)
-define(
    WSM_IFM_XMG,
    {<<"wsM_iFM_XMG">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% map(<<"wsM_iFM_XMG">>)
-define(
    WSM_IFM_XMGM,
    {<<"wsM_iFM_XMGM">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, []}}).
% group_by_sum(<<"wsM_iFM_XMGM">>)
-define(
    WSM_IFM_XMGMG,
    {<<"wsM_iFM_XMGMG">>, {?EXT_AWORBAG_INTERMEDIATE_TYPE, [list]}}).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

{InputBagItemId, InputBagItemType} = ?INPUT_ITEM,
{ok, _} = lasp:declare(InputBagItemId, InputBagItemType),
{InputBagWebSalesId, InputBagWebSalesType} = ?INPUT_WEB_SALES,
{ok, _} = lasp:declare(InputBagWebSalesId, InputBagWebSalesType),

{InterSelectionIId, InterSelectionIType} = ?IF,
{ok, _} = lasp:declare(InterSelectionIId, InterSelectionIType),

ok =
    lasp:filter(
        ?INPUT_ITEM,
        fun(
            {
                _, _, _, _, _, _, _, _, _, _, _, I_CATEGORY_ID,
                _, _, _, _, _, _, _, _, _, _}) ->
            I_CATEGORY_ID /= undefined
        end,
        ?IF),
```

```
{InterProjectionSIId, InterProjectionSIType} = ?IFM,
{ok, _} = lasp:declare(InterProjectionSIId, InterProjectionSIType),

ok =
    lasp:map(
        ?IF,
        fun(
            {
                I_ITEM_SK, _, _, _, _, _, _, _, _, _, _, I_CATEGORY_ID,
                _, _, _, _, _, _, _, _, _, _}) ->
            {I_ITEM_SK, I_CATEGORY_ID}
        end,
        ?IFM),

{InterProjectionWSId, InterProjectionWSType} = ?WSM,
{ok, _} = lasp:declare(InterProjectionWSId, InterProjectionWSType),

ok =
    lasp:map(
        ?INPUT_WEB_SALES,
        fun(
            {
                _, _, _, WS_ITEM_SK, _, _, _, _, _, _, _, _, _, _, _, _, _,
                WS_ORDER_NUMBER, _, _, _, _, _, _, _, _, _, _, _, _, _, _,
                _, _}) ->
            {WS_ORDER_NUMBER, WS_ITEM_SK}
        end,
        ?WSM),

{InterJoinPWSPSIId, InterJoinPWSPSIType} = ?WSM_IFM_X,
{ok, _} = lasp:declare(InterJoinPWSPSIId, InterJoinPWSPSIType),

ok =
    lasp:cond_product(
        ?WSM,
        ?IFM,
        fun({_WS_ORDER_NUMBER, WS_ITEM_SK}) ->
            WS_ITEM_SK
```

```
        end,
        fun({I_ITEM_SK, _I_CATEGORY_ID}) ->
            I_ITEM_SK
        end,
        ?WSM_IFM_X),

{InterProjectionJPWSPSIId, InterProjectionJPWSPSIType} = ?WSM_IFM_XM,
{ok, _} = lasp:declare(InterProjectionJPWSPSIId, InterProjectionJPWSPSIType),

ok =
    lasp:map(
        ?WSM_IFM_X,
        fun({{WS_ORDER_NUMBER, _WS_ITEM_SK}, {_I_ITEM_SK, I_CATEGORY_ID}}) ->
            {WS_ORDER_NUMBER, I_CATEGORY_ID}
        end,
        ?WSM_IFM_XM),

{GroupBySumPJPWSPSIId, GroupBySumPJPWSPSIType} = ?WSM_IFM_XMG,
{ok, _} = lasp:declare(GroupBySumPJPWSPSIId, GroupBySumPJPWSPSIType),

ok =
    lasp:group_by_sum(
        ?WSM_IFM_XM,
        fun(undefined, I_CATEGORY_ID) ->
            ordsets:add_element(I_CATEGORY_ID, ordsets:new());
        (SetI_CATEGORY_ID, I_CATEGORY_ID) ->
            ordsets:add_element(I_CATEGORY_ID, SetI_CATEGORY_ID)
        end,
        ?WSM_IFM_XMG),

{InterMakePairGPJPWSPSIId, InterMakePairGPJPWSPSIType} = ?WSM_IFM_XMGM,
{ok, _} = lasp:declare(InterMakePairGPJPWSPSIId, InterMakePairGPJPWSPSIType),

ok =
    lasp:map(
        ?WSM_IFM_XMG,
        fun({_WS_ORDER_NUMBER, SetI_CATEGORY_ID}) ->
            MakePairCategories =
```

```
                make_pair_sale_stores_items(SetI_CATEGORY_ID),
            ordsets:fold(
                fun({I_CATEGORY_ID1, I_CATEGORY_ID2}, AccInResult) ->
                    AccInResult ++ [{{I_CATEGORY_ID1, I_CATEGORY_ID2}, 1}]
                end,
                [],
                MakePairCategories)
        end,
        ?WSM_IFM_XMGM),

{GroupBySumPairGPJPWSPSIId, GroupBySumPairGPJPWSPSIType} = ?WSM_IFM_XMGMG,
{ok, _} =
    lasp:declare(GroupBySumPairGPJPWSPSIId, GroupBySumPairGPJPWSPSIType),

ok =
    lasp:group_by_sum(
        ?WSM_IFM_XMGM,
        fun(undefined, PairCnt) ->
            PairCnt;
        (PairCntSum, PairCnt) ->
            PairCntSum + PairCnt
        end,
        ?WSM_IFM_XMGMG),

{ok, {NewSubset, NewCDS, ReadResult}} =
    lasp:query_ext_consistent(?WSM_IFM_XMGMG, Subset, CDS),
```