UNIVERSITY OF CALIFORNIA
RIVERSIDE

Scalable Data-Parallel Processing of Semi-Structured Data

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Lin Jiang

Spetember 2021

Dissertation Committee:

Dr. Zhijia Zhao, Chairperson
Dr. Ahmed Eldawy
Dr. Mohsen Lesani
Dr. Amr Magdy

The Dissertation of Lin Jiang is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

First, I would like to thank my PhD advisor Prof. Zhijia Zhao for all the guidance and support over the past five years. His timely feedback and valuable suggestions are very helpful for my achievements and personal growth. Also thanks him for providing me the opportunities to work on these existing projects. I also want to thank all my committee members Prof. Ahmed Eldway, Prof. Ahmed Magdy, and Prof. Mohsen Lesani. Their critical suggestions and advice are very helpful for me to make further improvements on my research projects.

I am also thankful for all anonymous reviewers in Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2017 (PPOPP'17), Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating System, 2019 (ASPLOS'19), and Proceedings of the Very Large Data Bases (VLDB) Endowment, Volume 14, 2020-2021 for their timely and constructive feedback in Chapter 2, Chapter 3 and Chapter 4.

I am fortunate to have worked with talented and awesome lab mates and colleagues. Thanks to Junqiao Qiu, Umar Farooq, Amir Nohedi Sabet, Xiaofan Sun, Chengshuo Xu, Xizhe Yin and Soroush Saffari. Hope we will have some opportunities to corporate with each other in the near future.

Finally, I would like to express my gratitude to my family. Thanks to my parents, my parents-in-law, my wife, my daughter and all my relatives for their endless love and supports. I love you all!

This thesis is dedicated to my parents Wenping Jiang and Yanling Lin, my lovely wife Yanting Cao and my lovely daughter Annie Jiang.

For their endless love and supports.

ABSTRACT OF THE DISSERTATION

Scalable Data-Parallel Processing of Semi-Structured Data

by

Lin Jiang

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, Spetember 2021
Dr. Zhijia Zhao, Chairperson

Semi-structured data, like JSON, XML, and their derivatives, are essential in modern computing infrastructures, from cloud computing and microservice to NoSQL data stores and Internet of Things (IoT). However, existing software often fails to process such types of data in a scalable way due to their nested structures and the lack of effective data-level parallelism. The goal of this thesis is mainly to address two fundamental scalability issues in semi-structured data processing. First, how can semi-structured data analytics effectively leverage the abundant hardware parallelism that are offered by modern computer architectures (e.g., multi-cores and SIMD operations)? Second, how can semi-structured data analytics improve the data access efficiency (i.e., locality) and reduce the memory consumption when handling large semi-structured datasets? To answer these questions, this thesis proposes a series of parallelization techniques and streaming computation models dedicated to semi-structured data.

More specifically, this thesis first proposes a grammar-aware parallelization (GAP) scheme for XPath query evaluation, which leverages the data grammar (e.g., DTD file for

XML) to prune unnecessary paths during the enumerative execution. Then, it designs a streaming model for querying JSON data, which jointly compiles JSON grammar and JSONPath queries into a dual-stack pushdown automaton, and adopts a customized GAP scheme for its parallelization. Next, to effectively utilize both fine-grained (bitwise and SIMD) parallelism and coarse-grained (multi-core) parallelism, this thesis proposes a new design of bitwise structural index construction that is able to build leveled bitmaps for a single large JSON record in parallel, thanks to a set of parallelization techniques specialized to JSON structures. Finally, this thesis combines the ideas of bitwise index construction and the stream processing model, which brings in a novel on-demand parsing technique that can intelligently skip parsing irrelevant substructures of a JSON record based on the given JSON queries. All the above techniques have been systematically evaluated with real-world datasets and standard JSON/XML queries, and have demonstrated significant performance benefits over the existing solutions, in terms of both query evaluation time and memory consumption.

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
1.1 Parallelization of XPath Streaming Evaluation . . . 3
1.2 Parallel Streaming Model for JSONPath Evaluation . . . 3
1.3 Scalable Index Constructor for JSON Analytics . . . 4
1.4 Streaming JSON Data with Bit-Parallel Skipping . . . 5

2 Grammar-aware Parallelization of Scalable XPath Querying **6**
2.1 Introduction . . . 7
2.2 Background . . . 11
2.2.1 Querying Semi-Structured Data . . . 12
2.2.2 Pushdown Transducers . . . 13
2.2.3 Parallel Pushdown Transducers . . . 16
2.3 Overview . . . 18
2.4 Non-Speculative GAP . . . 21
2.4.1 Static Syntax Tree Generation . . . 22
2.4.2 Symbolic Execution of Pushdown Transducer . . . 23
2.4.3 GAP Pushdown Transducers . . . 26
2.5 Speculative GAP . . . 31
2.5.1 Partial Grammar Extraction . . . 31
2.5.2 Speculative GAP Pushdown Transducers . . . 32
2.6 Evaluation . . . 34
2.7 Related Work . . . 41
2.8 Summary . . . 43

3 Scalable JSONPath Querying **44**
3.1 Introduction . . . 45
3.2 Background . . . 50
3.3 Streaming Compilation . . . 53

3.3.1 Idea of Joint Compilation . . . 54
3.3.2 Path Expression Compilation . . . 54
3.3.3 JSON Syntax Compilation . . . 57
3.3.4 Automata Hooking . . . 58
3.4 Parallelizing Compilation . . . 61
3.4.1 Breaking Dependences . . . 61
3.4.2 Feasible Paths Inference . . . 64
3.4.3 Towards a Bounded Number of Paths . . . 68
3.4.4 Results Merging . . . 69
3.5 Runtime Optimization . . . 70
3.6 Evaluation . . . 72
3.6.1 Implementation . . . 72
3.6.2 Methodology . . . 73
3.6.3 Comparison with Existing Methods . . . 75
3.6.4 Performance of Parallel Execution . . . 77
3.6.5 Scalability . . . 81
3.7 Related Work . . . 82
3.8 Summary . . . 84

4 Scalable Bitwise Index Construction for JSON Analytics 85
4.1 Introduction . . . 86
4.2 Background . . . 89
4.2.1 JSON and Its Querying . . . 89
4.2.2 Structural Index Construction . . . 92
4.2.3 Scalability Issues for Bulky Records . . . 96
4.3 Data-Parallel Construction . . . 97
4.3.1 Dependences . . . 97
4.3.2 Dynamic Partitioning . . . 98
4.3.3 Contradiction-based Context Inference . . . 99
4.3.4 Speculative Parallelization . . . 101
4.3.5 Parallel Generation of Leveled Bitmaps . . . 103
4.3.6 Putting it All Together . . . 110
4.4 Locality Optimization . . . 111
4.5 Querying using Index . . . 113
4.6 Evaluation . . . 115
4.6.1 Methodology . . . 115
4.6.2 Index Construction Peformance . . . 118
4.6.3 Benefits and Costs Breakdown . . . 121
4.6.4 End-to-End Performance . . . 125
4.7 Related Work . . . 127
4.8 Summary . . . 129

5 Streaming Semi-Structured Data with Bit-Parallel Skipping 130
5.1 Introduction . . . 131
5.2 Background . . . 136
5.3 Streaming with Skipping . . . 140
5.3.1 Basic Streaming Model − A Decoupled Design . . . 141
5.3.2 Opportunities for Skipping . . . 146
5.3.3 Integration of Skippers . . . 149
5.4 Bit-Parallel Skippers . . . 152
5.4.1 Structural Intervals . . . 152
5.4.2 Skipping Algorithms . . . 156
5.5 Evaluation . . . 161
5.5.1 Methodology . . . 162
5.5.2 Overall Performance . . . 164
5.5.3 Benefits Breakdown . . . 167
5.5.4 Scalability . . . 169
5.6 Related Work . . . 169
5.7 Summary . . . 172

6 Conclusions 173

Bibliography 175

List of Figures

2.1 An illustration example (Facebook feed data [1]) 8
2.2 Scalability Comparison 10
2.3 A Diagram of The Pushdown Transducer 14
2.4 A running example with a recursive grammar. '*' after state 5 means a match. 15
2.5 Overview of GAP. Labeled numbers show the major steps when predefined input grammar is available (e.g., DTD or Schema). Blue lines show the speculative execution flow in the absence of a pre-defined grammar. 18
2.6 Static Syntax Tree 21
2.7 Algorithm 2 execution on the running example. 26
2.8 Speedup of Single-Query and Multi-Query Processing on A 20-core Machine 38
2.9 Scalability over Number of Cores 40
2.10 Scalability over Number of Queries 41

3.1 Challenge in Parallel JSON Data Processing[2]. 47
3.2 BNF Grammar of JSON. 50
3.3 Illustration of Joint Compilation. 53
3.4 Automaton for `$.rountes[*].steps[2].loc`. 56
3.5 Transition Rules of Parsing Automaton. 58
3.6 Transition Rules of Streaming Automaton. 59
3.7 An Example of Streaming Automaton Execution (input and query automaton are from Figures 3.1 and 3.4). 60
3.8 Two-Phase Parallelization. 61
3.9 Query Stack Feasibility Inference: XML vs. JSON 65
3.10 Performance of Streaming-based Methods*. 77
3.11 Comparison with Parsing-based Methods (Time). 78
3.12 Comparison to Parsing-based (Memory). 79
3.13 Speedups of JPStream-generated Automata on Xeon and Xeon Phi Servers. 79
3.14 Scalability over Number of Cores on Xeon-Phi 82

4.1 JSON Grammar and Example. 90
4.2 Example Structural Index Construction. 92
4.3 Querying via Structural Indices. 95

4.4 Dependences in Index Construction. 98
4.5 Examples of Context Inference. 100
4.6 Bitwise Rectification for Misspeculation. 102
4.7 Generation of Leveled Bitmaps: Existing Design (left) vs. New Design (right). 104
4.8 Two-Phase Index Construction. 109
4.9 Level Matching. 110
4.10 Index Merging. 110
4.11 Workflow of Data-Parallel Construction. 111
4.12 Index Construction Granularities. 112
4.13 Comparison of Index Construction Time of Different Methods on Bulky JSON Records. 118
4.14 Comparison of Index Construction Time of Different Methods on Small JSON Records. 120
4.15 Costs of Speculation and Recovering. 122
4.16 Scalability over Number of Threads. 123
4.17 Scalability over Record Size. 124
4.18 Comparison of Memory Consumption 125

5.1 Geo-referenced Tweet in JSON [128] 132
5.2 JSON Grammar 137
5.3 Preprocessing-based Query Evaluation 138
5.4 Streaming Query Evaluation 140
5.5 Transition Rules of Query Automaton 143
5.6 Examples of Basic Skipping Cases 143
5.7 Examples of Structural Intervals 153
5.8 Example of Interval Bitmap 154
5.9 Examples of Pairing Property 157
5.10 Performance Comparison on Large Records 164
5.11 Performance Comparison on Small Records 165
5.12 Memory Footprint Comparison 167
5.13 Scalability Comparison (BB1) 169

List of Tables

2.1 Feasible Paths Table 20
2.2 Method Versions in Evaluation 35
2.3 XML datasets (d means depth) 36
2.4 XPath queries. #sub shows the number of sub-queries in each query structure. 37
2.5 Average Number of Starting Execution Paths 39
2.6 Speculation Accuracy and Reprocessing Cost 39

3.1 Syntax Stack Feasibility Inference (* means any symbol). 67
3.2 Example Data Constraints Hash Table 71
3.3 Methods in Evaluation. 73
3.4 JSONPath queries. #sub shows the number of sub-queries in each query structure. 74
3.5 Dataset Statistics. 74
3.6 Number of Execution Paths (Xeon Phi) 80
3.7 Costs of Results Merging 80
3.8 Path Coverage Accuracy and Reprocessing Cost 81
3.9 Scalability over Data Size on Xeon 81

4.1 Methods in Evaluation 115
4.2 Dataset Statistics 116
4.3 JSONPath Queries 117
4.4 Time Breakdown by Steps 121
4.5 End-to-End Time (s) Comparison. 125

5.1 Five Groups of Skippers 149
5.2 Methods in Evaluation 162
5.3 Dataset Statistics 163
5.4 JSONPath Queries 164
5.5 Skipping Ratios by Skipper Groups 168

Chapter 1

Introduction

Semi-structured data, like XML, JSON, and their derivatives, are essential in the modern computing infrastructure, ranging from cloud computing [47, 91] and microservice architectures [130, 58], to Internet of Things (IoT) [132] and NoSQL data stores [87, 72]. In recent years, the volume of semi-structured data also grows rapidly [24, 92, 8]. Thus, the efficiency of semi-structured data processing becomes more critical to the modern software applications whose performance increasingly relies on them.

Conventionally, there are two basic ways for processing semi-structured data: i) preprocessing-based scheme and ii) streaming scheme.

Preprocessing-based Scheme. This traditional way of processing semi-structured data requires a full parsing (or other kinds of preprocessing) over the entire data record before getting the substructures of interests. However, for large input records, the parsing not only incurs a long delay at the beginning, but also occupies a large chunk of memory (for storing the parse tree) during the query evaluation.

Streaming Scheme. To avoid the above issues, a more lightweight approach, called streaming, would be preferred in many situations. It performs online parsing during query evaluation, without generating any in-memory parsing tree. In this way, it can process the entire data stream in a single pass with a bounded memory footprint. However, streaming model is not suitable for cases where the queries are unknown beforehand, since it has to re-traverse the data stream when a query is given.

Despite recent advances in processing semi-structured data, there are still two major issues that may limit its efficiency:

- *Lack of Data Parallelism.* A semi-structured data record follows a nested structure (e.g., a JSON object can be embedded in another object). A naive partitioning can easily break its internal structures, making it very challenging to expose data-level parallelism. On the other hand, modern CPU supports both fine-grained parallelism (bitwise and SIMD instructions) and coarse-grained parallelism (multi-core). Thus, a key research question lies in the exploitation of data parallelism for improving the performance of semi-structured data processing.

- *Lack of Efficient Streaming Models.* Though the streaming model for the conventional tag-based semi-structured data (e.g., XML) has been proposed, the streaming model for contemporary JSON-style semi-structured data remains an open question. Note that the syntax of JSON is fundamentally different from tag-based semi-structured data. Moreover, the traditional way of streaming requires to scan the entire stream character by character in order to recognize all tokens and syntactical structures. This comprehensive way of scanning seriously limits the best efficiency of data streaming.

To address the above challenges, this thesis introduces a series of parallelization techniques and new designs of streaming models to process XML/JSON data in an efficient yet scalable way. At the high level, the thesis is organized under four major research topics, as listed in the following sections.

1.1 Parallelization of XPath Streaming Evaluation

The first part of this thesis focuses on the parallelization of one existing streaming model for XPath query evaluation, which is a pushdown automaton automatically generated based on queries [68]. A recent work parallelized this model based on the idea of paths enumeration [106]. However, maintaining all enumerated execution paths remains a significant parallelization overhead, especially for complex queries or multi-query scenarios. To address this challenge, this thesis proposes a grammar-aware parallelization schema, namely GAP, to narrow down the feasible execution paths by learning data constraints from the grammar (e.g. DTD, XSD for XML). When certain grammar is not available, GAP may still prune unnecessary paths speculatively by inferring an incomplete grammar from training inputs. As elements in semi-structured data tend to follow certain patterns, speculative GAP often yields high speculation accuracy and low reprocessing costs.

1.2 Parallel Streaming Model for JSONPath Evaluation

To adopt a similar automata-based streaming scheme for JSON query evaluation, this thesis introduces a compilation system, namely JPStream, that compiles standard JSONPath queries into parallel executables with bounded memory footprints. The key idea

is formalized as a joint-compilation technique which compiles JSON grammar and JSONPath queries together into a single automaton (a dual-stack pushdown automaton). Based on this streaming model, this thesis further presents a set of customized parallelization techniques which can execute multiple automata in parallel on different partitions of a (large) JSON record, without suffering from the path explosion problem caused by path enumeration. Evaluation shows that JPStream can reduce the memory consumption significantly, meanwhile achieving near linear speedup.

1.3 Scalable Index Constructor for JSON Analytics

To leverage bitwise and SIMD parallelism that is available on modern computer architectures, a recent work, Mison [86], proposes a novel algorithm that generates bitwise indices for the structural characters (like colon `:` and comma `,`) in a JSON record, called *structural indices*. With such indices, a parser can directly jump into relevant positions of the JSON record to find query matches. This preprocessing-based approach has already demonstrated its high efficiency thanks to the bit-parallel operations. However, it does not scale well as records become larger and more complex due to its lack of coarse-grained (multi-core) data parallelism and memory-intensive design of some index construction steps. To address these limitations, this thesis introduces Pison − a more memory-efficient structural index constructor with supports of intra-record parallelism. Evaluation results confirm that Pison outperforms the state-of-the-art in terms of both index construction efficiency and end-to-end query evaluation performance.

1.4 Streaming JSON Data with Bit-Parallel Skipping

Lastly, this thesis addresses the comprehensive scanning limitation in the existing streaming models by exploring the possibility of quickly skipping substructures of a JSON record that are irrelevant to the queries. It first reveals a wide range of opportunities for skipping irrelevant substructures during the streaming query evaluation, then proposes a highly bit-parallel solution for implementing different skipping scenarios using bitwise and SIMD operations. These techniques lead to a new streaming framework that offers 15 bit-parallel skipping primitives, namely JSONSki. Evaluation results show that JSONSki achieves significant performance benefits compared to the state-of-the-art streaming and non-streaming approaches, meanwhile keeping the memory cost to a minimum.

Chapter 2

Grammar-aware Parallelization of Scalable XPath Querying

Semi-structured data (like XML and JSON) emerge in many domains, especially in web analytics and business intelligence. However, querying such data is inherently sequential due to the nested structure of input data. Existing solutions on XPath querying pessimistically enumerate all execution paths to circumvent dependencies, yielding sub-optimal performance and limited scalability.

In this chapter, we present `GAP`, a parallelization scheme that, for the first time, leverages the grammar of the input data to boost the parallelization efficiency. `GAP` leverages static analysis to infer feasible execution paths for specific contexts based on the grammar of the semi-structured data. It can eliminate unnecessary paths without compromising the correctness. In the absence of a pre-defined grammar, `GAP` switches into a speculative execution mode and takes potentially incomplete grammar extracted from prior inputs.

Together, the dual-mode `GAP` reduces the execution paths from all paths to a minimum, therefore maximizing the parallelization efficiency and scalability. The benefits of path elimination go beyond reducing extra computation − it also enables the use of more efficient data structures, which further improves the efficiency. An evaluation on a large set of standard benchmarks with diverse queries shows that `GAP` yields significant efficiency increase and boosts the speedup of the state-of-the-art from 2.9X to 17.6X on a 20-core machine for a set of 200 queries.

2.1 Introduction

Semi-structured data, like XML and JSON, is widely used in many application domains, such as web analytics [131], financial data processing [109], pub/sub applications [98], enterprise data exchanges [54], sensor networks [59], and smart buildings [121]. The volume of semi-structured data grows rapidly. Take Twitter as an example, it produces tweets in semi-structured format at a rate of 600 million per day [24]. To cope with fast data growth, users need efficient querying methods to extract information.

In comparison, the increase of modern CPU frequency has reached a plateau. Hence, exploiting parallelism is key to efficient query processing for semi-structured data. In this work, we focus on XPath queries for their basic roles in semi-structured data processing [68, 106, 88, 38].

However, parallelizing such queries is challenging due to the inherent nested structure of input data. Consider Facebook feed data [62] in Figure 2.1. Suppose using two threads to process a path query `feed/entry/id`, which aims to find the IDs of entries in

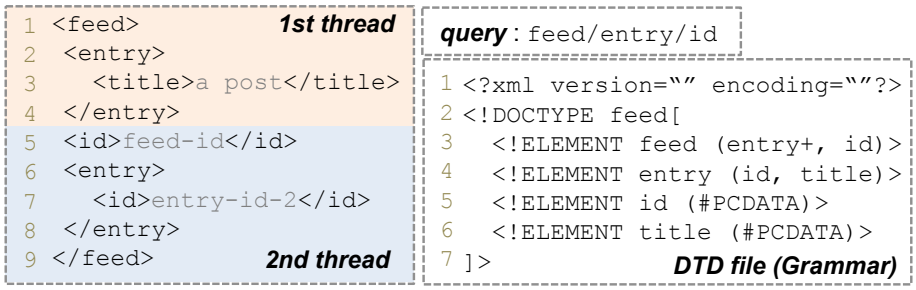

Figure 2.1: An illustration example (Facebook feed data [1])

the feed. Without examining the tags in the first half of the input, the second thread cannot determine if the `id` at line 5 is a match or not. Because it lacks the "context" of determining the answers to the query. As we will see later, such inherent dependence clearly manifests when the queries are formalized as *pushdown transducers* − a basic computation model for processing data that are defined by context-free grammars (Section 2.2).

State of The Art. Great efforts have been made on making parallel parsing possible for semi-structured data. With the parsing results, a DOM (Document Object Model) tree, the queries can be easily evaluated by traversing the tree. A practical issue with such methods is that the data could be too large to parse due to the huge memory consumption caused by constructing the parse tree. In addition, the query evaluation after parsing compromises data locality.

In comparison, Odgen and others [106] directly target the parallelization of XPath queries with a novel parallel pushdown transducer. By breaking the input into chunks and enumerating all execution paths for each chunk (except the first one), pushdown transducers are able to process different chunks in parallel. This method shows promising results

[1]For illustration purpose, the data is simplified and slightly reordered.

for single-query processing and processing a small set of concurrent queries. However, as the query becomes more complex or the number of concurrent queries increases, the total number of execution paths also raises, resulting in a dramatic increase of parallelization cost. Consequently, the speedup quickly drops from 11.1X to merely 2.9X on a 20-core Intel Xeon server, as shown in Figure 2.2.

In this work, we raise and address some fundamental questions in the parallelization of pushdown transducers: *is it necessary to enumerate all execution paths? If not, how to know which execution paths are unnecessary?*

The key to answering the questions lies in the special properties of input data – *they are defined by (context-free) grammar, either explicitly or implicitly.* For example, XML data are defined by document type definition (DTD) or XML schema. Similar ways are used to define JSON data as well [64]. The grammar not only defines the valid structure of the data, but also adds constraints on the possible execution paths under a specific context. Consider the example in Figure 2.1. By checking the first element `<id>`, the second thread would be able to infer that it may be under the `<feed>` or an `<entry>`, but definitely not under a `<title>`, as the grammar does not specify an ID for the title.

To leverage these insights, we propose a *grammar-aware parallelization (GAP)* for querying semi-structured data, which can substantially reduce the amount of execution paths up to 200X according to our experiments. It brings in two major benefits. First, a thread only needs to keep a small number of execution paths (typically < 5), hence runs more swiftly. Second, when the number of execution paths drops to one, which happens quite often based on our experiments, it becomes possible to switch to more efficient data

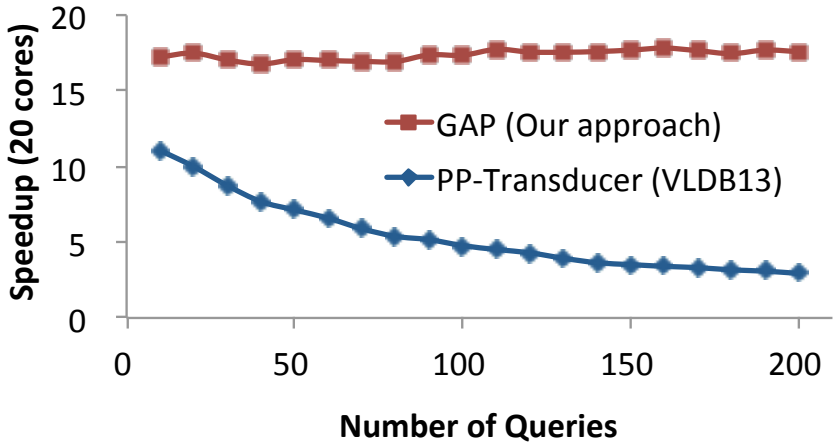

Figure 2.2: Scalability Comparison

structures (from trees to stacks) to execute the pushdown transducers. Together, they dramatically reduce the parallelization cost, making efficient parallelization of larger-scale query processing possible, as shown in Figure 2.2.

Specifically, depending on the availability of an explicit grammar, GAP works in two modes: non-speculative mode and speculative mode. In non-speculative mode (Section 2.4), GAP extracts a static syntax tree from the given grammar file (e.g., a DTD file), and symbolically executes the pushdown transducer over the syntax tree to infer feasible execution paths under a specific context (e.g., a token `<id>`). With such knowledge, a parallel pushdown transducer only maintains a set of feasible execution paths and automatically switches to a stack when it detects a unique feasible execution path is left. In certain scenarios, an explicit grammar may not be available. In such cases, GAP enters into the speculative mode, where it first collects some partial grammar from prior runs. As the grammar is partial, the inferred feasible paths might be incomplete. Hence, a pushdown

transducer runs speculatively with chances that the correct execution is left uncovered. To guarantee the correctness and reduce the penalty of misspeculation, a pair of validation and selective reprocessing mechanisms are proposed. With the dual-mode GAP, our approach is able to process semi-structured data either with or without a grammar file. Our evaluation results show that both non-speculative GAP and speculative GAP provide sustainable speedup, about 17.6X on a 20-core server, up to a set of 200 concurrent queries.

In sum, this work makes the following contributions:

- To our best knowledge, this work, for the first time, unveils the potential of leveraging input grammar to improve the efficiency of parallelization.
- It offers a rigorous inference of feasible execution paths from input grammar and an adaptive data structure switching mechanism to boost the efficiency of the basic computation model − pushdown transducers.
- It proposes a practical speculation scheme to cover the scenarios where the input grammar is unavailable.
- It, for the first time, enables a scalable parallelization of query processing that yields sustainable speedup as the query complexity increases.

2.2 Background

In this section, we first present some basic concepts for querying semi-structured data. Then we introduce the computation model for processing queries − *pushdown transducer* and its parallelization challenges.

2.2.1 Querying Semi-Structured Data

Querying semi-structured data is a routine operation in stream processing and many database systems that supports semi-structured data. For instance, YFilter [56] and XMLTK [39] leverage automata to simultaneously process a large number of XPath queries against an XML stream with a constant memory requirement. These studies focus on improving the expressiveness of queries, instead of exploiting data parallelism. On the other hand, database engines, like Microsoft SQL Server [110] and MonetDB [42] provide task-level parallelization for querying XML data, but require pre-computed index to accelerate query processing, which becomes unsuitable for single-pass processing model due to the substantial cost of index construction.

Pre-parsing v.s. On-the-fly Querying. There are two basic strategies to process queries: (i) first parse the semi-structure data into a parse tree (i.e., DOM tree in XML context), then answer the queries by searching the tree. (ii) Process the queries on-the-fly without constructing any tree structure. There are three limitations with the first strategy. First, parsing the semi-structured data requires a large memory footprint due to the construction of DOM tree, which is almost infeasible for large semi-structured data or streaming data. Second, parallel parsing itself is quite challenging, existing solutions suffer from either load imbalance or sequential bottleneck caused by a sequential preprocessing. At last, it needs to traverse the data again after the parsing, compromising the data locality. For the above reasons, we choose the second strategy in this work. To enable on-the-fly query processing, queries are implemented using stack-based computation model − pushdown

transducers [68, 106], which run over the semi-structured data stream and report matches as it proceeds. We next describe such a computation model.

2.2.2 Pushdown Transducers

Informally, a pushdown transducer is a finite automaton agumented with a stack and an output tape, as illustrated by Figure 2.3. With the stack, the pushdown transducer is able to memorize prior states (i.e., history). Such memorization is the essential for processing semi-structured data where certain "context" is needed. A pushdown transducer consumes the input stream by moving a pointer forward along the input tape one by one. Based on the input symbol, the control logic (defined as transitions) examines the current state and the status of the stack, then makes adjustments to the state and stack, and write symbols to the output tape.

A generic pushdown transducer can be defined as follows.

Definition 1 *A pushdown transducer is a 6-tuple (Σ, Γ, Δ, Q, q_0, δ) where Σ is the input alphabet, Q is the set of states, $q_0 \in Q$ is the initial state, Γ is the stack alphabet – a set of symbols that can be pushed onto stack, Δ is the output alphabet, and δ is a mapping $\delta : Q \times \Gamma \times \Sigma \to Q \times \Gamma \times \Delta$ which is also called the transition function.*

There are three factors that may affect the moving of a pushdown transducer: (i) current input symbol $c \in \Sigma$ from the input tape, (ii) current state $s \in Q$, and (iii) the element $e \in \Gamma$ on top of the stack. As a response, the pushdown transducer may take three actions: (i) move the current state to the next, (ii) update the top of the stack either by `push` or `pop`, and (iii) write a symbol $c' \in \Delta$ to the output tape. Note that, depending on the

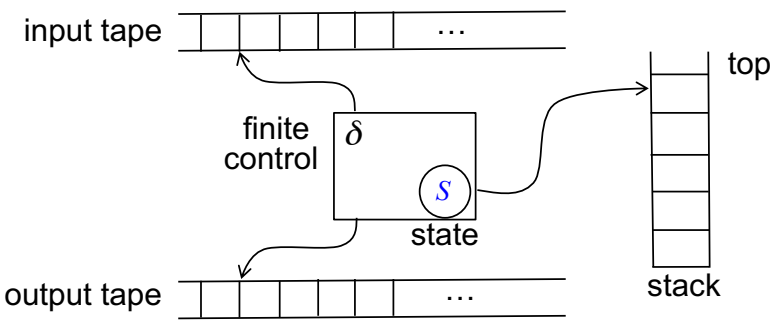

Figure 2.3: A Diagram of The Pushdown Transducer

definition of transitions, the three factors may not always take effects at every transition, and the same is true for the actions.

We next describe the *specific pushdown transducers* for query processing using the running example in Figure 2.4. It contains a grammar with recursion (between elements `a` and `b`) to demonstrate its generality.

States Q. For any give path query, a corresponding finite automaton can be created based on a simple automaton construction algorithm [68]. The states in the finite automaton are exactly the states in the pushdown transducer. The accept state(s) correspond to the matches of the query. In the running example, a path query `a/b/c` is converted to a finite automaton of five states, as shown in Figure 2.4-(c). `state 4` corresponds to the match of query.

Alphabets Σ ***,*** Δ ***, and*** Γ. In the context of XPath querying, the input alphabet Σ is the set of valid tokens (e.g., XML tags) resulted from a lexer. The output alphabet is actually the same as the input alphabet (i.e., $\Delta = \Sigma$) since the queries only return some parts of

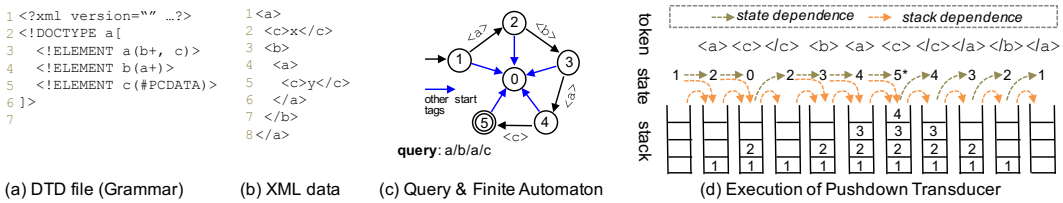

Figure 2.4: A running example with a recursive grammar. '*' after state 5 means a match.

the input data stream. As to the stack alphabet, it can be either the state set Q or input alphabet Σ. Based on the convention [106], we set $\Gamma = Q$.

Transitions δ_{plain} **,** δ_{push} **,** ***and*** δ_{pop}. There are three types of transitions in XPath querying pushdown transducers.

1. Plain transitions δ_{plain}: $Q \times \Sigma \rightarrow Q \times \Delta$. They do not influence the stack. For example, the transition after reading a text token `x` is a plain transition.

2. Push transitions δ_{push}: $Q \times \Sigma \rightarrow Q \times \Gamma \times \Delta$. When an start tag (e.g., `<a>`) is read, the transducer first pushes the current state onto the stack, then transition to the next state based on the finite automaton.

3. Pop transitions δ_{pop}: $Q \times \Sigma \times \Gamma \rightarrow Q \times \Delta$. When a close tag (e.g., `</d>`) is met, the transducer pops the stack and sets the current state to the popped state.

Example. Figure 2.4-(d) illustrates the transitions of the pushdown transducer for processing query `a/b/a/c`. Among the ten steps, five of them are push transitions and the others are pop transitions. When the pushdown transducer moves to the accept state (`state 5`), it reports a match to the query and outputs the match to its output tape.

Inherent Dependence. Based on the definition of pushdown transducers, it is obvious to notice two types of dependence in its execution depending on the variables: (i) At every step of the transition, the next state depends on either the immediate prior state or the element on top the stack (which is actually an even earlier state). We refer to such dependence as *state dependence*. (ii) Similarly, at each step of the transition, the content of stack depends on stack before the transition and sometimes also depends on the prior state. We refer to this type of dependence as *stack dependence*. In general, a tight dependence chain is formed from the first step to the current step, making the parallelization of such computation model quite challenging, as illustrated by Figure 2.4-(d).

2.2.3 Parallel Pushdown Transducers

To circumvent the inherent dependence in pushdown transducers, Ogden and others [106] propose a parallel pushdown transducer. Basically, it works in three major phases:

- *Split phase*. It first cuts the input data into equal-sized chunks, each of which may contain some broken tags.

- *Parallel phase*. For each chunk, except the first one, it enumerates all execution paths, each starting from a different state with an empty stack. A mapping m is maintained from each pair of starting state and stack (q_s, z_s) to a 3-tuple of finishing state, stack and output tape (q_f, z_f, o), that is $m = (q_s, z_s, q_f, z_f, o)$, where $m \in M = Q \times \Gamma^* \times Q \times \Gamma^* \times \Delta^*$. In our running example (Figure 2.4), it needs to enumerate all the six states (paths) and keeps a mapping for each of them.

During the processing of a chunk, different execution paths may lead to the same finishing state and stack (called *path convergence*). To improve the efficiency, a double-tree data structure is employed to compress the mapping and reduce some redundant computations when path convergence occurs.

For a chunk with missing start tags (due to partitioning), it is possible that a pop operation is needed while the stack is empty. In this case, it also has to enumerate every possible element that could be popped out from the stack (i.e. Γ). In such cases, a single path diverges to multiple ones, referred to as *path divergence*. In our running example, if the second thread starts from Line 5, when it processes the end tag `</a>` at Line 6, it encounters a path divergence, similarly to the end tags `</b>` and `</a>` at Lines 7 and 8.

- *Join phase*. Finally, mappings from different chunks are linked by matching the finishing state and stack in a chunk with the starting state and stack of its following.

An additional *filtering* phase may be added to enhance the expressiveness of the transducers (e.g., to handle predicates in XPath queries). Though the split, join and filter phases are sequential, they carry much less computations than the parallel phase and incur marginal cost [106].

Parallel pushdown transducers show promising results for single query or a small set of concurrent queries. However, as the query becomes more complex and as the number of queries increases, enumerating all execution paths tends to be less practical due to large amount of extra computation. To address this question, we propose a new scheme of parallelization, *grammar-aware parallelization (GAP)*, that leverages the grammar of the

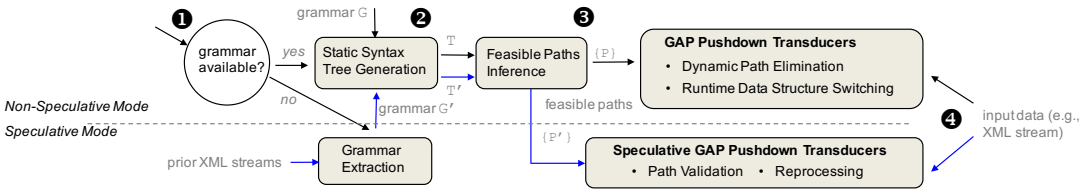

Figure 2.5: Overview of GAP. Labeled numbers show the major steps when predefined input grammar is available (e.g., DTD or Schema). Blue lines show the speculative execution flow in the absence of a pre-defined grammar.

semi-structured data to guide the design of parallelization. In the following, we first give an overview of GAP before elaborating its major components.

2.3 Overview

In this section, we give the high-level design of GAP, a new parallelization scheme that leverages input grammar to boost the efficiency of parallelization.

Grammar Availability and Dual-mode GAP. The key of GAP is the awareness of semi-structured data grammars. A substantial amount of semi-structured data come with grammars, such as the auction data from ebay developer API [55] and the publication data from DBLP [53]. Actually, six out of ten datasets from UW's XML data repository [126] contain some DTD file(s). On the other hand, it is not uncommon that the grammar of some semi-structured data is unavailable, either because there does not exist a pre-defined grammar, or because the access to the DTD or XSD is not granted.

To cover both scenarios, GAP can run in two modes: *non-speculative mode* and *speculative mode*. When the grammar of the semi-structured data is available (e.g., in the form of DTD), GAP enters into non-speculative mode, where the execution correctness is

always guaranteed, thanks to the rigorous feasible path inference from a complete grammar (as explained shortly). In another word, there is no speculation or prediction involved in this mode. On the other hand, when the grammar is unavailable, GAP switches into speculative mode. In this mode, GAP first collects some partial grammar by inferring it from previous runs (of the same data corpus). This part is illustrated as the first step of GAP in Figure 2.5.

Feasible Path Inference. After determining the execution mode, GAP obtains either a complete grammar or a partial grammar. With the grammar, GAP aims to infer which paths are feasible given a specific context, that is, a symbol from the input alphabet Q (e.g., a tag `<c>` in Figure 2.4-(b)).

To achieve this, GAP follows two steps: *static parse tree generation* and *symbolic execution of pushdown transducer*. In the first step, it takes the input grammar G as input and generates a static syntax tree T, which embodies the legal nesting structures of the input data. For example, element `b` has to be an immediate child of element `a` (see Figure 2.4-(a) and 2.4-(b)). In the second step, it executes the pushdown transducer symbolically by executing on every path of the static syntax tree to infer the feasible starting states when meeting a specific input symbol (i.e., the context). The results of the second step is a hash table, called *feasible path table*, where the key is input symbol and the value is the set of feasible paths, as shown in Table 2.1. Since the syntax tree is static and typically small, a full traversal is affordable. When the grammar is available beforehand, both steps can be done offline.

Table 2.1: Feasible Paths Table

Input symbol	Feasible paths/states
<a>	{1, 3}
	{2, 4}
<b>	{2}
...	...

GAP Pushdown Transducers. To leverage the results from feasible path inference, we design a new type of parallel pushdown transducer – *GAP pushdown transducers*. First, when it is needed, a GAP pushdown transducer can inquire the feasible path table to determine and remove unnecessary paths, which is referred to as *dynamic path elimination*.

Second, whenever a unique feasible path is left, either caused by path elimination or path convergence, the GAP pushdown transducer switches from the default double-tree data structure to a stack and executes exactly like a sequential pushdown transducer. This makes the GAP pushdown transducer run swiftly without any house-keeping work. This feature is referred to as *runtime data structure switching*.

Speculative GAP. As explained earlier in this section, when a pre-defined grammar is unavailable, only a partial grammar might be collected. Inferring feasible paths from such an incomplete grammar may result in some feasible paths missing. Consequently, feeding such incomplete information to a GAP pushdown transducer may cause the correct execution path being excluded. This scheme is generally referred to as *speculative execution*. There are two basic requirements for effective speculative execution. First, it has to ensure the correctness. GAP pushdown transducers ensure this in the join phase. When no match of mappings is found during the linking of two consecutive chunks, a misspeculation is reported. Under this situation, a reprocessing is initiated immediately on the missepculated chunk to

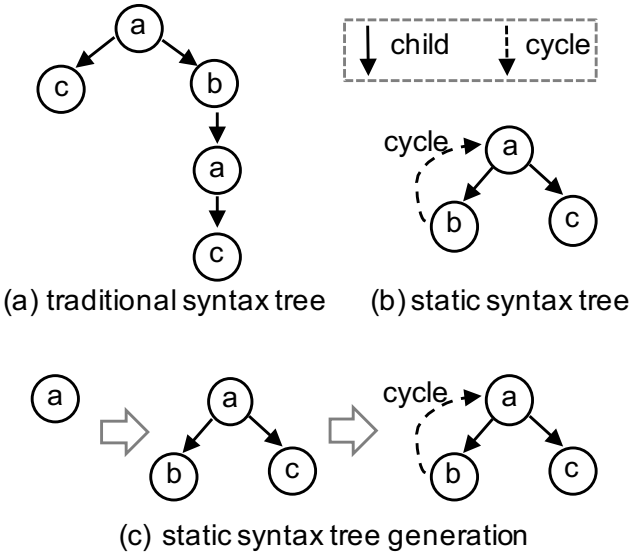

Figure 2.6: Static Syntax Tree

ensure the correctness. Second, the misspeculation penalty should be low enough so that it will not cancel out the benefits of speculation. To achieve this, GAP pushdown transducers carefully divide chunks into even smaller pieces such that reprocessing is done selectively.

Figure 2.5 illustrates both the basic and speculative GAP pushdown transducers. We next elaborate the two modes of GAP in the following two sections.

2.4 Non-Speculative GAP

When the grammar of the semi-structured data is available, GAP runs in non-speculative mode. In this section, we first describe the two steps to infer feasible paths: static syntax tree generation and symbolic pushdown transducer execution, then present more details about the new features of GAP pushdown transducers.

2.4.1 Static Syntax Tree Generation

Grammars written in textual rules are hard to interpret directly. To effectively leverage the grammar, we need a data structure to capture all the nesting relations among different input symbols/elements in a concise form.

In fact, the nesting relations can be naturally represented as a tree structure. A traditional syntax tree (or parse tree) indicates the derivation of a concrete sentence based on its (context-free) grammar. Figure 2.6-(a) shows the syntax tree of the running example (Figure 2.4-(b)). Note that such a syntax tree can grow to very large size as the input gets longer, meanwhile it may not capture all the possible derivation of its context-free grammar.

To represent all the derivation relations in the grammar with a concise format, we introduce *static syntax tree* − a tree generated purely based on the input grammar. As shown in Figure 2.6-(b), in a static syntax tree, each child element must appear and only appear once. The size of a static syntax tree depends on the input grammar, rather than input data, hence will not increase as data grows.

The construction of a static syntax tree is straightforward. Basically, a construction algorithm scans the grammar file, identifies the root element and its child/descendant elements iteratively until all leaf elements (elements without any child elements) are included, as shown in Algorithm 1. For recursive grammars, the algorithm labels nodes that refer to an ancestor by setting the field *cycle*. Figure 2.6-(c) illustrates the major steps of executing the algorithm on the running example. The time complexity of this algorithm is $O(n^2)$ where n is the number of elements in the DTD file.

Algorithm 1 Construction of Static Syntax Tree

```
1: procedure static_syntax_tree_construction(DTD)
2:     for each element el in DTD do
3:         if el is the first element then
4:             root = create_node(el); /* assume the 1st is root */
5:             root.cycle = null; /* represent recursion if exists */
6:             nodes = root;
7:         else/* search el in tree root */
8:             nodes = tree_search(root, el);
9:         for node in nodes do
10:            for each subelement sub of el do
11:                if sub.is_ancestor(node) then /* recursion */
12:                    node.cycle=sub; /* set a backward pointer */
13:                else
14:                    child = create_node(sub);
15:                    child.cycle=null;
16:                    node.add_child(child);
17:    return root;
```

2.4.2 Symbolic Execution of Pushdown Transducer

The goal of symbolic execution is to find out feasible execution paths for each input symbol. Formally, we define the feasible execution path as follows.

Definition 2 *Given an input symbol $c \in \Sigma$, for any syntactically correct inputs, if there exists an input such that the pushdown transducer can transition to state $s \in Q$ right before it reads c, then we call the execution path that starts from state s as a* feasible execution path *for symbol c.*

Algorithm 2 Feasible Paths Inference

```
1: procedure path_inference(root, init_state)
2:     current = init_state; /* current state in automaton */
3:     node = root; /* current node in static syntax tree */
4:     stack.push(root); /* for depth-first traversal */
5:     while stack is not empty do
6:         node = stack.pop();
7:         if node.cycle != null and node.done == false then
8:             unfold_cycle(hash, current, node); node.done = true; /* done with this cycle */
9:         if node.visited == false then
10:            next = next_state(current, node.start);
11:            hash.add(node.start, current); hash.add(node.end, next);
12:            current = next; node.visited = true;
13:        for each child of node do
14:            if child.visited == false then
15:                stack.push(child);
16:        if node is a leaf then current = next_state(current, node.end);
17:    return hash;
```

According to the definition, to infer the feasible paths for an input symbol, it needs to test all the possible inputs of syntactically correct semi-structured data, which is impossible. We address this by running the pushdown transducer symbolically on the static syntax tree. Basically, the transducer walks on every path of the static syntax tree and records the state right before meeting the start tag and end tag of a node on the path. For example, if the current state is `s` and the current node is `n`, then it adds a `key:value`

Algorithm 3 Feasible Paths Inference (Continue from Algorithm 2)

```
1: procedure unfold_cycle(hash, cur_state, node)
2:     /* current state and node in the cycle */
3:     current' = next_state(current, node.start);
4:     n = node.cycle; next = next_state(current', n.start);
5:     if next != null then
6:         queue.add(n);
7:         while queue is not empty do
8:             n = queue.remove(); next = next_state(current', n.start);
9:             hash.add(n.start, current'); hash.add(n.end, next);
10:            current' = next; nodes = def_children(n, current');
11:            if nodes is not empty then queue.add(nodes);
12:    return hash;
13: /* get node's children whose start tags are defined in state */
14: procedure def_children(node, state)
15:    for each child of node do
16:        if next_state(state, node.start) != null then
17:            defined_children.add(node);
18:    return defined_children;
```

pair of `<n>:s` and `</n>:s'` into the hash table, where `s'` is the next state after consuming symbol `<n>`.

Recursion Handling. For a grammar with recursion, the static syntax tree contains one or more `cycle`(s). In this case, the algorithm unfolds the cycles. However, a naive unfolding would result in dead loops. To address this, we use the finite automaton to guide the unfolding and only unfold it to the extent that is necessary for revealing feasible states.

Algorithm 2 shows the pseudocode of the symbolic execution. It takes the query automaton and static syntax tree as inputs and outputs feasible path hash table. The time complexity of symbolic execution is $O(n + g \cdot |query|)$, where n is the number of nodes in the syntax tree, g is the number of cycles in the syntax tree, and $|query|$ is the length of the query. The time complexity of dealing with cycles depends on the length the query.

Example. Figure 2.7 illustrates an execution of Algorithm 2 on the running example with the finite automaton in Figure 2.4-(c) and the static syntax tree in Figure 2.6-(c). Note that though there is a cycle in the static syntax tree (labeled at node `b`), the algorithm terminates quickly after it finds that `b` is not defined at state `4`.

```
/* H: hash table, S: stack, Q: queue */
node=a current=1 S{a} H{} // line 2-4 in Alg. 3
node=a current=2 S{b,c} H{<a>:1, </a>:2} // line 9-15 in Alg. 3
node=c current=0 S{b} H{<a>:1, </a>:2, <c>:2, </c>:0} // line 9-15 in Alg.3
since c is a leaf node, current=2 (i.e., the next state after </c>) // line 16-17 in Alg.3
node=b (b has a cycle) current'=3 n=a Q{a} // line 2-5 in Alg.2
current'=4, Q{c} H{<a>:{1,3}, </a>:{2,4}, <c>:2, </c>:0} // line 6-11 in Alg.2
/* since b is not defined at state 4, only c added to queue Q{c} */
current'=5 Q{} H{<a>:{1,3}, </a>:{2,4}, <c>:{2,4}), </c>:{0,5}} // cycle is done
node=b current=2, since b is still unvisited, process b  // line 9 in Alg.3
H{<a>:{1,3}, </a>:{2,4}, <b>:2, </b>:3, <c>:{2,4}, </c>:{0,5}}, S={}, // finish
```

Figure 2.7: Algorithm 2 execution on the running example.

2.4.3 GAP Pushdown Transducers

GAP pushdown transducers are based on the basic parallel pushdown transducers (Section 2.2.3) with two novel features:

- *Dynamic path elimination*: GAP pushdown transducers leverage feasible path inference to dynamically eliminate impossible execution paths at runtime, based on specific local context.

- *Runtime data structure switching*: Benefited from path elimination and path convergence, GAP pushdown transducers switch between a stack and a double-tree data structure at runtime to maximize the efficiency.

At high-level, GAP pushdown transducers follow the same flow as basic parallel pushdown transducers: (i) *split phase*, (ii) *parallel phase*, and (iii) *join phase*. The main differences between GAP pushdown transducers and basic parallel pushdown transducers lie in the parallel phase. In this phase, each transducer processes an input chunk while maintaining a set of possible execution paths through a set of mappings M. A mapping is defined as follows:

Definition 3 *Given an input chunk i, a mapping m^i for i is a 5-tuple $m^i = (q_s, z_s, q_f, z_f, o)$, $m^i \in M = Q\times\ \Gamma^* \times Q \times \Gamma^* \times \Delta^*$ where q_s and z_s are the starting state and stack right before processing i, q_f, z_f, and o are the finishing state, stack, and output tape after processing i.*

Basically, a mapping embodies the basic information of an execution path. For example, a mapping (1, ϵ, 4, 1:2:3, 1) says that if the transducer starts from state 1 with an empty stack, then after processing the input chunk, it will end at state 4 with a stack of 1:2:3 (":" separates items in a stack) and an output tape containing a match. Such information is used later for joining the results of different transducers.

Maintaining the mappings brings a significant amount of cost, comparing to a single path execution in a sequential transducer [106]. To reduce such parallelization cost, GAP transducers feature two new optimizations: *dynamic path elimination* and *runtime data structure switching*. Both of them happen in the parallel phase.

Dynamic Path Elimination. A GAP transducer leverages feasible path inference to prune impossible paths in three situations on the fly:

(1) At the beginning of a chunk (except the first one), a GAP transducer uses the first symbol of the chunk as the key to look up the feasible path hash table. In return, it obtains a list of feasible execution paths Q_{hash}, which is often a strict subset of all execution paths Q (i.e., $Q_{hash} \subseteq Q$). Then it processes the chunk with only the paths from Q_{hash}. For example, a GAP transducer that starts at Line 5 of the XML data in Figure 2.4-(b) would look up `<c>` in the hash table (the last line of Figure 2.7), and keeps only paths {2, 4}.

(2) During the processing of a chunk, path divergences might happen − when a pop operation is needed while the stack is empty, the parallel transducer has to enumerate all possible states that could be popped out from the stack. Prior work [106] extends the finite automaton by defining the transitions for all end tags, and only includes states that have a defined transition for the given end tag (denoted as Q_{fa}). Since the inference is only based on the finite automaton (i.e., query), regardless the input grammar, the state amount of reduction is limited [2]. In evaluation, we found it often fails to reduce the possibilities of popped-out states due to the inclusion of a special state that handles unrelated tags (e.g.,

[2]In evaluation, we found it often fails to reduce the possibilities of popped out states due to the inclusion of a special state that handles unrelated tags (e.g., state 0 in the running example).

state 0 in the running example). In comparison, when a path divergence occurs, a GAP transducer inquires the feasible path hash table that is built based on both the query and the input grammar. Using the given end tag as the key, it can obtain a potentially smaller set of possible states (i.e., $Q_{hash} \subseteq Q_{fa}$). Since the path divergence may happen multiple times during the processing of a chunk, the feasible path inference-based elimination has great potential to reduce the path maintenance cost. Following the example in situation (1), when the GAP transducer processes Line 6, it looks up `</a>` in the hash table and chooses paths {2, 4}, instead of {2, 4, 0} − the choice of prior work [106].

(3) When processing the first start tag right after a path divergence (e.g., the tag `<b>` in `</d></c><b>`, supposing that path divergence happened at both `</d>` and `</c>`), a GAP transducer also inquires the feasible path table with the first tag (`<b>`). The purpose is to further reduce the number of execution paths. This is achieved the taking the intersection between the set of execution paths before the first start tag Q_{old} and the returned path set from feasible path table inquiry Q_{hash} (i.e., $Q_{old} \cap Q_{hash}$).

Runtime Data Structure Switching. To reduce the cost of mapping maintenance, basic parallel pushdown transducers use a double-tree data structure [106], which compresses the mapping and reduces redundant computation in mapping updates. While being more efficient than maintaining each mapping separately, it is much slower than the stack-based execution of a sequential pushdown transducer.

An interesting observation we found is that the number of feasible execution paths often drops to one, thanks to the dynamic path elimination and path convergence. To

leverage this insight, we allow GAP transducers switch back to stack-based execution once a single feasible path is left.

More specifically, when a path elimination is performed, the GAP transducer checks the number of feasible execution paths. If the transducer finds only one path is left, then it switches to use a stack to continue its execution until the next path elimination check; otherwise it uses a double-tree data structure to maintain multiple execution paths. As discussed earlier, a path may diverge into multiple ones under certain circumstances (see Section 2.2.3). In such cases, the GAP transducers switch to a double-tree data structure right after the divergence. Since path elimination can happen multiple times for a GAP transducer depending on the contents in the chunk, the data structure switching can also occur multiple times. We hence refer to it as *runtime data structure switching*. Note that the switching cost is usually negligible as the switching typically occurs less than 5 times in millions of transitions, according to our experiments.

Finally, once each GAP pushdown transducer finishes its chunk, the mappings from different transducers are joint pair by pair using the same rules as basic parallel pushdown transducers [106]. Basically, to join a pair of mappings m^1 and m^2, it requires their states and stacks are matched appropriately (e.g., $m^1(q_f) = m^2(q_s)$).

In sum, GAP pushdown transducers are based on basic parallel pushdown transducers and work in three phases. With the two novel features, dynamic path elimination and runtime data structure switching, their execution cost can be potentially reduced.

2.5 Speculative GAP

In certain scenarios, a pre-defined grammar may not be available. For example, the grammar has not been explicitly defined or its access is not granted. GAP addresses this challenge with speculative execution. In this section, we first describe a method for extracting partial grammars from input data, then introduce the speculative execution of GAP pushdown transducers, including a local reprocessing technique to reduce the mis-speculation penalty.

2.5.1 Partial Grammar Extraction

In the absence of a pre-defined grammar, we leverage the insight that *input data implicitly dictate the grammar to a certain extent, though may not cover the entire grammar.* Specifically, we design a method that automatically extracts a partial static syntax tree from prior inputs.

Many applications process semi-structured data from the same source repetitively. For example, web analytics that receive semi-structured data from a specific social network website or data exchanges between two enterprises. In such scenarios, the input data from run to run tend to follow a similar *structure* as they are all defined by the same "hidden" grammar. Hence, it provides an opportunity to "learn" grammar from earlier runs.

Algorithm 4 shows a straightforward way to extract a (partial) static syntax tree directly from an input stream. More sophisticated grammar learning algorithms are also available from prior studies [118, 119].

Algorithm 4 Static Syntax Tree Extraction from Input Data

```
1: procedure static_syntax_tree_extraction(stream)
2:     while stream has next tag t do
3:         if t is the first element then /* root node */
4:             root = create_node(t); stack.push(root);
5:         else if t is a start tag then
6:             parent = stack.top();
7:             /* find a child of parent with tag t */
8:             child = find_child_by_tag(parent, t);
9:             if child != null then
10:                stack.push(child);
11:            else /* create a new child for parent */
12:                node = create_node(t);
13:                parent.add_child(node); stack.push(node);
14:        else if t is an end tag and t == stack.top().end then
15:            stack.pop();
16:        else
17:            print("errors in inputs");
18:    return root;
```

2.5.2 Speculative GAP Pushdown Transducers

Speculative GAP pushdown transducers are augmented with the capabilities of validating the speculative execution paths and initiating appropriate reprocessing when speculation fails. Since Section 2.4.3 already describes the basics of GAP pushdown transducers, here we focus on the aspects of speculative execution.

Speculative Execution. With the extracted grammar, GAP can infer feasible paths in the same way as inferring from a pre-defined grammar. The difference lies in the results − the feasible paths may be incomplete due to missing parts of the extracted grammar. Consequently, GAP transducers run speculatively, with the potential of missing the true path. Similarly to other speculation schemes, speculative GAP transducers first run with speculated data (i.e., feasible paths from extracted grammar), then rely on the validation and reprocessing to guarantee the correctness.

Note that when a speculative GAP transducer looks up an incomplete feasible path table, it may not even find any corresponding feasible paths for the inquired input symbol, simply because the symbol does not appear in the extracted grammar. In such situations, the GAP transducers degrade themselves to basic parallel pushdown transducers and enumerate all execution paths.

Another difference with the non-speculative transducers is about the third scenario of dynamic path elimination (see Section 2.4.3). When processing the first start tag after a path divergence, it does not take the intersection between two path sets Q_{old} and Q_{hash}. Instead, it simply uses Q_{hash} to replace Q_{old}, for two reasons: first, this creates chances to correct an earlier misspeculation; second, it allows the reprocessing to be performed selectively (as explained next).

Validation and Reprocessing. Once every GAP transducer finishes its chunk (speculatively), they enter into the join phase, where the validation and reprocessing are performed.

According to Section 2.4.3, the feasible path table is used in three scenarios: (1) at the beginning a chunk (except the first one); (2) in the occurrence of a path divergence;

(3) when processing the first start tag after a path divergence. Since the feasible path table is extracted from a partial grammar, it might miss the true execution path. Hence, a validation is required for each of the above three scenarios, respectively. For example, during the joining of a pair of mappings from two consecutive chunks, if the starting states of the latter chunk do not include the correct finishing state of the earlier chunk, a misspeculation is detected. In either case, when a validation fails, a GAP transducer would reprocess the failed part with the corresponding correct starting state. Note that a misspeculation at the beginning of of a chunk does not necessarily mean that the whole chunk needs to be reprocessed. Thanks to the other two dynamic path elimination scenarios, an earlier misspeculation might be corrected by a latter path elimination operation. Hence, the reprocessing of a chunk is usually performed selectively.

2.6 Evaluation

We evaluate GAP on a set of real-world XML benchmarks and examine its efficiency and scalability on a variety of path queries, including a set of 200 queries.

Implementation. We prototyped GAP in C language. It includes four major components: a static syntax tree generator that takes a DTD/XSD grammar as input and outputs a static syntax tree; a symbolic pushdown transducer executor that runs symbolically over a given static syntax tree; a multi-threaded GAP pushdown transducer implemented using Pthread and can be tuned into either *speculative mode* or *non-speculative mode* (by default, it is set to speculative mode), as well as a grammar extractor that can be enabled either online (for streaming data) or offline (for storage data).

Methodology. We compare GAP with the state-of-the-art parallel pushdown transducers [106], which has shown comparable performance to PugiXML [82], a popular C++ XML processing library and much better performance than Expat [52]. Table 2.2 summarizes the versions used in our comparison. For speculative GAP versions, we randomly choose a portion of the complete grammar [3].

Table 2.2: Method Versions in Evaluation

Method Versions	Abbreviation
Parallel pushdown transducer [106]	PP-Transducer
Non-speculative GAP	GAP-NonSpec
Speculative GAP with 20% grammar	GAP-Spec(20%)
Speculative GAP with 40% grammar	GAP-Spec(40%)
Speculative GAP with 80% grammar	GAP-Spec(80%)

All experiments run on a 20-core machine equipped with two Intel 2.13 GHz Xeon E7-L8867 processors. The machine runs openSUSE Leap 42.1 and has GCC 4.8.5. All programs are compiled with "-O3" optimization flag. The timing results reported are the average of 10 repetitive runs. We do not report 95% confidence interval of the average when the variation is not significant.

Benchmarks. Table 2.3 lists datasets used in our experiments as well as their statistics. Except XMark, all the datasets are from a commonly used UW XML data repository [126], which covers a variety of XML applications. To make a fair comparison with prior work [106], we also use a scaling factor that replicates the datasets, resulting in sizes from 600MB to 6GB. We do not report results for larger datasets because the measurements

[3]To ensure the partial grammar is meaningful, we randomly and recursively remove leaf elements from the original grammar.

are stable due to the large amount of repetitive computations involved in semi-structured query processing.

Table 2.3: XML datasets (d means depth)

Dataset	#tags	d_{max}	d_{avg}	Dataset	#tags	d_{max}	d_{avg}
Lineitem	34,781,152	3	2.94	DBLP	33,321,292	6	2.9
SwissProt	29,770,302	5	3.55	NASA	237,230,520	8	5.58
Protein	42,597,466	7	5.15	XMark	23,328,398	13	5.55

We use queries from XPathMark[63] for its designated purpose of XPath query evaluation and its realistic query structures. As listed in Table 2.4, the query set covers the whole set of A-type queries as well as two B-type queries in XPathMark. When predicates, parents or ancestors are used, the queries are translated into subqueries or rewritten, such that they can be merged into a single pushdown transducer. The right-most two columns show the number of subqueries and number of matches in the datasets.

Single-Query Performance. The speedup for single-query processing is shown in Figure 2.8 (left). The baseline is the sequential pushdown transducer.

Among the five versions, GAP-NonSpec yields the best speedup on all tested queries, leading to an average of 15X speedup. In comparison, PP-Transducer achieves least speedup except for benchmarks XM1 and XM2, in which cases the GAP-Spec(20%) performs the worst. It is easy to notice the clear trend among the three speculative versions. As the extracted syntax tree becomes more complete, the performance improves. However, the trend varies across different benchmarks, implying that the performance of speculative versions are less predictable. It is worth to note that even with 20% syntax tree, the GAP-Spec still yields better performance than PP-Transducer (13.2X v.s. 11.6X).

To better understand the performance results, we profiled the average number of starting execution paths, as shown in Table 2.5. The last row of single query block shows the geometrical mean of the number of starting execution paths for each version. It clearly shows a big discrepancy between PP-Transducer and GAP-NonSpec. The discrepancy confirms the effectiveness of the proposed dynamic path elimination and demonstrates its benefits in improving the performance.

Table 2.4: XPath queries. #sub shows the number of sub-queries in each query structure.

Query	Dataset	Query structure	#sub	#matches
NS1	NASA	/ds/d/tb/ts/tl/tit	1	2,119,760
NS2	NASA	//ds/d/tit	1	3,395,250
PT1	Protein	/pd/pe/r/ri/xs/x/u	1	279,402
PT2	Protein	/p/pe//u	1	1,949,416
Query	Dataset	Query structure	#sub	#matches
DP1	DBLP	/dp/ar/au	1	1,107,323
DP2	DBLP	//dp//ed	1	31,940
DP4	DBLP	/dp/ar[tit]/jn	3	558,046
XM3	XMark	//k/ancestor::li/t/k	3	241,556
NS3	NASA	/ds/d[descendant::tit or descendant::na or descendant::kw]/an	5	1,826,250
NS4	NASA	/ds/d[tit and al]/r/s/o/au/ln	4	191,250
PT3	Protein	/pd/pe/r[aci/acs or at or ct or nt]/ri/ats/at	6	5,668,287
DP3	DBLP	/dp[mt/au or mt/tit or mt/yr or pt/au or pt/tit or pt/yr or ...]/au	43	1,107,323
XM1	XMark	/s/r/*/item/[parent::af]/name	1	3,851
XM2	XMark	//s[r/*/item[parent::au parent::af ... parent::eu]/mb/m/t/k/ b or ...]/pp/ps/ name	18	178,500

Multi-Query Performance. To evaluate the performance of multi-query processing, we use 12 groups of queries, with three different sizes: 20 queries, 40 queries, and 80 queries.

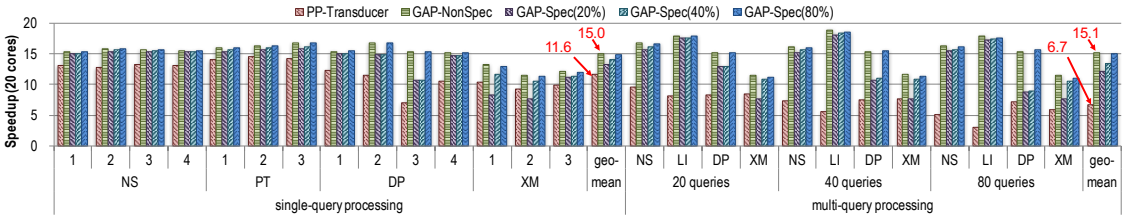

Figure 2.8: Speedup of Single-Query and Multi-Query Processing on A 20-core Machine

Multi-query processing speedup is shown in Figure 2.8 (right). Overall, the results follow a similar pattern as single-query processing. However, the performance differences among different versions become larger, especially between PP-Transducer and four GAP versions. Specifically, GAP-NonSpec produces 15.1X, similarly to its single-query performance, while PP-Transducer only yields 6.7X speedup. The reason is that when processing multiple queries concurrently, the pushdown transducers grow larger with more number of states. As a result, PP-Transducer ends up enumerating more paths, causing higher parallelization cost.

Table 2.5 reports the number of starting paths for all five versions. In single-query processing, the gap between PP-Transducer and GAP-NonSpec is about 10X (9.2 vs. 1.4 on average). While in multi-query processing, the gap quickly increases up to hundreds of times (188 vs. 2.1 on average).

Speculation Accuracy and Cost. Table 2.6 reports the speculation accuracy and the cost of misspeculation, including both single queries and query sets. The ones that are not listed do not have any misspeculation. The misspeculation cost is typically a very tiny portion of the total execution time, except queries to XM dataset which shows more than

Table 2.5: Average Number of Starting Execution Paths

Query Set		PP-Trans.	GAP-NonSpec	GAP-Spec 20%	GAP-Spec 40%	GAP-Spec 80%
single query	NS	7.7	1.0	5.7	4.8	1.1
	LI	8.4	1.1	5.3	1.3	1.1
	DP	9.0	1.7	3.1	3.1	1.6
	XM	12.4	1.9	6.3	5.5	3.7
	Geo	9.2	1.4	4.9	3.2	1.7
80 queries	NS	275.0	1.1	192.4	157.6	5.3
	LI	166.0	1.0	60.6	51.4	28.5
	DP	96.0	3.2	18.9	17.4	2.5
	XM	285.0	5.4	119.8	75.2	33.4
	Geo	188.0	2.1	71.7	57.0	10.6

Table 2.6: Speculation Accuracy and Reprocessing Cost

Query (set)		GAP-Spec(20%) cost	GAP-Spec(20%) acc.	GAP- Spec(40%) cost	GAP- Spec(40%) acc.
single query	DP1	0.003%	94.12%	0.003%	94.12%
	DP3	0.002%	94.12%	0.002%	94.12%
	DP4	0.003%	94.12%	0.004%	94.12%
	XM1	26.85%	62.50%	0%	100%
	XM2	25.10%	47.83%	0%	100%
query set	DP (20)	0.003%	88.24%	0.002%	88.24%
	DP (40)	0.002%	94.12%	0.003%	94.12%
	DP (80)	0.002%	96.43%	0.002%	96.43%
	XM (20)	24.18%	56.52%	0%	100%
	XM (40)	24.14%	56.52%	0%	100%
	XM (80)	26.02%	54.17%	0%	100%

24% misspeculation cost when using 20% grammar (GAP-Spec(20%)). There are two main reasons causing such relatively high cost: First, the speculation accuracy of these queries are relatively low, slightly higher than 50%, resulting in a substantial amount of reprocessing; Second, a few elements that appear quite often in the dataset are missing in the 20% partial grammar, in which cases, the speculative GAP transducers degrade themselves to basic parallel transducers and enumerate all execution paths.

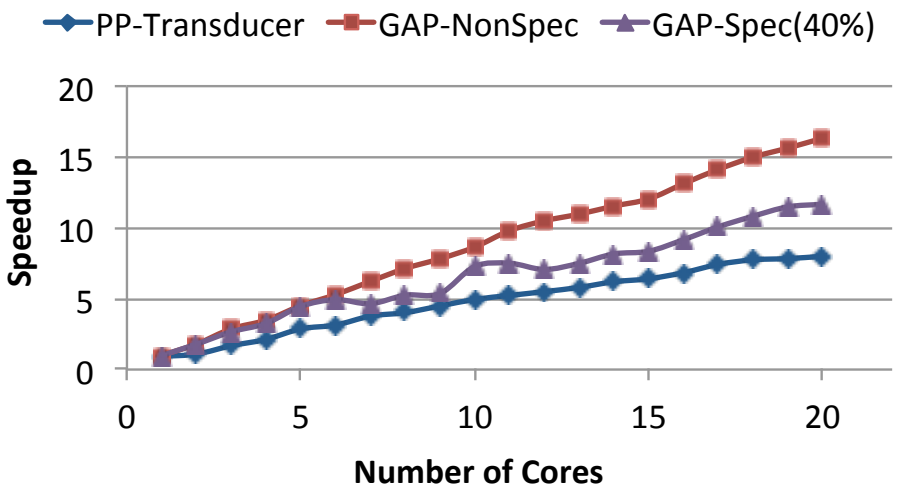

Figure 2.9: Scalability over Number of Cores

Scalability. We measured the scalabilities in terms of both the number of cores and the number of queries.

Figure 2.9 shows the scalability in terms of number of cores. All three versions show good scalability − the speedup linearly increases up to at least 20 cores. Meanwhile, it also clearly shows the trend of their differences − as the number of cores increases the performance gap among these three versions will become even larger.

Figure 2.10 shows the scalability in terms of number of queries. PP-Transducer shows a sharp decrease as the number of queries increases, which aligns well with the results reported by prior work [106]. In comparison, the two GAP versions show no degradation at all up to at least 200 queries, thanks to the dynamic path elimination.

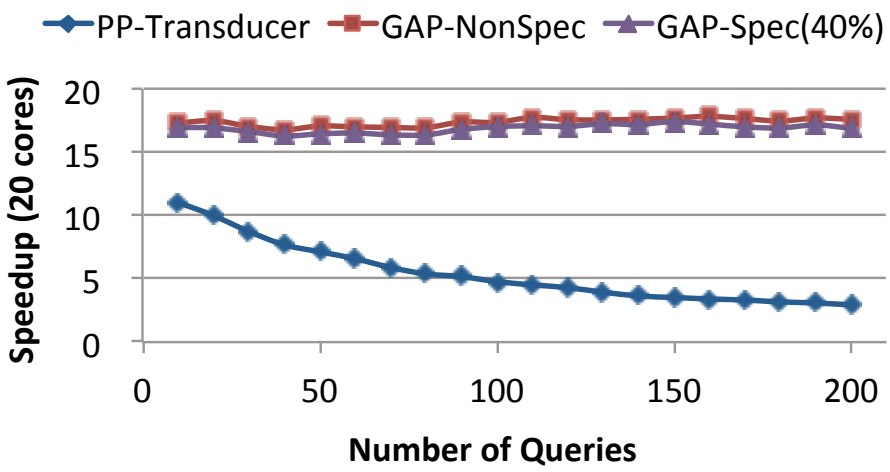

Figure 2.10: Scalability over Number of Queries

2.7 Related Work

There is a large amount of efforts on making parsing parallel for both context-free grammars and non-CFG grammars [105, 120, 127, 66, 67]. They provide valuable insights to this work. In particular, [66] leverages some rules extracted from the input structure to facilitate the parallel expression evaluation. Next, we will focus our discussions on XPath querying and its parallelism exploitation.

XML querying. Many prior work study the expressiveness of XPath querying and the execution of concurrent queries efficiently, including automata-based techniques [39, 134], array-based methods like TurboXPath [80] and stack-based algorithms like Twig2Stack [49]. This work uses an approach that combines a number of small queries into a single DFA and evaluates them simultaneously [68]. Y-Filter [56] and XMLTK [39] are based on this method and address state explosion with lazy DFA.

Some earlier work exploiting data parallelism in XPath queries rewrite an XPath query with predicates into several sub-queries, execute the sub-queries in parallel and merge their results sequentially [43]. However, this approach demands more hardware resources meanwhile exposes limited parallelism. The work by Zhang and others [134] executes multiple states in the NFA in parallel. Although each active state is handled by a different thread, the XML data is still processed sequentially. An alternative method proposed by Liu and others [90] uses a parallel structured join algorithm, where both query and join operations are parallelizable. However, constructing the required data structures is still a sequential step. This work is primarily based on the PP-Transducer [106] which can operate on arbitrarily framed XML chunks. More details are given in Section 2.1 and Section 2.2.3.

Parallel XML Parsing. There are two basic ways of data-level parallel parsing for XML: the partition-oriented and the merging-oriented [122, 112, 94, 133, 135]. The work by Lu and others, for example, first extracts the high-level structure of XML documents through a quick prescan [94], and parses each part of the document in parallel. In comparison, Wu and others [133]'s method cuts XML documents into chunks directly, parses them and merges the result together.

Other Parallelization. Recent work on parallelizing finite automata [137, 103, 136, 115] provides useful insights, but cannot be applied directly to pushdown transducers, due to the involvement of stack. Zhao and others design stack-based automata that predict future function calls to enable parallel Just-in-Time compilation [139]. Saeed and others [97] exploit the rank convergence in parallel dynamic programming, sharing some of the high-level ideas of this work.

2.8 Summary

This work proposes GAP, a novel parallelization scheme that leverages the grammar of semi-structured data to improve the parallelization efficiency. Depending on the availability of a pre-defined grammar, GAP can run in both speculative mode and non-speculative mode. In either case, GAP is able to infer feasible execution paths by generating a static syntax tree and executing the pushdown transducer symbolically on the tree. By feeding such information to the GAP pushdown transducers, unnecessary execution paths can be eliminated on the fly. In addition, GAP transducers feature a runtime data structure switching to further take advantage of path elimination and maximize the efficiency. Evaluation on real-world datasets and queries confirm the benefits of grammar-aware parallelization which yields consistent speedup to a large number of concurrent queries.

Chapter 3

Scalable JSONPath Querying

JSON (JavaScript Object Notation) and its derivatives are essential in the modern computing infrastructure. However, existing software often fails to process such types of data in a scalable way, mainly for two reasons: (i) the processing often requires to build a memory-consuming parse tree; (ii) there exist inherent dependences in processing the data stream, preventing any data-level parallelization.

Facing the challenges, developers often have to construct ad-hoc pre-parsers to split the data stream in order to reduce the memory consumption and increase the data parallelism. However, this strategy requires more programming efforts. Moreover, the pre-parsing itself is non-trivial to parallelize, thus introducing a new serial bottleneck.

To solve the dilemma, this work introduces a scalable yet fully automatic solution − a compilation system, namely ***JPStream***, that compiles standard JSONPath queries into parallel executables with bounded memory footprints. First, JPStream adopts a stream processing design that combines the querying and parsing into one pass, without generating any

in-memory parse tree. To achieve this, JPStream uses a novel joint compilation technique that compiles the queries and the JSON syntax together into a single automaton.

Second, JPStream leverages the "enumerability" of automaton to break the dependences and reason about the transition rules to prune infeasible states. It also features a runtime that learns structural constraints from the input to enhance the pruning. Evaluation on real-world JSON datasets with standard JSONPath queries shows that JPStream can reduce the memory consumption significantly, by up to 95%, meanwhile achieving near-linear speedup on multicore and manycore processors.

3.1 Introduction

JSON (JavaScript Object Notation) and its derivative data types (such as NetJSON [70], GeoJSON [69], JSON-LD [71], CoverageJSON [60], and etc.) form a family of contemporary semi-structured data (herein referred to as *JSON-family data types* or simply *JSON*). Together, they play a fundamental role in the modern computing infrastructure, ranging from cloud computing [47, 91] and microservice architectures [130, 58], to Internet of Things (IoT) [132] and NoSQL data stores [87, 72]. For example, major cloud providers, such as Azure [5], AWS [3], Firebase [61], and Oracle Cloud [48], all support JSON-based cloud services. Document data stores, such as MongoDB [99] and CouchDB [37], are built on JSON data.

Not only the popularity, but also the volume of JSON data grows quickly in recent years. For instance, the NASA Earth Exchange (NEX) project yielded over 20 TB climate data that is accessible from Amazon servers via JSON APIs [92]. Twitter produces tweets

as a JSON data stream at a rate of 600 million per day [24]. Public data sources, like data.gov [8], provide REST APIs to access a broad range of scientific data primarily in JSON format, with sizes quickly increasing.

To efficiently process large-volume text data, recent work proposes hardware-implemented automata, such as automata processor [57], cache automata [125], and in-SRAM pushdown automata [36]. Despite of their promising results, they are not yet readily available to the developers.

In comparison, this work focuses on the use of commodity multicore and many-core processors to accelerate JSON data processing, which, unfortunately, remains an open problem, due to the first two challenges shown in the first chapter.

Challenge I - Preprocessing Costs. Traditional strategies for processing semi-structured data require a full parsing over the entire data stream before extracting information (sub-structures of interests). This often creates a high memory pressure for large data streams. For example, JSON-C [12], a popular JSON parser requires 8GB memory to process a 1GB JSON stream. Even worse, loading a JSON file with hundreds of megabytes can easily raise an `out-of-memory` exception on CouchDB [37]. In addition to the high-memory cost, pre-processing introduces significant yet unnecessary delay as the substructures out of interests are also processed together. These high costs of preprocessing greatly limit the scalability of semi-structured data processing.

Challenge II - Inherent Dependences: Exposing effective parallelism is key to the scalable data processing on parallel computer architectures. Unfortunately, semi-structured data processing exposes limited parallelism due to its inherently nested structures. Fig-

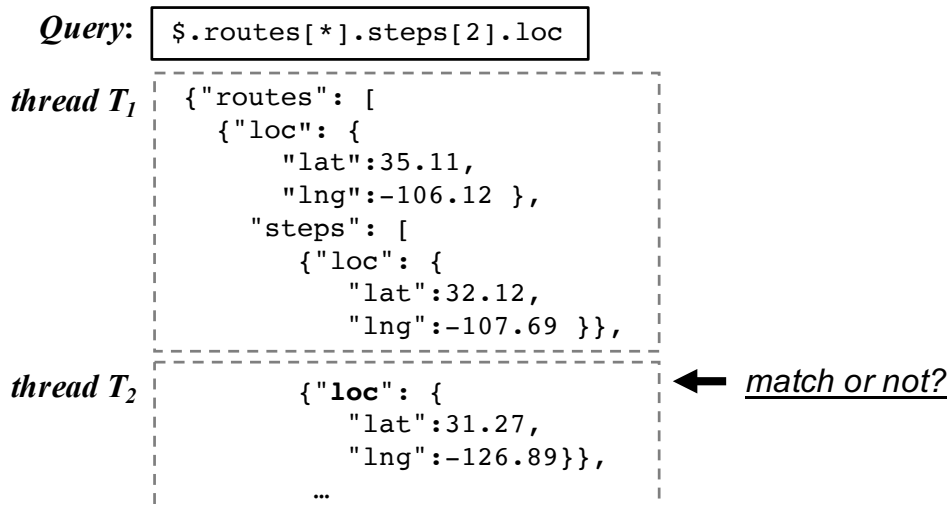

Figure 3.1: Challenge in Parallel JSON Data Processing[1].

ure 3.1 illustrates this challenge with a piece of Google route data. Suppose the user is interested in the location of the third step in each route, denoted as a path expression `$.routes[*].steps[2].loc`. Except the first thread, all the other threads cannot determine if the key `loc` satisfies the path conditions or not, due to the lack of relevant contexts. In general, such nested structures of the data serialize the processing of the whole JSON stream.

State of The Art. To cope with the preprocessing costs, stream programming paradigms, such as the SAX (Simple API for XML) model [75, 108], have been introduced. But, they simply provide callback APIs for handling the basic tokens, still requiring substantial programming efforts to design and implement a particular processing task. A more promising solution is to automatically generate deterministic automata to process the data in a streaming fashion [68]. However, so far, the automata are designed only for conventional XML. It

[1]Google route data is simplified and indented for better illustration.

remains an open question how to construct the automata for the stream processing JSON data, which follows quite different syntactical structures (see Section 3.2).

As to parallelization, prior efforts propose to pre-scan the data to find "good" boundaries for partitioning [96, 134, 95]. However, this strategy has several limitations. First, it adds an extra pass over the entire data stream, which could be very expensive for large data streams; Second, the pre-scan itself is a serial process; Third, despite the availability of "good" boundaries, they may not separate the data evenly, leading to potential load imbalance. At last, this strategy requires developers to write the pre-scanner for JSON data in a specific structure, adding extra programming burden.

Recent work [106, 79] has shown the possibility to partition an XML stream arbitrarily with aggressive parallelization techniques, thus avoiding the pre-scan. However, due to some unique features of JSON, these techniques suffer from the *path explosion problem* (see Sections 3.2 and 3.4), failing to bring performance benefits for parallel JSON processing.

Overview of This Work. To solve these problems, this work proposes a compilation system, namely ***JPStream***, that is particularly designed for JSON-family data processing. For a given set of JSONPath queries, JPStream can automatically generate automata-based parallel executables with bounded memory footprints for JSON data processing. We next briefly describe the major components of JPStream.

- *Streaming Compilation.* The key to avoid the expensive preprocessing is to adopt stream processing. However, this is non-trivial for a JSON stream which is defined by a Context-Free Grammar (CFG). The processing needs to perform both CFG parsing and query matching at the same time. JPStream addresses this with a *joint compila-*

tion. It first compiles the queries (i.e., path expressions) and JSON syntax into two separate automata. Unlike conventional automata, partially defined. JPStream then "hooks" the two automata together to form a new one, referred to this as *streaming automaton*, which turns to be a unconventional pushdown automaton with double stacks. Encoded with both queries and JSON syntax, a streaming automaton performs parsing and querying simultaneously in a single pass (see Section 3.3).

- *Parallelizing Compilation*. Unlike conventional automata that only start from the beginning of an input stream, the automata generated by JPStream can start from an arbitrary position. This flexibility is enabled by a set of parallelization techniques customized for JSON, including *streaming automaton state enumeration*, *JSON syntax-based state feasibility analysis* and *automata resetting*. Together, they eliminate the *path explosion problem* encountered by the conventional parallelization (see Section 3.4).

- *Runtime Optimization*. To further reduce parallelization cost, JPStream features a runtime that learns *hints* (i.e., structural constraints) from the data to eliminate certain unnecessary computations on the fly (see Section 3.5).

We prototyped JPStream in C language and used Pthreads for the generated code. Our evaluation using standard path expressions and a diverse set of JSON data has shown that JPStream-generated automata reduce the memory footprint by up to 95% comparing to preprocessing-based methods and scales with near-linear speedup on the commodity multicore and manycore processors (see Section 3.6).

In sum, this work makes a four-fold contribution:

```
  object  ::=  {} | {members}
 members  ::=  pair | pair,members
    pair  ::=  key:value
   array  ::=  [] | [elements]
elements  ::=  value | value,elements
   value  :=   object | array | primitive
```

Figure 3.2: BNF Grammar of JSON.

- To our best knowledge, this work introduces the first automata-based stream processing model for JSON data, with a novel joint compilation technique.
- It offers a set of parallelization techniques customized for JSON processing, making its possible to avoid the path explosion problem.
- It designs a data constraints learning scheme that can improve the parallelization efficiency at runtime.
- Finally, this work prototypes the proposed ideas and evaluates the system comprehensively.

Next, we provide the necessary background for this work.

3.2 Background

This section introduces the basic concepts of JSON and its processing, with a focus on its unique features comparing to XML, and how these features complicates the parallelization.

Grammar-based JSON. Originated from JavaScript, JSON has evolved into a language-independent data representation with a formal context-free grammar (CFG) (see Figure 3.2). The grammar follows the conventions of C-family languages, making it an ideal data-exchange language cross platforms.

Note that JSON grammar is fundamentally different from that of XML − a markup language based on tag pairs. Unlike JSON with a rich syntax, XML is almost "grammar-less" − as long as the tags are well paired, the data is considered to be valid [2]. As shown later, this "minimal grammar" feature makes the design of XML stream processing model easier. By contrast, CFG-based JSON requires a stream processing model that can cope with CFG, plus the querying. In fact, this needs a non-conventional automaton (an automaton with double stacks) to process. Next, we introduce the two major JSON structures: `object` and `array` and explain how they could complicate the design of data-level parallelization.

JSON Objects. A JSON `object` consists of zero or multiple `key:value` pairs, enclosed by a pair of curly parentheses (see Figure 3.2). For example, {`"lat":35, "lng":-106`} is a `loc` object with two key-value pairs (i.e., the attributes). This object-oriented design allows JSON to naturally fit with the data types in most object-oriented languages.

Comparing to XML tags, JSON objects are more efficient for encoding information. For example, XML needs a longer string to encode the same information in the prior example:

```
<loc><lat>35</lat><lng>-106</lng></loc>
```

[2]The XML grammar discussed here should not be confused with the user-defined grammar for a specific set of XML data, such as DTD or XSD.

Despite its conciseness, the syntax of JSON objects makes an effective parallelization more challenging. Intuitively, when an XML stream is partitioned, a pair of tags might be broken into different chunks. However, an end tag, like `</loc>`, still carries some context information − it was inside a `loc`. By contrast, a JSON object ends *anonymously*, with simply a right curly bracket }. As shown later in Section 3.4, lacking the information of object names is a key reason causing *path explosion*, a problem that can result in significantly high parallelization overhead.

JSON Arrays. The other major JSON component is `array`. A JSON `array` consists of an ordered list of `values`, embraced by a pair of square brackets (see Figure 3.2). For example, `steps` in Figure 3.1 is an array of `loc` objects.

Note that array is a new syntactical feature comparing to tag-based XML. Moreover, JSON allows the `elements` in an `array` to be `objects` or even `arrays`, creating nested structures that are potentially interleaved. On one hand, these new features make JSON more expressive than XML; On the other hand, similar to the *anonymous ending* feature, they complicate the parallelization. For example, in a broken piece of JSON data, it might be hard to tell if an object is under an array or an object, both of which can encapsulate objects. This syntactical uncertainty is another important reason for the path explosion problem (see Section 3.4).

In sum, the unique features of JSON make it a promising data language that gradually replaces conventional tag-based XML in many domains [129, 27, 123]. Meanwhile, they also complicate the designs of stream processing model and its parallelization.

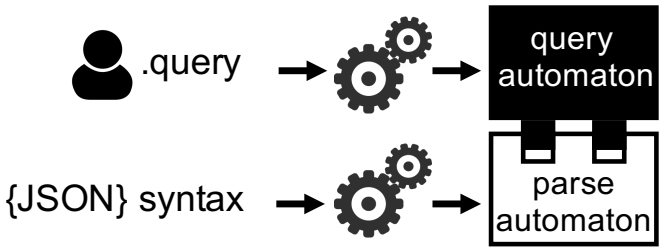

Figure 3.3: Illustration of Joint Compilation.

Querying JSON Data. In the processing of JSON data, the most basic queries are *path expressions* (a.k.a. JSONPath), which identify the substructures of interest [3]. JSONPath is the key building block of more advanced processing schemes (e.g., JSONiq [14]). As its name suggests, a path expression defines specific "paths" from the root element of the JSON data (an `object` or an `array`) to the substructures of interest. For example, `$.routes[1].steps` specifies the `steps` object under the first element of `routes` array. The dot `.` after `$` indicates that the root object is anonymous. Note that, unlike the path expressions for XML data, array indexes often appear in JSON path expressions for its support of array syntax. More details about JSONPath syntax can be found in [74].

Next, we explain how JPStream uses the path expressions and the JSON syntax for the purpose of stream processing.

3.3 Streaming Compilation

This section introduces the *joint compilation technique* that generates stream processing code for JSON.

[3]The counterpart for XML is XPath, which has the same expressiveness.

3.3.1 Idea of Joint Compilation

The basic idea of joint compilation is illustrated in Figure 3.3. First, the queries and JSON syntax are compiled separately into two automata: (i) a *query automaton* for matching query results and (ii) a *parsing automaton* for recognizing the JSON syntactical structure. Note that, different from conventional automata, the query and parsing automata are only *partially defined*, with some transition rules missing. For example, the transition rules for handling an ending curly bracket } or an ending square bracket] are undefined in the query automaton. On the other hand, there are no concrete states defined in the parsing automaton.

The partial definitions of both automata are designed on purpose to facilitate the next compilation step − *automata hooking*. Basically, the two partially defined automata are "hooked" together in a way that a deterministic and complete set of transition rules are created. The resulted automaton is referred to as *streaming automaton*, which is capable of performing parsing and querying simultaneously. Next, we elaborate the generations of query and parsing automata, and the automata hooking, respectively.

3.3.2 Path Expression Compilation

Given a set of path expressions, JPStream compiles them into a single finite automaton with transitions partially defined.

The basic idea is inspired by a seminal work that designs deterministic automata for XML querying [68]. We treat a path expression as a regular expression, where a `key` in the path expression is considered as a symbol in the regular expression. For path

expressions with predicates or logical operators, we break them down into subqueries, then group them into a set of regular expressions (also used in [106]). For example, a query `$.a[?(@.b||@.c)].d` is broken into four subqueries `$.a`, `$.a.b`, `$.a.c` and `$.a.d`, then grouped into a regular expression set {`a,ab,ac,ad`}. Then, following standard algorithms [31], the regular expressions are used to generate a deterministic finite automaton (DFA). Formally, the resulted finite automaton M_{query} is a 5-tuple:

$$M_{query} = (Q, \Sigma, \delta, s_0, F) \tag{3.1}$$

where Q is the set of states, Σ is the input alphabet, δ is the set of transition rules, s_0 is the initial state, and F is the set of accept states. When the finite automaton transitions into an accept state, a match for a subquery is found.

Two things worth to mention here:

- The transition rules set δ is partially defined. In fact, it only handles three kinds of input symbols: a `key`, an open square bracket `[`, and a comma `,`, despite the existence of other symbols. The last two symbols are used for handling array constraints, as we explain shortly.

- The DFA matches the results of the subqueries, which may still need to be merged and filtered to produce the final output. In the earlier example, a is finally outputted only if `a` match of `b` or `c` is found under `a`. Thus, there is another pass over the matches of subqueries.

Array Constraints Handling. Since JSON supports array syntax, its path expressions commonly include array indexes or index ranges, such as `steps[2]`, `steps[2:4]`

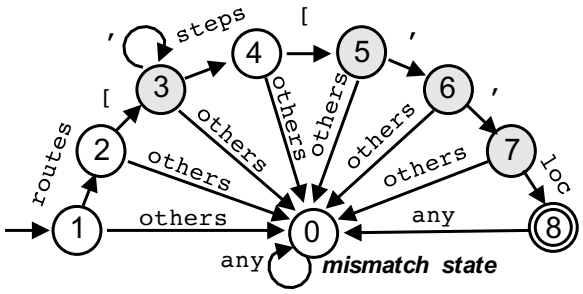

Figure 3.4: Automaton for `$.rountes[*].steps[2].loc`.

or `steps[*]`. To cope with these constraints, one option is to encode them into the automaton generation process. A trick for this is to reformat an array constraint with regular expressions of comma `,`, which separates array elements:

- A specific index range `[m:n]`[4] is reformatted to regular expression `[`, {`m,n`}, where {`m,n`} indicates the comma needs to be repeated `m` to `n` times.
- Similarly, the whole index range `[*]` is reformatted to regular expression `[`, `*`, where `*` indicates comma can be repeated any number of times.

Figure 3.4 shows the automaton for the path expression `$.routes[*].steps[2].loc`. The states with shadows are the divided states used for handling array indexes.

In practice, we can also augment the automaton with a set of counters, each for an array index constraint appeared in the path expressions. This solution adds extra condition checks to the execution, but reduce the number of states.

[4]Negative indexes like `[-5:]` are not supported (requires backtracking).

3.3.3 JSON Syntax Compilation

To precisely match the query results, it is critical to recognize the syntactical structures in JSON streams. As JSON is CFL, the recognition requires a pushdown automaton.

Stateless Parsing. Unlike the conventional CFG parsing that tracks a parsing state to build the parse tree, our pushdown automaton only needs to recognize the syntactical structures (e.g., `object`, `array`, and `key-value`), without connecting them into a tree. This critical difference makes a *stateless automaton* sufficient for our purpose. Later, we will see this "statelessness" is also important for the automata hooking.

A stateless pushdown automaton $M_{parsing}$ is a 3-tuple:

$$M_{parsing} = (\Sigma, \Gamma, \Delta) \tag{3.2}$$

where Σ = { {, }, `[ ]`, `,`, `prim`, and `key` } is the input alphabet, Γ = { {, `[`, and `key` } is the stack alphabet, and Δ is the transition rule set (listed in Figure 3.5). Each rule follows a format of $\Delta(c, s) \rightarrow s'$, where c, $c \in \Sigma$ is the current input symbol, and s is the stack content, which contains an ordered list of symbols from the stack alphabet Γ. Symbols in s are separated by comma `:`. Note that, for conciseness, the rules in Figure 3.5 show at most two top elements in the stack. The arrow $\rightarrow$ indicates a transition that changes the stack by performing `pop` or `push` operations. In general, the transitions rules are not difficult to follow. One detail worth to mention is that the colon `:` in a key-value pair is treated as part of a `key` token. For example, `"steps":` is recognized as a `key`.

So far, we have introduced the generations of both query and parsing automata. Next, we explain how to hook them.

$$
\begin{array}{rlrl}
\texttt{[Obj-S]} & \Delta(\{, *) \to \{ & \texttt{[Ary-S]} & \Delta(\texttt{[}, *) \to \texttt{[} \\
\texttt{[Key]} & \Delta(\texttt{key}, *) \to \texttt{key} & \texttt{[Obj-E]} & \Delta(\}, \varepsilon : \{) \to \varepsilon \\
\texttt{[Val-Obj-E]} & \Delta(\}, \texttt{key:}\{) \to \varepsilon & \texttt{[Elt-Obj-E]} & \Delta(\}, \texttt{[:}\{) \to \texttt{[} \\
\texttt{[Ary-E]} & \Delta(\texttt{]}, \varepsilon : \texttt{[}) \to \varepsilon & \texttt{[Val-Ary-E]} & \Delta(\texttt{]}, \texttt{key:}\texttt{[}) \to \varepsilon \\
\texttt{[Elt-Ary-E]} & \Delta(\texttt{]}, \texttt{[:}\texttt{[}) \to \texttt{[} & \texttt{[Key-Val]} & \Delta(\texttt{,}, \{) \to \{ \\
\texttt{[Elt-Pmt]} & \Delta(\texttt{prim}, \texttt{[}) \to \texttt{[} & \texttt{[Elt]} & \Delta(\texttt{,}, \texttt{[}) \to \texttt{[} \\
\texttt{[Val-Pmt]} & \Delta(\texttt{prim}, \texttt{key}) \to \varepsilon & &
\end{array}
$$

Figure 3.5: Transition Rules of Parsing Automaton.

3.3.4 Automata Hooking

The basic idea is to use the parsing automaton to drive the execution of query automaton, so that they can coordinate to accomplish the parsing and querying tasks.

Transitions Rule Embedding. When parsing automaton recognizes some JSON syntactical structures relevant to the queries (i.e., `key`, `[`, and `,`), it will feed them to the query automaton to trigger state transitions. For this purpose, we embed the transition rules of query automaton δ into three transition rules of the parsing automaton: `[Ary-S]`, `[Key]`, and `[Elt]`, which handle the alphabet of query automaton.

Query State Recording. Note that the progress of query matching may be lost when the processing moves to a lower JSON structural level. For example, after a `steps[]` array is processed, the automaton (see Figure 3.4) needs to return to the state before it encounters the `steps[]`. However, the query automaton itself is unable to memorize these older states. We solve this by introducing another stack − *query stack*. When the processing moves to a lower level (e.g., reading a `[`), the current query automaton state is pushed onto the query stack. Correspondingly, a state is popped out of the query stack and to serve as the current query state when the processing returns to a higher level of JSON structure.

[Obj-S]	$\Delta(q, \{, *, *) \rightarrow (q, *, \{)$
[Ary-S]	$\Delta(q, [, *, *) \rightarrow (\delta(q, [), q, [)$
[Key]	$\Delta(q, \texttt{key}, *, *) \rightarrow (\delta(q, \texttt{key}), q, \texttt{key})$
[Obj-E]	$\Delta(q, \}, \varepsilon, \varepsilon : \{) \rightarrow (q, \varepsilon, \varepsilon)$
[Val-Obj-E]	$\Delta(q, \}, q', \texttt{key:}\{) \rightarrow (q', \varepsilon, \varepsilon)$
[Elt-Obj-E]	$\Delta(q, \}, *, \texttt{:}\{) \rightarrow (q, *, [)$
[Ary-E]	$\Delta(q,], q', \varepsilon : [) \rightarrow (q', \varepsilon, \varepsilon)$
[Val-Ary-E]	$\Delta(q,], q' : *, \texttt{key:}[) \rightarrow (q', \varepsilon, \varepsilon)$
[Elt-Ary-E]	$\Delta(q,], q', \texttt{:}[) \rightarrow (q', \varepsilon, [)$
[Val-Pmt]	$\Delta(q, \texttt{primitive}, q', \texttt{key}) \rightarrow (q', \varepsilon, \varepsilon)$
[Elt-Pmt]	$\Delta(q, \texttt{primitive}, *, [) \rightarrow (q, *, [)$
[Key-Val]	$\Delta(q, \texttt{,}, *, \{) \rightarrow (q, *, \{)$
[Elt]	$\Delta(q, \texttt{,}, *, [) \rightarrow (\delta(q, \texttt{,}), *, [)$

Figure 3.6: Transition Rules of Streaming Automaton.

In this way, the parsing automaton coordinates with the query automaton, executing like a single automaton. Here, we refer to the hooked automaton as *streaming automaton*. Formally, a streaming automaton is an 8-tuple:

$$M_{streaming} = (\Sigma, \Gamma_q, \Gamma_s, \Delta, \delta, Q, s_0, F) \tag{3.3}$$

where the input alphabet Σ and syntax stack alphabet Γ_s are from the parsing automaton, the query stack alphabet Γ_q, along with Q, s_0, and F are from the query automaton, and Δ is the transition rule set, embedded with the ones δ from the query automaton. Figure 3.6 lists the rules in the format:

$$\Delta(s, c, qs, ss) \rightarrow (s', qs', ss') \tag{3.4}$$

saying when the automaton is in state s, after reading the symbol c, it will transition to state s', meanwhile, the query and syntax stacks qs and ss will be changed to qs' and ss' with pop/push operations, respectively.

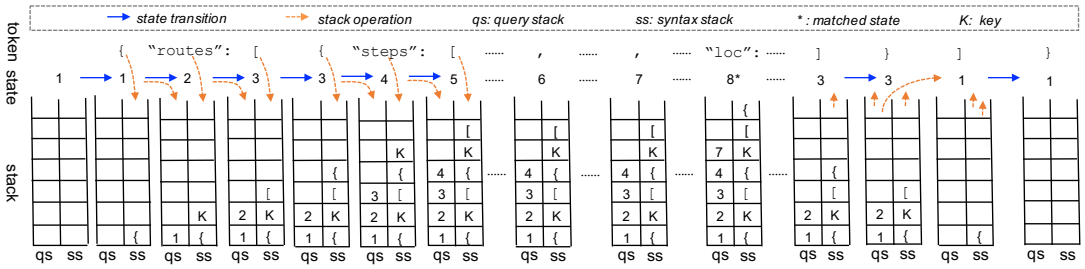

Figure 3.7: An Example of Streaming Automaton Execution (input and query automaton are from Figures 3.1 and 3.4).

Example. Figure 3.7 shows a streaming automaton execution. The input is from Figure 3.1 and the query automaton is given in Figure 3.4. For space limit, some transitions are omitted.

Initially, the automaton is in State 1 with both stacks empty. After reading the first symbol {, rule `[Obj-S]` pushes { onto the syntax stack without changing the state. For the next symbol `"routes":`, rule `[Key]` pushes `K` and State 1 onto the query and syntax stacks, respectively, then changes the state to 2 (see Figure 3.4 for state transitions). Next, after reading symbol `[`, rule `[Ary-S]` pushes the symbol and the current state to the stacks again, then updates the state to 3. By following the transition rules, this automaton processes the remaining input symbols in a similar way. Once it reaches state 8, an accept state, a match to the query is found. Finally, after finishing the last symbol }, the automaton halts with the query and syntax stacks both empty again.

However, the execution of a streaming automaton remains sequential. Next, we explain the ways to parallelize it.

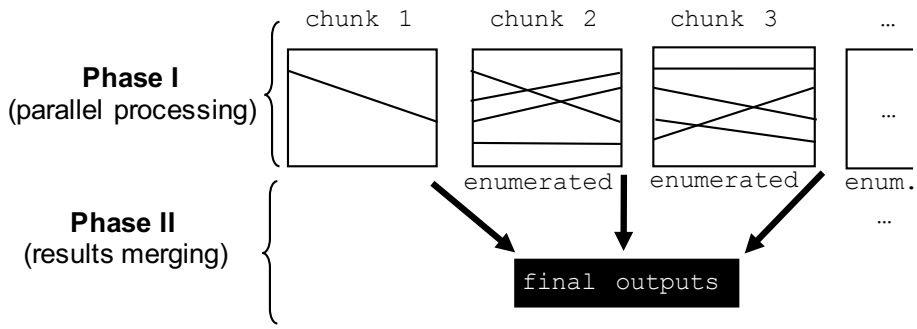

Figure 3.8: Two-Phase Parallelization.

3.4 Parallelizing Compilation

At a high level, the parallelization of a streaming automaton execution follows two phases: (i) parallel processing and (ii) results merging, as illustrated in Figure 3.8.

Basically, the data stream is first partitioned into chunks (almost) evenly [5], with each chunk processed by a thread (running a streaming automaton). After all threads finish, the results are merged to form the final outputs. The main challenge comes from the data dependences across the partitioned chunks where some JSON structures (e.g., a JSON object or array) might be broken into different chunks. In the following, we will discuss the techniques for breaking the dependences, analyze their costs, and more importantly, reduce the costs.

3.4.1 Breaking Dependences

As Figure 3.7 shows, the execution of a streaming automaton involves a series of transitions, each of which depends on the prior state and stacks. This can be symboli-

[5]The lexer is adjusted to avoid breaking a token.

cally reflected by the transition rule structure in Equation 3.4. Formally, we define *state dependence* and *stack dependence* as follows:

Definition 4 *In a streaming automaton execution, suppose T_i and T_j $(i < j)$ are two arbitrary transitions. Based on the transition rule structure in Equation 3.4, T_i writes to state s before T_j reads from it, thus indicating a true dependence (i.e., read-after-write) between the two transitions. We refer to this dependence as* ***state dependence****. Similarly, there also exist true dependences on the two stacks qs and ss between the two transitions, which we refer to as* ***query stack dependence*** *and* ***syntax stack dependence****, respectively.*

These three kinds of dependences serialize the execution of a streaming automaton from the beginning to the end. Next, we show how these dependences can be broken by leveraging some "enumerability" properties of automata.

- ***Breaking State Dependences***. Based on the definitions, the number of states in a streaming automaton is finite. Thus, we can enumerate all the states when the state is unknown. For example, by enumerating all the 9 states in Figure 3.4, we ensure the correct state is always covered.

- ***Breaking Stack Dependences***. Unlike state dependences, it is much harder to break the stack dependences because the stack can grow arbitrarily deep. Fortunately, we do not have to know all the contents of the stacks at once. According to the transition rules (see Figure 3.6), at most the top two elements of the stacks are accessed when the automaton reads an input symbol. This property allows us to *gradually* enumerate the stack contents *on-demand*. In specific, when a transition rule needs to access the stack(s)

while it is empty (due to the partitioning), we enumerate all the symbols in the stack alphabet (i.e., Γ_q or Γ_s).

For each enumerated case, we need to fork a new execution path. That is, enumeration may cause an execution path to *diverge* into multiple ones. If path divergence occurs frequently, the path maintaining overhead may outweigh parallelization benefits. We next show how fast the number of paths can grow.

Path Complexity Analysis. Suppose thread t_i is assigned to process chunk i and the number of paths is denoted as P.

At the beginning of chunk i, without knowing the state, thread t_i enumerates all the states. Thus the number of paths becomes $P = |Q|$, where Q is the state set. Then, thread t_i reads the first symbol c. Based on the transition rules (see Figure 3.6), if c is {, `[`, or `key` (i.e., the first three rules), then all the execution paths would proceed as normal. If c is }, then there are three potential rules (i.e., the 4th-6th rules) applicable. Which one is the actual depends on the stacks. Unfortunately, the stacks are all empty at this moment. Thus, thread t_i needs to enumerate all the three cases, with the assumptions that corresponding elements are in the stacks. Furthermore, the 5th rule `[Val-Obj-E]` needs to reset the current state with the top element of the query stack. Since the query stack is empty, thread t_i has to enumerate the query stack alphabet Γ_q (i.e., $|\Gamma_q|$ paths). Putting it together, when c is }, each of the existing path diverges into $|\Gamma_q| + 2$ paths. Thus, we have $P = |Q| \cdot (|\Gamma_q| + 2)$. Similarly, we can analyze the path numbers when c equals to other input symbols. In fact, the worst case happens to the symbol], where the number of paths $P = |Q| \cdot 3|\Gamma_q|$, because the query stack is accessed in all the three potential rules. If this

worst situation keeps happening in the following processing of this chunk, the number of paths would become $P = |Q| \cdot (3|\Gamma_q|)^k$, where k is the number of unmatched `]`. As $\Gamma_q = Q$, we have the following conclusion:

Theorem 5 *The worst-case path complexity in a parallel streaming automaton execution is:*

$$O(3^k \cdot |Q| \cdot |\Gamma_q|^k) = O(3^k \cdot |Q|^{k+1}) \tag{3.5}$$

where Q and Γ_q are the input and query stack alphabets, and k is the number of unmatched `]`.

There are two factors in Equation 3.5: $O(3^k)$ and $O(|Q|^{k+1})$. Both are caused by data partitioning. The first factor is due to the syntactical uncertainty (needed elements missing in the syntax stack) and the second factor is due to the state uncertainty (needed elements missing in the query stack). Hereinafter, we refer to the two uncertainties as *syntactical complexity* and *state complexity*, respectively.

Both complexities can cause the number of execution paths to increase exponentially, which may cause the path maintenance costs to surpass the parallelization benefits and even run out of the main memory for a relative small piece of chunk. We refer to this issue as *path explosion problem.*

3.4.2 Feasible Paths Inference

In this section, we discuss a couple of solutions that could help alleviate the path explosion problem by addressing the state and syntactical complexities, respectively.

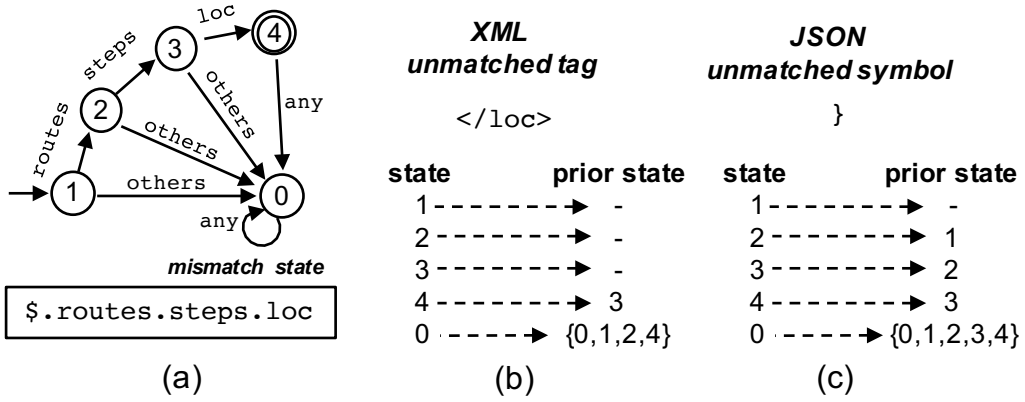

Figure 3.9: Query Stack Feasibility Inference: XML vs. JSON

Query Stack Feasibility Inference. Recent work [106, 79] has shown the possibility to leverage the state transitions of query automaton to infer the (top) elements in the query stack, for parallelizing XML's automaton execution.

The basic idea of the inference is illustrated in Figures 3.9-(a) and (b) using a simple query. When an unmatched end tag is met (e.g., `</loc>`), we can infer the top element in the query stack for each enumerated state. In fact, the top element is the state before meeting the open tag `<loc>`. Based on the structure of the query automaton, we can find all the possible states that transition to an enumerated state after reading `<loc>`. For example, only State 3 transitions to State 4 after reading `<loc>`. After the inference, we can find there are five execution paths left − the same as the number of states. In fact, it is provable that the number of paths is bounded by a constant (i.e., $|Q|$) with the query stack feasibility inference.

However, this approach fails to work effectively for the parallelization of JSON data processing for two reasons:

- ***Syntactical Complexity***. First, in the case of JSON, there are two complexities causing exponential path growth: state complexity and syntactical complexity. Query stack feasibility inference might help address the first, but not the second, which is not directly relevant to the states. Thus, the path explosion problem remains unsolved.

- ***Anonymous Ending Symbol***. Even for state complexity, the above inference cannot reduce the complexity to the linear level as in the case of XML. Because, unlike XML where an end tag (e.g., `</loc>`) carries the tag name, a JSON ending symbol (e.g., }) is anonymous, as shown in Figure 3.9-(c). Without the key name, there would be more states possibly serving as the prior state. For example, State 2 could transition to State 3 and State 1 could transition to State 2 (see Figure 3.9-(c)). Despite they correspond to different input symbols, they all have to be considered to ensure the correctness. Consequently, the number of execution paths increases from 5 to 8. This increase could be accumulated as more unmatched symbols are met.

In sum, due to the special features of JSON, query stack feasibility inference itself is insufficient for reducing the costs of parallelization. Next, we present a JSON-customized feasibility inference for the syntactical complexity.

Syntax Stack Feasibility Inference. With the partitioning, the syntax stack may lack needed element(s) to determine the next transition rule after reading a symbol. To address this syntactical uncertainty, our idea is *to infer the syntax stack by looking ahead more input symbols*. In fact, we have:

Theorem 6 *By looking forward at most two input symbols, the transition rule of a streaming automaton can be uniquely determined without the knowledge of the syntax stack.*

Table 3.1: Syntax Stack Feasibility Inference (* means any symbol).

Input	Following Two Symbols	Syntax Stack	Rule
}	}* or ,key	*:key:{	[Val-Obj-E]
}	,[or ,{ or]* or ,prim	*:[:{	[Elt-Obj-E]
}	ε or {*	ε:{	[Obj-E]
]	}* or ,key	*:key:[	[Val-Ary-E]
]	,[or ,{ or]* or ,prim	*:[:[	[Elt-Ary-E]
]	ε or [*	ε:[	[Ary-E]
prim	}* or ,key	*:key	[Val-Pmt]
prim	,[or ,{ or]* or ,prim	*:[	[Elt-Pmt]
,	key*	*:{	[Key-Val]
,	[* or {* or prim*	*:[	[Elt]

Theorem 6 can be proved with a brute-force enumeration. For each symbol that may cause syntactical uncertainties (4 such symbols), we enumerate all the possible combinations of its following two symbols, including the empty symbol ε ($7*7+7+1 = 57$ cases). Thus, there are $4*57 = 228$ cases overall. For each case, we run the streaming automaton on the three symbols. If the automaton halts before finishing, the case is infeasible; Otherwise, we record the correspondence between the input symbols and the first transition rule to the table. By exhaustively searching the correspondence space, we ensure the coverage of all the feasible situations.

Table 3.1 shows our findings. The first two columns list the input symbols that may cause syntactical uncertainty and the potential following two symbols, respectively. The last two columns show the contents that should be in the syntax stack (instead of being empty) and the corresponding transition rule that should be applied. With this technique, we reduce the syntactical complexity from $O(3^k)$ to $O(1)$.

3.4.3 Towards a Bounded Number of Paths

Though the syntactical complexity has been addressed in Section 3.4.2, the number of paths may still grow exponentially due to the state complexity, for which, the existing query stack feasibility inference fails due to the anonymous ending symbol property of JSON (Section 3.4.2). In this section, we discuss a new strategy to bound the number of paths.

On-demand Automata Resetting. The basic strategy is to reset a parallel streaming automaton once it fails to uniquely determine the next possible transition.

Recall the reason that an enumerated execution path may diverge is because the lack of information from the query and/or syntax stack(s) when reading certain input symbol. We refer to such a symbol as a *unmatched symbol*. When an unmatched symbol is met, we can apply the query and syntax stack feasibility inferences from Section 3.4.2. If they fail to uniquely determine the next transition, we will first skip the symbol, then reset the streaming automaton, which includes re-enumerating all the states as the current and emptying the stacks, just like processing a fresh new chunk. This strategy may leave some unmatched symbols unprocessed, for which the processing is postponed to the results merging phase. We refer to this scheme as *on-demand automata resetting*. As an execution path never diverge, on-demand automata resetting bounds the number of execution paths by the number of states $|Q|$, that is, $P = O(|Q|)$.

Data Units. Essentially, automata resetting further breaks the input chunks into even smaller pieces. To distinguish them from the original partitioning, we refer to these smaller pieces within a chunk as *data units*.

During the merging phase, the results of data units will be merged, with the unmatched symbols in-between processed. This may add extra costs to the merging phase, depending on the number of data units. According to our evaluation, the number of data units in chunk is quite manageable, resulting in only marginal merging costs.

3.4.4 Results Merging

First, during the parallel processing, for each data unit i, there is a mapping maintained:

$$\mathcal{M}_i(M_j^s) = M_j^e \tag{3.6}$$

where M_j^s is a possible starting configuration of unit i's streaming automaton and M_j^e is the corresponding ending configuration. The temporarily matched results are recorded separately for each stating configuration. After the parallel processing, the correct starting configuration of each unit can be identified one by one. For example, with the (always correct) ending configuration of the first data unit, a (serial) streaming automaton can further process the unmatched symbols between the first and second data units (if any). After that, the configuration of the automaton would be the correct starting configuration for the second data unit. In this way, we merge all the data units (along with unmatched symbols). During the units merging, the matched results on the correct path are separated as the actual matches of subqueries. Finally, the matches of subqueries are merged and filtered to produce the final outputs (see Section 3.3.2).

Input Errors. Sometimes, a JSON data stream may contain syntactical errors, which may alter the transitions of parallel streaming automata. However, since the merging of results

are performed sequentially, we can still ensure the detections of all syntactical errors that can be detected in serial streaming automaton during the merging phase.

3.5 Runtime Optimization

The prior section leverages the transition rules and local input symbols to reduce the number of enumerated paths. In this section, we exploit the data properties to further prune paths (extending GAP to JSONPath querying) [6]. The intuition is that the elements in JSON data streams often follow some patterns. For instance, in Figure 3.1, `steps` is always under `routes`, while `loc` could be under `routes` or `steps`. We refer to these casual relations among JSON objects and arrays as *data constraints*.

Knowing these data constraints might help further prune the execution paths. Consider the serial automaton execution in Figure 3.7. We can record the correspondence between the current state and its encountered input symbol. As shown in Table 3.2, there are only a subset of states actually happened to encounter a specific `key` or a `[`. For example, the second row says when the automaton meets `steps`, it is always at State 3. By feeding such information to the query stack feasibility inference, the enumerated paths could be further pruned. Note that, only `key` and `[` appear in Table 3.2, because, with automata resetting, the path enumeration can only happen before `key`, `[`, or {, so other symbols are not recorded. Symbol { is removed from the table as every state might encounter it, failing to provide any useful data constraints.

[6] Similar properties are also exploited in XML Data [79].

Table 3.2: Example Data Constraints Hash Table

Input Symbol	Feasible States (Paths)
`"routes":`	{1}
`"steps":`	{3}
`"loc":`	{5,6,7}
`"lat":`	{0}
`"lng":`	{0}
`[`	{2,4}

However, this observation-based optimization is not safe when the actual input does not follow the same "pattern" as the one used for collecting the data constraints. As a result, the correct path may be pruned. For example, suppose in the actual input, `lat` and `lng` appear under the `routes`, then using the feasible State 0 in Table 3.2 could be incorrect. Therefore, to ensure the correctness, we also need some correctness checking and reprocessing mechanisms.

Based on the above discussion, we provide JPStream an offline data constraint learner and an online component for constraint integration. The learner takes training inputs from the users to build a symbol-state correspondences, which is then exported as a hash table. The integration part tracks the uses of query stack feasibility inference and the resetting events during a parallel automaton execution. Once observed, the integration component uses the current symbol to query the hash table to get the feasible states. Then, it takes an intersection between the returned states and existing state set. Later, during the results merging phase, the integration component verifies if the correct state was covered. If not, it will reprocess the incorrectly interpreted unit, but never go beyond a unit, thanks to the automaton resetting.

3.6 Evaluation

This section evaluates the automata generated by JPStream.

3.6.1 Implementation

We prototyped JPStream in C language and used Pthread for the parallel implementation of streaming automata. JPStream supports the standard JSONPath queries [74].

Given a JSONPath query, JPStream first parses it into an abstract syntax tree (AST) that facilitates the generation of the query automaton. For less number of states, JPStream uses counters for handling array index constraints instead of state transitions (see Section 3.3). JPStream then hooks the query automaton to the pre-compiled parsing automaton to form a serial streaming automaton. For the parallelizing compilation, JPStream adopts a double-tree data structure to reduce the path maintenance costs [106, 79]. It implements the syntax stack feasibility inference (Section 3.4.2) and supports the on-demand automata resetting scheme (Section 3.4.3). In addition, JPStream also features an offline data constraint learner that takes additional training inputs to learn the data constraints, and a runtime integration module to apply the constraints (Section 3.5). To provide a bounded the memory footprint, JPStream offers a threshold limiting maximum size of data loaded each time. When a data stream is larger than the threshold, JPStream performs multiple load-process cycles. The default value of this threshold is set to 250MB.

3.6.2 Methodology

We compare the automata generated by JPStream with the state-of-the-art methods for parallel semi-structured data processing and also a group of state-of-the-practice JSON tools, in terms of performance and memory usage.

Table 3.3: Methods in Evaluation.

Abbr.	Methods
XMLStream	An adoption from parallel XML processor [106, 79]
JsonSurfer	A manually crafted tool for streaming JsonPath [21]
JSON-C	An open-source JSON parser in C [12]
Jackson-JSON	An open-source JSON parser in Java [15]
RapidJSON	A JSON parser in C++ from Tencent [17]
FastJSON	A parsing-based JSON tool in Java from Alibaba [4]
JPStreamNR	Non-restartable JPStream automata
JPStreamR	Restartable JPStream automata
JPStreamR+	JPStreamR with data constraints learning

Methods. Table 3.3 lists the methods used in our evaluation. XMLStream [106, 79] is a parallelization solution recently developed for XML data processing. Note that this solution cannot be directly applied to JSON data due to the syntax differences (see Section 3.2). Here, we ported their ideas to the JSON data processing. JsonSurfer [21], as far as we know, is the only stream processing implementation for JSONPath querying. Note that it is not based on automata. As it is Java implemented, for a fair comparison with our C-based automata, we also rewrote JsonSurfer in C. For parsing-based JSON processing, many tools exist. We pick JSON-C [12], Jackson-JSON [15], RapidJSON [17], and FastJSON [4] for their popularities and industrial supports. For Jackson-JSON and FastJSON, since they are implemented in Java, we first warmed up the JVM before measuring their performance,

Table 3.4: JSONPath queries. #sub shows the number of sub-queries in each query structure.

Query	Data	Query structure	#sub	#matches
BB1	BB	$.pd.shl[1:5].sel[1:2]	1	95
BB3	BB	$..p..st	1	0
TT1	TT	$[*].ur..is	1	312,460
TT2	TT	$[*].qs.en.um.[*].nm	1	9,580
NSPL2	NSPL	$.dt[20:200:2][:]	1	3,913
UHD1	UHD	$.dt[10:1000][:]	1	19,820
BB2	BB	$.pd[?(@.s || @.pid || @.nm || @.sr || @.tp || @.sd || @.nw)].cp[1:2].id	9	459,333
TT3	TT	$[*].qs.*.me[?(@.id && @.ids && @.idc && @.mu && @.mh && @.ud && @.url && (@.eu || @.tp))].sz.lg.w	9	32,750
NSPL1	NSPL	$.mt.*.co[?(@.id || @.fg || @.dtn || @.cc.lg)]..nm	6	44
AIL1	AIL	$.fe[?(@.tp && (@.id || @.gn)].pp.nm	5	21,546
AIL2	AIL	$.fe[1:7].fm.cd[:][:][10:20][:]	1	2,088
GMD1	GMD	$[-15:].rt[?@.cr].bd.*.lt	4	28
GMD2	GMD	$[*].rt[?(@.bd || @.cr].le[?@.sa].st[0:3].dt.vl	6	17,147
UHD2	UHD	$.mt.vi.co[?(@.id && @.nm && @.dtn)].cc.tp[1:120].it	6	169

for their best results. As to our methods, we evaluated three versions of JPStream: (i) the one with automata resetting disabled; (ii) the one with automata resetting; and (iii) the one with both automata resetting and data constraints learning.

Table 3.5: Dataset Statistics.

Data	#objects	#arrays	#key-value	#primitive	depth
BB	1,916,574	4,882,921	40,763,630	35,880,709	7
TT	2,898,910	4,197,990	31,472,660	30,910,730	10
NSPL	613	3,509,815	1,669	84,235,938	9
AIL	64,675	46,858,734	905,058	93,395,898	3
GMD	10,357,445	43,943	29,034,886	21,051,619	9
UHD	262	1,511,234	719	30,224,745	8

Platforms. All experiments run on two servers. The first one is a 16-core machine equipped with two Intel 2.10 GHz Xeon EE5-2620 v4 processors and 64 GB of RAM. The second

server is a 64-core machine equipped with Intel Xeon Phi 7210 processors and 96 GB of RAM. Both servers run on CentOS 7 and are installed with GCC 4.8.5. In the following, we refer to the two servers as *Xeon* and *Xeon-Phi*. All programs are compiled with `O3` optimization. The timing results reported are the average of 10 repetitive runs. All CPUs are warmed up before evaluation. We do not report 95% confidence interval of the average when the variation is not significant.

Datasets. Table 3.5 summarizes the statistics of JSON datasets used in our evaluation, including Best Buy product dataset [2], Twitter global tweet stream [22], National Statistics Post Code Lookup (NSPL) dataset for the United Kingdom [16], Indigenous Land Use Agreements dataset [9], Google Maps Directions dataset [6], and Washington State Department of Health dataset [7]. The training datasets for JPStreamR+ are extracted from the first 1% of the testing inputs.

Query Sets. Table 3.4 shows our evaluated queries. It covers all the basic types of JSONPath queries that are commonly used [74]. The rightmost two columns record the number of sub queries and the number of matches in each query structure.

3.6.3 Comparison with Existing Methods

Note that, due to the inherent dependences (see Section 3.2), all these methods only support sequential execution, except for XMLStream. These experiments were executed on Xeon.

Comparison with Streaming Methods. Figure 3.10 reports the execution time of stream processing methods. First, we notice XMLStream runs out of memory for all the queries.

Because it does not address the syntactical complexity (see Section 3.4.2), leading to exponential path growth that quickly consumes all the machine memory (64GB).

JsonSurfer successfully processed all the queries, but its performance is limited by the serial execution. Single-thread JPStreamR outperforms JsonSurfer, thanks to its use of a query automaton. By contrast, JsonSurfer uses the parsing stack to match queries, which may examine deeper in the stack for a complete comparison. This is especially inefficient for queries with .. wildcard, such as TT1 and NSPL2. As indicated by Figure 3.10 , the gaps in these two cases are even larger. On top of that, when using all the 16 cores, JPStreamR outperforms JsonSurfer by an order of magnitude.

Comparison with Parsing-based Methods. In Figure 3.11, JPStreamR is compared against four parsing-based methods. TT(20) and BB(20) are two query sets, each with 20 queries. For JSON tools without querying supports, we simply report their parsing time. Most parsers finish the parsing of our datasets within 10 secs, except for JSON-C (took over 30 secs). In comparison, single-threaded JPStreamR spent a little more than 10 secs, comparable to other JSON tools. On top of that, JPStreamR also finishes the querying at the same time, while FastJSON needs two separate phases. The total processing time of FastJSON surpasses JPStreamR in our evaluated cases. When all the 16 threads are used, JPStreamR outperforms all the evaluated parsing-based methods significantly.

Besides performance, JPStreamR also shows advantages in the memory footprint, as indicated in Figure 3.12. Thanks to its stream processing model, JPStreamR saves up

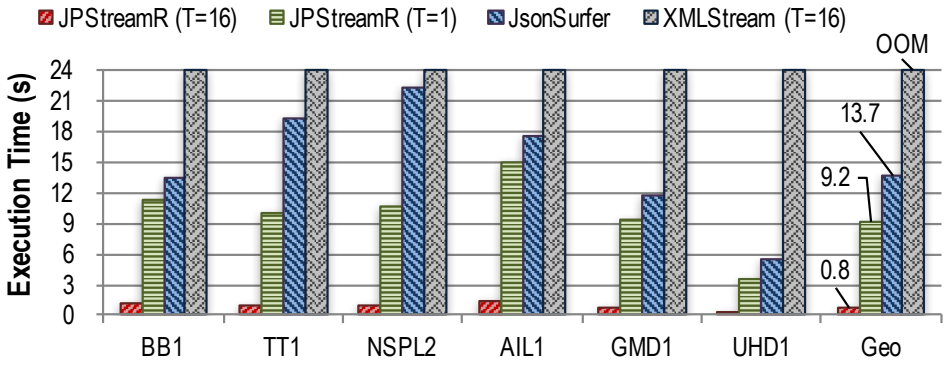

Figure 3.10: Performance of Streaming-based Methods*.

*The original JsonSurfer only supports a subset of queries. For the supported ones, its performance is 1-1.45X faster than JPStreamR (T=1), after the JVM warmup.

to 95% of memory consumption. In fact, unlike the other JSON tools that consume more memory for larger inputs, the memory footprint of JPStreamR is bounded.

3.6.4 Performance of Parallel Execution

Since the only other parallel method, XMLStream, runs out of memory in all the tested cases, we mainly focus on the three versions of JPStream-generated automata in this section.

Speedup. Figure 3.13 reports the speedups of parallel automata executions over the serial one. Among the three versions, JPStreamR+ yields the best speedup for all evaluated queries, thanks to its data constraint learning. On average, it achieves 12.1X on 16-core Xeon and 53.1X on 64-core Xeon Phi. For the other two versions, JPStreamR performs better in general, for its on-demand automata resetting scheme.

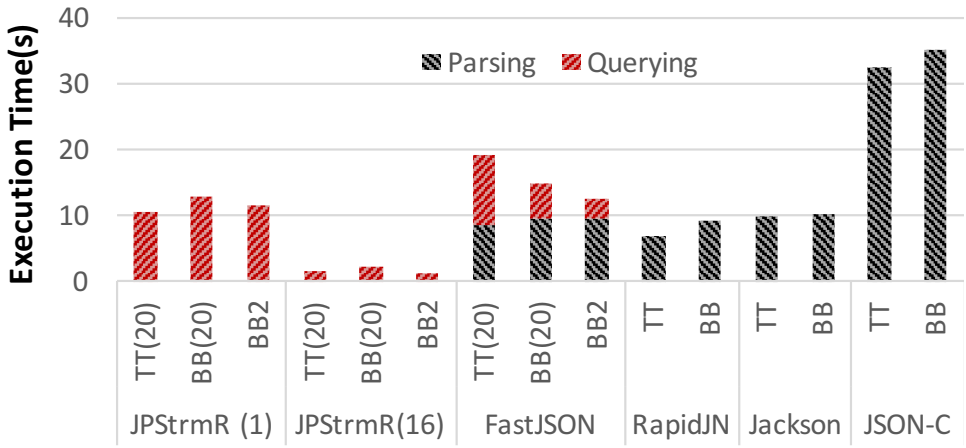

Figure 3.11: Comparison with Parsing-based Methods (Time).

To get deeper insights, we also examined the number of execution paths, the costs of results merging, and the costs of using the data constraint integration.

Number of Execution Paths. The more paths a parallel streaming automaton needs to maintain, the less benefits it brings. Table 3.6 reports the statistics about the number of execution paths. The "avg." columns show the average number of paths to maintain by a streaming automaton. As the last row (i.e., "Geo") shows, JPStreamNR maintains 2.29 paths on average across all the tested cases. In comparison, JPStreamR maintains 1.88 paths and JPStreamR+ maintains only 1.15 paths. These statistics largely echo the performance differences among the three versions of JPStream. The "max" columns show the maximum number of paths to maintain. We can see JPStreamNR maintains up to 119 paths. For the same case, JPStreamR+ only maintain 5 paths, due to the path pruning enabled by data constraint integration.

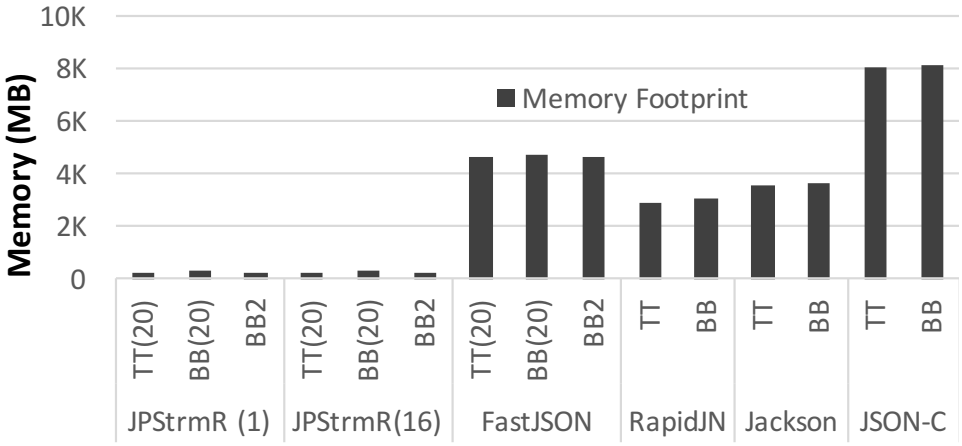

Figure 3.12: Comparison to Parsing-based (Memory).

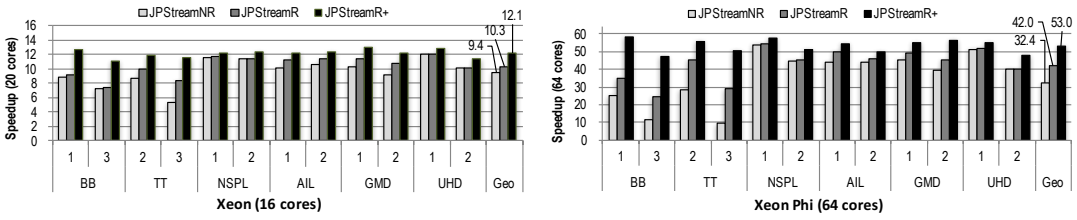

Figure 3.13: Speedups of JPStream-generated Automata on Xeon and Xeon Phi Servers.

Results Merging Costs. Table 3.7 reports the merging costs. First, the costs are consistently less than 0.001% in all the tested cases. Second, units merging does not add extra costs significantly comparing to the chunk merging, thanks to the relatively small number of units (3rd column).

Costs of Online Constraints Integration. In principle, the online data constraints integration is a double-edge sword, as it may eliminate a correct path due to the discrepancy between the training and testing inputs. For this reason, we examine the accuracy of

Table 3.6: Number of Execution Paths (Xeon Phi)

	JPStreamNR		JPStreamR		JPStreamR+	
Query	avg.	max	avg.	max	avg.	max
BB1	6.02	16.0	3.70	8.0	1.01	2.0
BB2	11.46	37.0	6.45	15.0	1.01	3.0
TT1	4.45	119.0	2.53	7.0	1.41	3.0
TT2	3.76	42.0	1.89	7.0	1.16	4.0
NSPL1	1.15	6.0	1.15	6.0	1.11	4.0
NSPL2	1.35	15.0	1.35	15.0	1.07	3.0
AIL1	1.75	25.0	1.74	10.0	1.19	3.0
AIL2	2.26	28.0	2.25	11.0	1.65	5.0
GMD1	1.07	23.0	1.06	10.0	1.01	2.0
GMD2	1.27	44.0	1.24	15.0	1.01	2.0
UHD1	1.26	6.0	1.26	6.0	1.13	3.0
UHD2	1.70	15.0	1.70	15.0	1.17	3.0
Geo	2.29	22.5	1.88	9.8	1.15	3.0

Table 3.7: Costs of Results Merging

Dataset		#Chunks	#Units	Chunk	Chunk+Unit
Xeon-Phi	BB	64	146	0.004%	0.004%
	TT	64	176	0.005%	0.006%
	NSPL	64	116	0.001%	0.001%
	AIL	64	292	0.003%	0.004%
	GMD	64	317	0.001%	0.002%
	UHD	64	64	0.002%	0.002%
	Geo	64	185	0.002%	0.003%

covering the correct execution path and the cost of reprocessing if the correct one is missed. The results are reported in Table 3.8. The queries that are not shown have 100% accuracies. The results indicate, by using the first 1% of the testing inputs as the training ones, for most test cases, JPStreamR+ successfully covered the correct paths. For the ones that it misses, the accuracy is beyond 90%, with reprocessing costs consistently less than 1%.

Table 3.8: Path Coverage Accuracy and Reprocessing Cost

Query		Acc.	Cost	Query		Acc.	Cost
Xeon	BB1	97.62%	0.13%	Xeon-Phi	BB1	95.21%	0.52%
	BB2	97.62%	0.08%		BB2	97.26%	0.26%
	TT1	92.86%	0.14%		TT1	96.59%	0.55%
	TT2	95.24%	0.13%		TT2	97.73%	0.56%
	TT3	92.86%	0.13%		TT3	95.45%	0.79%

Table 3.9: Scalability over Data Size on Xeon

Data Size	Exe. Time (1 core)	Exe. Time (16 cores)	Speedup
250 MB	3.022 s	0.245 s	12.335
1 GB	11.371 s	0.900 s	12.636
4 GB	46.573 s	3.619 s	12.868
16 GB	192.829 s	14.829 s	13.003

3.6.5 Scalability

Scalability over Number of Cores. Figure 3.14 presents the speedup curves of JPStream on Xeon-Phi. In general, all three versions exhibit linear speedup increase up to 64 cores, with different increasing rates. For JPStreamNR, the curve is less smooth, because the number of execution paths could vary significantly under different partitions. For the other two, the automaton resetting largely limits the maximal number of execution paths, yielding smoother curves.

Scalability over Input Sizes. Table 3.9 reports the speedup of JPStreamR+ with different data-size thresholds on Xeon. The results indicate that, in general, there is a tradeoff between the memory footprint and the performance gain.

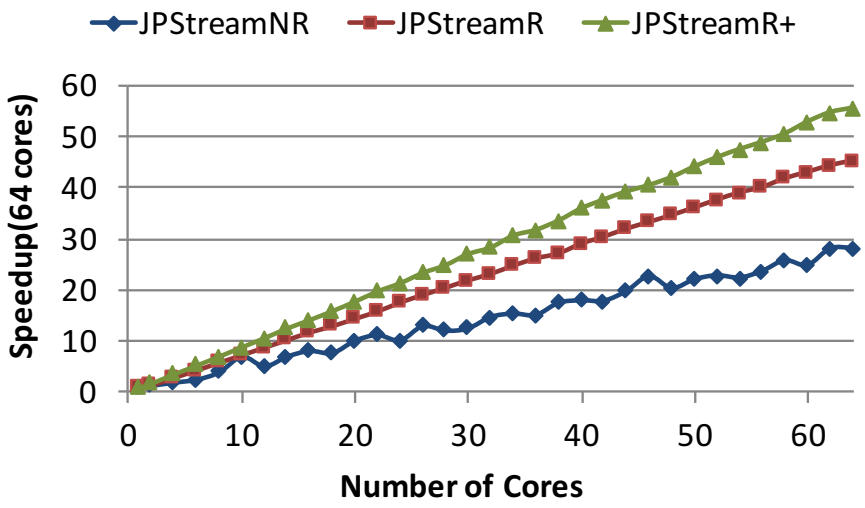

Figure 3.14: Scalability over Number of Cores on Xeon-Phi

3.7 Related Work

Semi-Structured Data Processing. A large body of prior work focuses on the expressiveness of semi-structured data querying and the efficiency of executing concurrent queries, especially for XML data. For evaluating XML path queries, representative methods include automata-based [68, 134], array-based [80], and stack-based algorithms [49]. Among them, a seminal idea by Green and others is to combine a set of XPath expressions into a deterministic finite automaton (DFA) to improve the query evaluation efficiency [68], which influences many other studies, including two widely used XML tools, Y-filter [56], XMLTK [39], as well as JPStream, though ours is for JSON data processing.

Unlike XML with well-developed streaming computation models, existing JSON tools [4, 12, 15, 17] usually rely on parsing the whole data stream to extract information. JsonSurfer [21] is the only stream processing tool for JSON that we are aware of, but

it runs less efficiently for the lack of automata. Barenghi and others [40] use operator precedence grammars (OPG) for parallel JSON parsing, without querying support. Unlike others, MISON [86] exploits bitwise parallelism with SIMD supports. It pre-constructs structural indexes for JSON data to speed up query processing. But the index construction can only execute sequentially due to dependences, thus cannot scale on multicore machines. Parabix [89] also uses bitwise parallelism to accelerate XML parsing, but not querying. Palkar and others apply filters [111] on the raw JSON byte stream before querying the data to accelerate the processing. Pavlopoulou and others [113] rewrite JSONiq queries into XQuery to leverage an existing XML framework (Apache VXQuery [1]) to support parallel processing of multiple JSON data streams, rather than a single stream.

Parallel Finite Automata. Finite automata (a.k.a. FSMs) are difficult to parallelize. A traditional parallelization is to use prefix-sum [84], which is also based state enumeration. An implementation of this method [104] has been optimized for SIMD instructions. Speculative parallelization of FSMs has also been proposed [115, 116, 136, 138] by breaking the transition dependences with state predictions. Though providing useful insights, the above work cannot be directly applied to the parallelization of semi-structured data processing, which essentially requires the use of stack-based automata.

Some recent work introduces parallel pushdown automata for processing XML data, such as PP-Transducer [106] and GAP [79]. They share some high-level ideas with JPStream. However, they suffer from the path explosion problem when ported to JSON. AT-GIS [107] is a recent work for parallel processing GeoJSON, a derivative of JSON. However, its applicability to the generic JSON remains unclear.

3.8 Summary

This work addresses the challenges in scalable JSON-family data processing with a compilation system, called JPStream. JPStream automatically generates parallel executables with low-memory footprints for JSON data processing. At its core is the design of a streaming computation model and a set of customized parallelization techniques. The model is enabled by a joint compilation idea that combines the queries and JSON syntax into a unified automaton. The parallelization leverages a syntactical feasibility inference, an on-demand automaton resetting scheme, and a data constraint learner to address the path explosion raised by the unique features of JSON. Finally, evaluation confirms the effectiveness of JPStream in generating highly scalable automata executables with bounded memory footprints, showing superiority over the state-of-the-art methods and existing JSON solutions in terms of performance and memory consumption.

Chapter 4

Scalable Bitwise Index Construction for JSON Analytics

JavaScript Object Notation (JSON) and its variants have gained great popularity in recent years. Unfortunately, the performance of their analytics is often dragged down by the expensive JSON parsing. To address this, recent work has shown that building bitwise indices on JSON data, called structural indices, can greatly accelerate querying. Despite its promise, the existing structural index construction does not scale well as records become larger and more complex, due to its (inherently) sequential construction process and the involvement of costly memory copies that grow as the nesting level increases.

To address the above issues, this work introduces Pison – a more memory-efficient structural index constructor with supports of intra-record parallelism. First, Pison features a redesign of the bottleneck step in the existing solution. The new design is not only simpler but more memory-efficient. More importantly, Pison is able to build structural

indices for a single bulky record in parallel, enabled by a group of customized parallelization techniques. Finally, Pison is also optimized for better data locality, which is especially critical in the scenario of bulky record processing. Our evaluation using real-world JSON datasets shows that Pison achieves 9.8X speedup (on average) over the existing structural index construction solution for bulky records and 4.6X speedup (on average) of end-to-end performance (indexing plus querying) over a state-of-the-art SIMD-based JSON parser on a 16-core machine.

4.1 Introduction

JSON (JavaScript Object Notation) has emerged as a popular data type in modern software applications [74]. Its derivatives, such as NetJSON [70], GeoJSON [69], JSON-LD [71], CoverageJSON [60], and others, span multiple domains. Together, they play critical roles in microservices [130, 58], Internet of Things (IoT) [132], NoSQL data stores [87, 72], and cloud computing [47, 91]. As the popularity of JSON increases, its data volume grows faster than ever. Many web applications, such as social networks (e.g., Twitter [24] and Facebook [25]) and online shopping (e.g., Bestbuy [2] and Walmart [26]) continuously produce a broad range of data in JSON format through open APIs. Public data sources, like Data.gov [8], host more datasets in JSON format of sizes easily reaching several gigabytes. However, analyzing JSON data requires parsing – an expensive task. Recent study [86] shows that JSON data parsing takes over 80% of the total time for complex queries and even a larger portion for simple queries. Hence, it is critical to accelerate the parsing in order to make the querying over JSON data performant.

State of The Art. Traditional ways of JSON parsing involve stack-based abstract machines, known as *pushdown automata*. Basically, an automaton traverses a JSON record in serial and recognizes the nested syntactical structures with a stack. Most popular JSON parsers, like Jackson [15] and GSON [11], fall into this category. Recent work JPStream [78] proposes a dual-stack automaton to carry out path queries simultaneously with the parsing. However, an inherent limitation with the automata-based solutions is that they have to traverse the JSON data stream *character by character* to perform the parsing and query matching.

In fact, it is possible to "skip" irrelevant parts of the data stream with the help of indexing techniques. Recently, Mison [86] proposes a novel algorithm that generates bitwise indices (bitmaps) for the structural characters in a JSON record (i.e., ":" and ",") – *structural indices*. With the indices, a parser can directly jump into relevant positions of the JSON record to find matches. As the construction can leverage bitwise and SIMD-level parallelism, it can process tens of or even hundreds of characters simultaneously [86].

Despite its promise, there are several issues in the existing design of structural index construction that may limit its scalability as the records become larger and more complex. First, most steps in the structural index construction are inherently sequential, preventing it from taking advantage of the coarse-grained parallelism. For bulky records, lack of intra-record parallelism fundamentally limits the efficiency of structural index construction. Second, one critical step in the current design of structural index construction [86] involves many costly memory operations, which gets worsen as the record becomes larger and more deeply nested. At last, the existing design assumes the record can fit into the

caches, which is not the case for bulky records – naively using the current design to process bulky records may suffer from poor data locality.

Overview of This Work. The primary goal of this work is to scale the structural index construction to larger and more complex JSON records. To achieve this, we identify all the dependences involved in each step of the index construction, then develop specialized parallelization solutions to "break" those dependences. Specifically, we address the dependences related to backlash sequences with *dynamic partitioning*; However, partitioning may still cut the JSON strings and the nested structures of a JSON record. To handle broken JSON strings, we propose a combination of techniques, including a *contradiction-based context inference*, a *speculation technique* with fast *bitwise reprocessing*. Finally, to handle broken nested JSON structures, we leverage a *reduction-based* parallelization technique.

Besides parallelization, we also identify the inefficiencies in the bottleneck step of the construction and propose a new design which is not only easier to implement but also more efficient to execute, thanks to its reduced memory accesses. Our evaluation shows the new design brings 1.4X speedup to the total serial execution time.

Moreover, to cope with the large working set in processing bulky records, we propose to build structural indices word by word, rather than step by step for the whole record. This locality optimization itself brings 1.7X speedup on average for bulky records.

Finally, we integrated the above techniques and developed a new structural index constructor called Pison. To ease the programming, Pison provides intuitive APIs that hide the details of index traversals. We evaluated Pison using a group of real-world JSON datasets with a focus on the bulky-record processing scenario. According to the results,

Pison outperforms the existing structural index construction solution Mison [86] by 9.8X (on average) for bulky records, and achieves 4.9X speedup (on average) of end-to-end performance over the popular SIMD-based JSON parser simdjson[85]. The results confirm the effectiveness of the proposed techniques.

Contributions. This work makes three main contributions:

- First, it presents a group of parallelization techniques that enable intra-record data parallelism to the JSON structural index construction.
- Second, it proposes two key optimizations to the serial design of structural index construction: a redesign of the bottleneck step and a locality optimization.
- Finally, it implements the proposed ideas into a C++ library (Pison) and compares it with multiple state-of-the-art JSON tools using real-world datasets. The source code of Pison is available at `https://github.com/AutomataLab/Pison`.

4.2 Background

This section introduces JSON basics, followed by an overview of the current structural index construction and its limitations.

4.2.1 JSON and Its Querying

JSON Syntax. JSON data follows a rigorous syntax, which can be defined by a context-free grammar (CFG), as shown in Figure 4.1. At high level, there are two major structures: object and array. An object always starts with a left brace { and ends with a right brace }.

```
  object ::= {} | {members}
 members ::= pair | pair, members
    pair ::= string:value
   array ::= [] | [elements]
elements ::= value | value, elements
   value ::= object | array |
             string | primitive
  string ::= "" | "characters"
```

(a) BNF Grammar of JSON

```
1 {"user": [ {
2     "id": 1,
3     "name": "\\\":A" },
4    {"id": 2 }
5   ]
6 }
```

(b) Sample JSON Data (Twitter)

Figure 4.1: JSON Grammar and Example.

Between them, there could be a series of key-value pairs, separated by commas, known as the attributes. By contrast, an array is placed between a pair of brackets `[` and `]`. Inside an `array`, there is a linear sequence of elements, separated by commas. Both object and array can be self-nested and also nested within each other. Here, we use the term "record" to refer to the top-level syntax structure in the nested JSON data, which could be either an object or an array.

Querying JSON Data. Essentially, each JSON record is a serialized hierarchy of an object or an array. A basic group of queries to JSON data is to identify the sub-structures

of interest inside the JSON hierarchy, known as *JSONPath queries* [74]. A JSONPath query defines specific paths from the root of the hierarchy to the sub-structures of interest. For example, `$.user[0].id` asks for the `id` of `user[0]`, the first element of array `user`. A dot `.` refers to the attributes of an object; two dots (like `$..id`) refers to all recursive descendant objects. TFor all elements in an array, use star `*`, as in `$.user[*].id`. More details regarding the JSONPath queries can be found in [74].

Conventionally, evaluating JSON queries requires parsing a JSON record into a tree structure. Queries are evaluated by walking down the tree from the root and matching tree nodes against the queries. Many popular JSON tools follow this approach, like Jackson [15], GSON [11], RapidJSON [18], and FastJSON [4]. Alternatively, the parsing and querying can be merged into a single pass, as shown in JPStream [78], which avoids building any parse tree. In either approach, the parsing essentially simulates a pushdown automaton that consumes the JSON data character by character (i.e., one byte per time). However, modern CPUs can perform 64-bit (i.e., 8-byte) calculation, and even 512-bit calculation (i.e., 64-byte) with SIMD units [117]. In this perspective, existing automata-based parsers underutilize the fine-grained parallelism in modern CPUs.

To effectively utilize bitwise and SIMD parallelism, recent work Mison [86] proposes a bitwise indexing scheme for JSON data. The basic idea is to construct bitwise indices on structural characters in a JSON record, such as ":" and ",". With the indices, a query can be quickly evaluated based on locations of object attributes and array elements of different levels. We next describe it.

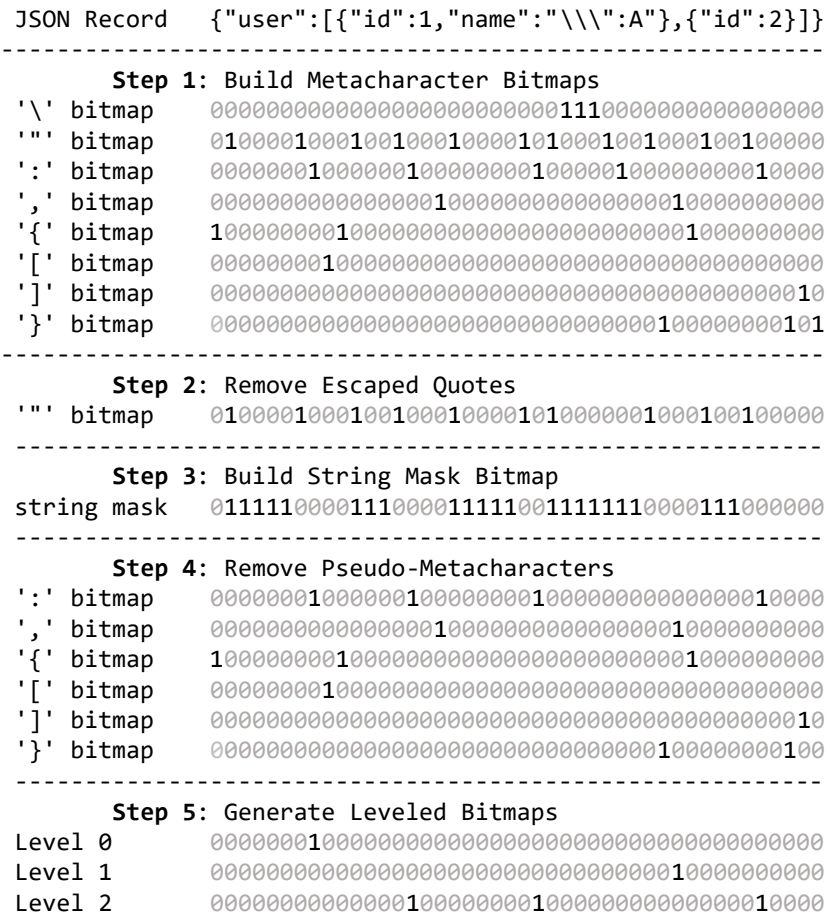

Figure 4.2: Example Structural Index Construction.

4.2.2 Structural Index Construction

This section summarizes the structural index construction proposed in Mison [86], with some steps adopted from simdjson [85]. In general, the construction has five steps [1], as illustrated in Figure 4.2.

[1] We slightly modified steps in [86] for better illustration.

Step 1: Build Metacharacter Bitmaps. It first builds a bitmap for each metacharacter in JSON, including `,`, `:`, `[`, `]`, {, }, `"`, and \. In the bitmap, 1s represent the positions of the metacharacter in the record. The bitmap can be constructed by comparing the metacharacter against the characters in the JSON record one by one. For better efficiency, SIMD instructions can be leveraged to compare multiple characters simultaneously. Assume that the SIMD width is 256-bit, then a 8-bit metacharacter is duplicated 32 times to populate a 256-bit vector ($\_mm256\_setr\_epi8$). The vector is then compared against the characters in JSON data, 256 bits (32 bytes) each time ($\_mm256\_cmpeq\_epi8$). The result is in a 256-bit vector where each 8-bit lane is either all 1s or all 0s. Finally, the most significant bit of each lane is selected and packed into a 32-bit integer ($\_mm256\_movemask\_epi8$). On a 64-bit machine, two such 32-bit integers are combined into a `long` type. This step runs in $O(\frac{8*n}{W})$ instructions, where n is the number of characters in the JSON record and W is the SIMD width (e.g., 256).

Note that the metacharacters may appear inside a string, in which case they are not actually effective in defining the JSON structure. To exclude them, we need to first identify the strings in JSON, which are marked by quotes. However, a quote may be escaped by \, which can be further escaped, as in `\\\"`. The next three steps are to find the actual strings in a JSON record and exclude the metacharacters in strings from the corresponding bitmaps.

Step 2: Remove Escaped Quotes. This step excludes all the escaped quotes − quotes following an odd number of consecutive \s, from the quote bitmap. To achieve this, Mison [86] iterates through all the quotes with backslashes ahead. In comparison, simdjson [85] pro-

poses a bitwise solution: it locates the starting and ending positions of backslash sequences, then finds odd-length backslash sequences based on the fact that a sequence of characters that starts at an odd (or even) position and ends at an even (or odd) position must have an odd length, thus, any following quotes should be escaped. More details about step and its implementation can be found in [85].

Figure 4.2 (Step 2) shows the updated quote bitmap, with one escaped quote (the 4th to the right) removed. This step needs $O(\frac{n}{w})$ instructions, where w is the word size.

Step 3: Build String Mask Bitmap. Next, it generates a string mask bitmap, where 1s correspond to characters in strings. Here, a bitwise approach can also be adopted from simdjson [85]. First, a prefix-sum of `XOR` is applied to the updated quote bitmap. The resulted bit at index i equals to the `XOR` of all bit values up to i (inclusive). In fact, this operation can be implemented by performing a carry-less multiplication `PCLMUQDQ` between the quote bitmap and a 64-bit value with all 1s. This step also runs in $O(\frac{n}{w})$ instructions.

Step 4: Remove Pseudo-Metacharacters. This step removes those metacharacters that appear in strings, from their corresponding bitmaps. It can be implemented as an `ANDNOT` operation between the metacharacter bitmaps and the string mask bitmap.

Step 5: Generate Leveled Bitmaps. The final step is to separate the colon and comma bitmaps based on the levels that colons and commas appear in the record, dictated by the brackets `[`, `]`, {, }. As shown in Figure 4.2 (step 5), the top level is an object with one key-value pair, so there is a `1` at Level 1, corresponding to a colon. Next, as the value in the key-value pair is an array of two elements, Level 2 has one `1`, corresponding to a comma

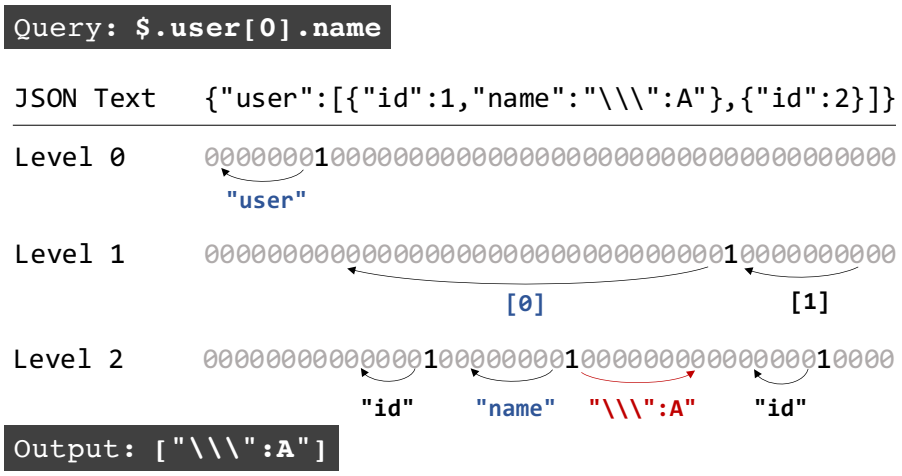

Figure 4.3: Querying via Structural Indices.

in the array. This process repeats until it reaches the deepest level in the record (or in the path query). These leveled bitmaps are the final outputs − structural indices.

To implement this step, Mison [86] proposes to scan the brackets ([,], {, }) bitmaps and employ a stack to recognize the nesting levels of colons and commas. Though the idea is intuitive, this solution actually involves expensive memory copies operations. We will elaborate this step in detail later in the paper.

In summary, the structural indices can be effectively constructed by taking advantages of bitwise and SIMD parallelism. For example, on a 64-bit processor, the construction can operate on 64-bit vectors, where a bit corresponds to one character (byte) in the JSON record. That is, the structural index construction can process 64 characters simultaneously. With the use of SIMD instructions, some of these operations can operate on vectors of 256 bits or even 512 bits, which can significantly accelerate the JSON data processing.

Querying Structural Indices. With the structural indices, we can quickly evaluate JSONPath queries. As illustrated in Figure 4.3, for a given JSONPath query (`$.user[0].name`), the evaluation starts from the top level of JSON structure (an object in the example), locates its key (`"user"`) through backward parsing from all the `1`s in this level, then finds the range in the bitmap for its value − between this `1` and the next `1`, and moves to the next level. This process stops when no match is found in a certain level or continues until the whole query expression has been matched.

4.2.3 Scalability Issues for Bulky Records

Despite its exploration of fine-grained bitwise parallelism, the existing structural index construction [86] fails to scale well to bulky JSON records with complex structures, due to three reasons.

- ***No Intra-Record Data Parallelism.*** Existing construction processes each record in serial. For bulky records, the serial processing can seriously limit the scalability, resulting significant delay for answering a query.

- ***Heavy Memory Operations.*** The last step (bottleneck step) of the existing construction requires to *duplicate* the colon and comma bitmaps K times, where K is the number of nesting levels in the record or the query expression. For bulky records, making copies of whole bitmaps are very expensive. Then, the algorithm masks bits with overlapped ranges, with an increasing overhead as it moves to deeper levels. Finally, it employs a stack which also makes intensive memory copies during the `push` and `pop` operations.

- ***Poor Data Locality***. A simple adoption of the existing index construction algorithm to bulky records may suffer from poor data locality, as the generated intermediate bitmaps may not fit into the CPU cache(s)

The primary goal of this work is to bring coarse-grained data parallelism to the structural index construction such that a bulky record can be processed in parallel effectively. Moreover, this work also tries to improve the memory access efficiency by redesigning the critical step of structural index construction algorithm (Step 5) and offering a more locality friendly construction process.

4.3 Data-Parallel Construction

For a sequence of small JSON records, parallelism naturally exists among different records. The challenge lies in the parallelization of building indices for a single bulky JSON record, which could cause significant delay when it is processed sequentially.

4.3.1 Dependences

When a bulky JSON record is partitioned, as illustrated in Figure 4.4, most steps of the structural index construction (except Steps 1 and 4) involve certain kinds of dependences. For Step 2 (removing escaped quotes), when a sequence of backslashes are broken into two partitions, we need to know the number of backslashes appearing at the suffix of the prior partition to tell if a following quote is escaped or not. For Step 3 (building string mask bitmap), we need to know whether the current partition starts inside a string or not.

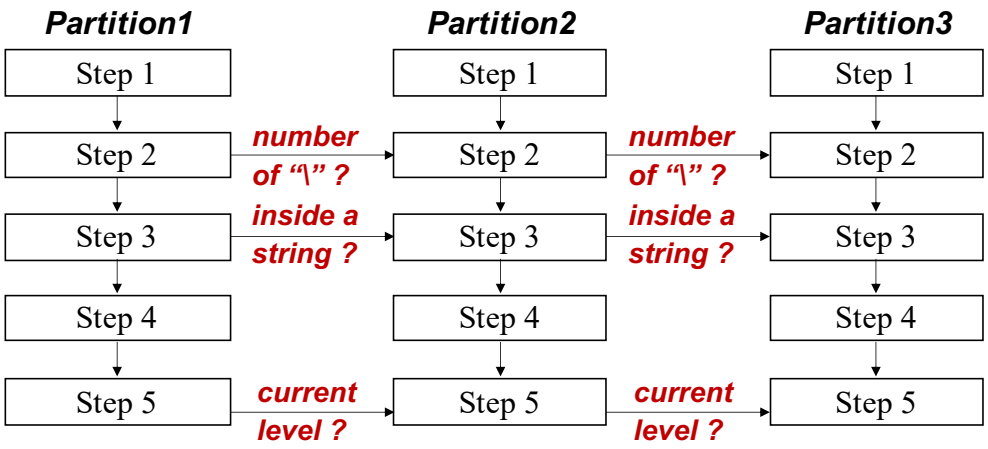

Figure 4.4: Dependences in Index Construction.

Finally, for Step 5 (generating leveled bitmaps), we need to know at which level the current chunk begins.

To construct the structural indices in parallel, we need to effectively address the above dependence challenges. In the following of this section, we will introduce an assembly of parallelization techniques that are integrated together to solve the dependences, including (i) *dynamic partitioning*, (ii) *contradiction-based context inference*, (iii) *speculation*, and (iv) *reduction*-style parallelization. Note that the last one (reduction-style parallelization) is based on a redesign of the Step 5 in the existing construction algorithm.

4.3.2 Dynamic Partitioning

Naively partitioning a JSON record may break a string, a keyword (e.g., `true` or `null`), or a sequence of backslashes into different chunks. The first and third cases are related to the dependences in Step 3 and Step 2, respectively (see Figure 4.4). One way to

avoid such cases is adjusting the boundary (e.g., moving it to the left) between two chunks dynamically such that no strings, keywords, or backslash sequences get cut. However, avoiding a bad cutting is non-trivial. Considering a piece of JSON data $\cdots$`"`$\cdots$, in general, it is hard to tell if the quote is the start or the end of a string. For this reason, we only use this dynamic partitioning to avoid cutting of backslash sequences and keywords. Broken strings will be addressed next with more advanced techniques.

4.3.3 Contradiction-based Context Inference

In fact, issues related to broken strings also arise in other contexts of parallel processing. Recently, Ge and others [65] addressed the ambiguity issue caused by broken strings in parallel CSV parsing with a pattern-based approach. Two string patterns (each with two chars) are carefully designed for CSV data, such that once they are observed, the parsing ambiguity can be immediately resolved. Though the idea is inspiring, it is unclear how similar patterns can be designed for JSON whose syntax is much more complex than CSV. In the following, we will present a *contradiction*-based string context inference. Unlike the prior work [65], this inference does not rely on the specific syntax of data type, thus may also work for data types beyond JSON.

The key insight behind the contradiction-based inference is that *characters inside a string, most of the time, do not form valid tokens.* For example, in string `"user"`, characters `user` cannot be interpreted as any valid JSON token(s). In fact, any string with some alphabet letters ([a-zA-Z]), except keywords[2] and scientific numbers (e.g., `2e+7`), cannot be

[2]Alphabetical keywords in JSON: `true`, `false`, and `null`.

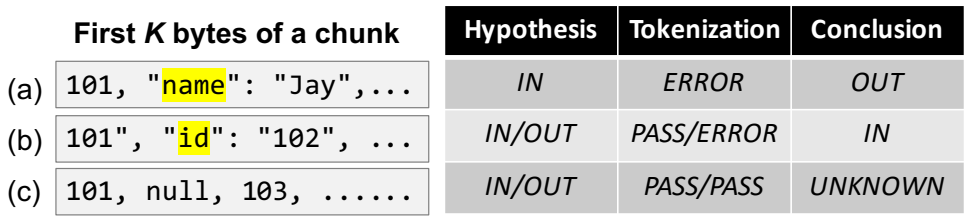

Figure 4.5: Examples of Context Inference.

tokenized successfully. Based on this insight, we propose to build *hypothesis* and leverage *contradiction* to infer the string status at the beginning of a JSON partition.

Algorithm 5 shows the contradiction-based context inference. First, it assumes that the input chunk starts inside a string (i.e., $hypothesis$ = IN). Based on the assumption, it tries to recognize the tokens in the first K bytes of the chunk. Once it hits an error of tokenization, it gets a contradiction − the current assumption is wrong, therefore it returns the opposite status − OUT (Line 9-10), as the conclusion. An example of such cases is Figure 4.5-(a). Otherwise, if the first K bytes are all successfully tokenized (Line 6), the algorithm would fail to draw any conclusion, because some sequences of characters can be interpreted in either way correctly: part of a string or valid JSON tokens, like the example in Figure 4.5-(c). In this case, the algorithm tries the other assumption − $hypothesis$ = OUT, to tokenize the first K bytes again. If it encounters a tokenization error (i.e., a *contradiction*), it would return IN as the conclusion, which happens to the case in Figure 4.5-(b). Finally, if both attempts fail to draw any conclusion, the algorithm returns UNKNOWN, as in the case of Figure 4.5-(c).

Algorithm 5 Contradiction-based Context Inference

```
1: /* return string status at beginning, head represents the first K bytes of a chunk */
2: procedure context_inference(head)
3:     while hypothesis = {IN, OUT}  do
4:         while TRUE do
5:             /* reach the end of the head */
6:             if head.hasNextToken() == END then
7:                 break;
8:             /* cannot recognize the next token */
9:             if head.hasNextToken() == ERROR then
10:                return opposite(hypothesis);
11:            head.nextToken();
12:    return UNKNOWN;
```

Note that, the more bytes (i.e., larger K) the algorithm checks, usually the higher chances it can draw a conclusion, but this will also incur higher overhead. In evaluation, we set $K = 64$ by default, and found it rarely fails to draw the conclusion. But, what if it does fail to draw any conclusion? We next address such cases with speculation.

4.3.4 Speculative Parallelization

Note that even when the conclusion is UNKNOWN, the chances for being inside a string and being outside a string are usually not equal. In fact, most of the time, they are highly biased, for the same reason mentioned earlier: it is rarely seen that characters inside a JSON string form valid JSON tokens. In another word, if the inference has successfully recognized the first K bytes as valid JSON tokens, it is very unlikely they are part of a string. Therefore, if we speculatively consider the status is OUT, there is a high chance that it is true.

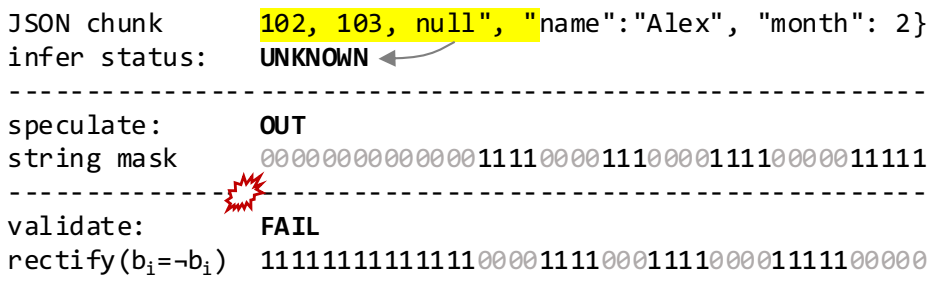

Figure 4.6: Bitwise Rectification for Misspeculation.

Based on this intuition, we propose a speculative parallelization scheme for structural index construction. If the context inference of one chunk returns UNKNOWN, the construction automatically enters into speculative mode, in which chunks with UNKNOWN inference results are processed speculatively by assuming they start from a position outside a string. The next question is when when the speculation is validated. If we validate it after all the five steps are finished (on each chunk), once a speculation fails, we have to rerun all the steps on the corresponding chunk. To minimize the misspeculation cost, we should validate the speculation right after Step 3. To achieve this, we add a synchronization between Steps 3 and 4, where chunks in speculative modes are validated against the ground truth (IN or OUT) of prior non-speculative chunks. In cases some speculation fails, a naive handling is reprocessing the corresponding chunks (up to Step 3). In fact, this is an overkill for this particular kind of misspeculation. We will present a faster alternative shortly.

Similar to our situation, the pattern-based approach [65] used in parallel CSV parsing may also fail to resolve the parsing ambiguity sometimes, thus turns to specula-

tion for parallelization. However, as we show next, for structural index construction, the misspeculation can be handled more efficiently at the bit level with *bitwise rectification*.

Bitwise Rectification. Instead of reprocessing the JSON chunk that was misinterpreted (from Step 1 to Step 3), we found it is possible to directly recover the correct bit values from the incorrect string mask bitmap with a simple bitwise logic operation $b_i = \neg b_i$. Figure 4.6 illustrates this idea with an example JSON chunk. In the example, the string status inference fails to draw any conclusion (i.e., UNKNOWN), thus the index construction switches to the speculative mode and assumes the string status is OUT. However, during the validation, it turns out that the correct string status is IN. Interestingly, as shown in Figure 4.6, the correct string mask bitmap is exactly the "opposite" of the incorrect bitmap. Essentially, this is due to the *parity* of quotes in defining strings − a string always starts from an odd-number quote and ends at the next (even-number) quote. An incorrect interpretation of the quote parity would flip all the following string definitions. Based on this observation, we can generate the correct bitmap by flipping values in the incorrect bitmap, that is, $b_i = \neg b_i$. As expected, we found this bitwise rectification is much faster than reprocessing the JSON chunk.

4.3.5 Parallel Generation of Leveled Bitmaps

The last dependence to address is the "level" at the beginning of a chunk in Step 5 (generating leveled bitmaps). Before introducing the solution, we first summarize the existing design of Step 5 from Mison [86] and present a new design for this step which is

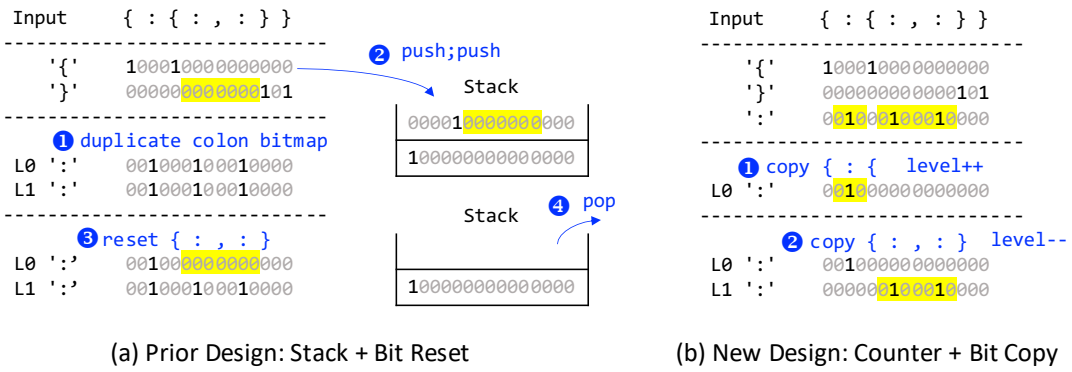

Figure 4.7: Generation of Leveled Bitmaps: Existing Design (left) vs. New Design (right).

simpler yet more efficient. After that, we will explain how the new design can be parallelized based on the reduction-style parallelism.

Redesign of Step 5

The main task of Step 5 is to separate colons and commas of different levels into different bitmaps.

Existing Design. Figure 4.7-(a) illustrates the basic idea of Step 5 in Mison [86]. For easier explanation, the figure only shows curly brackets and colons that capture the structures of JSON objects, but the idea can be easily extended to cover JSON arrays. First, it duplicates the colon bitmap K times, where K is the number of levels in the JSON record or in the query expression (e.g., two times for `$.user.name`). The duplicated bitmaps, denoted as $L_0 \cdots L_{K-1}$, will be turned into the leveled bitmaps; Then, it tries to locate the objects from inner levels to outer levels by traversing the bitmaps of { and } with the help of a stack. Basically, from left to right, the bitmap corresponding each { is pushed onto the

stack. The stack top has the beginning position of the inner most object (the position with `1`). From there, Mison scans forward to find the first } − the ending position of the inner most object. Then, it resets the bits in the same range in L_0, as highlighted in Figure 4.7-(a). After that, Mison pops the stack, moves to the second inner most level, and repeats the same operations. This process continues until Mison reaches the outer most level of the JSON structure. In practice, we found that implementation based on the above design executes efficiently for small JSON records, taking 30-40% of the total construction time. However, for bulky records, this ratio is raised to 60-70%, making Step 5 the bottleneck of structural index construction. Our further investigation reveals two main factors limiting its efficiency:

- *Intensive Bit Value Updates.* First, the colon bitmap duplication involves intensive memory writes to (pre-allocated) bitmaps. Second, from an inner level to an outer level, as the object range becomes larger, more bits need to be reset. The complexity of the bit resetting is $O(\frac{n}{w} \cdot b)$, where n is the number of bytes, w is the word size, and b is the number of brackets.

- *Expensive Stack Operations.* Employing a stack to record nesting level and the start positions of objects requires copying bitmaps onto and off the stack (`push` and `pop`), which are relatively expensive operations.

To address the efficiency issues in the existing design of Step 5, we next introduce a new algorithm for generating leveled bitmaps.

Algorithm 6 Generating Leveled Bitmaps

```
1: /* generate leveled bitmaps l[0...k − 1] */
2: procedure generate_leveled_bitmaps(b_colon, b_comma, b_lbracket, b_rbracket)
3:     level = -1;
4:     for each word w in bitmaps do
5:         w_bracket = w_lbracket ∨ w_rbracket;
6:         if w_bracket == 0 then /* if no brackets, copy all bits from w_colon or w_comma */
7:             if w_colon ≠ 0 then w_l[level] = w_colon; /* part of an object */
8:             else w_l[level] = w_comma; /* part of an array */
9:         else
10:            w_begin = 1; /* beginning position of an interval */
11:            w_range = get_interval_bitmap(w_bracket, w_begin, level);
12:            while w_range ≠ 0 do /* iterate over intervals separated by brackets */
13:                if w_colon ∧ w_interval ≠ 0 then
14:                    w_l[level] = w_l[level] ∨ (w_colon ∧ w_range);
15:                else
16:                    w_l[level] = w_l[level] ∨ (w_comma ∧ w_range);
17:                if w_end ∧ w_lbracket > 0 then level = level + 1;
18:                if w_end ∧ w_rbracket > 0 then level = level - 1;
19:                w_range = get_interval_bitmap(w_bracket, w_begin, level);
20:    return l[0...k − 1]
```

New Design. Figure 4.7-(b) illustrates the basic idea of the new design. First, instead of duplicating the colon bitmap and resetting the bits, the new design first allocates leveled bitmaps with all 0s (`calloc()`), then copies bits of different levels from the colon bitmap to the corresponding leveled bitmaps, which reduces the bit value updates.

Second, the new design recognizes the objects from the outer levels to inner levels – the opposite of the existing solution. This avoids the use of stacks for recording the

Algorithm 7 Generating Leveled Bitmaps (Continue from Algorithm 6)

```
1: procedure get_interval_bitmap(w_bracket, w_begin, level)
2:     if w_bracket ≠ 0 then /* locate next interval */
3:         w_end = E(w_bracket); /* end of an interval */
4:         w_bracket = R(w_bracket);
5:         w_interval = w_end − w_begin;
6:         w_begin = w_end;
7:     else/* locate the last interval */
8:         w_end = 2^(|w|−1); /* end of the word */
9:         w_interval = (w_end − w_begin) ∨ w_end;
10:    return w_interval
```

beginning positions in the existing design. In this case, a counter, like variable `level` in Figure 4.7-(b), is sufficient for recognizing the levels of objects.

More details of this new design are shown in Algorithm 6, which covers the generation of leveled bitmaps for both fields inside an object and members inside an array. The inputs to the algorithm include colon bitmap b_{colon}, comma bitmap b_{comma}, and left and right bracket bitmaps $b_{lbracket}$ and $b_{rbracket}$, where $b_{lbracket}$ combines bitmaps of { and `[`, and $b_{rbracket}$ combines bitmaps of } and `]`. Since both kinds of brackets add levels, there is no need to distinguish between them (the same as in [86]). The outputs of the algorithm are leveled bitmaps, each of which consists of the `1`s from the colon and comma bitmaps in the corresponding level.

Initially, the algorithm marks the current `level` to -1 (Line 3). Then, it iterates over the bitmaps word by word. For better efficiency, it combines the words from left and right bracket bitmaps $w_{lbracket}$ and $w_{rbracket}$ into $w_{bracket}$ (Line 5). If $w_{bracket}$ contains only `0`s (no brackets), it simply copies the current word of colon/comma bitmap to the

corresponding leveled bitmap (Line 6-8); Otherwise, the algorithm iterates over all the intervals separated by two consecutive brackets. Algorithm 7 are to find the current interval. Here, the algorithm uses functions E and R defined in Mison [86] to extract the right most bracket and remove it, respectively. Once the interval is located, Algorithm 6 copies the word from colon/comma bitmap to the same interval of the corresponding leveled bitmap (Line 13-16). Finally, it updates the `level` based on the bracket is left or right. The same process repeats until all the words of the bitmaps are processed.

With the new design, next we will show that Step 5 can be parallelized based on a reduction-style parallelism. Note that, in theory, the prior design of Step 5 [86] might also be parallelized in a similar fashion, but the reduction in that case would become more involved for its stack-based design and the efficiency may suffer even more with multiple threads, given its intensive memory operations.

Reduction and Index Merging

Partitioning a JSON record makes it difficult to tell at which level the beginning of a chunk is. To "break" the level dependence among JSON chunks, we adopt a *reduction*-style parallelization with two phases: (i) parallel leveled index generation and (ii) index merging, as illustrated in Figure 4.8.

In the first phase, all JSON chunks are processed in parallel under the assumption that they all start from Level 0. The outputs of this phase are the partial leveled bitmaps for individual JSON chunks, as depicted in Figure 4.9. After all chunks have been processed, the construction enters into the second phase, where the levels of different chunks are aligned

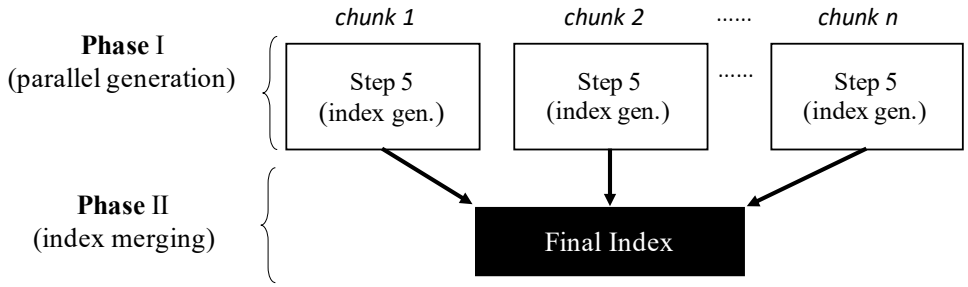

Figure 4.8: Two-Phase Index Construction.

with those in the prior chunks one by one − the ending level of chunk i is the beginning level of chunk $i+1$. In the example shown in Figure 4.9, thread T1 ends at Level 1, which is matched with Level 0 of thread T2. As a result, the original Level 1 of thread T2 becomes Level 2, which is then aligned with Level 0 of thread T3. Thus, levels -2 and -1 of T3 turn into levels 0 and 1, respectively.

After adjusting the levels, the partial bitmaps are connected based on their actual levels, forming an array of linked lists, as shown in Figure 4.10. Note that the outputs from parallel index construction are slightly different from those in serial index construction, where each leveled bitmap is a single array. In principle, this difference may increase the cost of bitmap accessing. However, since the number of chunks (the same as the number of CPU cores) is relatively very small comparing to the number of bits, this extra cost is small (within 2%).

In summary, with multiple types of dependences involved in the structural index construction with an assembly of customized parallelization techniques. Next, we integrate them.

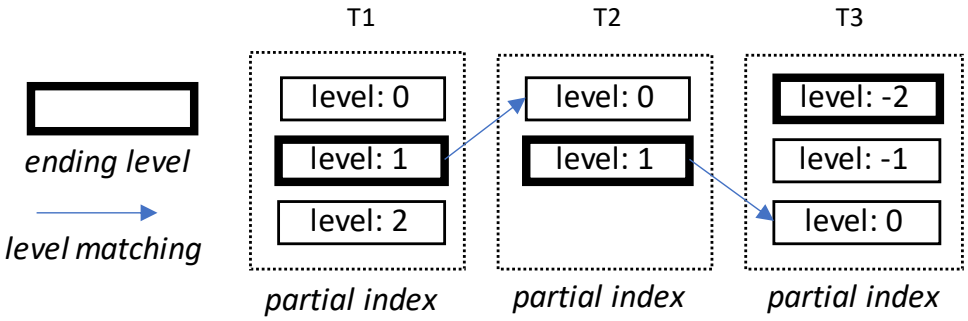

Figure 4.9: Level Matching.

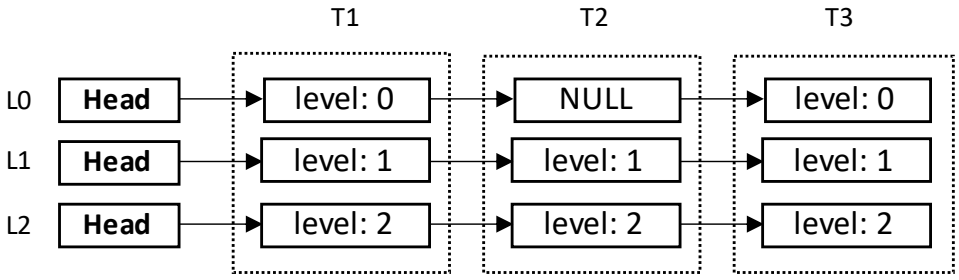

Figure 4.10: Index Merging.

4.3.6 Putting it All Together

Figure 4.11 shows the high-level workflow of data-parallel structural index construction. From the top to the bottom, the construction process is divided into five stages. The first stage partitions the JSON record into chunks with *dynamic partitioning* to avoid breaking any keywords and backlash sequences. By default, the number of chunks n is set to the number available CPU cores. After partitioning, the construction leverages the *contradiction-based context inference* to find out if a chunk starts inside a string or not. When some inferences fail, *speculation* is immediately triggered to process the corresponding chunks speculatively. If speculation is enabled for at least on one JSON chunk, the

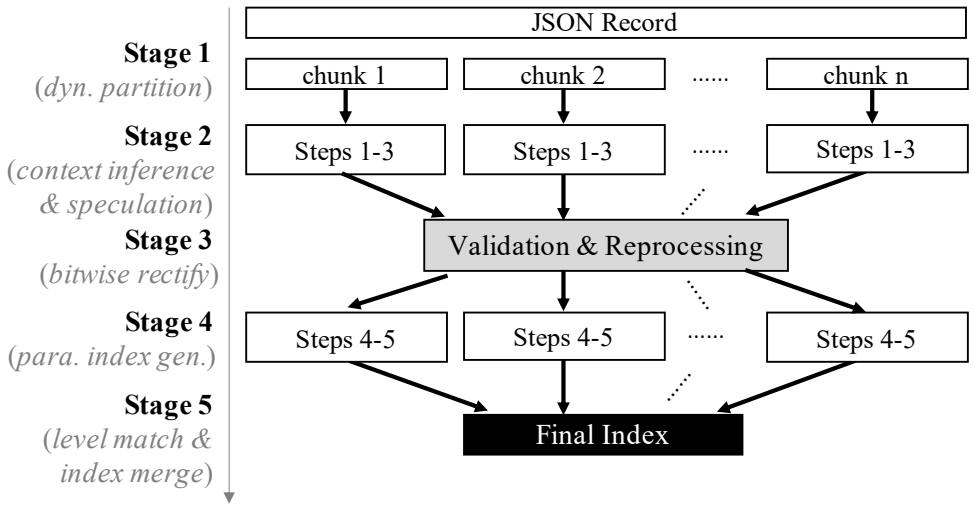

Figure 4.11: Workflow of Data-Parallel Construction.

construction would next enter into the Stages 2, 3 and 4, one by one; Otherwise, it skips Stage 3 and combines Stages 2 and 4 (i.e., all five steps are completed without synchornization). Before entering Stage 3, all the parallel executions of different chunks (in Stage 2) should have been finished, that is, a *barrier* is required. In Stage 3, chunks with speculation are validated *in order* based on the actual string statuses (i.e., inside a string or not). When some misspeculation is detected, the corresponding input chunk would be reprocessed (only Step 3). For fast reprocessing, *bitwise rectification* would be applied. The outputs of after Stage 4 are partial structural indices built for each input chunk. Finally, Stage 5, performs the index level matching and merging to produce the final structural indices.

4.4 Locality Optimization

So far, our design of parallel structural index construction (including a new Step 5) is performed step by step. In each step, it traverses the JSON record and/or its (inter-

mediate) bitmaps entirely. This is consistent with the ways of description of the original design of Mison [86]. Since steps share the access of some bitmaps (e.g., one step writes to it and another step reads from it), repetitively traversing them cause poor data locality. This issue might less be a concern for small JSON records, but could become serious when processing large JSON records, as the memory footprint easily exceeds the cache capacities.

Instead of constructing the bitmaps step by step, in fact, we can build them word by word [3], where a word consists of only a few bytes (e.g., 8 bytes on 64-bit machines). Figure 4.12 illustrates the two granularities of bitmap construction.

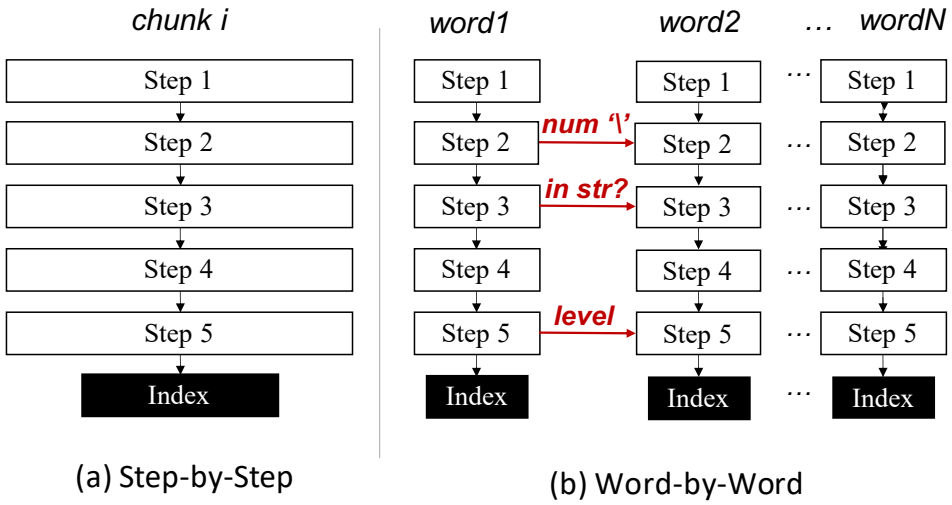

Figure 4.12: Index Construction Granularities.

To implement the word-by-word bitmap construction, the construction algorithm has to generate and consume bitmaps partially and incrementally. This requires recording some "context" of each step between two adjacent words. For example, after finishing one word at Step 2, the algorithm needs to record how many consecutive \s have been observed

[3]Note that word-by-word processing could also be adopted in Mison [86]

by the end of the current word and resume the counting when processing the next word. Despite these extra work of "bookkeeping", the benefits of word-by-word processing are significant for bulky JSON records, as we will report later.

4.5 Querying using Index

In this section, we first present the APIs for accessing the bitmaps and demonstrate their usage in evaluating common JSON path queries, then we further discuss the strategies for enabling parallel query evaluation on bulky JSON records.

APIs. To simplify the programming, we hide the low-level bitmap traversal details into a set of high-level JSON data accessing APIs that are similar to the existing tools [85]. The APIs include a *BitmapConstructor* class for generating the leveled bitmaps in parallel and a bitmap traversal class *BitmapIterator* for navigating through the leveled bitmaps and locating the keys in objects or elements in arrays.

Algorithm 8 demonstrates an example usage of these APIs for evaluating query `$.user[0].name`. Note that even though the bitmap indices are constructed in parallel (line 3), the bitmap traversal (line 5-10) remains sequential. To make JSON data analytics scalable, we next describe a strategy that enables parallel bitmap traversals.

Parallel Query Evaluation. Given the indices of many small JSON records, we can easily traverse them in parallel and evaluate the query on different records independently. However, for a bulky JSON record, it is non-trivial to traverse its indices in parallel. Our parallelization of the index traversal stems from a simple yet critical observation [4] – a bulky

[4] According to real-world datasets [28, 16, 23, 2, 6, 26].

Algorithm 8 API Usage Example (query `$.user[0].name`)

```
1: procedure query_execution_example(input)
2:     BitmapConstructor bc = new BitmapConstructor()
3:     Bitmap bm = bc.construct(input, 16) /* with 16 threads */
4:     BitmapIterator iter = bc.getIterator(bm)
5:     if iter.isObject() && iter.moveToKey("user") then
6:         iter.down() /* value of "user" */
7:         if iter.isArray() && iter.moveToIndex(0) then
8:             iter.down() /* "user[0]" */
9:             if iter.isObject() && iter.moveToKey("name") then
10:                result = getValue()
11:    return result
```

JSON record usually consists of a JSON array at an upper level (closer to the root level) which dominates the size of the JSON records and divides lower levels of the JSON record into many smaller elements. We refer to such an array as the *dominating array*. Based on this observation, we propose to first locate the index level of the dominating array, from where we then traverse different elements in the array in parallel. The criteria for defining the dominating array are configurable. In our experiments, we require the array to occupy 80% of the whole record in size and consist of at least 256 elements. Elements of the dominating array are processed using a thread pool where each thread gets a copy of the current bitmap iterator and proceeds independently. For better load balancing, elements of the dominating array are inserted into a *worklist* and then consumed on demand by worker threads.

4.6 Evaluation

This section presents evaluation results of the proposed techniques, with a focus on the parallel performance on bulky JSON records.

4.6.1 Methodology

We implemented the parallel JSON structure index constructor in C++, namely Pison, and used Pthread for its parallelization. We compare Pison with the existing solution Mison [86]. As Mison [86] itself is not publicly available, we use a third-party implementation of Mison, called Pikkr [13], as well as our own implementation of Mison for this comparison. Moreover, we also implemented an optimized version of Mison, denoted as Mison+, where Steps 2 and 3 are implemented using ideas from simdjson [85] (see Section 4.2).

In addition, we also compare Pison with simdjson [85] − a popular SIMD-based JSON processing tool, RapidJSON [18] − a popular JSON parser based on character by character processing, and JPStream [78] − a streaming-based JSON tool. Table 4.1 lists all the methods in our evaluation.

Table 4.1: Methods in Evaluation

Method	Brief Description
simdjson	A SIMD-based JSON parser [85]
CMison	Our C++ implementation of Mison [86]
Mison+	Improved Mison (Steps 2-3) based on simdjson [85]
Pikkr	Third-party implementation of Mison in Rust [13]
RapidJSON	A JSON parser in C++ from Tencent [18]
JPStream	A parallel streaming JSON processor in C [78]
Pison(SbS)	Pison with step-by-step processing (this work)
Pison(WbW)	Pison with word-by-word processing (this work)

We evaluate Pison for processing a sequence of small JSON records and individual bulky JSON records in terms of both processing time and memory consumption. Table 4.2 reports the statistics of JSON datasets used in our evaluation. These include Best Buy (BB) product dataset [2], tweets from Twitter (TT) developer API [23], Google Maps Directions (GMD) dataset [6], National Statistics Postcode Lookup (NSPL) dataset for UK [16], Walmart (WM) product dataset [26], and Wikipedia (WP) entity dataset [29]. The default size of each dataset is approximately 1GB for easier comparison. Each dataset forms a single large JSON record. To create scenarios of small records processing, we manually extracted the dominating array from each dataset, broke it into smaller records, and inserted a new line after each small record − a common way to organize small JSON records. The number of small records for each dataset is shown in the column #subrec. in Table 4.2.

Table 4.2: Dataset Statistics

Data	#objects	#arrays	#K-V	#prim.	#subrec.	depth
TT	2.39M	2.29M	26.5M	24.3M	150K	11
BB	1.91M	4.88M	40.7M	35.8M	230K	7
GMD	10.3M	43K	29.0M	21.0M	4.44K	9
NSPL	613	3.50M	1.66K	84.2M	1.74M	9
WM	333K	34K	8.19M	9.92K	275K	4
WP	17.3M	6.53M	53.2M	35.0M	137K	12

To evaluate the querying performance, we first include 8 queries used in Mison [86] on Twitter dataset (TT1-TT8), then for each other dataset, we created a JSONPath query, as shown in Table 4.3. Note that 8 of out the 13 queries consist of two subqueries. For bulky records made of small records, we add a prefix `[*].` to each of its queries. Together,

they cover common patterns of path queries, as well as queries of different complexities and selectiveness.

Table 4.3: JSONPath Queries

Queries TT1-TT8 are from Mison [86]

ID	Query structure	#matches	#visited fields
TT1	{ ur.id }	150,135	1,979,268
TT2	{ ur.id, rtct }	300,270	3,076,477
TT3	{ ur.id, ur.la }	300,270	4,681,698
TT4	{ ur.nm, rp }	300,270	2,279,538
TT5	{ ur.la, la }	300,270	6,767,260
TT6	{ id, rtst }	252,524	3,243,333
TT7	{ id, en.urls[*].url }	239,016	3,615,763
TT8	{ id, en.urls[*].idc[*] }	327,897	3,862,762
BB	{ pd[*].cp[1:3].id }	459,332	7,362,758
GMD	{ rt[*].lg[*].st[*].dt.tx }	1,716,752	5,215,576
NSPL	{ mt.vw.co[*].nm }	44	121
WM	{ it[*].bmpr.pr, it[*].nm }	288,391	7,867,449
WP	{ cl.P[*].ms.pty }	15,603	1,974,693

All the experiments were conducted on a 16-core machine equipped with two Intel 2.1GHz Xeon E5-2620 v4 CPUs and 64GB RAM (referred to as *Xeon server*) The CPUs support 64-bit ALU instructions, 256-bit SIMD instruction set, and the carry-less multiplication instruction (`pclmulqdq`). Both servers run on CentOS 7 and are installed with G++ 7.4.0 and JDK 1.8.0_191. All C++ programs are compiled with "-O3" optimization flag. In the case of small records, they are stored in a single array with the beginning position of each record stored in a separate `offset` array. The timing results are the average of 10 runs; no 95% confidence interval is shown when the variation is not significant.

Next, we first report the performance of serial and parallel index constructions of different methods.

4.6.2 Index Construction Peformance

Figure 4.13 reports the time of index construction for bulky JSON records using different methods. We exclude JPStream since it does not construct any indices. Among them, RapidJSON and simdjson create parse trees while the other methods create structural indices.

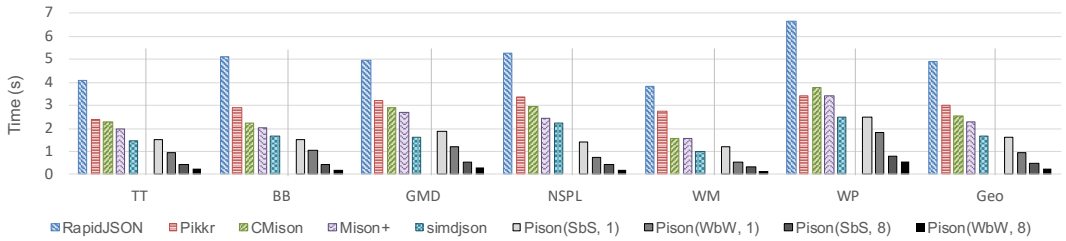

Figure 4.13: Comparison of Index Construction Time of Different Methods on Bulky JSON Records.

RapidJSON vs. Others. First, according to the results, RapidJSON takes substantially longer to construct indices than other evaluated methods, even though it is known for its superior performance than many other popular JSON parsers (such as FastJSON [4]). The main reason is that RapidJSON does not leverage bitwise parallelism and SIMD instructions. The results confirm the effectiveness of leveraging fine-grained parallelism in JSON data processing.

Mison vs. simdjson vs. Serial Pison. For methods with bitwise parallelism and SIMD supports, the two versions of Mison (Pikkr and CMison) take slightly longer than others. In comparison, Mison+ runs faster than the prior two, by 31% and 11%, thanks to its optimized Steps 2 and 3 based on simdjson. Following these, simdjson and serial Pison(SbS) show similar performance on average. Note that the only difference between

Mison+ and serial Pison(SbS) is the the new design of Step 5 (Section 4.3.5). Therefore, the time difference between the two shows the benefits of this new design − 1.41X speedup on average, which is substantial for an optimization of one step. Finally, serial Pison(WbW) performs the best among all the serial methods. The performance gap between Pison(WbW) and Pison(SbS) indicates the benefits of locality optimization (Section 4.4), which is 1.67X speedup on average, demonstrating the importance of the proposed locality optimization. Adding the benefits of the new design of Step 5 and the locality optimization together, we find that serial Pison(WbW) runs 3.1X faster than Pikkr, 2.6X faster than CMison, and 2.3X faster than Mison+.

So far, the comparison are among only serial methods. Next, we discuss the performance of parallel Pison − the main contribution of this work. Note that we cannot run other methods in parallel due to the lack of parallelization.

Parallel Pison vs. Others. First, running two versions of Pison (SbS and WbW) in parallel with eight threads, achieves 3.3X and 3.8X speedups over the serial counterparts, respectively. The sub-linear speedups indicate that the structural index construction is not only computation-intensive, but also memory-intensive; while the parallelization helps address the former, the performance could be limited by the latter. Despite this limitation, when compared to the other indexing methods, parallel Pison still shows significant performance improvements − 19X faster than RapidJSON, 11.6X faster than Pikkr, 9.8X faster than CMison, and 8.8X faster than Mison+.

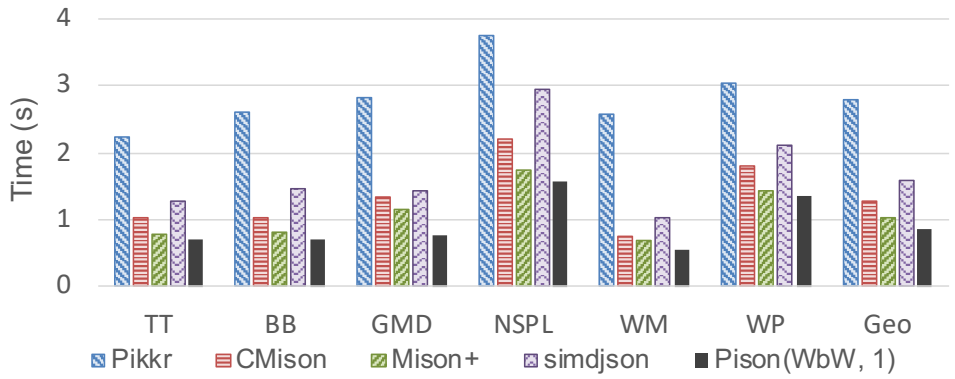

Figure 4.14: Comparison of Index Construction Time of Different Methods on Small JSON Records.

So far, the performance results are for building indices on bulky JSON records – the focus of Pison. For completeness, we also report the performance of small record indexing.

Small Record Indexing. For many small records, parallelism can be easily achieved at the task level, we only report the serial performance of Pison for fairness. Figure 4.14 shows the performance results of different methods, which are consistent with those in large record indexing, except that the two implementations of Mison and its optimized version Mison+ run relatively faster than they do in large record indexing. The reason is that when the records are small, they can easily fit into the caches; the locality of their step-by-step processing gets much improved.

Next, we break down the benefits of parallelization by steps and evaluate the effectiveness of the parallelization techniques in detail.

Table 4.4: Time Breakdown by Steps

Entry is [seq. time(s) : para. time(s) w/ 8 threads] of Pison(SbS)

	Step 1	Step 2	Step 3	Step 4	Step 5
TT	0.48 : 0.14	0.03 : 0.01	0.06 : 0.01	0.12 : 0.05	0.83 : 0.25
BB	0.59 : 0.18	0.05 : 0.01	0.06 : 0.01	0.15 : 0.06	0.65 : 0.20
GMD	0.65 : 0.19	0.05 : 0.02	0.06 : 0.01	0.21 : 0.06	0.92 : 0.27
NSPL	0.65 : 0.18	0.06 : 0.03	0.07 : 0.02	0.20 : 0.06	0.42 : 0.15
WM	0.59 : 0.17	0.06 : 0.01	0.08 : 0.01	0.19 : 0.06	0.27 : 0.10
WP	0.67 : 0.19	0.06 : 0.01	0.06 : 0.01	0.15 : 0.07	1.55 : 0.51
Geo SP	3.55X	5.04X	6.54X	2.83X	3.04X

4.6.3 Benefits and Costs Breakdown

Time Breakdown by Steps. Table 4.4 reports the sequential and parallel execution times of for each step in the structural index construction, as well as the averaged parallelization speedup of each step. The results show that parallelization improvements vary among steps. Step 3 achieves the highest speedup − 6.5X, while Step 4 achieves the lowest − 2.8X. As discussed earlier, the variation of benefits mainly depends on the memory-computation ratio. Step 3 is relatively more complex computation-wise, meanwhile only writes results to one bitmap (see Figure 4.2). By contrast, Step 4 only involves a single bitwise operation (`ANDNOT`), but needs to write to six bitmaps. As a result, Step 4 is more memory-bound than Step 3, leading to less speedups. Similar reasoning also holds to the other three steps. After reporting the time and speedups, we next further evaluate the main parallelization techniques used in different steps.

Context Inference. To parallelize Step 3, in Section 4.3.3, we proposed contradiction-based context inference to find if a chunk starts inside a string. To confirm its effectiveness, we profiled the success rate of context inferences in all the parallel executions reported in

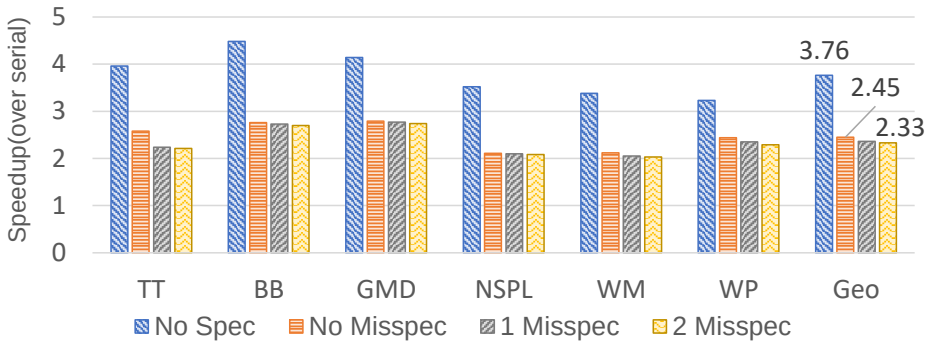

Figure 4.15: Costs of Speculation and Recovering.

Figure 4.13. The results show that all the inferences succeeded, which indicates that a contradiction is derived in all the evaluated cases.

Speculation. Since no context inferences failed in evaluation, we artificially failed a few context inferences when 8 threads are used to examine the performance under speculative mode. Thus, 20 threads entered into the speculative mode across 6 datasets Interestingly, none of them failed, which demonstrates the effectiveness of the heuristic used by the speculation − usually characters inside a string do not form valid tokens (see Section 4.3.4). Despite the high speculation accuracy, the performance still suffers from another aspect − loss of data locality. As discussed in Section 4.3.5, once entering speculative mode, the validation stage would isolate the word-by-word optimizations within Steps 1-3 and Steps 4-5, respectively. Figure 4.15 shows the cost of speculation − reducing speedups from 3.76X to 2.45X.

In principle, misspeculation can still happen, in which cases there are costs of recovering. To measure these costs, we manually flipped the speculated string status for 1

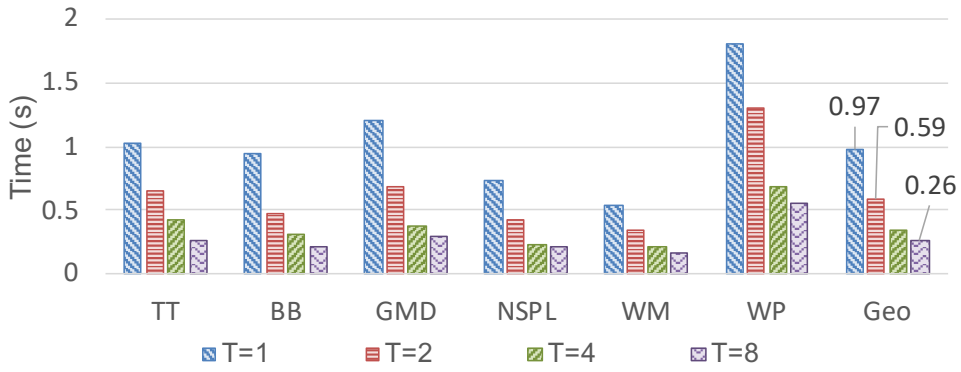

Figure 4.16: Scalability over Number of Threads.

and 2 out of 4 speculative threads, respectively. Figure 4.15 reports the costs of reprocessing. With 2 cases of misspeculation, the speedup only drops from 2.45X to 2.33X. This is due to the bitwise rectification introduced in Section 4.3.4. Without this technique, we have to reprocess the chunk, in which case the cost would be quite significant.

Costs of Index Merging. The costs associated with index merging are of two types: (i) the cost of index merging − the second phase of the two-phase parallel index construction and (ii) the extra cost of accessing of merged (linked) indices. For the former, we profiled the cost of each phase and found that the merging cost is less than 0.01% for all test cases. This is because the index merging only needs to connect the partial indices of different chunks together (see Figure 4.10), including the adjustment of index levels (Figure 4.9), both of which are low-cost operations. For the latter, our profiling results show that overhead of accessing linked indices is 1.5% on average, comparing to the single-array indices from serial construction.

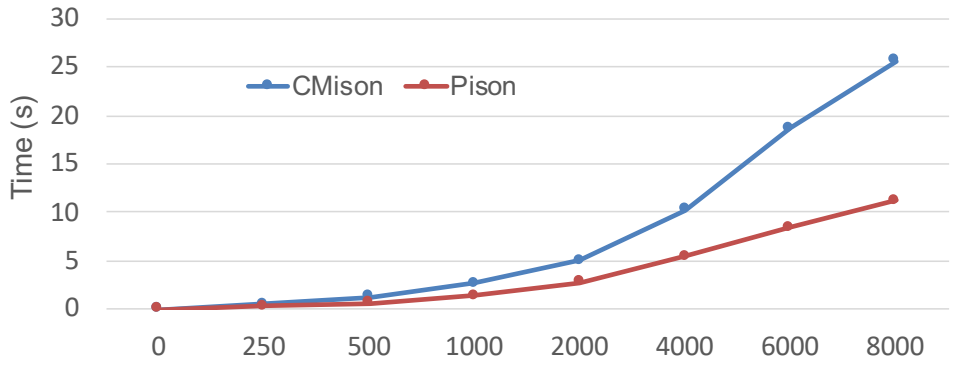

Figure 4.17: Scalability over Record Size.

Scalability. Figure 4.16 reports the parallel performance of Pison with different numbers of threads. The results show that the additional benefits from more threads diminish slightly as the number of threads increases. This is due to the fact that structural index construction is both computation and memory intensive. As computations get parallelized with more threads, the memory bandwidth gets more saturated, limiting the performance benefits.

To better understand how Pison and Mison scale as the record size increases, we further vary the size of the record in the BB dataset from 256MB to 8GB. Figure 4.17 shows the scalabilities of Mison and Pison as the record size increases. As the trend indicates, the larger the records are, the more time saving Pison provides comparing to Mison.

Memory Consumption. The other concern in processing large records is the memory consumption. As shown in Figure 4.18, the memory footprint of Pison is about 3.0GB on average, which is the least among the indexing or parsing-based methods. Comparing to step-by-step processing, the word-by-word processing of Pison eliminates the needs of storing large intermediate bitmaps, resulting in 40.8% less memory footprint. The Figure

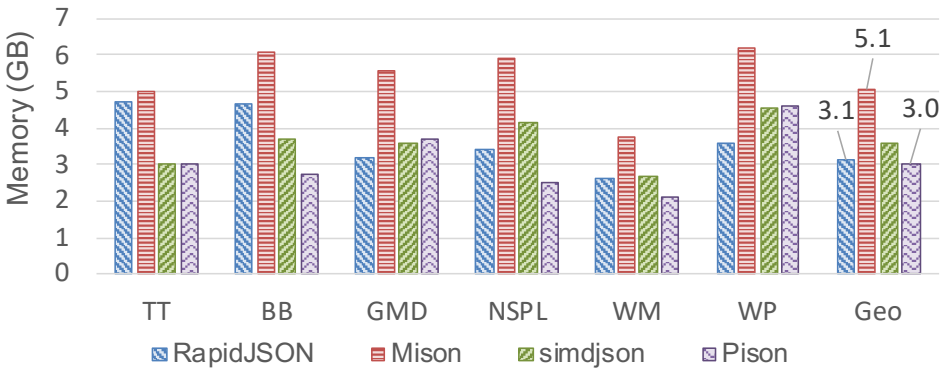

Figure 4.18: Comparison of Memory Consumption

Table 4.5: End-to-End Time (s) Comparison.

	JPS(8)	RapidJ	Mison+(1)	simdjson	Pison(1)	Pison(8)
TT	6.64	4.73	4.92	2.13	4.10	0.86
BB	1	5.27	2.61	1.84	1.55	0.34
GMD	1.01	5.18	3.64	1.88	2.32	0.58
NSPL	1.02	5.26	2.46	2.22	0.76	0.22
WM	0.43	3.98	2.14	1.19	1.2	0.25
WP	1.38	6.7	3.67	2.64	2.21	0.6
Geo	1.26	5.12	3.11	1.93	1.76	0.42
Sum	11.48	31.12	19.44	11.9	12.14	2.85

does not include the memory footprint of JPStream, which in fact is configurable (set it to 1GB) thanks to its streaming-style processing strategy.

4.6.4 End-to-End Performance

Finally, we evaluate end-to-end performance that includes both index construction and query evaluations. The queries used for this evaluation are from Table 4.3.

Table 4.5 reports the end-to-end performance of different methods under evaluation. Except for JPStream, the time for all the other methods include both the index

construction time and the querying time. For bulky JSON records, we cache the indices and reuse them for multiple queries. For dataset TT, this means the 8 queries share the constructed indices. In the case of JPStream, a streaming-based method, its execution time is only about the querying.

JPStream vs. Others. First, from the performance results of JPStream, we observe a large gap (over 5.2X) between the case of TT and cases of other datasets. In fact, JPStream performs the worst in the case of TT among all evaluated methods. The reason is that, for TT, multiple individual queries are evaluated. Without any indices, the evaluation has to traverse the entire raw dataset once for each query, causing repeated overhead, which is expected as JPStream is designed for streaming scenarios. For other datasets where a single query is evaluated, JPStream clearly shows better performance than other methods, except for parallel Pison. On average, parallel Pison runs 3X faster than JPStream when both of them running with 8 threads.

Parallel Pison vs. Others. Among the indexing-based methods, parallel Pison achieves 12.2X speedup over RapidJSON, 7.4X speedup over Mison+, 4.6X speedup over simdjson, and 4.2X over its own serial version. Note that for Mison+, we do not have the parallel implementation for its querying. However, we find that its indexing time alone (2.28s) already takes significantly longer than the total running time of parallel Pison (0.42s).

In summary, the above results confirm that the efficiency of parallel Pison, showing substantial performance boosts over a set of state-of-the-art JSON processing tools.

4.7 Related Work

This section discusses existing research on raw semi-structured data processing and the parallelization of large-record processing

Raw Semi-Structured Data Processing. There is a rich body of research on processing raw semi-structured data such as XML and JSON. Typical solutions often involve in some forms of automata [39, 134, 56] and stacks [50, 82], which are essential in recognizing the nesting structures and matching the queries. Most existing JSON tools [4, 12, 15, 18] follow this direction and convert the JSON data stream into in-memory tree structures before querying. However, it takes time and memory to generate the parsing trees [78]. One way to avoid the cost of parsing is to adopt streaming schemes [68, 81, 106, 21, 78]. For instance, JSONSurfer [21] directly evaluates path queries without any pre-parsing of the JSON data. However, it is inefficient in performing the query matching due to a lack of the capability of tracking the matching status. JPStream [78] improves this by compiles a set of JSONPath queries and JSON syntax into a dual stack pushdown automaton which records both the parsing status and the matching status of queries.

One common limitation with the above methods is the "one-character-each-time" processing strategy. To deal with this limitation, Mison [86] proposes to build structural indices with bitwise operations, which can process tens of or even hundreds of characters simultaneously with the help of bitwise and SIMD-level parallelism. This idea originates from NoDB [32, 33, 76, 83], which builds structural index on raw CSV files and adaptively uses the index to load the data. In fact, simdjson [85] also adopts this bitwise processing in its first stage of JSON parsing. Besides JSON data, the idea of bitwise processing is also

seen in other contexts, including regular expression matching [45], XML parsing [89], as well as some database systems [30, 41, 101, 51]. Based on bitwise processing, Sparser [111] applies filtering before performing the parsing to further accelerate the processing.

Parallel Processing of Large Records. In addition to the exploration of bitwise and SIMD-level parallelism, it is also important to exploit coarse-grained parallelism (such as multicores) in the processing of individual large records. In fact, many efforts have been put into the parallelization of XML stream processing [96, 112, 122, 106, 79], including the use of hardware accelerators [100, 98]. The key in enabling parallel processing of individual XML record is "breaking" the dependences in the data processing. For example, in [96, 112], a pre-scan is applied to the XML record first to partition it according its high-level structures. In another work [106], the authors design parallel pushdown transducers that enumerate all the possible states at the beginning of an arbitrary XML partition to break the state dependences. To reduce the cost of state enumeration, GAP [79] leverages the XML grammar to prune infeasible states. By contrast, there are few studies in the parallelization of JSON stream processing. JPStream [78] adopts ideas from parallel XML processing [106, 79] to JSON processing so that an individual large JSON record can be effectively processed in parallel. This shares a similar goal with Pison.

Besides the above work, there are also prior studies that design speculative parallelization for parallel processing other types of raw data [135, 137, 136, 116, 124]. For example, HPar [135] proposes speculative parsing of HTML documents. In [137, 136, 116], FSMs are executed speculatively to perform pattern matching over unstructured textual data. Recently, Parparaw [124] leverages speculation techniques to process delimiter-separated

raw data in parallel, and PBS [114] speculatively parallelizes bitstream processing by modeling it into FSM computations. As far as we know, Pison is the first work that leverages speculation for parallel indexing of JSON data.

4.8 Summary

Constructing structural indices for JSON data has shown promises in accelerating JSON analytics, but its serial design makes it difficult to scale to large and complex JSON records. This work addresses this challenge by introducing intra-record parallelism and redesigning the structural indices construction process. It proposes an assembly of parallelization techniques that make it possible to construct the structural indices of individual JSON records in parallel. Evaluation on datasets collected from real-world applications show that the developed system − Pison, surpasses the performance of state-of-the-art tools, including simdjson and Mison, in both small-record and large-record processing scenarios.

Chapter 5

Streaming Semi-Structured Data with Bit-Parallel Skipping

Semi-structured data, like JSON, is the de facto standard for data communication on the Web and data representation in document-based data stores. Streaming analytics combines semi-structured data parsing and query evaluation into one pass to avoid generating any parse tree. Though promising, its conventional design requires to scan the data stream character by character to recognize all the syntactical structures, which fundamentally limits the efficiency of data streaming.

This work challenges the above design: is it necessary to scan the entire data stream in detail for query evaluation? In fact, based on the query and the data stream structures, this work finds a wide range of ***opportunities for skipping*** certain "irrelevant substructures" of the data stream. However, there is a dilemma − identifying the irrelevant substructures itself appears to need comprehensive scanning. To circumvent this "chicken-

egg" issue, an alternative yet faster way to identifying the irrelevant substructures is desired. For this purpose, this work presents a ***highly bit-parallel solution*** which intensively utilizes bitwise and SIMD operations to identify the irrelevant substructures during the streaming. The keys to this solution include a new streaming model − recursive-descent streaming, for easy adoption of skipping optimizations, an abstraction − structural intervals, for partitioning the data streams, and a group of bit-parallel algorithms for implementing different skipping cases. The whole solution is packaged into a JSON streaming framework, called JSONSki. It offers 15 bit-parallel skippers that can be naturally integrated into the streaming to dynamically "jump over" irrelevant substructures. Evaluation using real-world JSON datasets and standard path queries shows that JSONSki can achieve 19.0× speedup on average over the existing JSON streaming tool, and 7.7× speedup on average over a popular SIMD-based parser (simdjson).

5.1 Introduction

Recent years have seen a surge in adopting semi-structured data, like JSON (JavaScript Object Notation) and its variants, in the computing infrastructures, ranging from data transfer among loosely-coupled services in the cloud [35, 20, 48, 129] to the underlying data representation of popular document-based data stores, like MongoDB [99] and Firebase [61], to public data release through open APIs (e.g., data.gov [8]). Thus, the efficiency of semi-structured data processing is getting more critical to many modern software applications that require low data processing latency and strict memory budget.

```
{ "coordinates" : [
    40.74118764, -73.9998279
  ],
  "user" : {
    "id" : 6253282
  },
  "place" : {
    "name" : "Manhattan",
    "bounding_box" : {
      "type" : "Ploygon",
      "pos" : [
        [-74.026675, 40.683935], …
] } } }
```

Figure 5.1: Geo-referenced Tweet in JSON [128]

Figure 5.1 shows some attributes in a JSON Tweet object [1]: `coordinates`, `user`, and `place` [128]. Note that the attributes themselves could be objects or arrays. In general, there are two processing schemes for such semi-structured data:

- *Preprocessing-based scheme* preprocesses the data stream to build some in-memory data structures before extracting data of interests. For example, common JSON processing tools, such as RapidJSON [19], FastJSON [4], Gson [11], and simdjson [85], first parse the JSON data according to its grammar [10] to create an in-memory parse tree, then evaluate queries by traversing the tree. Though intuitive, this scheme suffers from a significant upfront delay due to the pre-parsing. Moreover, the parse tree occupies a chunk of memory that grows as the data stream becomes larger.

[1] The object is simplified and slightly modified for illustration purpose.

- *Streaming scheme* can naturally avoid the above issues by immediately consuming the semi-structured data stream, locating and extracting the data of interests on the fly. For example, JsonSurfer [21] and JPStream [78] evaluate path queries dynamically as they traverse the data stream without generating any in-memory trees. In this way, they only need to scan the data in one pass with a bounded memory cost.

Despite its promises, the conventional design of streaming scans the entire data stream character by character to recognize each token and each syntactical structure (e.g., objects and arrays), to evaluate the queries. This detailed way of scanning fundamentally limits the efficiency of streaming. In this work, we raise a key question in semi-structured data streaming: *is it possible and practical to dynamically "skip" certain parts of the data stream to accelerate the processing?*

Opportunities. Interestingly, we find that, by leveraging the query and the syntax of semi-structured data, certain data segments become "irrelevant" to the query evaluation, thus can be safely skipped. Consider the JSON object in Figure 5.1 as part of the data stream. Assume the query is to find the attribute `name` of object `place` (i.e., path query `$.place.name`), then the following skipping opportunities present:

- *Skip attributes of unmatched types.* First, the streaming can safely skip the *details* of the first attribute, including its name (`"coordinates"`) and value, because it is an array-type attribute, while the query looks for an object (based on that `place` has an attribute `name`).

- *Skip values of unmatched names.* Then, when it comes to the second attribute, though it is an object, its name (`"user"`) fails to match that in the query (`"place"`), therefore the streaming can skip its value − an object, entirely.

- *Skip remaining attributes after matching.* Finally, after the `"name"` attribute is also matched, the streaming can safely skip the following attributes of the current object `place` (i.e., `"bounding_box"`) because, according to its syntax, attributes in an object cannot share names, thus no matches are possible in these remaining attributes.

The above cases are for queries involving objects. Similar skipping opportunities also exist for queries involving arrays. In this work, we systematically explore the above skipping opportunities and categorize them into basic groups.

However, without a priori knowledge about the input stream, a naïve way of implementing the above skipping techniques still needs to traverse the entire data stream in detail in order to find out all the skipping opportunities (e.g., identifying an attribute and its type) and to mark the boundary of each data segment (e.g., an object) to be skipped. These circular dependences essentially create a "chicken-egg" problem.

Practicality. To address the practical aspect of skipping, an alternative yet faster solution to skipping is desired. For this purpose, we introduce *bit-parallel skipping* which intensively utilizes bitwise and SIMD operations to identify the skipping opportunities and perform the skipping. The key components include i) a new streaming model for easy integration of the skipping optimizations − *recursive-descent streaming*, ii) an abstraction for partitioning the data stream into basic logical units − *structural intervals*, and iii) an assembly of *bit-parallel algorithms* for implementing different skipping scenarios.

First, recursive-descent streaming decouples the parsing logic from the existing automaton-based streaming model with the help of a recursive-descent parser. This new design exposes the skipping opportunities in a more intuitive way, and also makes it easier to integrate the skipping optimizations thanks to the use of recursive functions. Second, by dividing the data stream into structural intervals, the complexity in handling the nesting structures can be greatly reduced. This enables a batched, bracket/parenthesis counting-based strategy to identify the nesting levels and the boundaries of irrelevant substructures. Finally, based on this strategy, the skipping algorithms naturally expose bit-parallelism, hence they can effectively take advantage of the bitwise and SIMD operations that are prevalent on modern architectures.

Thanks to the bit-parallel skipping, the streaming no longer needs to scan the characters one by one, instead, it can process a word-size of characters all at once (e.g., 64 characters on a 64-bit machine). In fact, some recent work, like simdjson [85], Mison [86], and Pison [77], also leverages the bit-parallelism in semi-structured data processing. However, their uses are limited to locating the metacharacters in JSON data stream and they all belong to the preprocessing-based scheme.

Based on the above proposed techniques, this work presents a new streaming framework for JSON-style data − JSONSki. It features 15 bit-parallel skipping primitives (called skippers), that can be naturally integrated into the streaming for faster processing. To demonstrate its efficiency, this work compares JSONSki with a group of state-of-the-art JSON processing tools, including both streaming and preprocessing-based ones, as well as those supporting bit-parallelism. With real-world datasets and standard path queries, the

evaluation shows that JSONSki on average achieves 19.0× speedup over a state-of-the-art streaming library, JPStream [78], and 7.7× speedup over the popular SIMD-based parser, simdjson [85].

Contributions. To summarize, this work makes the following key contributions to semi-structured data processing.

- First, it reveals the opportunities of skipping data segments during the semi-structured data streaming (Section 5.3).
- Second, it introduces *recursive-descent streaming*, a new streaming model for easy adoption the skipping ideas, and a group of bit-parallel skipping algorithms based on the concept of *structural intervals* (Section 5.4).
- Finally, this work systematically compares the proposed solution with several state-of-the-art semi-structured data processing tools and confirms its efficiency (Section 5.5).

Next, we provide more background of this work.

5.2 Background

We first introduce the basics of semi-structured data, then present the details of the two basic processing schemes.

Semi-Structured Data. Unlike structured data, such as tables in relational data stores, semi-structured data do not have to follow a strict schema. In fact, the data itself carries its type information, structures, and values. This makes them more flexible in organizing the data. In general, semi-structured data have two flavors: i) tag-based semi-structured

```
    object  ::=  { } | { attributes }
attributes  ::=  attribute  |  attribute, attributes
 attribute  ::=  attribute-name :  value
     array  ::=  [ ] | [ elements ]
  elements  ::=  element  |  element, elements
   element  ::=  value
     value  ::=  object | array — primitive
```

Figure 5.2: JSON Grammar

data, like XML, and ii) programming language-based semi-structured data, like JSON. The former carries data in a hierarchy of pair tags, such as `<id>6536</id>`. XML was popular in the early days of web development. However, recent trends [129, 27] show that they are getting replaced by the JSON-style data.

JSON, originated from JavaScript language, uses a compact data representation. Figure 5.2 shows in its grammar. It has two basic structures: an *object* − a list of attributes enclosed by a pair of parentheses, such as {`"id":653, "name":"John"`}, where an attribute name and its value are separated by a colon, and an *array* − an ordered list of elements enclosed by a pair of brackets, like `[653, "John"]`. Moreover, an attribute value or an array element itself can be an object or array. Together, they can be nested recursively to form a complex multi-level structure, like the Tweet object in Figure 5.1. More details of JSON syntax can be specified in its standard [10]. We refer to the root-level object or array as a *JSON record*. Depending on the scenarios, a JSON data stream may consist an individual bulky JSON record, or a sequence of (small) JSON records.

Querying Semi-Structured Data. Semi-structured data can be queried via *path expressions*, like `$.place.name`, each of which specifies a path from the root of a JSON record to

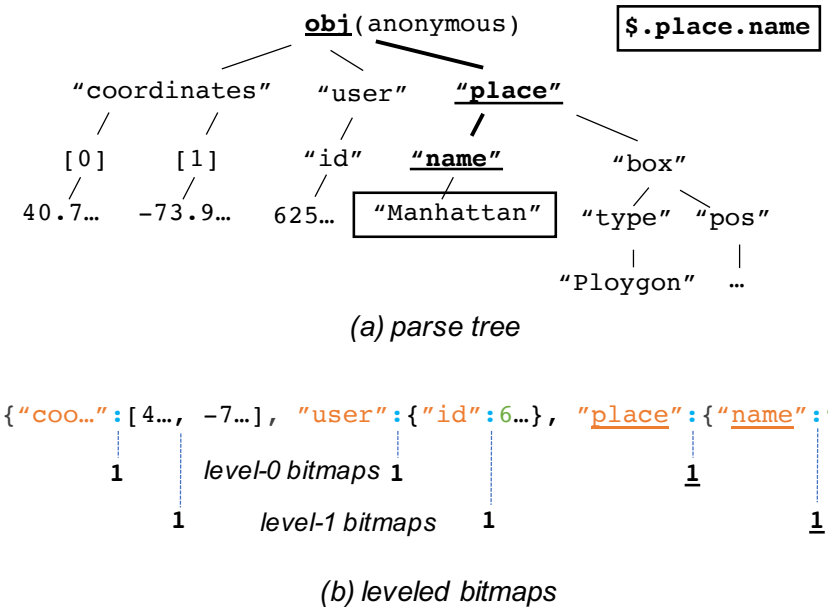

Figure 5.3: Preprocessing-based Query Evaluation

one or multiple object attributes. Since the root (an object or an array) is anonymous, a path query uses a `$` sign to refer to it. For array-type attributes, the query may carry index constraints. For example, `$.places[0:2]` refers to the first two elements in the `places` array. To select all array elements, wildcard $*$ should be used, like `$.places[*]`. More details about path queries are available in [74].

As mentioned earlier, to evaluate JSON path queries, there are *preprocessing-based* and *streaming* schemes.

Preprocessing-based Scheme. This scheme first constructs an in-memory data structure, like a parse tree, for a JSON record, then traverses the tree top-down to evaluate the path queries. Figure 5.3-(a) illustrates the parse tree and query matching for the running

example. Most existing JSON processing tools fall into this category, such as JSON-C [12], RapidJSON [19], FastJSON [4], Gson [11], simdjson [85], and among others.

Instead of generating a parse tree, Mison [86] and Pison [77] build indices for metacharacters at different levels of a JSON record, called *leveled bitmaps*. For each level in the record, two bitmaps are created: a *colon bitmap* for accessing the attribute, and a *comma bitmap* for accessing the array elements. Figure 5.3-(b) shows the positions of 1s in the bitmaps for the top two levels of the JSON record. Based on them, the evaluation can quickly locate the attribute names (appearing before colons of current level) or array elements (appearing before commas of current level), and match them against those in the query.

Regardless which in-memory data structure is constructed, preprocessing the data stream introduces a significant delay upfront. Moreover, the constructed in-memory data structure itself may consume a substantial portion of memory, especially for data stream with large records. These limitations seriously restrict the use of preprocessing-based scheme for applications that are sensitive to latency and memory consumption.

Streaming Scheme. This scheme can avoid the limitations of preprocessing-based scheme by combining the conventional parsing and the path query evaluation into a single pass [21, 78]. It immediately consumes the data as the stream is being traversed and its structures are being recognized. The key to this design is maintaining two stacks simultaneously: a *query stack* for tracking the matching status at different levels of a JSON record and a *syntax stack* for recognizing the syntactical structures of JSON [78]. Figure 5.4 illustrates the basic idea of streaming evaluation using the example JSON query and data. The state machine

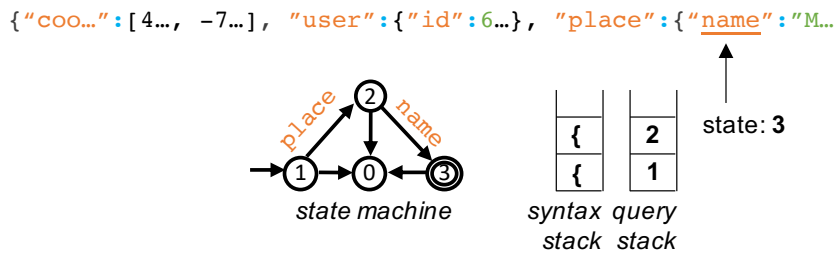

Figure 5.4: Streaming Query Evaluation

captures the two attribute names to match in the query. The syntax stack shows that the evaluation is currently under the second-level nested object. The query stack shows that it has reached state 2 at the first level. After the token `"name"` is consumed, the current state becomes 3, which is an accept state, indicating a match.

Streaming evaluation directly consumes data in the cache − no writing back to the memory (except for outputs), thus, its efficiency tend to be higher than the preprocessing-based scheme, which constructs some data structures in the memory. However, the existing designs for streaming [21, 78] need to recognize every token in the data stream, based on which they update the state and stacks. This token-by-token processing limits the efficiency of streaming evaluation.

In the following, we will present the ideas of skipping which eliminates the needs of comprehensive scanning.

5.3 Streaming with Skipping

This section first introduces a new basic streaming model, then presents the skipping ideas under this model.

5.3.1 Basic Streaming Model – A Decoupled Design

As mentioned earlier, existing work [78] models the streaming evaluation of path queries over a JSON record as a dual-stack automaton (Figure 5.4). One stack (query stack) tracks the query matching progress at different levels of a record, while the other stack (syntax stack) recognizes the syntactical structures. An automaton manages the two stacks cooperatively based on a set of transition rules. Though being formal and rigorous, this model is not ideal for the skipping optimizations. First, it would be less intuitive to find the skipping opportunities as the parsing logic is encoded in the transition rules. Moreover, as shown later, to make it practical, the skipping optimizations will not follow token-by-token processing, thus making them unsuitable to be expressed as formal transition rules.

For the above reasons, we introduce an alternative design of the streaming model which is more amenable to the skipping optimizations. The key idea is to decouple the parsing logic from the streaming automaton and employ a *recursive-descent parser* to drive the query matching automaton. The new design brings two major benefits: First, it simplifies the transition rules as only the query stack needs to be maintained explicitly; the syntax stack becomes a call stack which is automatically managed by the operating system; Second, the use of recursive functions makes it easier to program the skipping logic into the parsing procedure. Next, we describe it major components: a *query automaton* and a *recursive-descent streaming* strategy.

Query Automaton. A path expression, like `$.place.name`, can be converted into a pushdown automaton (a state machine plus a stack) to track its matching progress [68, 106, 78]. The idea is to treat a path expression like a regular expression, so that the matching

progress can be captured by a finite state machine. For example, Figure 5.4 shows a four-state machine for the example path expression, where state 1 is the starting state, state 3 is the accept state, and state 0 is the unmatched state. Unlike regular expressions, strings in a path expression need to be matched according to their levels in the data record, therefore, a stack is also needed for holding the current state at each level. For queries with array index constraints (like `pos[2:4]`), a counter will be associated with the state at the corresponding level to match the specified range. More details regarding this conversion can be found in prior work [78].

Thanks to the decoupling, the pushdown automaton only consumes five types of tokens, rather than eight types as in the dual-stack streaming model [78]: K, V, `[`, `]`, and `,`, where K is an attribute name, like `"place"`, V is an artificial token that symbolically represents all possible values of an attribute, and the other three are metacharacters in JSON. Note that the comma tokens are only those used to separate array elements. As shown later, all the other occurrences of comma will be filtered out by the recursive-descent parser.

For each type of tokens, there is a transition rule, as listed in Figure 5.5, where each rule is in the format:

$$\Delta(state, counter, token, stack) \rightarrow (state, counter, stack)$$

Rule `[Key]` says, when an attribute name K is consumed, the automaton pushes the current state q onto the stack (stack elements are separated by colons), then updates the current state based on the finite state machine transition $\delta(q, K)$. Rule `[Val]`, on the other hand, pops the stack top q' and uses it for the current state, upon receiving a value

$$
\begin{array}{rl}
\texttt{[Key]} & \Delta(q, c, K, *) \rightarrow (\delta(q, K), c, q : *) \\
\texttt{[Val]} & \Delta(q, c, V, q' : *) \rightarrow (q', c, *) \\
\texttt{[Ary-S]} & \Delta(q, c, \texttt{[}, *) \rightarrow (\delta(q, \texttt{[}), 0, qc : *) \\
\texttt{[Ary-E]} & \Delta(q, c, \texttt{]}, q'c' : *) \rightarrow (q', c', *) \\
\texttt{[Com]} & \Delta(q, c, \texttt{,}, *) \rightarrow (q, c + 1, *)
\end{array}
$$

Figure 5.5: Transition Rules of Query Automaton

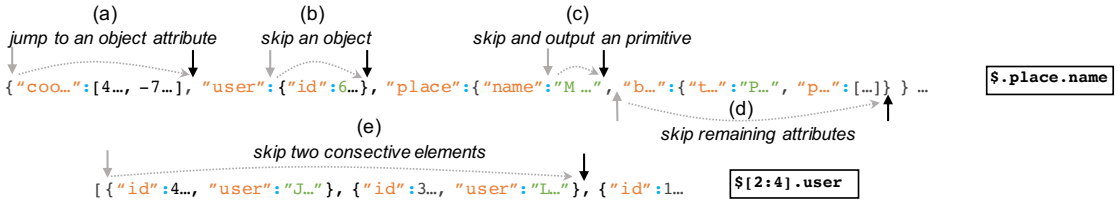

Figure 5.6: Examples of Basic Skipping Cases

token V. Rules `[Ary-S]` and `[Ary-E]` work in a similar way, when brackets are consumed, except for two differences: First, when `[` is consumed, `[Ary-S]` pushes the current counter along with the current state onto the stack, then it resets the counter to zero, indicating the start of a new array. Correspondingly, when `]` is consumed, both the state and counter are reassigned with those values from the stack top. Finally, upon consuming a comma (in an array), rule `[Com]` simply increments the counter.

Note that the existing dual-stack streaming automaton needs 13 transition rules [78] as it encodes both the query matching and parsing logic. Next, we explain how the query automaton can be driven by a recursive-descent parser.

Recursive-Descent Streaming. As a classic type of top-down parsing algorithm, recursive-descent parsing is known for its uses of recursive functions to simplify the parser design [93].

Algorithm 9 Recursive-Descent Streaming

```
1:  function object()
2:      consume("{")
3:      while hasMoreAttributes() do
4:          attribute = getAttributeName()
5:          consume(":")
6:          type = getAttributeType()
7:          if matched == true then   /* already in matched status */
8:              switch type do
9:                  case "object": output(attribute + ":" + object())  break
10:                 case "array":  output(attribute +":" + array())  break
11:                 case "primitive": output(attribute + ":" + primitive())
12:         else  /* not yet matched */
13:             qa.transition(attribute)  /* update query automaton */
14:             if qa.status() == ACCEPT then matched = true
15:             switch type do
16:                 case "object": object()  break
17:                 case "array": array()  break
18:                 case "primitive":  primitive()  /* consume a primitive */
19:             matched = false
20:             qa.transition(V)  /* V: a symbolic representation for values */
21:     consume("}")
```

The basic idea is to create a function for each non-terminal in the grammar, whose job is to recognize the structures of this non-terminal. For non-terminals that are recursively defined, their functions naturally become recursive functions.

In the case of JSON grammar (see Figure 5.2), there are two major non-terminals: `object` and `array`, after inlining all other non-terminals. Algorithm 9 presents the two functions: `object()` and `array()`, respectively [2]. Both functions consist of recursive calls to themselves and to each other. Thanks to the use of recursion, the two functions are

[2]Due to limited space, the pseudocode of `array()` is separated into Algorithm 10.

Algorithm 10 Recursive-Descent Streaming (Continue from Algorithm 9)

```
1:  function array()
2:      consume("[")
3:      qa.transition("[")
4:      if notEmpty() then
5:          while true do
6:              if qa.status() == ACCEPT then matched = true
7:              type = getElementType()
8:              switch type do
9:                  case "object": object()  break
10:                 case "array": array()  break
11:                 case "primitive":  primitive()
12:             matched = false
13:             if hasMoreElements() then
14:                 consume(",")
15:                 qa.transition(",")
16:             else break
17:     consume("]")
18:     qa.transition("]")
```

straightforward to follow − they recognize the structures of objects and arrays simply following their grammar rules. To achieve streaming query evaluation, we embed the query automaton transitions into the recursive descent parsing, which are highlighted (in blue) in Algorithm 9 and Algorithm 10. In `object()`, the two statements at Line 13 and 20 correspond to the `[Key]` and `[Val]` transitions in Figure 5.5, respectively. They are executed before and after an attribute value is recognized (Line 15-18). Similarly, in `array()`, the two statements at Line 3 and 18 correspond to the `[Ary-S]` and `[Ary-E]` transitions. They are executed when an array starts and ends, respectively. Finally, the statement at Line 15 corresponds to the `[Com]` transition and it is executed after each comma in the array is

encountered. Besides the above transitions, to output matches, the parser also checks the query automaton for the matching status (at Line 14 in `object()` and Line 6 in `array()`).

We name the above approach *recursive-descent streaming*. Next, we present the skipping ideas under this new model.

5.3.2 Opportunities for Skipping

Under the basic streaming model, the data stream is processed token by token, and every token needs to be recognized and consumed. Moreover, these tokens are fed into the query automaton to trigger transitions. To avoid this comprehensive streaming, we find that, the query (a path expression) often carries "hints", when intelligently leveraged, along with the syntax of semi-structured data, they can enable various opportunities for skipping data segments, without compromising the integrity of the results.

One of the "hints" is the *data type*, which can be inferred from the query, but has not yet been leveraged by the basic streaming model. For example, from `$.place.name`, we can infer that `place` is an object, because it has an attribute called `name`. By contrast, expression `$.places[2:4].name` implies that `places` is an array. In this way, we can obtain the types for all attribute names in the path expression, except for the last (the inner-most), which can be of any type. Based on such type information, the streaming may skip data segments with unmatched types. Besides this, the streaming may also leverage other query information, such as attribute names (e.g., `place`) and index constraints (e.g., `[2:4]`) to skip irrelavent syntactical structures in the data stream.

Next, we summarize these skipping opportunities into five groups based on their intuitions. Assume that a global variable, `pos`, is maintained during the streaming, which

marks the current position of streaming in the data stream. The skipping effects can be achieved by advancing `pos`.

G1: Jump to a Type-Specific Attribute/Element. As shown in JSON grammar (see Figure 5.2), an object consists of a sequence of attributes, where each attribute carries a value that could be an object, an array, or a primitive. Based on the attribute type of interest (extracted from the query), we can skip all type-unmatched attributes, and directly jump to the attribute of the matched type. As shown in Figure 5.6-(a), based on the type information, the streaming skips the first attribute − an array, and directly jumps to the second attribute − an object.

Similar scenarios also occur when the streaming traverses an array with heterogeneous-typed elements, it may directly jump to the element of matched type. However, for queries with index constraints, we need to make sure the skipping does not go beyond the specified index range.

G2: Skip an Unmatched Attribute Value. For attributes with matched type, the streaming further extracts the attribute name and feeds it to the query automaton for updating the matching status. Note that this attribute name extraction (tokenization) cannot be skipped. But, if the query automaton fails to make matching progress − the attribute name leads the automaton to the `unmatched` state (e.g., state `0` in Figure 5.4), it would be unnecessary to further examine its attribute value. Depending on the type of the value, the streaming may skip an object, an array, or simply a primitive. Figure 5.6-(b) presents such an example, where an object is skipped as its name `"user"` fails to match the `"place"` in the query.

G3: Skip and Output a Matched Value. Consider the scenario where the query automaton has reached the `accept` state, so it should output the attribute value or the array element. In this case, the streaming may perform skipping and outputting together, as long as it can find the end position of the value. This avoids examinating the details of the output. Figure 5.6-(c) shows such a case where the query is fully matched, so the streaming skips and outputs its value entirely.

G4: Skip Remaining Attributes after Matching. Within an object, once an attribute name is matched, the state of the automaton would be `in-progress` or `accept`. In this case, the streaming may skip all following attributes of this object, as the JSON syntax prohibits an object from owning duplicated attribute names, meaning no matches could be found in these remaining attributes. Figure 5.6-(d) illustrates such a case, where the attribute that appears after `name` is skipped.

G5: Skip Out-of-Range Elements. Finally, some queries may carry array index constraints, typically a range, such as `[2:4]`. These constraints expose opportunities to skip consecutive array elements that are out of the specified range. As shown in Figure 5.6-(e), the streaming skips the first two array elements as the query asks for matches between the third and forth array elements (i.e., `[2:4]`). Similarly, skipping is also possible after the end of the specified range is reached.

Table 5.1 lists the skipping cases discussed above as functions that can be called to achieve the corresponding skipping effects. We refer to these functions as *skippers*. Before showing their algorithmic details (in Section 5.4), we first illustrate how they can be naturally integrated into the streaming.

Table 5.1: Five Groups of Skippers

G1	goToObjAttr()	jump to an object attribute
	goToAryAttr()	jump to an array attribute
	goToObjElem()	jump to an object element
	goToAryElem()	jump to an array element
	goToObjElem(K)	jump to an object within K elements
	goToAryElem(K)	jump to an array within K elements
G2	skipObject()	skip an object
	skipArray()	skip an array
	skipPrimitive()	skip a primitive
G3	outputObject()	skip and output an object
	outputArray()	skip and output an array
	outputPrimitive()	skip and output a primitive
G4	skipLeftAttrs()	skip all remaining attributes
G5	skipElems(K)	skip K consecutive array elements
	skipLeftElems()	skip all remaining array elements

5.3.3 Integration of Skippers

Due to space limits, we use the `object()` function from the recursive-descent streaming (see Algorithm 9) as an example to illustrate the integration of skippers. Algorithm 11 shows the new `object()` function after the skipper integration.

Comparing to the original `object()`, there are four places where skippers are integrated. First, it extracts the expected type from the query automaton (Line 3), based on which, it invokes the corresponding skipper to jump to the type-matched attribute directly (Line 5-8). Note that it is possible that the object contains no more type-matched attributes. In this case, the skipping reaches the end of the object (Line 9-10). Second, after the query automaton takes the attribute name (Line 13), it starts to process its value (Line 14). Here, the pseudocode of `process_value()` function is shown in Algorithm 12 (continue from Algorithm 11). Basically, it checks the matching status (Line 1). If the last

Algorithm 11 Streaming with Skipping (Partial)

```
1:  function object()
2:      consume("{")
3:      type_expected = qa.typeExpected()  /* qa: query automaton */
4:      while hasMoreAttributes() do
5:          if type_expected != "unknown" then
6:              switch type_expected do
7:                  case "object":  result = goToObjAttr()  break
8:                  case "array":  result = goToAryAttr()
9:              if result == OBJECT_END then  /* has reached object end */
10:                 consume("}") return
11:         type = getAttributeType()
12:         attribute = getAttributeName()
13:         qa.transition(attribute)  /* consume attribute name */
14:         matched_flag = process_value(qa)
15:         if matched_flag == MATCHED then
16:             break while-loop
17:     if hasMoreAttributes() then
18:         skipLeftAttrs()
19:     consume("}")
```

transition leads to an `unmatched` state, then the corresponding skipper would be called to skip the attribute value (Line 4-7). Otherwise, if the last transition leads the automaton to the `accept` state, then the outputting-related skippers would be called to output the values without examining their details. Finally (back to `object()` in Algorithm 11), if the query automaton just made matching progress (Line 15-16), the function would stop examining the following attributes of this object in the future (Line 17-18) by calling skipper `skipLeftAttrs()`.

Algorithm 12 Streaming with Skipping (Partial) (Continue from Algorithm 11)

```
1:  function process_value(qa)
2:      if qa.status() == UNMATCHED then
3:          switch type do
4:              case "object":  skipObject()  break
5:              case "array":  skipArray()  break
6:              case "primitive":  skipPrimitive()
7:              matched_flag = UNMATCHED
8:      else if qa.status() == ACCEPT then
9:          switch type do
10:             case "object":  outputObject()  break
11:             case "array":  outputArray()  break
12:             case "primitive":  outputPrimitive()
13:             matched_flag = MATCHED
14:     else  /* in-progress */
15:         switch type do
16:             case "object":  object()  break
17:             case "array":  array()  break
18:             case "primitive":    /* impossible */
19:         matched_flag = MATCHED
20:     qa.transition(V)  /* V: a symbolic representation for values */
21:     return matched_flag
```

In a similar way, the skippers can be integrated into the other function `array()` in the basic recursive-descent streaming. As mentioned earlier and demonstrated here, the use of recursive functions, instead of automaton transition rules, makes it much more intuitive to adopt the skipping optimizations.

So far, we have present the ideas of skipping under the new streaming model, without showing any implementation details. In fact, how to achieve these skipping goals is a key challenge by itself, which we will address in the next section.

5.4 Bit-Parallel Skippers

Though the ideas of skipping sound attractive, it is non-trivial to make them work effectively in practice. A key question in achieving the skipping goals (e.g., skipping an object) is:

how can the target position of skipping be identified?

Conventionally, this question itself is a parsing problem, thus requires scanning every character and generating every token. If the skippers are implemented in this way, the only benefit they bring is bypassing some query automaton transitions as they are not needed during the skipping. The main cost of streaming − comprehensive scanning, still remains. To address this dilemma, we need an alternative yet much faster solution to implement the skippers. In this section, we show a highly bit-parallel solution to implement the skippers, which bypasses the conventional detailed scanning. The keys to this solution include an abstraction for partitioning of the semi-structured data stream into basic logic units, namely, *structural intervals*, and an assembly of counting-based algorithms for identifying the nesting levels and boundaries of objects and arrays.

5.4.1 Structural Intervals

Designing bit-parallel algorithms is notoriously challenging due to the low-level bit-manipulations. This is compounded by the recursive and nested structures of the data stream. To reduce the design complexity, we propose to partition the data stream into some basic data segments.

Definition 7 *Given a metacharacter of interest, denoted as α, a* ***structural interval*** *for α is a consecutive sequence of characters located between the current streaming position* `pos` *(inclusive) and the following closest α (exclusive).*

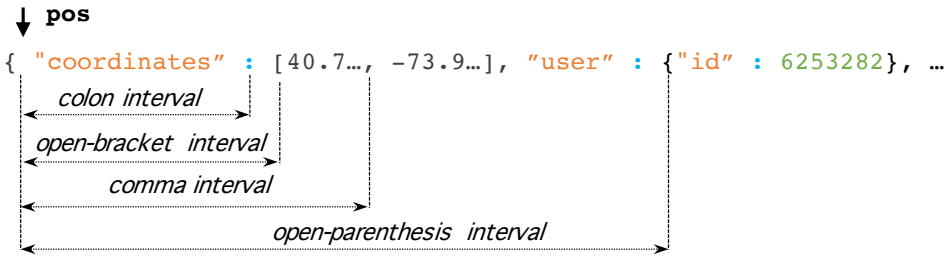

Figure 5.7: Examples of Structural Intervals

Depending on the metacharacter of interest, there could be different kinds of structural intervals (see Figure 5.7). Note that pseudo-metacharacters inside strings should not be counted.

A critical property of structural intervals is that they can be *constructed and accessed efficiently with bit-parallelism.*

Construction. Given the current streaming position `pos`, and a metacharacter of interest α, the structural interval constructor builds an *interval bitmap* for characters in the data stream, and only the bits within the interval are set to 1s, as illustrated by the open-bracket interval bitmap in Figure 5.8. By default, the size of an interval bitmap is the same as a word, denoted as W. For example, on a 64-bit machine, the bitmap consists of 64 bits, representing 64 consecutive characters in the data stream. However, an interval bitmap may span multiple words, as shown by the open-parenthesis interval bitmap in Figure 5.8. In this case, the constructor builds a *partial interval bitmap* for each W characters overlapped with the interval. Note that a partial interval bitmap should be constructed after the prior one has been used and destroyed. The constructor should not buffer them to create a whole

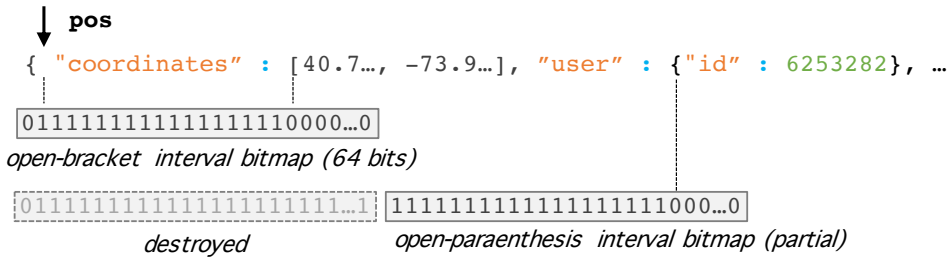

Figure 5.8: Example of Interval Bitmap

interval bitmap, which in theory can be arbitrarily long, thus contradicting the streaming design.

Function `builtInterval(pos, char)` from Algorithm 13 describes the construction process. First, it builds a bitmap for the given metacharacter (Line 3). The basic idea [86, 85, 77] is shown by function `builtMetacharBitmap(char)`. Here, we need to remove all the pseudo-metacharacters. To achieve this, a string bitmap is created (Line 17), then a logic `AND` is taken between the string bitmap and the raw metacharacter bitmap (Line 20). After this, the constructor marks the start position (Line 4), resets bits up to the start position to 0s (Line 5-6), marks the end position (Line 7), and finally generates the interval bitmap by taking a bitmap subtraction (Line 8).

Besides the above construction, Algorithm 14 presents an alternative constructor, `nextInterval(char)`, which is for faster construction of a sequence of structural intervals. Also, Algorithm 14 shows a function for quickly obtaining the end position of an inter-

Algorithm 13 Structural Interval Construction and Access

```
 1:  /* Build an interval bitmap for a metacharacter */
 2: function buildInterval(pos, char)
 3:     bitmap = buildMetacharBitmap(char)
 4:     b_start = 1 << pos  /* mask start position */
 5:     mask_start = b_start ^ (b_start – 1)  /* mask bits up to start */
 6:     bitmap = bitmap & ~mask_start  /* reset bits up to start to 0s */
 7:     b_end = bitmap & –bitmap  /* mask end position */
 8:     b_interval = b_end – b_start  /* create the interval bitmap */
 9:     return b_interval
10:
11:  /* Construct the bitmap for a given metacharacter */
12: function buildMetacharBitmap(char)
13:     if bitmap_char != null then  /* to avoid rebuilding */
14:         return bitmap_char
15:      /* construct a bitmap to mask all the characters inside strings */
16:     if bitmap_string == null then
17:         bitmap_string = buildStringBitmap()
18:     bitmap_char = buildRawCharBitmap(char)
19:      /* remove metacharacters in strings */
20:     bitmap_char = bitmap_char & bitmap_string
21:     return bitmap_char
```

val, `intervalEnd(interval)`. It uses the leading-zero counting instruction to locate the mirrored end position [3], then converts it to the actual end position.

In the next section, we will show how the structural intervals can be leveraged to facilitate the skipping algorithm design.

[3] In fact, a metacharacter bitmap is always created in mirrored way [86].

Algorithm 14 Structural Interval Construction and Access (Continue from Algorithm 13)

```
1:  /* Build an interval bitmap between first two 1s in a bitmap */
2:  function nextInterval(char)
3:      bitmap = buildMetacharBitmap(char)
4:      b_start = bitmap & –bitmap  /* get rightmost 1 */
5:      bitmap = bitmap & (bitmap – 1)  /* remove rightmost 1 */
6:      b_end = bitmap & –bitmap  /* get rightmost 1 again */
7:      b_interval = (b_end – b_start)  /* create the interval bitmask */
8:      return b_interval
9:
10:  /* Get the position of the end of an interval (i.e., rightmost 1) */
11:  function intervalEnd(interval)
12:     pos = lzcnt(interval)  /* position of leftmost 1 */
13:     pos = wordSize() – pos  /* mirror the position */
14:     return pos
```

5.4.2 Skipping Algorithms

We start with the main design ideas, then present the algorithm in detail for each group of skippers.

Main Design Ideas. The basic question in skipping is to find the target position. For example, to skip an object, we need to locate the close parenthesis of the object that pairs with its open parenthesis. To avoid character-by-character scanning, skippers process the data stream *interval by interval*. First, they convert every W characters from the data stream into some relevant interval bitmaps (see Section 5.4.1). Then, based on these bitmaps, they use a *counting-based pairing strategy* to locate the close parenthesis/bracket of an object/array. The strategy is backed up by the following property:

Lemma 8 *In a JSON object, assume α and β are two closest open parentheses (marking an open-parenthesis interval), and the number of close parentheses between them is n_{close}.*

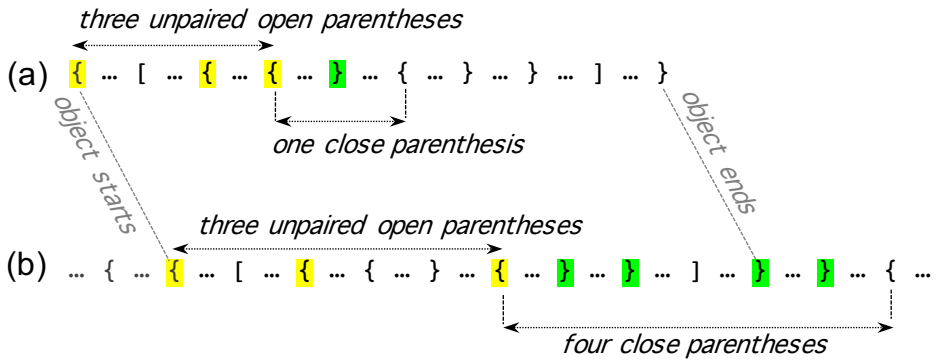

Figure 5.9: Examples of Pairing Property

Also assume the number of unpaired open parentheses before α (including α) is n_{open}. Then, we have $n_{close} < n_{open}$. The same conclusion holds for JSON arrays.

Figure 5.9-(a) illustrates the above property. Between the two closest open parentheses marked on the bottom, there is one close parenthesis − no more than the number of unpaired open parentheses up to the first open parenthesis (i.e., three).

However, when a JSON object is part of the data stream, the property must *fail* to hold when the object ends. As illustrated in Figure 5.9-(b), there are four close parentheses between the last two open parentheses − more than the number of unpaired open parentheses from the object start position up to the first one of the two open parentheses (i.e., three). This means the object ends between the last two open parentheses. In fact, the object ends at the third close parenthesis − the same as the number of unpaired open parentheses n_{open}. This is also true when a JSON array is part of the data stream.

Based on the above observation, we present the algorithms for skipping a type-specific value (G2), then we expand their ideas to other skipper groups. For easy recognition, bit-parallel operations are highlighted in blue in the pseudo-code.

G2: Skip a Type-Specific Value. Take the skipper for skipping an object `skipObject()` as an example. Algorithm 15 shows its basic steps. First, it consumes the open parenthesis of the object and sets the number of unpaired open parentheses to 1 (Line 2-3). Then, it traverses the open parenthesis intervals. For each interval, it builds the close parenthesis bitmap (Line 9) to count the number of close parentheses in this interval (Line 10-11). Based on that, it updates the number of unpaired open parentheses, by subtracting the number of paired ones, plus the new one that appears at the beginning of this interval (Line 13). If there are more close parentheses than those to be paired, the object must end in this interval, and its close parenthesis is the `num_{`-th one (Line 15), whose position is exactly the target of this skipping (Line 16).

The algorithm for skipping an array `skipArray()` is the same as `skipObject()`, except that all the occurrences of parentheses are replaced with the corresponding brackets.

By contrast, the algorithm for skipping a primitive is simpler. As described in `skipPrimitive()` in Algorithm 16, it first builds a comma interval (Line 2), then collects the close parentheses within this interval (Line 3-4). If no close parenthesis exists in this interval, it simply goes to the end of the interval; otherwise, it jumps to the position before the object ends (Line 5-7).

Algorithm 15 G2: Skip a Type-Specific Value

```
1:  function skipObject()
2:      consume("{")
3:      num_{ = 1   /* number of unpaired open parentheses */
4:      while true do
5:          if interval == null then
6:              interval = buildInterval(pos,"{")
7:          else
8:              interval = nextInterval("{")
9:          bitmap_} = getMetacharBitmap("}")
10:         bitmap_} = bitmap_} & interval
11:         num_} = __popcnt(bitmap_})  /* count "}"s in this interval */
12:         if num_} < num_{ then
13:             num_{ = num_{ − num_} + 1
14:         else  /* enough or more "}"s found */
15:             pos = getPosition(bitmap_}, num_{)
16:             goto pos  /* position of "}" that ends to-be-skipped object */
```

G1: Jump to a Type-Specific Attribute/Element. With G2 skippers, we can compose the algorithms for G1 skippers.

Algorithm 17 shows the `goToObjAttr()` function. To jump to an object attribute, the algorithm traverses the attributes one by one, and checks their types. For each attribute, it builds a colon interval to quickly go to its colon (Line 3-4), so that it can directly check the type of its value without extracting the attribute name (Line 5). In cases the value type is unmatched, the algorithm calls corresponding skippers to skip the value.

The other G1 skippers can be implemented in a similar way, except that for jumping to a type-specific array element, there is no need to construct the colon interval, and for queries with index constraints, the jumping should track a counter to avoid reaching an out-of-range element. In addition, Algorithm 17 shows an enhanced skipper `skipPrimAttrs()`,

Algorithm 16 G2: Skip a Type-Specific Value (Continue from Algorithm 15)

```
1: function skipPrimitive()
2:     interval = buildInterval(pos, ",")
3:     bitmap_} = getMetacharBitmap("}")
4:     bitmap_} = bitmap_} & interval
5:     if bitmap_} == 0 then /* current object does not end */
6:         goto intervalEnd(interval)
7:     goto getPosition(bitmap_}, 1) − 1
```

for skipping a sequence of primitive attributes or elements together, The idea is to go as far as the next open parenthesis and open bracket (Line 12), and if the object/array has not ended (Line 15), it has skipped these primitive attributes (elements); otherwise, it goes to the end of the object/array (Line 18).

G3: Skip and Output a Matched Value. G3 skippers are very similar to G2 skippers, except that they also need to output the contents that they skipped. We can easily embed the outputting statements into the skipping at the interval level.

G4: Skip Remaining Attributes after Matching. Though for a different scenario, function `skipLeftAttrs()`, in fact, can be implemented like `skipObject()`, because both of them want to reach the end of an object. The only subtle difference is that `skipLeftAttrs()` occurs inside an object (between attributes), while `skipObject()` happens before an object. By removing the `consume("{")` at Line 2 in Algorithm 15, we get the algorithm for `skipLeftAttrs()`.

G5: Skip Out-of-Range Elements. The algorithms for G5 skippers also resembles some of the above skippers. Function `skipElems(`K`)` skips K consecutive elements regardless of their types, which can be achieved by traversing the comma intervals with an

Algorithm 17 G1: Jump to A Type-Specific Attribute/Element

```
1:  function goToObjAttr().  /* Jump to the next object attribute */
2:      while hasMoreAttributes() do
3:          interval = buildInterval(pos, ":")
4:          pos = intervalEnd(interval) + 1
5:          type = getAttributeType()
6:          switch type do
7:              case "primitive": skipPrimAttrs()  break
8:              case "object":  break while-loop
9:              case "array":  skipArray()
10:
11: function skipPrimAttrs()  /* Skip consecutive primitive attributes */
12:     interval = buildInterval(pos, "{", "[")
13:     bitmap_} = getMetacharBitmap("}")
14:     bitmap_} = bitmap_} & interval
15:     if  bitmap_} == 0 then  /* object has not ended */
16:         goto intervalEnd(interval)
17:     else
18:         goto getPosition(bitmap_}, 1)
```

index counter. For each element encountered, a corresponding skipper is called. Function `skipLeftElems()` is similar to the function `skipLeftAttrs()`, with parenthesis bitmaps replaced by bracket bitmaps.

So far, we have presented the algorithms for all the five groups of skippers listed in Table 5.1.

5.5 Evaluation

This section evaluates the proposed streaming with bit-parallel skipping and compares it with existing methods.

Table 5.2: Methods in Evaluation

JPStream	A state-of-the-art JSON streaming library [78]
RapidJSON	A popular conventional JSON parser from Tencent [19]
simdjson	A popular SIMD-based JSON parser [85]
Pison	A structural index-based JSON preprocessor [77]
JSONSki	JSON streaming with bit-parallel skipping (this work)

5.5.1 Methodology

We implemented the 15 skippers in Table 5.1 and the recursive descent streaming in C++. We name this streaming framework *JSONSki*. It supports standard JSON path query structures [74] with basic array index constraints, such as `[a:b]` and `[a:]`. Based on the availability of SIMD instructions (e.g., AVX512), it tries to maximize the overall bit-parallelism. By default, JSONSki transparently invokes the skippers whenever they apply, based on the structure of the JSON path query. However, advanced developers can use the skipping APIs for exploiting opportunities in other JSON analytics.

Methods in Comparison. We compare JSONSki with several representative JSON processing tools, including simdjson [85], RapidJSON [19], JPStream [78], and Pison [77]. A summary of these methods are listed in Table 5.2. Among them, simdjson and Pison also utilize bitwise and SIMD parallelism, but their uses are limited to the identification of metacharacters. Also, RapidJSON, simdjson, and Pison belong to the preprocessing scheme, while JPStream and JSONSki are streaming-based. Pison and JPStream also support speculative execution, which is orthogonal to the skipping optimizations. For the following evaluation, this feature is disabled.

Table 5.3: Dataset Statistics

Data	#objects	#arrays	#attr	#prim.	#sub	depth
TT	2.39M	2.29M	26.5M	24.3M	150K	11
BB	1.91M	4.88M	40.7M	35.8M	230K	7
GMD	10.3M	43K	29.0M	21.0M	4.44K	9
NSPL	613	3.50M	1.66K	84.2M	1.74M	9
WM	333K	34K	8.19M	9.92K	275K	4
WP	17.3M	6.53M	53.2M	35.0M	137K	12

Datasets. The datasets used for this evaluation are collected from real-world applications, including tweets stream from Twitter (TT) developer API [23], product dataset from Best Buy (BB) [2], National Statistics Postcode Lookup (NSPL) dataset from United Kingdom [16], Google Maps Directions (GMD) dataset [6], Walmart (WM) product dataset [26], and Wikipedia (WP) entity dataset [29]. Table 5.3 summarizes the statistics regarding their structures. For easier comparison, we made each dataset approximately the same size: 1GB. Each dataset can be either one single large record or a sequence of small records. The column #sub in Table 5.3 lists the number of small records in each dataset.

Path Queries. Table 5.4 lists the JSONPath queries used in the evaluation. For each dataset, we constructed two queries. The last column lists the number of matches. Together, they cover the common path query structures, as well as different levels of complexities and selectivities. Similar query structures have been used for evaluation by the prior work [78, 86, 77].

All experiments were conducted on a server equipped with two Intel 2.1GHz Xeon E5-2620 v4 CPUs and 64GB RAM. The CPUs support 64-bit ALU instructions and 256-bit SIMD instruction set. This server runs on CentOS 7 and is installed with G++ 7.4.0. All

Table 5.4: JSONPath Queries

ID	Query structure	#matches
TT1	$[*].en.urls[*].url	88,881
TT2	$[*].text	150,135
BB1	$.pd[*].cp[1:3].id	459,332
BB2	$.pd[*].vc[*].cha	8,857
GMD1	$[*].rt[*].lg[*].st[*].dt.tx	1,716,752
GMD2	$[*].atm	270
NSPL1	$.mt.vw.co[*].nm	44
NSPL2	$.dt[*][*][2:4]	3,509,764
WM1	$.it[*].bmpr.pr	15,892
WM2	$.it[*].nm	272,499
WP1	$[*].cl.P150[*].ms.pty	15,603
WP2	$[10:21].cl.P150[*].ms.pty	35

C++ programs were compiled with "-O3" optimization flag. All inputs are preloaded into the memory before the processing. Each input with small records is stored in an array, along with an offset array (for starting positions).

5.5.2 Overall Performance

We first compare the execution time of the different methods, then examine their memory overhead.

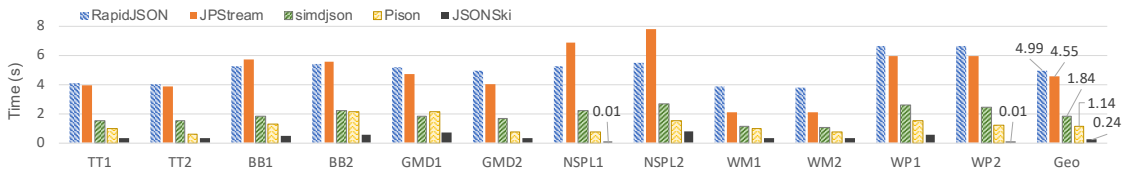

Figure 5.10: Performance Comparison on Large Records

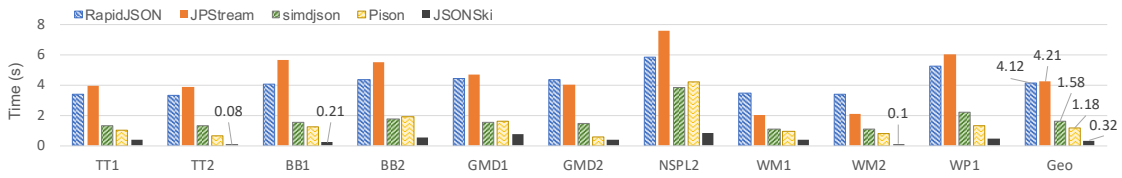

Figure 5.11: Performance Comparison on Small Records

Performance on Large Records. First, we evaluated different methods on data streams of a single large JSON record, where each record is around 1GB. Figure 5.10 reports the execution time. For preprocessing-based methods, it includes both the preprocessing time and querying time.

From the results we can find that JPStream and RapidJSON run significantly slower than the other three methods. This is because both of them process the data stream character by character, due to lack of bit-parallelism. In comparison, the other three methods can simultaneously process a batch of characters with bitwise/SIMD operations. In particular, JPStream, the existing streaming processor, runs 19.0× slower than JSONSki, which clearly demonstrates the importance of adopting bit-parallel skipping to the streaming.

Among the three methods with bit-parallelism, JSONSki outperforms simdjson and Pison, achieving 7.7× and 4.8× speedups on average, respectively. The benefits mainly come from three aspects: First, JSONSki not only utilizes bit-level parallelism for identifying metacharacters, but also uses it for locating the boundaries of objects and arrays (see Section 5.4). Second, JSONSki can quickly "jump" over a large ratio of the data stream, without examining its details (see Section 5.3.2). Third, with streaming design, JSONSki processes the data mostly inside the caches. By contrast, simdjson needs to find all the

tokens in the data stream and parse them into the tree structure. Unlike simdjson, Pison constructs structural indices before the parsing and querying. This allows it to skip parsing certain parts of the data stream in detail. However, it still needs to build the structural indices for the entire data stream, and its non-streaming design also limits the overall performance.

In addition, we noticed that JSONSki takes significantly less time (around 0.01s) in two cases: NSPL1 and WP2. It turns out that the queries find all matches in the early part of the data stream, so JSONSki skips a high ratio of data stream. The effectiveness of skipping will be discussed in Section 5.5.3.

Performance on Small Records. The performance results on data streams with small records are similar to those on the large-record data streams, except that, most of the evaluated methods become slightly faster, thanks to the better cache locality − some small records can be processed in the caches. Note that JSONSki shows a bit longer running time on average (from 0.24s to 0.32s). This is actually due to the exclusion of two cases in the large-record scenarios (NSPL1 and WP2) which are not applicable to the small-record scenarios.

Memory Overhead. Besides performance, the memory cost of data processing is also critical to many semi-structured data applications. Figure 5.12 reports the overall memory footprints of different methods during the processing of large records. The results clearly separate JPStream and JSONSki from the rest, thanks to their streaming design. In fact, the memory footprints for JPStream and JSONSki are mainly for holding the data stream and outputs (around 1GB). While the other three methods need a substantial amount

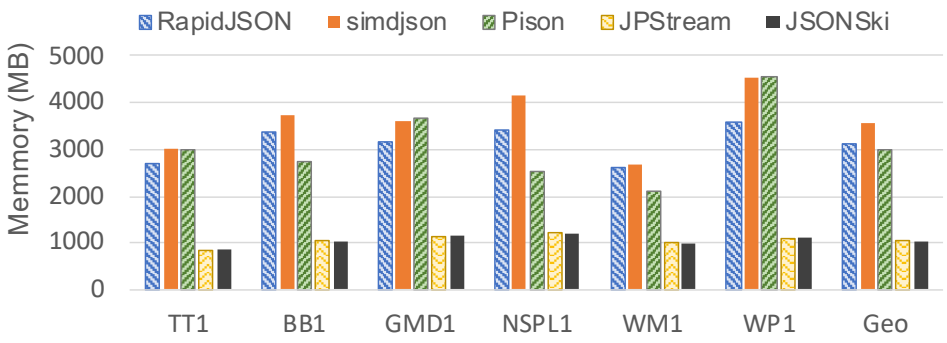

Figure 5.12: Memory Footprint Comparison

of extra memory for holding the preprocessing results, either a parse tree (simdjson and RapidJSON) or structural indices (Pison). On average, their memory sizes are tripled (around or above 3GB).

5.5.3 Benefits Breakdown

To further examine the benefits of JSONSki, we introduce a metric called *skipping ratio* – the ratio between the characters skipped and the total length of the data stream, to estimate the effectiveness of skippers. Table 5.5 reports these ratios for different groups of skippers under different querying scenarios. Note that G5 skippers are for queries with index constraints, so they are applicable only to some of the queries evaluated.

First, from the last column of Table 5.5, we can find the total skipping ratio for all five groups of skippers is very high across all the evaluated queries – all above 95%. This confirms the potential of skipping opportunities in practice. For the less than 5% characters that are not skipped, they are mostly the strings of attribute names that have

Table 5.5: Skipping Ratios by Skipper Groups

Query	G1	G2	G3	G4	G5	Total
TT1	**12.80%**	**78.22%**	0.22%	**8.20%**	–	99.44%
TT2	0.00%	1.17%	2.28%	**95.62%**	–	99.07%
BB1	**14.34%**	0.72%	0.49%	**82.19%**	0.75%	98.49%
BB2	**89.24%**	**8.73%**	0.02%	¡ 0.01%	–	97.99%
GMD1	**13.18%**	0.04%	1.06%	**83.13%**	–	97.41%
GMD2	0.02%	**99.97%**	¡ 0.01%	0.00%	–	99.99%
NSPL1	¡ 0.01%	¡ 0.01%	¡ 0.01%	**99.99%**	–	99.99%
NSPL2	**83.45%**	0.00%	1.55%	¡ 0.01%	**10.94%**	95.94%
WM1	**97.97%**	0.13%	0.01%	1.66%	–	99.77%
WM2	¡ 0.01%	0.33%	1.90%	**96.56%**	–	98.79%
WP1	1.47%	**83.08%**	0.01%	**14.77%**	–	99.33%
WP2	¡ 0.01%	0.02%	¡ 0.01%	0.01%	**99.96%**	99.99%

to be extracted for checking the matching status, as well as some metacharacters that have to be consumed individually (see Algorithm 11).

Second, from the results, we find that skippers of different groups often contribute unevenly to the overall skipping ratio. Skippers with substantial contributions ($> 5\%$) are highlighted in Table 5.5. For example, under TT1, G2 skippers achieve a skipping ratio of 78.22%, while G3 skippers only skip 0.22% of the data stream. Similar contribution variations also occur under other querying scenarios. In general, the contributions of skippers highly depend on the matching process: the query structure and the data stream structure. Except for G3, we find all other four groups of skippers made substantial contributions in at least one querying scenario. In the case of G3 skippers, their effectiveness actually depends on the querying selectivity, as they are designed for outputting. For queries with lower selectivity, their contributions would become higher.

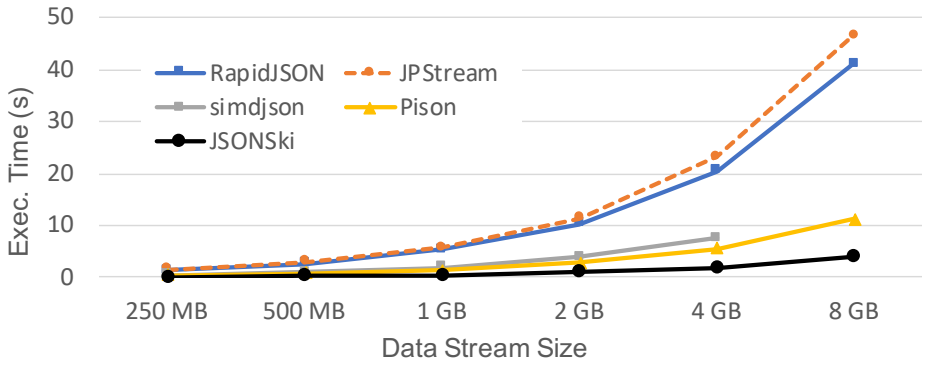

Figure 5.13: Scalability Comparison (BB1)

(simdjson supports records of size up to 4G)

5.5.4 Scalability

Finally, we briefly demonstrate the scalability of JSONSki. We constructed data streams of different sizes from the BB dataset, ranging from 250MB to 8GB. Figure 5.13 reports the execution time of different methods under query BB1. The results show that, as the data stream size increases, the execution time of JSONSki tends to grow slower than those of the other methods. This projects even more benefits of JSONSki for data streams with larger records.

5.6 Related Work

This section summarizes the existing work on semi-structured data under three topics: i) character-by-character processing, ii) bit-parallel processing, and iii) skipping and filtering.

Character-by-Character Processing. Most existing work on querying raw semi-structured data (such as XML and JSON) follow a character-by-character processing style. For

XML querying, existing methods are either based on automata [39, 134, 56, 68, 106], arrays [81], or stacks [50, 82]. Following similarly directions, most existing JSON query evaluators [4, 12, 15, 19] convert each JSON record into in-memory tree structure before getting the substructure of interests.

However, as discussed earlier, preprocessing introduces an upfront delay and requires additional memory, especially for large data streams. To address such issues, streaming scheme has been proposed, first for XML data [68, 106] then for JSON data, such as JSONSurfer [21] and JPStream [78]. The basic idea is to traverse the data stream in one pass, and perform the on-the-fly parsing for evaluating the queries. More details regarding to the streaming can be found in Section 5.2.

Bit-Parallel Processing. To further optimize the processing of semi-structured data, some recent work turn to some advanced features of modern CPUs, Mison [86] creates structural indices using SIMD and bitwise instructions, which can process tens of or even hundreds of characters simultaneously. However, its index construction process is sequential for a single JSON record. To improve the scalability, Pison [77] further brings intra-record parallelism, with the help of some customized parallelization techniques (e.g., speculation). Some of these ideas are inspired by NoDB [32, 33, 76, 83], which builds structural index on raw CSV files. Unlike the above methods, simdjson [85] adopts bitwise processing to speedup the parsing tree generation. All these methods are preprocessing-based, they use bit-parallelism for constructing the in-memory data structures. In comparison, JSONSki leverages bit-parallelism for accelerating the skipping in the streaming scheme.

Besides JSON processing, bitwise parallelism has also been applied in other textual data processing scenarios, like regular expression matching [45], substring searching [45, 111], XML parsing [89], delimiter-separated data parsing [124], as well as data processing in some database systems [30, 41, 101, 51].

Skipping and Filtering. To boost the performance of JSON data processing, Sparser [111] filters out certain JSON records before parsing and querying them. The filtering is based on the substrings appeared in the query. If a JSON record has no specified substrings at all, it would be filter out. Obviously, the effectiveness of the filtering depends on the query selectivity. If a query can find matches in most records, the filtering only brings limited benefits (or even slows down the processing). Note that this filtering is orthogonal to the skipping proposed in this work. The former is at the inter-record level and it is NOT aware of the record structure, while our skipping is at the intra-record level and it is based on the record structure.

For XML data, some earlier work also mentioned skipping optimizations, but both the contexts and the high-level ideas are different from this work. First, XHints [73] proposes to insert artificial "hints" into the XML data stream to indicate the portions for skipping. This ad-hoc solution may work well for some cases, but it sacrifices the generality. In comparison, Amagasa and others [34] propose to skip XML elements that fail to match the path queries. This shares similarities with our G2 skippers. To implement this, they designed a streamlined parser that can skip attributes of unmatched XML elements.

5.7 Summary

Streaming is a promising scheme for semi-structured data processing. However, its conventional design requires to scan the entire data stream in detail. In this work, we propose the idea of "streaming with skipping". In particular, we reveal a series of opportunities for skipping certain substructures of the data stream that are irrelevant to the query evaluation. To tap into the full potential of the skipping, we propose a group of bit-parallel algorithms for implementing different skipping cases. With these techniques, we developed a new JSON streaming framework, called JSONSki. Our evaluation shows that JSONSki can skip over 95% of the data stream for common query structures, bringing significant speedups over the existing streaming and non-streaming methods.

Chapter 6

Conclusions

In this thesis, we tackled several major challenges in scalable processing of semi-structured data. Specifically, we first proposed GAP, a grammar-based parallelization scheme that leverages the grammar of the input data to narrow down the feasible execution paths in the enumerative parallelization model of XPath query evaluation [106]. It supports both non-speculative mode and speculative mode, depending on the availability of pre-defined grammar. Following this work, we designed a stream processing model for querying JSON data, namely JPStream, which is enabled by a joint compilation technique that combines JSON grammar and input queries into a unified pushdown automaton with two stacks. The parallelization of this model is enabled by leveraging a syntactical feasibility inference, an on-demand automaton resetting scheme, and a data constraint learner (adopted from GAP). Together, they avoid the path explosion issue raised by the unique features of JSON. Next, to accelerate the index construction in preprocessing-based scheme, we designed a highly efficient structural index constructor by exploiting both fine-grained (SIMD/bitwise)

parallelism and coarse-grained (multi-core/multi-thread) parallelism, namely Pison. Based on an existing solution named Mison [86], Pison redesigns its structural index construction process for less memory access and addresses the parallelization challenges by breaking the data dependences. Lastly, we explored the syntactical structure of JSON record and found a series opportunities to skip certain types of substructures that are irrelevant to the query evaluation during the streaming query evaluation. These skipping cases are implemented using a group of bit-parallel algorithms and integrated into a versatile recursive-descent streaming model. These techniques are packed into a JSON streaming framework, called JSONSki. Our evaluation demonstrates the benefits of all proposed techniques over the existing solutions in terms of both processing time and memory consumption.

Bibliography

[1] Apache VXQuery. https://vxquery.apache.org/.

[2] Best Buy developer API. https://bestbuyapis.github.io/api-documentation/. Retrieved: 2018-07-01.

[3] Best practices for reading JSON data. https://docs.aws.amazon.com/athena/latest/ug/. Retrieved: 2018-07-01.

[4] A fast JSON parser/generator for Java. https://github.com/alibaba/fastjson/. Retrieved: 2018-07-01.

[5] Getting started with JSON features in Azure SQL database. https://docs.microsoft.com/en-us/azure/sql-database/sql-database-json-features. Retrieved: 2018-07-01.

[6] Google Map directions API. https://developers.google.com/maps/documentation/directions/start/. Retrieved: 2018-07-01.

[7] Health care provider credential data. https://data.wa.gov/api/views/qxh8-f4bd/. Retrieved: 2018-07-01.

[8] The home of the U.S. government's open data. https://www.data.gov/. Retrieved: 2018-07-01.

[9] Indigenous land use agreements. https://data.gov.au/dataset/indigenous-land-use-agreements-registered-or-in-notification/. Retrieved: 2018-07-01.

[10] Introducing JSON. https://www.json.org/. Retrieved: 2019-07-01.

[11] A java serialization/deserialization library to convert java objects into json and back. https://github.com/google/gson/. Retrieved: 2019-05-20.

[12] JSON-C - a JSON implementation in C. https://github.com/json-c/json-c. Retrieved: 2018-07-01.

[13] JSON parser which picks up values directly without performing tokenization in Rust. https://github.com/pikkr/pikkr. Retrieved: 2020-05-10.

[14] JSONiq: The JSON query language. http://jsoniq.org/. Retrieved: 2018-07-01.

[15] Main portal page for the Jackson project. https://github.com/FasterXML/jackson/. Retrieved: 2018-07-01.

[16] National statistics postcode lookup UK. https://data.gov.uk/dataset/national-statistics-postcode-lookup-uk/. Retrieved: 2018-07-01.

[17] RapidJSON. http://rapidjson.org/. Retrieved: 2018-07-01.

[18] RapidJSON. http://rapidjson.org/. Retrieved: 2020-10-12.

[19] RapidJSON. http://rapidjson.org/. Retrieved: 2020-10-12.

[20] Run javascript everywhere. https://nodejs.dev/. Retrieved: 2019-07-01.

[21] A streaming JsonPath processor in Java. https://github.com/jsurfer/JsonSurfer/. Retrieved: 2018-07-01.

[22] Twitter developer API. https://developer.twitter.com/en/docs/. Retrieved: 2018-07-01.

[23] Twitter developer API. https://developer.twitter.com/en/docs/. Retrieved: 2019-07-01.

[24] Twitter usage statistics. http://www.internetlivestats.com/twitter-statistics/. Retrieved: 2018-07-01.

[25] Using the graph API. https://developers.facebook.com/docs/graph-api/using-graph-api/. Retrieved: 2019-05-10.

[26] Walmart developer API. https://developer.twitter.com/en/docs. Retrieved: 2019-05-20.

[27] Why JSON will continue to push XML out of the picture. https://www.ctl.io/developers/blog/post/why-json-will-continue-to-push-xml-out-of-the-picture. Retrieved: 2018-07-01.

[28] Wikidata:database download. https://www.wikidata.org/wiki/Wikidata:Database_download. Retrieved: 2019-05-10.

[29] Wikimedia miscellaneous files. https://archive.org/details/wikidata-json-20150202. Retrieved: 2020-05-20.

[30] Azza Abouzied, Daniel J Abadi, and Avi Silberschatz. Invisible loading: access-driven data transfer from raw files into database systems. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 1–10, 2013.

[31] Alfred V Aho. *Compilers: principles, techniques and tools (for Anna University), 2/e*. Pearson Education India, 2003.

[32] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. NoDB: efficient query execution on raw data files. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 241–252, 2012.

[33] Ioannis Alagiannis, Renata Borovica-Gajic, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. NoDB: efficient query execution on raw data files. *Communications of the ACM*, 58(12):112–121, 2015.

[34] Toshiyuki Amagasa, Mana Seino, and Hiroyuki Kitagawa. Energy-efficient xml stream processing through element-skipping parsing. In *2013 24th International Workshop on Database and Expert Systems Applications*, pages 254–258. IEEE, 2013.

[35] Amazon. Extracting data from JSON. `https://docs.aws.amazon.com/athena/latest/ug/extracting-data-from-JSON.html`, 2021. Retrieved: 2021-07-01.

[36] Kevin Angstadt, Arun Subramaniyan, Elaheh Sadredini, Reza Rahimi, Kevin Skadron, Westley Weimer, and Reetuparna Das. ASPEN: A scalable In-SRAM architecture for pushdown automata. In *Proceedings of the 51th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2018.

[37] Apache Fundation. Apache CouchDB. `http://couchdb.apache.org/`. Retrieved: 2018-07-01.

[38] Diego Arroyuelo, Francisco Claude, Sebastian Maneth, Veli Makinen, Gonzalo Navarro, Kim Nguyen, Jouni Siren, and Niko Valimaki. Fast in-memory xpath search using compressed indexes. *Software: Practice and Experience*, 45(3):399–434, 2015.

[39] Iliana Avila-Campillo, Todd J Green, Ashish Gupta, Makoto Onizuka, Demian Raven, and Dan Suciu. Xmltk: An xml toolkit for scalable xml stream processing. 2002.

[40] Alessandro Barenghi, Stefano Crespi-Reghizzi, Dino Mandrioli, Federica Panella, and Matteo Pradella. Parallel parsing made practical. *Sci. Comput. Program.*, 112:195–226, 2015.

[41] Spyros Blanas, Kesheng Wu, Surendra Byna, Bin Dong, and Arie Shoshani. Parallel data analysis directly on scientific file formats. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 385–396, 2014.

[42] Peter Boncz, Torsten Grust, Maurice Van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. Monetdb/xquery: a fast xquery processor powered by a relational engine. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 479–490. ACM, 2006.

[43] Rajesh Bordawekar, Lipyeow Lim, Anastasios Kementsietsidis, and Bryant Wei-Lun Kok. Statistics-based parallelization of xpath queries in shared memory systems. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, pages 159–170, 2010.

[44] Tim Bray. The javascript object notation (JSON) data interchange format. *RFC*, 8259:1–16, 2017.

[45] Robert D Cameron, Thomas C Shermer, Arrvindh Shriraman, Kenneth S Herdy, Dan Lin, Benjamin R Hull, and Meng Lin. Bitwise data parallelism in regular expression matching. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 139–150. IEEE, 2014.

[46] Robert D. Cameron, Thomas C. Shermer, Arrvindh Shriraman, Kenneth S. Herdy, Dan Lin, Benjamin R. Hull, and Meng Lin. Bitwise data parallelism in regular expression matching. In José Nelson Amaral and Josep Torrellas, editors, *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, pages 139–150. ACM, 2014.

[47] Roy H. Campbell and Reza Farivar. Cloud computing applications, part 1: Cloud systems and infrastructure. `https://www.coursera.org/learn/cloud-applications-part1`. Retrieved: 2018-07-01.

[48] Arijit Chakraborty. An introduction to REST and JSON. `https://blogs.oracle.com/cloud-platform/an-introduction-to-rest-and-json`. Retrieved: 2018-07-01.

[49] Songting Chen, Hua-Gang Li, Jun'ichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, and K. Selçuk Candan. Twig^{2}stack: Bottom-up processing of generalized-tree-pattern queries over XML documents. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 283–294, 2006.

[50] Songting Chen, Hua-Gang Li, Junichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, and K Selçuk Candan. Twig2stack: bottom-up processing of generalized-tree-pattern queries over xml documents. In *Proceedings of the 32nd international conference on Very large data bases*, pages 283–294, 2006.

[51] Yu Cheng and Florin Rusu. Parallel in-situ data processing with speculative loading. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1287–1298, 2014.

[52] James Clark, C Cooper, and F Drake. The expat xml parser, 2011.

[53] DBLP Team. Welcome to DBLP. `http://dblp.uni-trier.de/`, 2016. Retrived: 2016-07-01.

[54] Frank S de Boer, Marcello M Bonsangue, Joost Jacob, Andries Stam, and L Van der Torre. Enterprise architecture analysis with xml. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, pages 222b–222b. IEEE, 2005.

[55] Ebay developers program. What is the ebay api? `https://go.developer.ebay.com/what-ebay-api`, 2016. Retrived: 2016-07-01.

[56] Yanlei Diao, Peter Fischer, Michael J Franklin, and Raymond To. Yfilter: Efficient and scalable filtering of xml documents. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 341–342. IEEE, 2002.

[57] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3088–3098, 2014.

[58] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[59] Adam Dunkels et al. Efficient application integration in ip-based sensor networks. In *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 43–48. ACM, 2009.

[60] W3C Editor. Overview of the CoverageJSON format. `https://w3c.github.io/sdw/coverage-json/`. Retrieved: 2018-07-01.

[61] Chris Esplin. Firebase data modeling. `https://howtofirebase.com/firebase-data-modeling-939585ade7f4`. Retrieved: 2018-07-01.

[62] Facebook. Public feed API. `https://developers.facebook.com/docs/public_feed`, 2016. Retrieved: 2016-07-01.

[63] Massimo Franceschet. Xpathmark: Functional and performance tests for xpath. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[64] Francis Galiegue and Kris Zyp. Json schema: Core definitions and terminology. *Internet Engineering Task Force (IETF)*, 2013.

[65] Chang Ge, Yinan Li, Eric Eilebrecht, Badrish Chandramouli, and Donald Kossmann. Speculative distributed csv data parsing for big data analytics. In *Proceedings of the 2019 International Conference on Management of Data*, pages 883–899, 2019.

[66] Alan Gibbons and Wojciech Rytter. *Efficient parallel algorithms*. Cambridge University Press, 1989.

[67] Alan Gibbons and Wojciech Rytter. Optimal parallel algorithms for dynamic expression evaluation and context-free recognition. *Information and Computation*, 81(1):32–45, 1989.

[68] Todd J Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing xml streams with deterministic automata. In *International Conference on Database Theory*, pages 173–189. Springer, 2003.

[69] GeoJSON Working Group. The GeoJSON specification (RFC 7946). `http://geojson.org/`. Retrieved: 2018-07-01.

[70] Netork Working Group. NetJSON: data interchange format for networks. `http://netjson.org/rfc.html`. Retrieved: 2018-07-01.

[71] W3C JSON-LD Community Group. JSON for linking data. `https://json-ld.org/`. Retrieved: 2018-07-01.

[72] Venkat N Gudivada, Dhana Rao, and Vijay V Raghavan. NoSQL systems for big data management. In *Services (SERVICES), 2014 IEEE World Congress on*, pages 190–197. IEEE, 2014.

[73] Akhil Gupta and Sudarshan S Chawathe. Skipping streams with xhints. Technical report, 2004.

[74] Stefan Gössner. JSONPath - XPath for JSON. `http://goessner.net/articles/JsonPath/`. Retrieved: 2018-07-01.

[75] IBM. XML-SAX (parse an XML document). `https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_73/rzasd/zzxmlsa.htm`. Retrieved: 2018-07-01.

[76] Stratos Idreos, Ioannis Alagiannis, Ryan Johnson, and Anastasia Ailamaki. Here are my data files. here are my queries. where are my results? In *Proceedings of 5th Biennial Conference on Innovative Data Systems Research*, number CONF, 2011.

[77] Lin Jiang, Junqiao Qiu, and Zhijia Zhao. Scalable structural index construction for JSON analytics. *Proc. VLDB Endow.*, 14(4):694–707, 2020.

[78] Lin Jiang, Xiaofan Sun, Umar Farooq, and Zhijia Zhao. Scalable processing of contemporary semi-structured data on commodity parallel processors - A compilation-based approach. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 79–92. ACM, 2019.

[79] Lin Jiang and Zhijia Zhao. Grammar-aware parallelization for scalable XPath querying. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 371–383. ACM, 2017.

[80] Vanja Josifovski, Marcus Fontoura, and Attila Barta. Querying XML streams. *VLDB J.*, 14(2):197–210, 2005.

[81] Vanja Josifovski, Marcus Fontoura, and Attila Barta. Querying xml streams. *The VLDB Journal*, 14(2):197–210, 2005.

[82] Arseny Kapoulkine. pugixml: a light-weight, simple and fast xml parser for c++ with xpath support. `http://pugixml.org/`. Retrieved: 2016-07-01.

[83] Manos Karpathiotakis, Miguel Branco, Ioannis Alagiannis, and Anastasia Ailamaki. Adaptive query processing on RAW data. *Proceedings of the VLDB Endowment*, 7(12):1119–1130, 2014.

[84] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, 1980.

[85] Geoff Langdale and Daniel Lemire. Parsing gigabytes of JSON per second. *VLDB J.*, 28(6):941–960, 2019.

[86] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. Mison: A fast JSON parser for data analytics. *PVLDB*, 10(10):1118–1129, 2017.

[87] Yishan Li and Sathiamoorthy Manoharan. A performance comparison of SQL and NoSQL databases. In *Communications, computers and signal processing (PACRIM), 2013 IEEE pacific rim conference on*, pages 15–19. IEEE, 2013.

[88] Leonid Libkin, Wim Martens, and Domagoj Vrgoc. Querying graph databases with xpath. In *Proceedings of the 16th International Conference on Database Theory*, pages 129–140. ACM, 2013.

[89] Dan Lin, Nigel Medforth, Kenneth S. Herdy, Arrvindh Shriraman, and Robert D. Cameron. Parabix: Boosting the efficiency of text processing on commodity processors. In *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012*, pages 373–384, 2012.

[90] Le Liu, Jianhua Feng, Guoliang Li, Qian Qian, and Jianhui Li. Parallel structural join algorithm on shared-memory multi-core systems. In *The Ninth International Conference on Web-Age Information Management, WAIM 2008, July 20-22, 2008, Zhangjiajie, China*, pages 70–77, 2008.

[91] Logicworks. The future of AWS' cloud: Infrastructure as an application. `https://www.cloudcomputing-news.net/news/2016/jun/02/the-future-of-aws-cloud-infrastructure-as-an-application/`. Retrieved: 2018-07-01.

[92] Isaac Lopez. Amazon hosting 20 TB of climate data. `https://www.datanami.com/2013/11/12/amazon_hosting_20_tb_of_open_climate_data/`. Retrieved: 2018-07-01.

[93] C Louden Kenneth. Compiler construction: Principles and practice. *Course Technology*, 1997.

[94] Wei Lu, Kenneth Chiu, and Yinfei Pan. A parallel approach to xml parsing. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, GRID '06, pages 223–230, 2006.

[95] Wei Lu, Kenneth Chiu, and Yinfei Pan. A parallel approach to XML parsing. In *7th IEEE/ACM International Conference on Grid Computing (GRID 2006), September 28-29, 2006, Barcelona, Spain, Proceedings*, pages 223–230, 2006.

[96] Wei Lu and Dennis Gannon. Parallel XML processing by work stealing. In *Proceedings of the 2007 workshop on Service-oriented computing performance: aspects, issues, and approaches*, pages 31–38. ACM, 2007.

[97] Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. Parallelizing dynamic programming through rank convergence. *ACM SIGPLAN Notices*, 49(8):219–232, 2014.

[98] Abhishek Mitra, Marcos Vieira, Petko Bakalov, Walid Najjar, and Vassilis Tsotras. Boosting xml filtering with a scalable fpga-based architecture. *arXiv preprint arXiv:0909.1781*, 2009.

[99] MongoDB. MongoDB extended JSON. `https://docs.mongodb.com/manual/reference/mongodb-extended-json/`. Retrieved: 2018-07-01.

[100] Roger Moussalli, Robert J Halstead, Mariam Salloum, Walid A Najjar, and Vassilis J Tsotras. Efficient XML path filtering using GPUs. In *ADMS@ VLDB*, pages 9–18. Citeseer, 2011.

[101] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Instant loading for main memory databases. *Proceedings of the VLDB Endowment*, 6(14):1702–1713, 2013.

[102] Wojciech Mula and Daniel Lemire. Faster base64 encoding and decoding using AVX2 instructions. *CoRR*, abs/1704.00605, 2017.

[103] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. Data-parallel finite-state machines. In *ASPLOS '14: Proceedings of 19th International Conference on Architecture Support for Programming Languages and Operating Systems*. ACM Press, 2014.

[104] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. Data-parallel finite-state machines. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 529–542, 2014.

[105] Antinus Nijholt. The cyk approach to serial and parallel parsing. 1991.

[106] Peter Ogden, David Thomas, and Peter Pietzuch. Scalable XML query processing using parallel pushdown transducers. *Proceedings of the VLDB Endowment*, 6(14):1738–1749, 2013.

[107] Peter Ogden, David B. Thomas, and Peter R. Pietzuch. AT-GIS: highly parallel spatial query processing with associative transducers. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1041–1054, 2016.

[108] Oracle. Parsing an XML file using SAX. `https://docs.oracle.com/javase/tutorial/jaxp/sax/parsing.html`. Retrieved: 2018-07-01.

[109] Seán O'Riain, Edward Curry, and Andreas Harth. Xbrl and open data for global financial ecosystems: A linked data approach. *International Journal of Accounting Information Systems*, 13(2):141–162, 2012.

[110] Shankar Pal, Vishesh Parikh, Vasili Zolotov, Leo Giakoumakis, and Michael Rys. Xml best practices for microsoft sql server 2005. *Retrieved*, 11:2004, 2004.

[111] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Filter before you parse: Faster analytics on raw data with sparser. *Proceedings of the VLDB Endowment*, 11(11):1576–1589, 2018.

[112] Y. Pan, W. Lu, Y. Zhang, and K. Chiu. A static load-balancing scheme for parallel xml parsing on multicore cpus. In *Proc. of the 7th International Symposium on Cluster Computing and the Grid (CCGRID)*, 2007.

[113] Christina Pavlopoulou, E. Preston Carman Jr., Till Westmann, Michael J. Carey, and Vassilis J. Tsotras. A parallel and scalable processor for JSON data. In *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018.*, pages 576–587, 2018.

[114] Junqiao Qiu, Lin Jiang, and Zhijia Zhao. Challenging sequential bitstream processing via principled bitwise speculation. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 607–621. ACM, 2020.

[115] Junqiao Qiu, Zhijia Zhao, and Bin Ren. Microspec: Speculation-centric fine-grained parallelization for fsm computations. In *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*, pages 221–233. IEEE, 2016.

[116] Junqiao Qiu, Zhijia Zhao, Bo Wu, Abhinav Vishnu, and Shuaiwen Leon Song. Enabling scalability-sensitive speculative parallelization for FSM computations. In William D. Gropp, Pete Beckman, Zhiyuan Li, and Francisco J. Cazorla, editors, *Proceedings of the International Conference on Supercomputing, ICS 2017, Chicago, IL, USA, June 14-16, 2017*, pages 2:1–2:10. ACM, 2017.

[117] James Reinders. Intel AVX-512 instructions. `https://software.intel.com/en-us/articles/intel-avx-512-instructions`. Retrieved: 2019-06-01.

[118] Yasubumi Sakakibara. Learning context-free grammars from structural data in polynomial time. *Theoretical Computer Science*, 76(2-3):223–242, 1990.

[119] Yasubumi Sakakibara. Efficient learning of context-free grammars from positive structural examples. *Information and Computation*, 97(1):23–60, 1992.

[120] Giorgio Satta and Oliviero Stock. Bidirectional context-free grammar parsing for natural language processing. *Artificial Intelligence*, 69(1):123–164, 1994.

[121] Lars Schor, Philipp Sommer, and Roger Wattenhofer. Towards a zero-configuration wireless sensor network architecture for smart buildings. In *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 31–36. ACM, 2009.

[122] B. Shah, P. Rao, B. Moon, and M. Rajagopalan. A data parallel algorithm for xml dom parsing. *Database and XML Technologies, Lecture Notes in Computer Science*, 5679, 2009.

[123] Sqlizer. A brief history of JSON. `https://blog.sqlizer.io/posts/json-history/`. Retrieved: 2018-07-01.

[124] Elias Stehle and Hans-Arno Jacobsen. ParPaRaw: Massively parallel parsing of delimiter-separated raw data. *Proc. VLDB Endow.*, 13(5):616–628, January 2020.

[125] Arun Subramaniyan, Jingcheng Wang, Ezhil RM Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. Cache automaton. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 259–272. ACM, 2017.

[126] Dan Suciu. XML data repository. `http://www.cs.washington.edu/research/xmldatasets/`, 2003. Retrieved: 2016-07-01.

[127] Ninomiya Takashi, Torisawa Kentaro, Kenjiro Taura, and Jun'ichi Tsujii. A parallel cky parsing algorithm on large-scale distributed-memory parallel machines. 1997.

[128] Twitter. Data dictionary: Standard v1.1. `https://developer.twitter.com/en/docs/twitter-api/v1/data-dictionary/object-model/geo`, 2021. Retrieved: 2021-07-01.

[129] TwoBitHistory. The rise and rise of JSON. `https://twobithistory.org/2017/09/21/the-rise-and-rise-of-json.html`. Retrieved: 2018-07-01.

[130] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *Computing Colombian Conference (10CCC), 2015 10th*, pages 583–590. IEEE, 2015.

[131] Alex Hai Wang. Don't follow me: Spam detection in twitter. In *Security and Cryptography (SECRYPT), Proceedings of the 2010 International Conference on*, pages 1–10. IEEE, 2010.

[132] Philipp Wehner, Christina Piberger, and Diana Gohringer. Using JSON to manage communication between services in the internet of things. In *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th International Symposium on*, pages 1–4. IEEE, 2014.

[133] Zhiqiang Yu Yu Wu, Qi Zhang and Jianhui Li. A hybrid parallel processing for xml parsing and schema validation. In *Balisage: The Markup Conference 2008*.

[134] Ying Zhang, Yinfei Pan, and Kenneth Chiu. A parallel xpath engine based on concurrent NFA execution. In *16th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2010, Shanghai, China, December 8-10, 2010*, pages 314–321, 2010.

[135] Zhijia Zhao, Michael Bebenita, Dave Herman, Jianhua Sun, and Xipeng Shen. Hpar: A practical parallel parser for html–taming html complexities for parallel parsing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):44, 2013.

[136] Zhijia Zhao and Xipeng Shen. On-the-fly principled speculation for FSM parallelization. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 619–630, 2015.

[137] Zhijia Zhao, Bo Wu, and Xipeng Shen. Challenging the "embarrassingly sequential": Parallelizing finite state machine-based computations through principled speculation. In *ASPLOS '14: Proceedings of 19th International Conference on Architecture Support for Programming Languages and Operating Systems*. ACM Press, 2014.

[138] Zhijia Zhao, Bo Wu, and Xipeng Shen. Challenging the embarrassingly sequential: parallelizing finite state machine-based computations through principled speculation. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 543–558. ACM, 2014.

[139] Zhijia Zhao, Bo Wu, Mingzhou Zhou, Yufei Ding, Jianhua Sun, Xipeng Shen, and Youfeng Wu. Call sequence prediction through probabilistic calling automata. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 745–762. ACM, 2014.