Implicit Parallelization of the Nested Relational Calculus for Semi-Structured Data

Master's Thesis

Pieter Adrianus Hameete

Implicit Parallelization of the Nested Relational Calculus for Semi-Structured Data

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE
TRACK INFORMATION ARCHITECTURE

by

Pieter Adrianus Hameete
born in Albrandswaard

Web Information Systems
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
http://wis.ewi.tudelft.nl

Implicit Parallelization of the Nested Relational Calculus for Semi-Structured Data

Author: Pieter Adrianus Hameete
Student id: 4025539
Email: phameete@gmail.com

Abstract

Large volumes of data are produced, published and exchanged over the Internet. Such data is often in a semi-structured format which is typically irregular and therefore challenging to analyze. High-level data analysis languages are built on top of implicit parallel data processing platforms that handle distribution of computations and data. Currently, work is being performed on the Nested Relational Calculus for Semi-structured Data (sNRC), which combines well-known formalisms from the Nested Relatonal Calculus for querying nested data with modern approaches for large-scale data analysis.

This work presents a first of its kind system for parallel evaluation of sNRC queries built on top of an implicit parallel framework called Flink. Previous work on an optimization called input projection recombined and modified to present an input projection algorithm for sNRC. This optimization has as goal to improve the performance and scalability of the parallel sNRC system by reducing the size of the input dataset.

The system is evaluated with the XMark benchmark on a cluster of up to 16 quad CPU nodes, and for datasets of up to 141 GB. We show that the presented parallel sNRC system is capable of processing large-scale datasets and that it can facilitate future work on sNRC. Moreover, it is shown that the presented input projection algorithm strongly improves the performance and scalability for sNRC queries that require partitioning.

Thesis Committee:

Chair:	Prof. dr. ir. G.J. Houben, Faculty EEMCS, TUDelft
University supervisor:	Dr. ir. J. Hidders, Faculty EEMCS, TUDelft
Committee Member:	Dr. ir. A. Iosup, Faculty EEMCS, TUDelft

Preface

Before you lies my Master's thesis, the work that concludes my 6 and a half years as a Computer Science student at the Delft University of Technology. I feel like this project has been the most challenging as well as satisfying and educational project that I have worked on as a student. I have learned a lot during this project, both scientifically as well as personally. Looking back at this point I am proud of what I have achieved, but it would never have been possible without the advice and support of the many people that I would like to thank in this preface.

First of all I would like to thank my daily supervisor Jan Hidders. You have guidance has been very important to me throughout the project, always pointing me into the right direction while still allowing me to choose my own path. I would like to thank you specifically, but also my fellow graduate students for the discussions that we had. These discussions were very educational and have forced me to critically think about my own work and to dig deeper when necessary. I would like to also thank the other two members of my Thesis committee, Alexandru Iosup and Geert-Jan Houben, for their efforts and feedback and I would like to thank Friso Abcouwer for proofreading my thesis.

I am very grateful for the wonderful people that I have met over the last 6 years. We have had wonderful times together and we have had the opportunity to learn so much from each other. A final thanks goes out to my family, in particular to my parents and to my girlfriend Eva, for supporting me in everything that I do and for always being there for me.

Pieter Adrianus Hameete
Rotterdam, the Netherlands
May 30, 2016

Contents

List of Figures

Chapter 1

Introduction

As the Internet is continuously developing, social networks keep growing in popularity and mobile communications contribute to an enormous amount of data being generated. A large portion of the data that is published on the Internet or exchanged between distributed applications or web applications is in a semi-structured data format such as the popular XML or JSON formats [27]. Semi-structured data is nested and typically irregular, and the data may have no schema or an incomplete schema. It is most often encountered during the exploratory phases of data analysis [10].

Analysis of semi-structured data has many interesting use cases, but it is also a very challenging topic. The amount and complexity of semi-structured data require a huge amount of effort to be invested in order to retrieve valuable insights. Approaches for efficient analysis of large amounts of semi-structured data are very desirable and have therefore become a much researched topic by both industries and academia.

Long before the 'big data' trend, much research had already been performed in the area of nested relational databases [47, 40, 43, 21]. This research was later formalized in a Nested Relational Calculus (NRC) and Nested Relational Algebra [13, 47, 33] which influenced well-known query languages for semi-structured data such as XQuery [11].

In order to analyse large amounts of data **implicit parallel** data processing frameworks [17] can be leveraged. By implitly parallel we mean that the framework handles distribution of computations and data accross the available computation nodes. Examples of such implicit parallel frameworks are the well-known MapReduce [23], Spark [49] or Flink [5]. Because existing query languages for semi-structured data were not capable of handling large amounts of data in terms of volume and velocity, new high-level data analysis languages [36, 10, 44] were developed on top of such implicit parallel data processing frameworks.

Recently work has been presented that connects the formalisms provided by the NRC and NRA with modern approaches for large-scale data analysis. It has been shown that the NRC can provide a solid basis for a data flow language [30] and that NRC is suitable for expressing MapReduce optimisations [28]. Ongoing work is performed on a dialect of the NRC, called the Nested Relational Calculus for Semi-structured Data (sNRC). The sNRC uses the well-known formalisms from the NRC

to design a language for querying semi-structured data [29]. This language has as its goals to be well-understood in terms of semantics and expressive power, to be easily implementable on an implicit parallel data processing framework to process large-scale data and to allow application of well-known optimisation techniques.

The goal of this work is to explore the capabilities of the sNRC to fulfil its goals, and to create opportunities of further research on the topic of sNRC. First, a reference system for parallel evaluation of sNRC expressions is presented that uses a well-known technique called input projection. This reference system is improved upon by developing a more sophisticated input projection algorithm. Furthermore, the improved parallel sNRC system is evaluated and compared to the reference system on a cluster using a popular benchmark for semi-structured data. The experiments provide insights in the performance of the parallel sNRC system and the results of the input projection optimization. In the remainder of this chapter these goals will be linked to a set of research objectives, the contributions of this thesis will be summarized and an outline of the thesis will be provided.

1.1 Research Objectives

The goal of this thesis as stated above can be described by a set of research questions. These research questions are answered based on our findings in the remainder of this thesis.

RQ1: *Which state-of-the-art parallel data processing platform is suitable for implicit parallelization of the Nested Relational Calculus for Semi-Structured Data?*

As was already stated in the introduction, there are several different parallel data processing platforms available. By researching the qualities and shortcomings of each of the data processing platforms and aligning these with the goals for a parallel sNRC system, a platform needs to be chosen for implicit parallelization of the sNRC.

RQ2: *How can an input semi-structured dataset be fragmentized to allow processing by a streaming data processing platform?*

Like with other parallelized systems for querying semi-structured data [17, 36, 10] we cannot always load a complete semi-structured dataset into memory. This also complies with our goal to investigate whether the sNRC is suitable for processing large-scale datasets. The selected data processing platform features a streaming data processing engine. Such a streaming data processing engine is not built to load a complete semi-structured dataset into memory. Instead it is designed to process smaller chunks or elements and therefore it is required to fragmentize the input dataset into a collection of smaller elements. After answering RQ1 and RQ2 a **Reference Parallel sNRC System** can be designed.

RQ3: *How can a reference input projection approach be improved upon to further reduce the size of the input semi-structured dataset?*

The approach for the Reference Parallel sNRC system is to apply a technique called input projection to fragmentize the input dataset to allow processing on a parallel data processing platform. The initial approach taken for the reference parallel sNRC implementation can be improved upon in particular for queries that use partitioning operations. Further reducing the size of the input semi-structured dataset is expected to result in improved performance and scalability. Therefore, it is investigated how to improve upon the reference parallel sNRC input projection to develop an **Improved Parallel sNRC System**.

RQ4: *What is the effect of the improved input projection on the performance in terms of runtime and scalability of the improved parallel sNRC system compared to the reference system?*

As a final research question we investigate how and when applying the improved input projection improves the performance in terms of runtimes and scalability of the reference parallel sNRC system. To do this a suitable benchmark should be chosen. Both the reference and improved system should be evaluated on datasets of varying sizes and clusters of varying sizes to answer this research question.

1.2 Contributions

In this section we summarize the contributions of this work. Based on the work presented in the remainder of this thesis to answer the above research questions we identify the following contributions:

- **A Parallel sNRC System Implementation:** A working, first of its kind, parallel sNRC system is presented in this thesis. It is demonstrated how sNRC fulfils one of its goals, by indeed allowing implementation on an implicit parallel data processing framework. The qualities and flaws of the available data processing systems are outlined and discussed. We motivate and present the design of the parallel sNRC system and discuss how challenges were overcome. Having a first system is crucial to allow further research in the area of (parallel) sNRC.

- **Input Projection for sNRC:** We have applied a first optimization to the parallel sNRC system in the form of input projection. A necessary reference system is presented to adapt sNRC to an implicit parallel data processing framework. Next, an improved version is presented that uses a static sNRC expression analysis algorithm and combines existing state-of-the-art approaches for input projection to further reduce the size of the input dataset.

- **Evaluation of the Parallel sNRC System and Input Projection:** Using a set of experiments on both a single node, as well as a cluster of up to 16 nodes and datasets of up to 141 GB we evaluate the parallel sNRC system with the reference and improved input projection approaches. It is shown how the improved input projection is a strong improvement over the reference, and consistently

improves on an existing approach regarding the size of the input dataset. Moreover, the systems are evaluated using the well-known XMark [41] benchmark for semi-structured data. Queries from the XMark benchmark are categorized and the performance in terms of runtime and scalability for both systems is investigated and discussed for each of the categories.

1.3 Thesis Outline

The remainder of this thesis is outlined in this section. In Chapter 2 we provide background information consisting of scientific and industrial work that is relevant for the work presented in this thesis. In Chapter 3 a choice for a implicit parallel data processing framework is motivated. It is discussed what the challenges are of implicitly parallelizing sNRC on this framework, and how these challenges were overcome. In particular the need for input projection is motivated and a reference approach is presented. In Chapter 4 the need for improving upon the reference input projection approach is motivated. An improved approach that uses static sNRC expression analysis is presented, along with a proof of correctness. Additional optimizations based on existing input projection approaches are performed and a more sophisticated algorithm for applying the projection is provided. In Chapter 5 the reference and improved parallel sNRC systems are evaluated and compared. The relative reduction of the input dataset by the input projection algorithms is investigated. By using a benchmark for semi-structured data, the runtimes and scalability of both systems are evaluated and compared on clusters of up to 16 nodes and datasets of up to 141 gigabytes. Finally, in Chapter 6 this work is concluded by summarizing the obtained insights. The work and process are reflected on and possible directions for future research are provided.

Chapter 2

Background and Related Work

This chapter aims to provide the reader with sufficient background information to understand the content and decisions made in the upcoming chapters of this thesis. A background introduction to query languages for semi-structured data is presented as well as the history of such query languages. Additionally, data processing platforms that allow for parallelized processing of large datasets, state-of-the-art query languages for semi-structured data, approaches to implicitly parallelize existing query languages for semi-structured data and approaches to handle datasets larger than the available amount of memory are discussed.

2.1 Data Processing Platforms

Processing huge quantities of data in parallel is a challenging task. To help developers with fragmentation of the data over a cluster of computers and with distribution of the computations, *parallel data processing frameworks* have become widely adopted [17].

A very popular framework is MapReduce [23] and its open source implementation Hadoop. MapReduce programs take as input a set of key/value pairs and output a set of key/value pairs as well. The computation consists of two functions that are written by the user. The *Map* function takes an input key/value pair and produces a set of intermediate key/value pairs. All intermediate key/value pairs are then grouped and all pairs with the same key are passed to a *Reduce* function. The Reduce function takes the set of all values that belong to a key and produces a set of output values. The Hadoop implementation of MapReduce also comes with the widely used Hadoop Filesystem (HDFS) [42] which is a distributed filesystem that features fault-tolerance and high throughput through replication. The HDFS is capable of storing very large datasets. Examples exist of companies that store petabyte scale datasets on the HDFS [1].

The Dryad [31] platform by Microsoft models each program as a Directed Acyclic Graph (DAG). In this model a data processing task is modelled as a directed graph starting at the input sources and ending at a data sink. The vertices are computational units that can be programmed by the users through a functional interface. The edges are channels for communicating data through files, TCP pipes and shared-memory. Dryad schedules handles parallel execution of the DAG and achieves parallelism by

[1] `https://wiki.apache.org/hadoop/PoweredBy`

scheduling the vertices simultaneously on multiple cores or nodes in a cluster. Further optimizations are performed at runtime, for example to reduce network communication by introducing aggregation vertices. A set of language extensions called DryadLINQ [48] allows writing imperative or declarative operations for datasets. These programs are automatically translated into a distributed execution plan for Dryad.

The simplicity of the MapReduce model is also a shortcoming. The *Map* and *Reduce* operations alone are not a good fit for naturally and efficiently expressing more complex data processing tasks [8]. Moreover, MapReduce and Dryad use an acyclic model that does not support reusing a dataset accross multiple operations [49]. Modern platforms use more powerful abstractions for parallel data processing [17] to facilitate more complex operations and reusing datasets accross multiple operations. Two of such processing platforms are Spark [49] and more recently Flink (previously Stratosphere) [5]. Both these platforms support iterative processing through cyclic dataflows. Spark was designed with the reusability of datasets in mind, filling gaps in areas such as iterative machine learning algorithms. To achieve this, Spark introduces an abstraction called *Resilient Distributed Datasets* (RDDs) to partition a collection of read-only objects across a set of machines. RDDs are typically kept in memory and can easily be reconstructed when a partition is lost. Similar to Dryad, Spark allows users to use operations such as the well-known *map*, *reduce*, *filter* and *foreach* by passing functions. These operations are performed in parallel for the RDDs.

Flink uses a programming model of *Parallelization Contracts* (PACTs) [8] which is a generalization of the MapReduce model that also works on key/value pairs. A PACT consists of an *Input Contract* and an *Output Contract* that define properties on the input and output data of the PACTs *User Function* (UF). Each PACT is responsible for partitioning the input values and the execution of its UF. The default set of PACTs in Flink contain the *Map* and *Reduce* PACT that we know from the MapReduce model as well as three PACTs that process two sets of input values: the *Match*, *CoGroup* and *Cross*. Flink features an optimizer that performs a cost-based optimization on a PACT program. After optimization the PACT program is compiled into a DAG and provided to Flink's execution engine Nephele. Similar to Dryad, each vertex contains PACT code and each edge is a communication channel. The input and output contracts ensure that the UF in each PACT gets the data in its required format. Finally, the DAG is *spanned* to create a parallel data flow where each vertex may have multiple parallel instances that will be executed. This parallelization is determined automatically by the PACT compiler.

Both Spark and Flink offer an API for processing streams of data. To achieve this, Spark uses micro-batches. Flink on the other hand uses *true* streaming based on Google's Dataflow Model [4], resulting in a lower latency of pipelined operations [50]. In contrast to Spark, Flink offers automatic off-heap memory management [24]. Flink's features operators that are capable of running with very low memory available, and can spill data to disk very efficiently. This ensures that Flink will not run out of memory, regardless of the size of the dataset.

2.2 Nested Relational Calculus

The Nested Relational Calculus (NRC) [47] is a query language that can be used to write functional programs that work with and iterate over collection types such as bags, sets and lists. NRC uses a predefined set of base data types that can be combined to introduce nesting and to create finite collections.

The NRC originates from Codd's well-known powerful model for representing relational data in large databases [20]. Codd separated the internal representation in the database from the model, allowing more flexibility to change the internal representation when needed. Codd's work imposes a first normal form requirement on relational databases. This was not considered as the correct approach for modern databases [47], so a variety of nested relational database designs followed [40, 43, 21]. These designs allow use of nested relations without losing the expressive power of Codd's original model.

Finally, Wong introduces the Nested Relational Calculus (NRC) as a nested relational language and the Nested Relational Algebra (NRA) [13, 47, 33]. Wong's goal is to formalize the calculi and algebras from the previous nested relational database designs, and to reduce the complexity that was introduced over time without losing expressive power. The NRC and NRA have some useful properties: for example Wong shows that functions that are definable in the algebra have polynomial time complexity, and he presents a set normalization rules [46].

Hidders, Kwasnikowska, Sroka, Tyszkiewicz and Van Den Bussche [30] have demonstrated that the NRC is very useful as a basis for developing a formal and graphic workflow notation for dataflows. By combining the workflow formalisms of Petri nets with the NRC operations for complex values they design a language that allows construction of dataflows that are guaranteed to generate output and that have a convenient graphical representation.

In different work Grabowski, Hidders and Sroka [28] show that the NRC can also be used to represent existing optimizations for data processing systems. They state that using the NRC as a basis for such optimizations is a more general and elegant approach that combines classical optimizations and system-specific 'ad-hoc' formalisms.

2.3 Semi-structured Data

Semi-structured data has become widely used on the internet: consider for example HTML websites and the XML and JSON data formats that are hugely popular for exchanging data on the modern web. We have also seen a growth in popularity for database systems that use a semi-structured data model for storage, such as MongoDB or eXistdb. This increased usage of semi-structured data models also requires new methods for effectively handling, manipulating and querying (large amounts of) semi-structured data.

Semi-structured data is typically irregular. This means that the data has no schema or a partial schema. Data is not always present and the same concepts may be represented in different ways. The schema and the values itself are intertwined in the

dataset. Semi-structured data is most often found during the exploration phase of data analysis [10].

2.4 Query Languages for Semi-Structured Data

The Nested Relational Calculus, the Object Exchange Model (OEM) for exchanging semi-structured data and OEM's query language [37] have served as inspiration for designing query languages for semi-structured data [45]. The UnQL [16] language introduces a tree model for semi-structured data. The authors focus mainly on XML data and shows how the tree model can be queried using structural recursion. StruQL [26] was built to deal with managing websites of increasing size. The authors use a graph model based on OEM to handle semi-structured data from heterogenous sources. Queries are optimized by pushing the queries to the data sources, and by exploiting indices when available. Quilt [18] served as the basis for the standardized XML query language XQuery [11]. Quilt formalized concepts from previous XML query languages, abbreviates its path expressions from XPath [9] and takes important lessons from SQL.

Recently, following the Big Data hype and the introduction of mainstream data processing platforms such as the open source MapReduce [23] implementation Hadoop, more languages that allow analysis of large semi-structured datasets have come to see the light. Yahoo has introduced Pig Latin [36], which seeks to fit in between the declarative SQL language and the procedural low-level MapReduce approach. Pig Latin uses high-level relational algebra style primitives to allow for traditional database optimizations and uses a nested data model of atomic values, tuples, bags and maps. Pig Latin programs are compiled to a set of Hadoop MapReduce jobs. Facebook has introduced Hive [44] as a data warehousing solution on top of Hadoop. Hive's query language HiveQL does not uniformly handle nested data collections, but currently does have support for querying JSON data. IBM has developed Jaql [10], a scripting language designed for the analysis of large semistructured datasets. Jaql shares many goals with Pig, but focuses on reusability and composability of the scripts and allowing the use of partial data schemas.

Fegaras, Li, Gupta and Philip introduced their language MRQL for querying XML on top of a MapReduce environment in 2011 [25]. Their work originates from a desire to have a declarative query language that is amenable to optimization for a MapReduce environment. They state that it is not possible to apply traditional relational query optimization techniques to the MapReduce domain. Where Pig and Hive perform several rule based optimizations, MRQL distinguishes itself by applying a richer set of cost-based optimizations. The optimizer focuses only optimizing the join operators however. Currently, MRQL is only an Apache Incubator project supporting various raw data formats, such as XML, JSON and CSV, and data processing platforms, such as Hadoop, Spark and Flink.

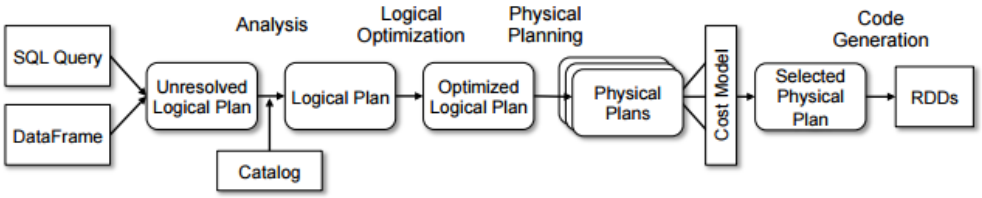

Figure 2.1: Phases of query planning in Spark SQL. All rounded rectangles represent Catalyst trees. From [6].

In 2015 a new module called SparkSQL [6] for the popular general purpose data processing framework Spark was released to support relational processing for Spark's Resilient Distributed Datasets as well as other data sources. This module includes support for semi-structured data sources such as JSON. SparkSQL operates on DataFrames, which are equivalent to tables in a traditional relational database. For optimization SparkSQL uses its newly developed Catalyst optimizer. Catalyst is an extensible optimizer that considers a SparkSQL program as a tree with rules that can be applied to manipulate the tree. Developers can add new optimization rules themselves. The optimizer is used during multiple phases of query planning for SparkSQL as shown in figure 2.1. To handle JSON sources, SparkSQL automatically infers the schema from the data and registers the dataset as a table.

PAXQuery [17] follows a different approach by applying implicit parallelization to the existing XQuery language for XML. The authors convert the Xquery to an equivalent in the XML Tuple Algebra [34] where the query is unnested. The XML tuple data is then converted to a custom Flink nested data format and the algebraic expressions are converted to a sequence of Flink operations on the data. By automatically translating XML queries into Flink plans, PAXQuery relies on the automatic optimizations that are performed by Flink. PAXQuery is evaluated using a set of modified queries from the XMark [41] benchmark and it is shown to scale up well with moderate overhead as the data volume and number of nodes increases.

2.5 Nested Relational Calculus for Semi-Structured Data

As we have seen in the previous section there have been many developments in the area of querying semi-structured data in the last two decades. The query languages have become more sophisticated over time, though they have drifted away from theoretical calculi such as the NRC. Hidders et al. [29] have developed a dialect of the Nested Relational Calculus, called the Nested Relational Calculus for Semi-structured Data (sNRC). The goal of sNRC is to design a language that (1) is well-understood, (2) can be leverage implicit parallelization of existing data processing platforms and (3) allow the application of well-known optimization techniques. Moreover, sNRC is a solid basis for designing a data workflow notation [29, 22]. The work on sNRC connects the trend of implicit parallelization of query languages for semi-structured data with the well-understood theoretical basis of the NRC.

2.5.1 sNRC Data Model

The sNRC uses a simple nested data model called the *sNRC Data Model* which uses the following concepts [29]:

1. $\mathcal{B}$ is the set of basic value constants such as strings, booleans and integers.
2. Bags are denoted as $\{\{1,2,3,a,b,c\}\}$ and an empty bag is denoted as $\emptyset$. The additive bag union is denoted as $\uplus$ and $\{\{f(\bar{x}) \mid \varphi(\bar{x})\}\}$ denotes bag comprehension.
3. $\langle x,y\rangle$ is an ordered pair containing values x and y.
4. $\mathcal{V}$ is the set of nested values. An element in $\mathcal{V}$ is a bag where each element is either a basic value in $\mathcal{B}$ or an ordered pair $\langle x,y\rangle$ with $x,y \in \mathcal{V}$. Bags may be heterogeneous, and only finitely nested values are permitted.

2.5.2 sNRC Syntax

The original syntax for the calculus over bags that sNRC uses is defined as follows:

$$E ::= \mathbf{in} \mid X \mid C \mid \langle E,E\rangle \mid E.1 \mid E.2 \mid \tag{2.1}$$

$$\emptyset \mid E \uplus E \mid \{[E \mid X \in E,\ldots,X \in E]\} \mid \tag{2.2}$$

$$B(E) \mid \dot{\mathbf{set}}(E) \mid E \doteq E \tag{2.3}$$

Here X are variables, C are basic value constants, $\{[e|\Delta]\}$ is the flattening bag comprehension and B are user-defined functions. To ensure the definedness of the result of an expression E, all input and output values are assumed to be bags. For operators that do not expect a bag, a rule of thumb is that they are mapped over the elements of the bag instead, flattening the result if needed.

2.6 Input Projection for Semi-Structured Data

As we have seen in the previous sections, query languages for semi-structured data rely on the optimizers of the data processing platform on which they are built [6, 17, 25]. *Input Projection* techniques were developed to help single machines cope with nested datasets that are too large for memory. Input Projection is an optimization that given a dataset and a query produces a projected dataset of at most the same size and on which evaluation of the query generated the same result.

One goal of this thesis is to evaluate whether input projection is a useful optimization for modern query languages for semi-structured data, that make use of implicit parallelization. We expect that reducing the size of the input can provide speedups for such languages by improving throughput for partitioning steps, by reducing the time required for serialization and by reducing the overall memory footprint of the dataset resulting in less time spent on garbage collection. In this section we will provide an overview of the most popular input projection techniques.

Jagadish, Lakshmanan, Srivastava and Thompson [32] present a Tree Algebra for XML (TAX) that is complete for the relational algebra and uses a labeled tree as a data

model. An important part of their work is the concept of a *Pattern Tree*. A pattern tree is a node-labeled and edge-labeled tree, where each node has a distinct label and each edge describes a parent-child (pc) or ancestor-descendant (ad) relationship. A pattern tree is accompanied by a set of boolean predicates that can be applied to the nodes. See Figure 2.2 for an example of an XML dataset modelled as a tree and two pattern trees. Given a collection of data trees and a pattern tree it is now possible to find embeddings of the pattern tree in the data trees such that the nodes in the embedding match the structure of the tree, and the nodes in the embedding verify the set boolean predicates. Though the authors state to be working on an implementation of TAX, called Timber, no experiments were performed unfortunately.

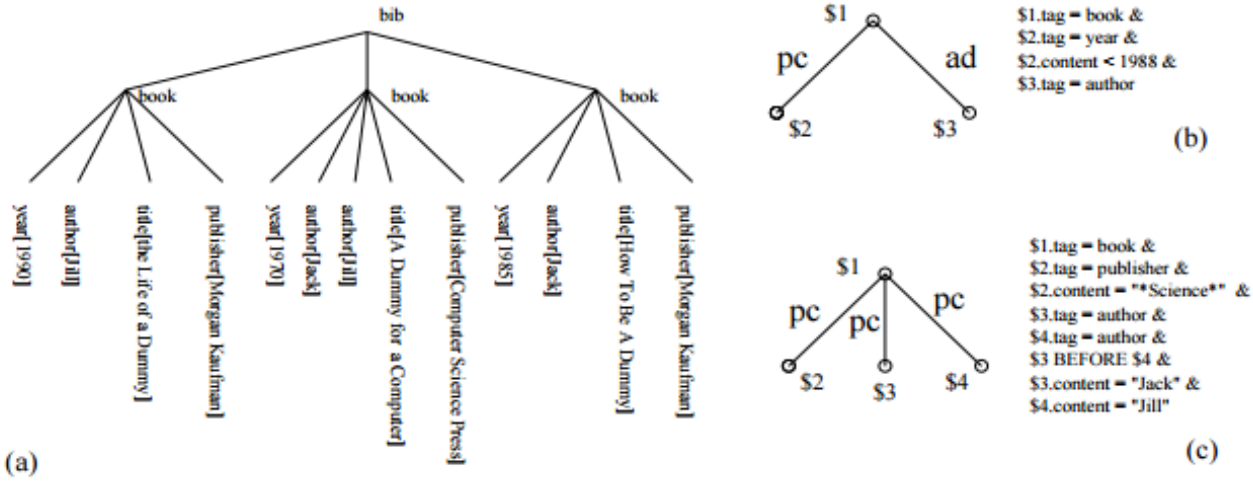

Figure 2.2: (a) A one-tree XML database and (b,c) Two pattern trees. From [32]

Many XML querying systems followed that model, representing XML selection patterns as a tree with parent-child and ancestor-descendant relationships. The tree is generally matched by first matching the structural relationships against the database, and then joining these matches together. Bruno, Koudas and Srivastava [15] generalize two approaches for the two steps into an algorithm called TwigStack. The work on TwigStack is further generalized in the Twig2Stack algorithm, which uses the Generalized Tree Pattern model that is more suitable for modelling XPath and XQuery statements [19].

Marian and Siméon [35] present an XML input projection algorithm to improve the capability of XQuery engines to work on datasets that are larger than memory. The full data model needed to be built in memory to allow using the full complexity of XQuery. Their approach uses a static path analysis algorithm that analyses a query before it is executed by applying a set of inference rules in a bottom-up fashion. Marian and Siméon use a different model than the pattern tree. Instead, their static path analysis algorithm results in a set of projection paths, each describing one part of the input nested dataset that are required for computing a complete and correct answer to the query. A loading algorithm is then used to match each of the paths to obtain the

projected dataset of reduced size. Their approach produces paths that include the full path from the root of the document. Moreover, the presented loading algorithm is inefficient when the number of projection paths increases. Bressen et al. [14] build on the work of Marian and Siméon to further reduce the size of a projected dataset by pruning unnecessary siblings and descendants. To achieve this, they require the use of an index structure to effectively perform the transformations.

Chapter 3

Reference Parallel sNRC System

One of the goals of this thesis is to implicitly parallelize the Nested Relational Calculus for Semi-structured Data (sNRC). In this chapter a first, or reference, approach for an implicitly parallelized sNRC system is described and the decisions that were made while designing the reference system are motivated.

3.1 Apache Flink as Implictly Parallel Data Processing Platform

In the previous chapter several popular implicitly parallel data processing platforms were outlined. For this thesis Apache Flink is chosen as the data processing platform that will be used to implicitly parallelize sNRC. In this section this decision will be motivated.

In the background information chapter it was discussed how XQuery is inspired by the Nested Relational Calculus. It has already been shown that Flink can be used to efficiently implicitly parallelize XQuery [17]. Because XQuery and NRC show many similarities, we are confident that efficient implicit parallelization of sNRC using Flink as a data processing platform is also possible.

When we compare the alternatives we again find that Flink is a suitable data processing platform to use. When we consider MapReduce we find a very mature platform in Hadoop and a very general programming model that has been shown to scale very well. However, this model is not a natural fit to more complex operations such as joins [8] which requires the programmer to work around the programming model, potentially resulting in bad performance [38]. Moreover, the Flink model contains the *map* and *reduce* operators as well as many other operators. Dryad was discontinued by Microsoft in favor of a Hadoop implementation for its cloud computing platform Azure.

There is no up-to-date scientific comparison of Flink and Spark, but the two are frequently compared on different channels such as specific conferences [7], the well-known Question and Answer site for programmers StackOverflow [3, 1, 2], interviews with Spark or Flink core team members [50] as well as many blogs of individual people. Because the Spark and Flink core members with in-depth knowledge of the plat-

forms actively participate in these comparisons we have chosen to utilize this information to come to a decision.

The main differences between Flink and Spark are that Flink is a streaming engine that can emulate batch processing, while Spark is a batch engine that can emulate streaming using micro-batches. As also discussed in the background section this means that Flink can pipeline the data between different operators without having to wait for intermediate results. This can result in lower latencies. Additionally, Flink has efficient mechanisms for spilling data to persistent disks when datasets get too large for memory and easy-to-use automatic configuration and optimization mechanisms. Spark is a more mature platform that supports technologies that Flink does not support at this moment. These technologies are not relevant for the work in this thesis however, and therefore we choose Flink as a data processing platform based on the advantages listed above.

For this thesis Apache Flink version 0.10.1 is used. Flink offers both a Java and a Scala API. Both the Scala and the Flink Scala API are used for the query implementations, and Java was used for developing the data model and data loading algorithms.

3.2 sNRC Data Model

The sNRC Data Model for which the input projection is defined is described in [29] and briefly described in this Thesis in Section 2.5.1. To make further discussion easier we provide several definitions of properties of sNRC data.

Definition 3.1 (Direct sub-element relationship)**.** *x is a* direct *sub-element of y if:*

- *x is a basic value in $\mathcal{B}$ or an ordered pair $\langle a,b\rangle$, y is a bag, and $x \in y$*
- *x is a bag, y is a tuple and $x \doteq y.1$ or $x \doteq y.2$*

Definition 3.2 (Sub-element relationship)**.** *x is a sub-element of y if:*

- *x is a direct sub-element of y*
- *x is a basic value in $\mathcal{B}$ or an ordered pair $\langle a,b\rangle$, y_2 is a bag, $x \in y_2$, and y_2 is a sub-element of y*
- *x is a bag, $x \doteq y_2.1$ or $x \doteq y_2.2$, and y_2 is a sub-element of y*

Using the concepts of the sNRC data model, a definition of an sNRC Dataset can also be provided:

Definition 3.3 (sNRC Dataset)**.** *An **sNRC Dataset** is a 2-tuple (N,V) where:*

- *N is a collection of elements from the set of nested values $\mathcal{V}$ contained by the dataset. Each element in N except for those in V have a super-element in N.*
- *V is a bag of top-level sNRC values in N. These are the values that do not have a super-element.*

3.2.1 sNRC Data Model Implementation

The description of the sNRC data model was provided in the Background chapter in Section 2.5.1 and useful properties are defined earlier in this section. In this subsection it will be described how the sNRC data model was implemented for the reference implicit parallel sNRC system. Figure D.1 in Appendix D shows an UML diagram of how the data model is structured.

The implementation of the Data model strictly follows the sNRC data model. We distinguish between a *Bag* and *Data*. A *Bag* contains sNRC *Data* elements, which are either a *Basic Value* from $\mathcal{B}$ or an *Ordered Pair* $\langle x, y \rangle$ with $x, y \in \mathcal{V}$.

The original sNRC data model is untyped. In the implementation all data is initially untyped by instantiating the values as a String. Though a type system is hidden from the user, there is a type system used in the background in order to allow comparison of values. When comparing two values, the system attempts to lazily convert the value to a numeric value to allow comparison of numbers and booleans. Tuples can only be compared to tuples, and only *Singleton Bags* may be compared to other *Singleton Bags* or *Basic Values* by applying flattening. Tuples are excluded here to prevent expensive comparisons due to unknown levels of nesting. See Table 3.1 for an overview of currently supported comparisons.

Table 3.1: Comparison of sNRC - Flink types

	String	Int	Double	Boolean	Tuple	Bag
String	Yes	Yes	Yes	Yes	No	Singleton
Int	Yes	Yes	Yes	Yes	No	Singleton
Double	Yes	Yes	Yes	Yes	No	Singleton
Boolean	Yes	Yes	Yes	Yes	No	Singleton
Tuple	No	No	No	No	Yes	No
Bag	Singleton	Singleton	Singleton	Singleton	No	Singleton

All Data objects allow the same set of basic operations on the data such as comparisons and hashcode computation. Moreover, all data objects allow projection operations: selecting the first or second field or selecting by key. These operators only yield a result for *Ordered Pairs* however. These projection operations will be discussed in more detail later in this chapter. For *Bags* the projection operations are mapped over all elements contained by the bag, as described in the original sNRC document [29].

3.3 Adapting to a Streaming Data Processing Engine

In the beginning of this chapter it was explained that Flink is a streaming dataflow engine following Google's Data Flow model [4]. Even though for this thesis the Flink batch API is used, Flink treats batch applications as special cases of stream processing applications. This means that operators are efficiently pipelined wherever possible, reducing latency and increasing throughput by forwarding data from one operator to another where possible.

Like with other parallelized systems for querying semi-structured data [17, 36, 10] the entire dataset can not be loaded into memory. This also complies with one of the goals of this thesis to process datasets that are much larger than the available main memory. Instead it is necessary to fragmentize the dataset into a bag of smaller elements, each of which fit into main memory. These fragments can then be processed one at a time by the Flink operators, and streamed to the next operator when processing is completed.

In this Thesis data generated by the XMark benchmarking tool [41] is used. XMark generates semi-structured datasets in an XML format. The XMark benchmark will be discussed in more detail in Chapter 5. In Figure 3.1 you can find a tree model that describes the nested structure of an XMark generated dataset.

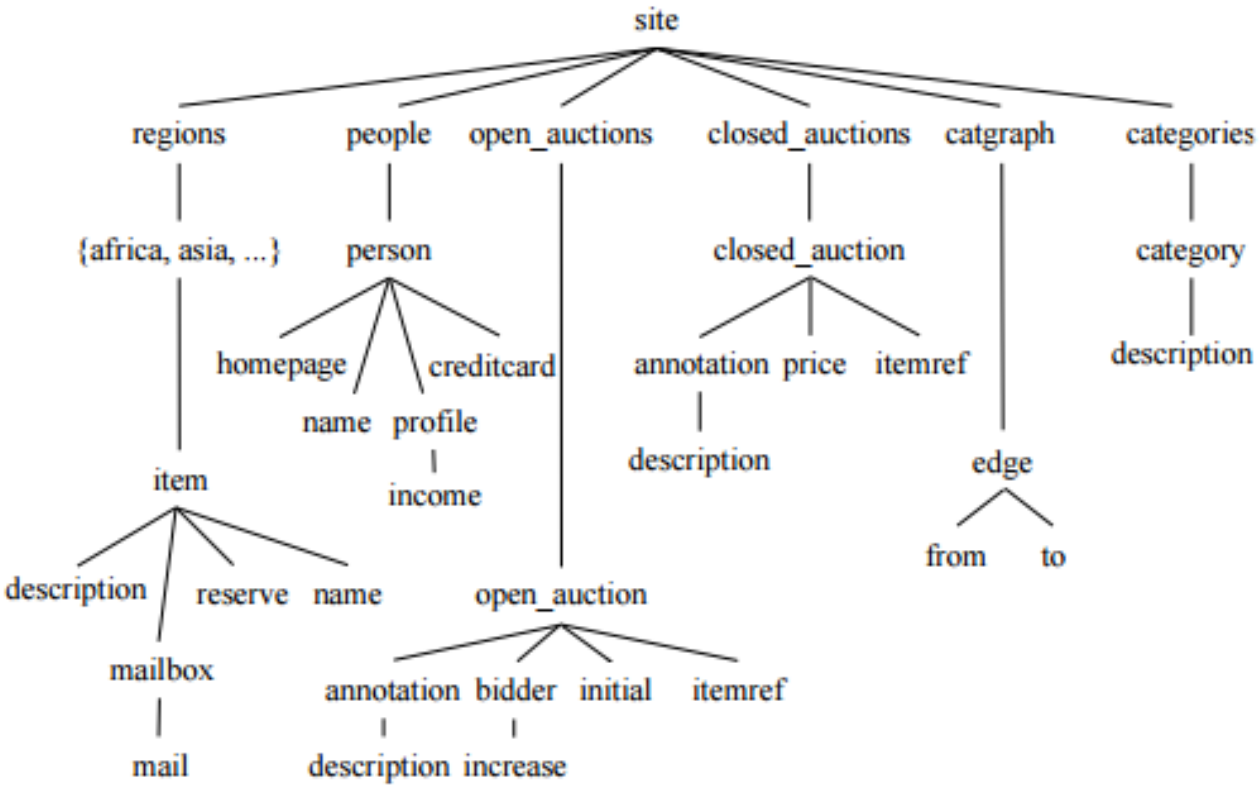

Figure 3.1: The nested data structure of the XMark benchmark dataset. From [41].

The Dataset has a single top-level nested value, with a key *site*. The datasets that are used for this thesis are sufficiently large that the *site* nested value will not fit in main memory of a computing node. If we consider the XMark Query 1 example and examine the query in 3.2 more closely, we see that to answer this query only the sub-elements with key *person* are required. Each of these elements with key *person* is only a fraction of the site of the total dataset, and is a small enough fragment be processed by Flink's streaming engine. An example of the semi-structured data of such a person object in XML format is shown in Listing 3.4. In the next sections it will be discussed how input projection is used to fragmentize the dataset for the XMark queries by this

reference system.

3.4 Extended sNRC Syntax

Before discussing how the actual input projection is implemented for the reference system, an extended sNRC syntax is introduced in this section. The original sNRC Syntax is described in [29] and briefly described in this thesis in Section 2.5. For convenience and readability the extended sNRC syntax is used in this thesis from this point onwards. The extended sNRC syntax is defined as follows:

Definition 3.4 (Extended sNRC Syntax)**.**

$$\begin{aligned} E ::= & \boldsymbol{in} \mid X \mid C \mid \langle E, E \rangle \mid E.1 \mid E.2 \mid \\ & \emptyset \mid E \uplus E \mid \{[E \mid X \in E, \ldots, X \in E]\} \mid \\ & B(E) \mid \dot{\boldsymbol{set}}(E) \mid E \doteq E \mid \dot{\boldsymbol{size}}(E) \mid \\ & E.[key] \mid E..[key] \end{aligned}$$

The expressions that were appended to the original syntax can be (re-)written to the original syntax as follows:

1. $\dot{\textbf{size}}(E)$ computes the number of elements contained by a bag.
2. $E.[key]$ is an alias for $\{[a \mid a \in E, a.1 \doteq \{\{key\}\}]\}$. This expression selects tuples that are direct sub-elements of an sNRC value based on their key.
3. $E..[key]$ is similar to $E.[key]$, but instead works on *all* sub-elements instead of only the direct sub-elements. This is similar to the 'descendant' relationship that is well known from other languages such as the XQuery language [11].
   For example we could write $E..[key]$ where E is a bag with arbitrary elements as follows:

   $$\begin{aligned} & X = E.[key] \\ & \dot{\textbf{while}}(X.2.[key] \neq \emptyset) \\ & \quad X = X \uplus X.2.[key] \end{aligned}$$

   Note that a while loop is not present in the sNRC syntax, so it is assumed that sNRC is implemented in a language that supports while loops. In this case Scala [1] is used. Flink also supports iterative distributed processing of datasets.

3.5 Input Projection

In order to fragmentize the input dataset we make use of a technique called *Input Projection*. Input projection is a well known technique that has already been applied to the domain of querying semi-structured data, in particular XML data. In Section 2.6 we have discussed these existing approaches. The definition of input projection that will be used in this thesis is the following:

[1] http://www.scala-lang.org/

Definition 3.5 (Input Projection)**.** Input Projection *is function that given an input dataset and a query produces a* projected dataset*. A projected dataset is a dataset of at most the same size as the original dataset for which evaluation of the query yields the same result as for the original dataset.*

For the reference system the first concern is to fragmentize the dataset to allow parallelization of sNRC on Flink. Following the used definition for input projection, the fragmentized dataset should contain at least the data that is required to compute a correct result for an sNRC query. In Chapter 4 an improved input projection approach is presented that attempts to select at most those elements that are necessary to correctly answer an sNRC query. Before providing the reference input projection algorithm, some additional definitions and notations are introduced in the next section.

3.5.1 Projection Expressions

By using a fragment of the extended sNRC syntax we can define *projection expressions*. A set of these projection expressions can be used to describe a subset of the original dataset.

The following fragment of the Extended sNRC syntax is used for describing projection expressions:

Definition 3.6 (Projection Expression Syntax)**.** $E ::= E.1 \mid E.2 \mid E.[key] \mid E..[key]$

And a *Projection Expression* is then defined as follows:

Definition 3.7 (Projection Expression)**.** *A projection expression is an sNRC expression using a fragment of the extended sNRC syntax that is described in Definition 3.6 that describes a subset of the input dataset.*

3.5.2 Projected sNRC Dataset

By taking a finite set of projection expressions and an input sNRC dataset, a projected sNRC dataset can be determined. Two concepts need to be defined for projection. First the single projection operation is defined:

Definition 3.8 (Projection Operation)**.** *The operation* ***Project****(P,D) returns an sNRC dataset resulting from evaluating projection expression P on sNRC dataset D.*

And next we can generally define projection for an sNRC dataset:

Definition 3.9 (Projection)**.** *Given an sNRC Dataset* $D = (N,V)$*, and a set of projection expressions* $(P_1, P_2, \ldots, P_n)$*. A projected sNRC Dataset* $D' = (N',V')$ *is defined as follows:*

1. $N' \subset N$.
2. $n \in N'$ *if:*
   a) $\exists i$ *such that* $n \in$ ***Project****(*P_i*, D).*
   b) $\exists n_2, i$ *such that* $n_2 \in$ ***Project****(*P_i*, D),* n *is a sub-element of* n_2.
3. V' *= a bag containing all elements from* N' *that do not have any super-elements*

3.5.3 Correctness Definition

One of the main goals of Projection Expressions as described in subsection 3.5.1 is to ensure that the evaluation of a query on a projected sNRC dataset yields the same results as on the original dataset. For an input projection algorithm to be correct, the following theorem must hold:

Theorem 1 (Correctness)**.** *Let D be a sNRC dataset and E be an sNRC query. Let PE be the set of projection expressions resulting from the input projection algorithm. Take D' to be the result of the projection of PE on D. It then holds that the results of the evaluation of E on D and results of the evaluation of E on D' are identical.*

3.6 Reference Loading Algorithm

In this section it is described how projection is implemented for the reference parallel sNRC system. The goal of this algorithm is to apply input projection to the input dataset to fragmentize the input.

For the reference system the projection expressions are manually determined from the original XMark queries. The chosen projection expressions are based on the elements over which is iterated by the original queries. For example for XMark query 1 as listed in Listing 3.2 a projection expression **in**.[*site*].2.[*people*].2.[*person*] is chosen. Because all 'person' elements and their sub-elements are retrieved, this obviously includes all data required to answer the query. The projection expressions that are chosen for the reference input projection for each of the queries are listed in Appendix A.

We will next describe how we apply the projection for the reference system based on the chosen set of projection expressions. Taking inspiration from [32, 15] each projection expression is modeled as a node-labeled and edge-labeled *projection tree*. An edge may be labeled either 'pc' (parent-child) for a direct sub-element relationship or 'ad' (ancestor-descendant) for all sub-elements following the extended sNRC syntax from Section 3.4. The node label is the selection criterion, to select the input, the first or second value, or to select based on a key value. In the case of the reference system each node has at most one (direct) sub-element, so the projection tree could be considered as a linked list. Figure 3.2 shows the projection expression for XMark Query 1 as a pattern tree.

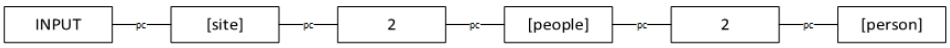

Figure 3.2: Projection Expression for XMark Query 1 modeled by an Input Graph

The reference algorithm also maintains a pattern tree that represents the location of the parser with respect to the input. We will refer to this pattern tree as the *input path*. Upon encountering a new element, it is appended to the input path. When encountering the end of the element, it is removed from the input path. The input path and pattern trees may be very efficiently compared using the Algorithm 3.6.2. False matches can be found in $O(d)$ while matches are confirmed in $\Theta(d)$, with d the depth of the pattern tree. Most matches are completed in $O(1)$, when the keys do not match.

The reference system uses an event-based parser for parsing the XMark datasets. The event based parser generates an event every time the start or ending of an element occurs. The pseudocode presented in Algorithm 3.6.1 represents the code that handles these events, maintains the input path and checks whether there is a match with the projection expressions using the matchInput function defined in Algorithm 3.6.2. Whenever there is a match, the complete current element is parsed to the sNRC data model using the approach of Subsection 3.7.1. The sNRC element is then forwarded as input to the Flink program that computes the answer to the query.

Algorithm 3.6.1: FINDNEXT()

global *Reader*
local $InputPath, ProjExp$
while Reader has next
do
- $next \leftarrow$ next Reader event
- **switch** *next*
  - **case** $START\_ELEM$
    - $CurrentTag \leftarrow$ current element tag
    - $InputPath \leftarrow InputPath + InputNode(CurrentTag)$
    - **if** $matchInput(InputPath, ProjExp)$ **then return** (**true**)
  - **case** END_ELEM remove last element from InputPath
  - **case** $START\_DOC$ $InputPath \leftarrow EmptyPath$
  - **case** END_DOC **return** (**false**)

return (**false**)

Algorithm 3.6.2: MATCHINPUT($InputPath, ProjExp$)

if $ProjExp.value \neq InputPath.value$
then return (**false**)
if $ProjExp$ has parent
then
- **if** $InputPath$ has parent
  - **then return** $(matchInput(InputPath.parent, ProjExp.parent))$
  - **else return** (**false**)

if $ProjExp$ has ascendant
then
- **if** $InputPath$ has ascendant
  - **then return** $(matchInput(InputPath.ascendant, ProjExp.ascendant))$
  - **else return** (**false**)

return (**false**)

This algorithm follows the definition of Input Projection from Definition 3.5. The fact that the algorithm verifies the correctness theorem (Theorem 1) follows from the selection procedure for the projection expressions in combination with the fact that for each input element all sub-elements are loaded as well. In the evaluation chapter, Chapter 5, it is shown through experimentation that the algorithm indeed results in a successful fragmentation, allowing processing of datasets larger than main memory using Flink. Additionally, it is shown that the algorithm produces projected datasets that are smaller than the input dataset.

3.7 Reference Parallel sNRC - Flink Architecture

In this section the architecture of the reference parallel sNRC system is described. This system is built to execute sNRC equivalents of the queries from the XMark benchmark. In Appendix D you can find the UML that describes the structure of the most important classes of the reference parallel sNRC system.

The system is centered around a benchmark suite that consists of benchmark objects. Each of these objects represents a single sNRC XMark query implemented on top of Flink. Tools to measure metrics such as the runtime wrap each benchmark object.

The queries themselves are converted from the original XQuery definitions to Scala sNRC - Flink code that uses the sNRC data model and the Flink Batch Processing API. This translation from sNRC expressions to Flink operators is very natural. See for example the code in for XMark query 1 in XQuery, sNRC and sNRC-Flink respectively in Listings 3.1, 3.2 and 3.3. The bag comprehension naturally translates to a flat map, and the conditional translates to a filter. All sNRC versions of the XMark queries can be found in Appendix A. The sNRC - Flink implementations can be found in the source code repository [2]).

Listing 3.1: XMark Query 1 in XQuery

```
let $auction := doc("auction.xml") return
for $b in $auction/site/people/person[@id = "person0"]
       return $b/name/text()
```

Listing 3.2: XMark Query 1 in sNRC

$$\{[b.2.name.2 \mid b \in \mathbf{in}.site.2.people.2.person, b.2.@id.2 \doteq \text{"}person0\text{"}]\}$$

Listing 3.3: XMark Query 1 in sNRC-Flink

```
val input : DataSet[SNRCData] =
       env.readFile(new XML2SNRCInputFormat("[site].2.[people].2.[person]"), inPath)
val person = input filter { _.select(".2.@id") equals "person0" }
val result = person flatMap { _.select(".2.name") }
```

The XML2SNRCInputFormat in Listing 3.3 handles parsing of the XML and fragmentizes the input by applying input projection based on the projection expressions of the query. The input projection results in a bag of elements that is used for answering the query. In the case of XMark query 1, this is a bag of ordered pairs that represent the 'person' elements. Note how the Flink Dataset fulfils the role of an sNRC bag. Both are an unordered collection of elements that may contain duplicates and that allow parallel processing of the contained elements.

3.7.1 XML to sNRC Data Model

The sNRC data model is designed to allow simple conversion of other semi-structured data formats to the sNRC data model. For this thesis we will make use of a dataset

[2] https://github.com/PHameete/flink-snrc

in the XML [12] format. In this section it is described how XML is converted to the sNRC data format.

In Listing 3.4 a fragment from an XML dataset can be found that is used in this thesis. The XML data model is centered around tags. The opening tag and closing tag mark the beginning and end of an XML element. For example in Listing 3.4 '<name>' and '</name>' mark the beginning and ending of the 'name' element. All characters between the opening and closing tag are the content of the element. This content can consist of sequences of characters (this is the case for the 'name' element) as well as other elements (this is the case of the 'address' element). Multiple elements with the same tag are allowed to coexist. An element may also have attributes defined in the opening tag, such as the attribute 'id' in the opening tag for the person element. Unlike elements, attributes have only a single value and an attribute can appear at most once for each element.

Listing 3.4: Person element from an XMark dataset in XML format

```
<person id="person0">
	<name>Huei Demke</name>
	<emailaddress>mailto:Demke@uu.se</emailaddress>
	<address>
		<street>95 Grinter St</street>
		<city>Macon</city>
		<country>United States</country>
		<zipcode>32</zipcode>
	</address>
	<creditcard>8182 1228 4493 3873</creditcard>
	<profile income="55386.86">
		<education>High School</education>
		<gender>male</gender>
		<business>No</business>
		<age>37</age>
	</profile>
</person>
```

As was stated earlier, parsing XML to the sNRC data model is straightforward. Each element in the XML dataset is converted to an sNRC ordered pair. The first field of the ordered pair is a singleton bag containing the XML element's tag value. The second field of the ordered pair is a bag that contains the content and attributes of the element. Character sequences are parsed as basic values. Attributes are parsed as ordered pairs where the first field is a singleton bag containing the attribute's name prepended with a '@' character to distinguish between attributes and content elements. The second field of the attribute is also a singleton bag containing the value of the attribute. All sub-elements of the parsed the element can be parsed as ordered pairs recursively following this same approach.

When the XML data in Listing 3.4 is parsed using the above approach we end up with the sNRC dataset in Listing 3.5.

Listing 3.5: Person element from an XMark dataset in sNRC format

```
{<{person}, {
	<{id}, {person0}>,
```

```
        <{name}, {Huei Demke}>,
        <{emailaddress}, {mailto:Demke@uu.se}>,
        <{address}, {
                <{street}, {95 Grinter St}>,
                <{city}, {Macon}>,
                <{country}, {United States}>,
                <{zipcode}, {32}>
        }>,
        <{creditcard}, {8182 1228 4493 3873}>,
        <{profile}, {
                <{income}, {55386.86}>,
                <{education}, {High School}>,
                <{gender}, {male}>,
                <{business}, {No}>,
                <{age}>, {37}>
        }>
}>}
```

Chapter 4

Improved Parallel sNRC

In this chapter an improved input projection algorithm which computes projection expressions based on static analysis of an sNRC query is presented. It is proven by applying induction that the resulting projected dataset contains all elements required to correctly answer the query. Finally it is shown how a set of projection expressions can be combined into an annotated projection tree, and an improved loading algorithm that applies the projection to the input dataset by using such a projection tree is presented.

4.1 Motivation for Improving the Reference System

In the previous section the design of the reference parallel sNRC that is built top of the implicitly parallel data processing platform Apache Flink is discussed. By using an input projection algorithm only the relevant elements are selected from the input dataset. The reference parallel sNRC system from Chapter 3 selects all the sub-elements of these elements as well. This approach results in a fragmentation that allows using the streaming engine of Flink for the evaluation of sNRC queries. There is still room for improvement however: as we will see in chapter 5 where the system is evaluated, the reference system does not perform and scale well for more complex queries that for example join two parts of the input dataset.

The goal of the improved input projection algorithm is to improve upon the reference system in terms of the size of the input data. We expect that reducing the size of the input data will result in better performance and scalability for more complex queries. Such complex queries have one or more partitioning steps, where the data needs to be serialized and transmitted between the different nodes in the Flink cluster. Appendix B shows all Flink execution plans for the used queries. For this work we expect three effects of reducing the size of the input nested data. Each effect is expected to reduce the runtime, and improve scalability for 'complex' sNRC queries that contain one or more partitioning steps. The three effects that we expect are the following:

1. When Flink operators cannot be directly chained (for example at partitioning steps) the data needs to be serialized. Smaller nested values allow for quicker serialization.

2. When partitioning takes place, the serialized nested values are transmitted between the nodes in the Flink cluster. Smaller nested values are expected to result in more throughput when sending values from one node to another.

3. Many nested values are processed by Flink's operators. These values are read as input, processed and then forwarded to the next operator. Values that are no longer required by the operator are cleaned up by the garbage collector. Smaller nested values have a smaller memory footprint, which we expect to result in less time spent on garbage collection.

The following example describes how the reference system input projection can be improved upon. Queries described by an sNRC expression are evaluated against a dataset that can contain information about many different subjects as can be seen in Figure 3.1. Many of these subjects may not be relevant for the evaluation of a particular sNRC query. For XMark Query 1, the reference input projection algorithm selects the person elements including all sub-elements as shown in Listing 3.4. However, to answer XMark Query 1, only the id and name of the person are required. All other information is obsolete. Listing 4.1 shows a person element, where only the required elements are highlighted. Obviously, by selecting only the required elements the input dataset can be reduced. The degree of the reduction depends on the query. For example for XMark Query 10 (see Appendix A), the reduction is expected to be much less than for XMark Query 1, because most sub-elements of the person element are required to correctly answer that query.

Listing 4.1: Person element from an XMark dataset. Only the highlighted parts are required to answer XMark query 1.

```
<person id="person0">
 <name>Huei Demke</name>
       <emailaddress>mailto:Demke@uu.se</emailaddress>
       <address>
              <street>95 Grinter St</street>
              <city>Macon</city>
              <country>United States</country>
              <zipcode>32</zipcode>
       </address>
       <creditcard>8182 1228 4493 3873</creditcard>
       <profile income="55386.86">
              <education>High School</education>
              <gender>male</gender>
              <business>No</business>
              <age>37</age>
       </profile>
</person>
```

The remainder of this chapter describes the improved input projection algorithm. The improved algorithm selects only the required sub-elements by static analysis of an sNRC query to generate a set of more extensive (or more specific) projection expressions than the set of projection expressions that the reference system uses to perform input projection.

4.2 Static sNRC Query Analysis

In this section the algorithm that is designed to analyze an sNRC query to compute a set of projection expressions is presented. The approach that is followed is similar to the approach that [35] used for analysis of XQuery expressions.

Though the sNRC syntax is more compact than the syntax of XQuery, analysis of sNRC expressions provides similar challenges to the analysis of XQuery expressions, each of which will be discussed in more detail.

4.2.1 sNRC Syntax and Normalization

A single query can be described by many different sNRC expressions. A simple example can be provided based on sNRC XMark query 6:

$\dot{\texttt{size}}(\{[b..[item] \mid b \in \texttt{in}.[site].2.[regions]]\})$

Given that the same schema is used, this sNRC query can be rewritten to the following sNRC query:

$\dot{\texttt{size}}(\{[b \mid b \in \texttt{in}.[site].2.[regions]..[item]]\})$

Both sNRC queries will yield the same result when evaluated on the same dataset. By rewriting each sNRC query to a normal form this challenge can be overcome. Where [35] could make use of the normalization mechanisms that are present in the XQuery core, sNRC does not yet have a set of formally defined normalization mechanisms. Normalization is possible for NRC [29, 46], and therefore likely possible for sNRC. However, defining such a set of normalization rules and implementing a normalization engine falls outside the scope of this thesis. Instead, the sNRC queries are analyzed and executed as they are defined in Appendix A. It will be proved that each set of projection expressions derived by the algorithm results in projected dataset that allows correct evaluation of the query.

4.2.2 Variables

The sNRC syntax allows implicit declaration of variables, and binding of variables during bag comprehension operations. An sNRC expression can be bound to a variable in the environment by the static query analysis algorithm. The environment will then bind the set of projection expressions used by the bound sNRC expression to the variable name. Other sNRC expressions that access the variable can retrieve the set of bound projection expressions from the environment to extend these projection expressions if needed. In other words, *the environment allows mappings a variable name to a projection blueprint*.

For simplicity and following the original sNRC definition in [29] it is assumed that each variable has been renamed so that each iterator has a distinct variable name in the environment.

The following notation will be used for variables in the static expression analysis algorithm:

$$Env \vdash Var \Rightarrow E_1$$

To map a variable with name *Var* to an expression E_1 in the environment. The environment will bind the set of projection expressions used by E_1 to the variable named *Var*.

$$Env \vdash Get(Var)$$

To retrieve the set of projection expressions bound to the variable with name *Var* in the environment.

4.2.3 sNRC Expression Composability and Context

From the used sNRC syntax (see Section 3.4) it is clear that sNRC expressions can be composed in many different ways given that the same schema is used. Similar to [35] this can be solved by deriving the projection expressions from the sNRC query following a bottom-up approach.

Each distinct sNRC expression from the extended syntax works in the *context* of 0 or more other sNRC expressions. For example, an expression $E_1 \doteq E_2$ uses the expressions E_1 and E_2. We will refer to this by saying that E_1 and E_2 are the context expressions of $E_1 \doteq E_2$. The input expression **in** is a data source, and therefore has no context expressions. For the sNRC expression $E.2$ that selects that second field(s) works in the *context*, the expression E is its context expression. It is clear from these examples that each type of expression needs to be analysed individually to see how they are composed, and which expressions form their context. These context expressions co-determine the result of the expression.

The role that each context expression plays in the evaluation of a particular expression requires careful analysis as well. By showing a step-by-step example the approach of the improved algorithm is demonstrated, as well as the challenge imposed by sNRC expression composability. Consider for this example the following sNRC query:

$$\{[\{[x \mid y \in \texttt{in}.[auctions].2.[auction], x.2.[id] = y.2.[id]]\} \mid x \in \texttt{in}.[people].2.[person]]\}.2.[name]$$

This query reads the input and then joins two parts of the dataset based on an id and then selects the name sub-element from each result of the join. The input expression has no context expression and requires an empty projection expression corresponding to the top-level nested value or 'root' of the sNRC dataset.

Following a bottom-up approach, the outter flattening bag comprehension can split into two parts $\{[Target_1 \mid Cond_1]\}$ where $Target_1$ and $Cond_1$ are both sNRC expressions that form the context of the flattening bag comprehension. Both *Target* and *Cond* can be composed of (multiple) other sNRC expressions.

The required projection expressions are first computed for the $Cond_1$ context, and next for the $Target_1$ context, because the $Cond_1$ expression determines when the flattening bag comprehension generates output, and it binds the variables used for iterating

over the input.

The $Cond_1$ expression itself is composed of a single expression. This expression yields an obvious result, binding a projection expression $\mathbf{in}.[people].2.[person]$ to variable x in the environment. The $Target_1$ expression is an inner flattening bag comprehension, which can again be split into two parts $Target_2$ (x) and $Cond_2$ ($y \in \mathbf{in}.[auctions].2.[auction], x.2.[id] = y.2.[id]$). Again the first expression of $Cond_2$ is obvious, binding a projection expression $\mathbf{in}.[auctions].2.[auction]$ to the variable y in the environment. The second expression of $Cond_2$ is a comparison, which works in the context of two other expressions: $x.2.[id]$ and $y.2.[id]$.

Both these expressions extend the expression paths of their *context* expressions (variables x and y) to generate the projection expressions $\mathbf{in}.[people].2.[person].2.[id]$ and $\mathbf{in}.[auctions].2.[auction].2.[id]$. The target expression of the inner flattening bag comprehension $Target_2$ simply reuses the projection expressions bound to variable x.

The final set of projection expressions used by the inner flattening bag comprehension (or $Target_1$) merges the sets of projection expressions used by $Target_2$ and $Cond_2$:

$\mathbf{in}.[people].2.[person]$
$\mathbf{in}.[people].2.[person].2.[id]$
$\mathbf{in}.[auctions].2.[auction].2.[id]$

This allows us to determine the final set of projection expressions used by the outer flattening bag comprehension, by combining the sets of projection expressions used by $Target_1$ and $Cond_1$:

$\mathbf{in}.[people].2.[person]$
$\mathbf{in}.[people].2.[person].2.[id]$
$\mathbf{in}.[auctions].2.[auction].2.[id]$

The final step of the query is to perform a selection ($.2.[name]$) on the results of the outer flattening bag comprehension. Following the bottom-up approach this means that the projection expressions of the context expression are extended to get the following set of projection expressions:

$\mathbf{in}.[people].2.[person].2.[name]$
$\mathbf{in}.[people].2.[person].2.[id].2.[name]$
$\mathbf{in}.[auctions].2.[auction].2.[id].2.[name]$

This shows that an approach using a single set of projection expressions is not sufficient. The final selection ($.2.[name]$) was applied to all projection expressions that were used in its context expression (the outer flattening bag comprehension). However, not all of the projection expressions that are used by the flattening bag comprehension are returned as output. In this case, only the projection expression $\mathbf{in}.[people].2.[person]$ describes the *actual output* of the flattening bag comprehension. The other paths, $\mathbf{in}.[people].2.[person].2.[id]$ and $\mathbf{in}.[auctions].2.[auction].2.[id]$ are only *used to determine which values are returned as output*.

The example above illustrates the need to distinguish between projection expres-

sions that describe which input elements can become part of the output, and projection expressions that describe input elements that are used to determine which elements become part of the output. To overcome this, the definition of a projection expression pair is introduced:

Definition 4.1 (Projection Expression Pair)**.** *A projection expression pair* (PE, UPE) *consists of two sets of projection expressions that describe the output of an sNRC expression. PE is a set of* ***output projection expressions*** *describing the input elements that can become part of the output of the expression. UPE is a set of* ***used projection expressions*** *describing the input elements that are used to determine the output of the expression.*

This new definition can now be applied to the example above. Consider the following part of the sNRC query of the example above:

$$\{[\{[x \mid y \in \texttt{in}.[auctions].2.[auction], x.2.[id] = y.2.[id]]\} \mid x \in \texttt{in}.[people].2.[person]]\}$$

The following projection expression pair describes the output of this sNRC query:

$$(\{\texttt{in}.[people].2.[person]\}, \{\texttt{in}.[people].2.[person].2.[id].2.[name], \texttt{in}.[auctions].2.[auction].2.[id].2.[name]\})$$

Observe that the first set in the projection expression pair is the set of output projection expressions of the query. These projection expressions describe the part of the input that is output by the sNRC expression, in this case the 'person' elements. The second set of the projection expressions is the set of used projection expressions. These expressions are indeed only determined *which* 'person' elements are output by the sNRC query, but are not part of the output itself.

The main notation that will be used to describe the static sNRC query analysis is the following:

$$E \Rightarrow V$$

Where E is an sNRC expression, and V is a projection blueprint defined as follows:

Definition 4.2 (Projection Blueprint)**.** *A projection blueprint V is a set of sNRC nested values that describes the structure of the output of expression E. Differently from normal sNRC nested values, in V bags are sets and each basic value is a projection expression ordered pair.*

Several operations on projection blueprints are used in the static query analysis. Union for projection blueprints is the same as union for sNRC bags. V.PE produces the set that contains all output projection expressions contained by the nested structure of V. V.UPE produces the set that contains all used projection expressions contained by the nested structure of V.

4.3 Static sNRC Query Analysis Algorithm

In this section an improved algorithm used for projection is defined. This algorithm uses a static analysis of an sNRC query to compute a set of projection expressions. As

was found in the previous sections, each expression from the extended sNRC syntax needs to be analyzed separately to define how the projection expressions can be derived using a bottom-up approach. We therefore provide for each expression in the sNRC syntax an inference rule [39] that can be applied.

4.3.1 Basic Value Constants

Basic value constants from $\mathcal{B}$, the empty tuple $\langle\rangle$ and the empty bag $\emptyset$ do not output or use any input elements.

$$\frac{}{\mathcal{B} \Rightarrow \emptyset}$$
$$\langle\rangle \Rightarrow \emptyset$$
$$\emptyset \Rightarrow \emptyset$$

There are no statements above the inference rule to indicate that there are no preconditions for these statements.

4.3.2 Sequences

A sequence of expressions occurs for example when multiple expressions are composed in the condition of a flattening bag comprehension, for example:

$$\{[x \mid x \in site.2.auctions, x \neq \langle\rangle]\}$$

Here $x\ in site.2.auctions, x \neq \langle\rangle$ are two separate expressions that are sequenced.

An empty sequence of expressions does not output or use or output any values from the input dataset.

$$\frac{}{() \Rightarrow \emptyset}$$

When the sequenced expressions have a projection blueprint these projection blueprints are merged.

$$\frac{E_1 \Rightarrow V_1 \quad E_2 \Rightarrow V_2}{E_1 \uplus E_2 \Rightarrow V_1 \cup V_2}$$

4.3.3 Unions

Taking the union of two expressions is similar to sequencing two expressions, so again the projection blueprints of the two expressions are merged.

$$\frac{E_1 \Rightarrow V_1 \quad E_2 \Rightarrow V_2}{E_1, E_2 \Rightarrow V_1 \cup V_2}$$

4.3.4 Set

The set expression removes duplicate from the results of an sNRC expression. Because in a projection blueprint all bags are sets this means that the set expression uses the unchanged projection blueprint of its context expression.

$$\frac{E_1 \Rightarrow V_1}{\dot{\mathbf{set}}(E_1) \Rightarrow V_1}$$

4.3.5 Reading input

Reading input provides access to the input dataset for the remainder of an sNRC query and its static analysis. This is represented by the projection expression **in**. No input elements are required to output the input elements.

$$\frac{}{\mathbf{in} \Rightarrow \{(\{\mathbf{in}\}, \emptyset)\}}$$

4.3.6 Comparisons

A comparison expression never results in output elements originating from its context expressions. Instead, a comparison expression always outputs a basic value (a boolean) that represents the result of the comparison. All projection expressions of the context expressions are therefore used projection expressions and not a part of the output projection expressions.

$$\frac{\begin{array}{c} E_1 \Rightarrow V_1 \\ E_2 \Rightarrow V_2 \end{array}}{E_1 \doteq E_2 \Rightarrow \{(\emptyset, V_1.PE \cup V_1.UPE \cup V_2.PE \cup V_2.UPE)\}}$$

This rule holds not just for '$\doteq$', but for all types of comparisons between two sNRC expressions. Note also that the structure of the output in the projection blueprints of the context expressions is lost. This does not affect the correctness though, because the comparison produces no output projection expressions and output structure can therefore be disregarded.

4.3.7 Size

The size expression counts the number of elements contained by the bag that is the output of its context expression. Similar to a comparison, the size expression outputs basic values that do not relate to any input elements. Therefore, all projection expressions contained by the context expression projection blueprint are used projection expressions.

$$\frac{E_1 \Rightarrow V_1}{\dot{\mathbf{size}}(E_1) \Rightarrow \{(\emptyset, V_1.PE \cup V_1.UPE)\}}$$

Similar to the comparison expression, the structure of the output of the context expressions is lost. Again this does not affect the correctness, because the size expression produces no output projection expressions and structure can therefore be disregarded.

4.3.8 Variables

As described above when a variable is bound to an sNRC expression E_1, the environment binds the projection blueprint V_1 that is used by E_1 to the variable named *Var* in the environment.

$$\frac{\begin{array}{c} Env \vdash Var \Rightarrow V_1 \\ E_1 \Rightarrow V_1 \end{array}}{Env \vdash Get(Var) \Rightarrow V_1}$$

4.3.9 Element Of and Iteration

The element of or iteration sNRC expression is crucial for bag comprehension. This expression binds the iteration variable *Var* to the projection blueprint V_1 that is used by the sNRC expression E_1 which produces the output over which is iterated.

$$\frac{\begin{array}{c} Env \vdash Var \Rightarrow V_1 \\ E_1 \Rightarrow V_1 \end{array}}{Env \vdash Get(Var) \Rightarrow V_1}$$

Iterating over expression E_1 with iteration variable Var means that the output projection expressions of E_1 are not extended further, unless they are accessed through the iteration variable *Var*.

4.3.10 Bag Comprehension

As discussed in Section 4.2.3 a flattening bag comprehension is considered to be composed of two elements: $\{[Target \mid Cond]\}$. The *Cond* expression binds the iteration variables and determines which elements are to be passed on to the evaluation of *Target*. The *Target* expression then determines the output of the bag comprehension. Note that both *Cond* and *Target* can be sequences of expressions to which the 'Sequence' rule can be applied recursively. The *Cond* expression always contains at least one 'Element Of' expression to bind an iteration variable to the expression over which is iterated.

$$\frac{\begin{array}{c} Target \Rightarrow V_1 \\ Cond \Rightarrow V_2 \end{array}}{\{[Target \mid Cond]\} \Rightarrow V_1 \cup \{(\emptyset, V_2.PE \cup V_2.UPE)\}}$$

Note that for the *Target* the projection blueprint is used as-is, preserving the output structure. For the *Cond* an new projection expression pair where all projection expressions from the *Cond* projection blueprint are added as used projection expressions. Note that this causes the structure of the *Cond* expression to be lost. Because the *Cond* expression produces no output elements, the structure can be ignored. If the iteration variable is used in the *Target* expression, its projection blueprint will be accessed through the variable.

4.3.11 Ordered Pair

The 'Ordered Pair' expression modifies the structure of the projection blueprint, by introducing an extra level of nesting. Because the projection blueprint uses the sNRC data model, this nesting can also be introduced in the projection blueprint to correctly maintain the structure of the output.

$$\frac{E_1 \Rightarrow V_1 \quad E_2 \Rightarrow V_2}{\langle E_1, E_2 \rangle \Rightarrow \{\langle V_1, V_2 \rangle\}}$$

4.3.12 Selection Expressions

The set of selection expressions consists of the sNRC expressions that are used to select sub-elements of nested values $\mathcal{V}$. This set of expressions is the same set that is used to describe projection expressions:

$$E ::= E.1 \mid E.2 \mid E.[key] \mid E..[key] \mid E..1 \mid E..2$$

What makes these steps important is that they extend the projection expressions used in the projection blueprints. More specifically they retrieve the projection expressions from their context expressions and append to these to generate a new set of projection expressions.

$$\frac{E_1 \Rightarrow V_1}{E_1.[key] \Rightarrow \{[x.[key] \mid x \in V_1]\}}$$

$$\frac{E_1 \Rightarrow V_1}{E_1..[key] \Rightarrow \{[x..[key] \mid x \in V_1]\}}$$

$$\frac{E_1 \Rightarrow V_1}{E_1.1 \Rightarrow \{[x.1 \mid x \in V_1]\}}$$

$$\frac{E_1 \Rightarrow V_1}{E_1.2 \Rightarrow \{[x.2 \mid x \in V_1]\}}$$

The selection expressions are mapped over the elements of the projection blueprint. For projection expression pairs, the selection expression is applied to the first field (output projection expressions). Sets of projection expressions, each projection expression is extended with the selection expression. For ordered pairs the 6th normalization rule of Wong [46] is applied. A selection of the first or second field on an ordered pair expression will select the projection blueprint stored in the first field or second field respectively.

4.3.13 User-Defined Functions

In this thesis no static sNRC query analysis for user-defined functions is defined. Because of the endless variety of operations that can be applied in a user-defined function this is a topic of research by itself. If user-defined functions are used in a sNRC query it is assumed that a correct projection blueprint is provided.

4.3.14 Finishing the analysis

When the entire sNRC query has been analyzed using the bottom-up approach and the above inference rules a single projection blueprint V is obtained. The set of output projection expressions can be extracted by applying the V.PE operation. Similarly, the used projection expressions can be extracted by applying the V.UPE operation. These sets can be merged to obtain a final set of projection expressions.

4.4 Correctness

One of the main goals of Projection Expressions as described in subsection 3.5.1 is to ensure that the evaluation of a query on a projected sNRC dataset yields the same results as on the original dataset. It will be shown that the algorithm based on the inference rules of section 4.3 verifies the correctness theorem (Theorem 1 in the previous chapter).

To prove this by using induction on the inference rules for each expression defined in section 4.3 the following lemmas are needed:

Lemma 1 (Output Projection Expressions)**.** *Given the projection blueprint V of an sNRC query E. Let D be the result of a projection of the output projection expressions (OPE's) V.PE on an sNRC dataset D. It holds that D' contains all the elements that are present in the result of the evaluation of E on D.*

Proof (Output Projection Expressions)**.** *This lemma is validated by using induction on the inference rule for each expression in section 4.3:*

- ***Basic Value Constants:*** *The OPE's of these expressions is the empty set. Basic Values do not return any elements from the input and therefore the set of returned input elements is correctly contained by the empty set.*

- ***Sequences:*** *The OPE's of a sequence are the union of the OPE's of its context expressions. If it is assumed by induction that the context expressions produce a correct set of OPE's then the inference rule for sequences will also produce a set of OPE's that contains all elements output by the sequence expression. An empty sequence returns no elements and therefore the set of returned input elements is correctly contained by the empty set.*

- ***Unions:*** *Similar to sequences, the OPE's of a union are the union of the OPE's of its context expressions. If it is assumed by induction that the context expressions return a correct set of OPE's then the inference rule for unions will produce a set of OPE's that contains all input elements output by the union expression.*

- ***Set:*** *The set rule uses the OPE's of its context expression. The set of elements output by a set expression is a subset of the bag of elements output by its context expression. Assuming by induction that the OPE's of the context expression are correct, the set of OPE's generated the set rule will correctly contain all elements output by the set expression.*

- ***Reading input:*** *Reading the input returns the top level element of the sNRC dataset. This is the same element that is accessed by projecting the input expression* ***in*** *to the dataset.*
- ***Comparisons:*** *Comparisons return a basic value that do not relate to elements from the input dataset. Therefore, no input elements are output by these expressions and that the set of output elements is indeed contained by the empty set.*
- ***Size:*** *The same reasoning as for comparisons holds here.*
- ***Bag Comprehension:*** *The output of the bag comprehension expression is the output of its Target context expression. The bag comprehension rule includes the total projection blueprint of the Target context expression in the projection blueprint the bag comprehension. If it is assumed by induction that the OPE's contained by the projection blueprint of the Target context expression are correct, then the set of OPE's of the bag comprehension correctly contains all output elements.*
- ***Ordered Pair:*** *The projection blueprint of an ordered pair expression contains an ordered pair that contains the exact two projection blueprints of the two context expressions. Now assuming by induction that the OPE's of both context expressions were correct, the set of OPE's of the projection blueprint of the ordered pair expression will correctly contain the output elements for both context expressions.*
- ***Navigation Expressions:*** *The navigation expression rule by definition extends all OPE's of its context expression with the navigation expression itself by mapping over all elements of the projection blueprint. It is therefore trivial that the output elements of the navigation expressions are also contained by the extended OPE's.*

Lemma 2 (Variables)**.** *The output projection expressions (OPE's) that are bound to a variable in the environment allows reaching all elements that the variable iterates over.*

Proof (Variables)**.** *A set of OPE's can only be bound to a variable in the environment by two expressions: implicit binding of a variable to an expression and iteration over an expression through the element of expression. In both cases the variable is instantiated with the projection blueprint V that is required by the expression E that it is associated with. As shown in the proof of Lemma 1 the set of OPE's $V.PE$ contains all the input elements that are output by E.*

Lemma 3 (Completeness)**.** *The projection blueprint V that is determined for an sNRC expression E using the static sNRC query analysis algorithm from section 4.3 contains all projection expressions that are determined for its context expressions.*

Proof (Completeness)**.** *When observing the inference rules of section 4.3 this is obvious. Every inference rule propagates either the entire projection blueprint, or the sets of projection expressions of its context expressions into V. This means that all*

projection expressions are propagated, and therefore that V contains all projection expressions of E's context expressions.

Proof (Correctness)**.** *For Theorem 1 to be valid, D' must contain all elements from D that are needed to evaluate the expression E. With lemmas 1 and 2 it is shown using the projection expressions of a context expression that the elements required to evaluate that context expression are contained by D'. To validate that the final projection is correct, the final set of projection expressions must contain all projection expressions that are required by the expressions context expressions. This is shown in Lemma 3.*

4.5 Projection Trees

Previously in this chapter we have provided the definition of a rule-based static sNRC query analysis algorithm that produces a set of projection expressions that allows computing a correct projection on the input dataset. The reference system applies each projection expression separately to the input dataset. This approach works, because the projection expressions used by the reference system describe the the input elements, which are loaded including all sub-elements. The projection expressions produced by the static sNRC query analysis program are more complex, also describing which sub-elements of the input elements should be loaded. This introduces a problem that we will illustrate based on the set of projection expressions that was determined for XMark Query 1 by the static analysis:

Listing 4.2: Projection Expressions for sNRC XMark Query 1

```
in.[site].2.[people].2.[person].2.[name].2
in.[site].2.[people].2.[person].2.[@id].2
```

The projection expression determined for the reference system was **in**.[*site*].2.[*people*].2.[*person*] which loaded each person element and all its sub-elements when performing the input projection.

For the set of projection expressions determined for the improved system, it is not clear what the entry point of the input should be (in this case the 'person' element). Taking inspiration from the work on a Tree Algebra for XML and Holistic Twig Joins [32, 15] we can instead combine the projection expressions required by an sNRC query into a tree. As will be discussed in the remainder of this chapter, such a tree structure allows retrieving additional information about the structure of the input. Additionally all projection expressions can be applied in a single pass over the input dataset at the expense of a more complex loading algorithm.

4.5.1 Definition

As discussed in the previous section, projection expressions need to be combined into a projection tree to allow derivation of additional information to perform the input projection. A formal definition of a projection tree is now presented.

Definition 4.3 (Projection Tree)**.** *A Projection Tree is a node- and edge-labeled tree (V,E) such that:*

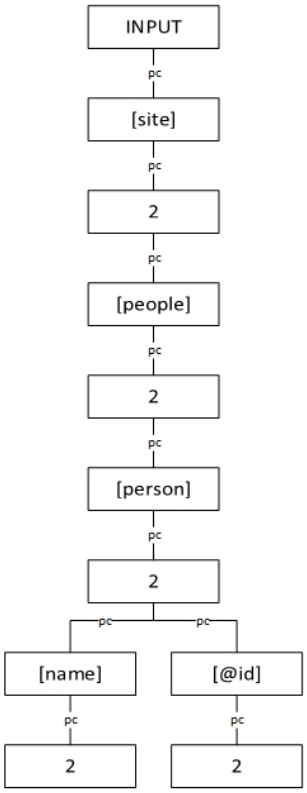

Figure 4.1: Projection Tree for XMark Query 1

- *Each node in V represents a selection expression: either selecting the input, the first field of an element, the second field of an element, or an element based on a key value.*
- *Each edge in E is labeled* pc *for direct sub-elements or* ad *for all sub-elements.*

In Figure 4.1 projection tree equivalent is shown for the set of projection expressions that are determined for sNRC XMark Query 1.

4.5.2 Annotated Projection Expressions

In this section we describe an extension of the static sNRC query analysis algorithm to annotate the derived projection expressions. One annotation is required to find the input elements. We define an *input element* as follows:

Definition 4.4 (Input Element)**.** *Given a projection tree (V,E), an input element is a node in V that will become a top-level element in the projected dataset.*

In the example of sNRC XMark Query 1 and the projection tree in Figure 4.1, the person element is an input element. This means that when a person element is located in the input dataset, the loading algorithm should from that point on start parsing the input data. The first annotation that is introduced in this chapter is used to automatically determine the input elements in a projection tree.

A second annotation is is used as an optimization for counting operations. When a projection expression describes input that is only read to be counted, the content of this input is irrelevant and need not be loaded.

The first annotation is applied in the 'Element of and Iteration' rule of the static sNRC query analysis algorithm:

$$\frac{\begin{array}{c} Env \vdash Var \Rightarrow V_1 \\ E_1 \Rightarrow V_1 \end{array}}{\begin{array}{c} V_1' \vdash \{[x.annotateVar(Var) \mid x \in V_1]\} \\ Env \vdash Get(Var) \Rightarrow V_1' \end{array}}$$

When binding a projection blueprint to a variable in the environment, each output projection path in the projection blueprint is annotated with the variable name. This information is propagated when the projection blueprint is accessed through the variable. Using this information it can be determined which variables are extended by which projection expressions.

The second annotation is applied in the 'Size' rule of the static sNRC query analysis algorithm:

$$\frac{E_1 \Rightarrow V_1}{\dot{\mathbf{size}}(E_1) \Rightarrow \{(\emptyset, \{[x.annotateSize \mid x \in V_1.PE]\} \cup V_1.UPE)\}}$$

For a size expression, the output projection expressions of its context expression (in this case $V_1.PE$) describe the input for the size expression. All these output projection expressions are therefore annotated with a size annotation ('#') to indicate that these elements are counted.

Note that both of these modified rules do not change the projection blueprints that are determined by the original rules. The projection expressions contained by the projection blueprints are only annotated with additional information. The correctness is therefore unaffected.

4.5.3 Annotated Projection Tree

When converting a set of annotated projection expressions to a projection tree, the annotations from the projection expressions are propagated into the projection tree. The 'size' annotation is used by the loading algorithm to determine whether the subelements of an element need to be parsed. This is described in the next section. The iteration variable annotation allows determining the input elements in the projection tree.

When two or more projection expressions extend a projection expression based on the same iteration variable, the projection expression stored in this iteration variable is an input element. This is well-illustrated by sNRC XMark Query 1:

$$\{[b.2.[name].2 \mid b \in \mathbf{in}.[site].2.[people].2.[person], b.2.[@id].2 \doteq \{\{person0\}\}]\}$$

This shows that it is not possible to select a more deeply nested input element than: **in**.[*site*].2.[*people*].2.[*person*], because the 'name' and '@id' sub-elements that are selected must belong to the same 'person' element bound to the iteration variable. When only one projection expression extends a projection expression that is stored in an iteration variable, no such interdependency exists and the final projection expression can be used as input element. An example of that is sNRC XMark Query 6 which can be found in Appendix A.

4.6 Improved Loading Algorithm

In this section we will describe the algorithm that is used to perform the projection on the input dataset based on the annotated projection tree. Where the reference system algorithm used XML parsing events, the improved loading algorithm uses a more generic approach. The improved algorithm introduces a layer of abstraction by processing sNRC parsing events. This allows the loading algorithm to work on any type of nested data for which an adapter is available, as will be discussed in more detail in the next section.

Similar to the reference loading algorithm, the improved loading algorithm maintains a projection tree that represents the location with respect to the input that we will refer to as the *input path*. The reference algorithm consisted of two steps: first finding the input element, and then parsing the entire input element and all its sub-elements. The improved algorithm also follows a two step approach. First, the *findNext*, *tupleFind* and *secondFieldFind* functions are used to find the input elements. The second step is not as straightforward as for the reference system. The improved algorithm uses the *parseTuple*, *parseBag* and *parseNextBag* functions to parse the input element and only the sub-elements that are described by the projection tree.

Notice how all functions make use of a *selectAction* function. This function matches the input path to a matching node in the projection tree, by following the input path through the projection tree from the root. Then, based on the outcome it outputs the following action:

- If the input path leads to a dead end: skip the current element (*SKIP*)
- If the input path leads to a leaf node that is not annotated with a count annotation: parse the current element and all sub-elements (*PARSE_ALL*)
- If the input path leads to a leaf node that is annotated with a count annotation: parse the current element without sub-elements (*PARSE_EMPTY*)
- If the input path leads to a node that is marked as input node: indicate that an input node is found and start parsing from this element (*INPUT*)
- If none of the above holds: move to process the next element (*MOVE*)

The following pseudocode fragments describe the functions that are used by the improved algorithm as described above.

Algorithm 4.6.1: FINDNEXT()

global $Reader, InputPath, ProjTree$
while Reader has next
do $\begin{cases} next \leftarrow \text{next Reader event} \\ \textbf{switch } next \begin{cases} \textbf{case } START\_TUPLE \textbf{ return } (tupleFind()) \\ \textbf{case } END\_TUPLE \text{ remove last element from InputPath} \\ \textbf{case } END\_DOC \textbf{ return } (null) \end{cases} \end{cases}$
return $(null)$

Algorithm 4.6.2: TUPLEFIND()

global $InputPath, ProjTreekey \leftarrow$ reader.parseFirst()
$InputPath \leftarrow InputPath + SelectKeyNode(key)$
$action \leftarrow selectAction(InputPath, ProjTree)$
switch $action$ $\begin{cases} \textbf{case } SKIP \text{ reader.skip()} \\ \textbf{case } PARSE\_ALL \\ \textbf{case } PARSE\_EMPTY \\ \textbf{case } INPUT \textbf{ return } (parseTuple(key)) \\ \textbf{case } MOVE \textbf{ return } (secondFieldFind()) \end{cases}$

Algorithm 4.6.3: SECONDFIELDFIND()

global $InputPath, ProjTreeInputPath \leftarrow InputPath + SelectSecondNode$
$action \leftarrow selectAction(InputPath, ProjTree)$
switch $action$ $\begin{cases} \textbf{case } SKIP \text{ reader skip tuple} \\ \textbf{case } PARSE\_ALL \\ \textbf{case } PARSE\_EMPTY \\ \textbf{case } INPUT \textbf{ return } (parseBag()) \end{cases}$

Algorithm 4.6.4: PARSETUPLE(*key*)

global $Reader, InputPath, ProjTree$
$action \leftarrow selectAction(InputPath, ProjTree)$
switch $action$ $\begin{cases} \textbf{case } PARSE\_ALL \begin{cases} \text{remove last element from InputPath} \\ \textbf{return } (reader.parseCompleteTuple()) \end{cases} \\ \textbf{case } PARSE\_EMPTY \begin{cases} \text{remove last element from InputPath} \\ \textbf{return } (orderedpairwithkeyandemptybag) \end{cases} \\ \textbf{case } INPUT \\ \textbf{case } MOVE \begin{cases} InputPath \leftarrow InputPath + SelectSecondNode \\ \textbf{return } (parseBag()) \end{cases} \end{cases}$

Algorithm 4.6.5: PARSEBAG()

```
global Reader, InputPath, ProjTree
action ← selectAction(InputPath, ProjTree)
switch action {
    case PARSE_ALL {
        remove last element from InputPath
        return (reader.parseSecond())
    case PARSE_EMPTY {
        remove last element from InputPath
        return (emptybag)
    case INPUT
    case MOVE return (parseNextBag())
```

Algorithm 4.6.6: PARSENEXTBAG()

```
global Reader, InputPath, ProjTree
result ← new Bagwhile Reader has next
do {
    next ← next Reader event
    switch next {
        case START_TUPLE
        {
            action ← selectAction(InputPath, ProjTree)
            switch action {
                case SKIP reader.skip()
                case PARSE_ALL
                case PARSE_EMPTY
                case MOVE
                case INPUT result ← result + parseTuple(key)
        case END_TUPLE remove last element from InputPath
        case END_DOC return (null)
return (result)
```

As expected, the evaluation of all sNRC XMark queries yields identical results for projections computed by both the reference system loading algorithm as well as the improved loading algorithm. In the evaluation chapter, Chapter 5 it is shown that the improved loading algorithm also results in a successful fragmentation of the input dataset, allowing processing by Flink. Moreover experiments are conducted to show the improvement of this algorithm over the reference approach, and the improvement compared to the algorithm of Marian and Simeon [35].

4.7 Improved Parallel sNRC - Flink Architecture

In order to support the static sNRC query analysis algorithm and the improved loading algorithm several extensions implemented for the sNRC - Flink architecture with respect to the reference system architecture. An UML diagram showing the structure of the most important clases can be found in Appendix D.

Firstly, all sNRC expression definitions and projection blueprint classes were added. Each sNRC XMark query implementation now has a corresponding sNRC expression. From the sNRC expression the projection blueprint can be determined using the rules defined in this chapter. Secondly the projection tree classes are added, as well as a parser to convert a set of projection expressions into a projection tree. The most important change is the addition of the improved loading algorithm that applies projection defined by the projection tree to the input dataset. Moreover, the improved parser

works with an abstract nested data parser. This allows easy integration with other nested data types such as JSON by adapting the respective parser to the interface.

Chapter 5

Evaluation

In this Chapter we will discuss how we have evaluated the parallel sNRC implementations. The reference system from Chapter 3 with the reference input projection algorithm and the system with the improved input projection algorithm from Chapter 4 will be evaluated and compared. The benchmark that is used for the evaluation, XMark, is discussed in more detail first. Next the single node setup, experiments and outcomes are shown and discussed. This is followed by the setup, experiments and outcomes of the experiments on a SURFSara cluster of up to 16 nodes, and datasets of up to 141GB.

5.1 XMark Benchmark

The benchmark for semi-structured data that is used to evaluate the parallel sNRC systems introduced in this thesis is the widely used XMark benchmark [41]. XMark offers a set of 20 varying queries that are modelled after typical real-world scenarios. Each of the queries is designed to pose a particular challenge to an XML query processor. These challenges vary from simple lookups to lookups of elements that are deeply nested and from complex joins to aggregations.

The queries are run on a single dataset that contains information about auctions, including information about the auctioned items, buyers, sellers, and categories. A tree overview of the nested data structure of the XMark dataset is shown in Figure 5.1. XMark allows generating datasets of arbitrary sizes, and generation of split datasets. Each split will contain a complete and valid structure from the root element of the dataset, to where the previous split ended.

5.1.1 sNRC - XMark Benchmark Queries

The set of XMark queries that is used to test the system developed for this thesis does not contain all 20 original XMark queries. The sNRC data model [29] also described in Section 2.5.1 relies on the use of bags and tuples. The elements of a bag are by definition not ordered. Though the data model allows modelling of an ordering of the elements in a bag (by wrapping the elements in tuples) this is not a natural approach for the sNRC data model.

XMark queries 2,3 and 4 as well as XMark queries 18 and 19 are not included in the benchmark suite for this work, because their challenges to the query processor

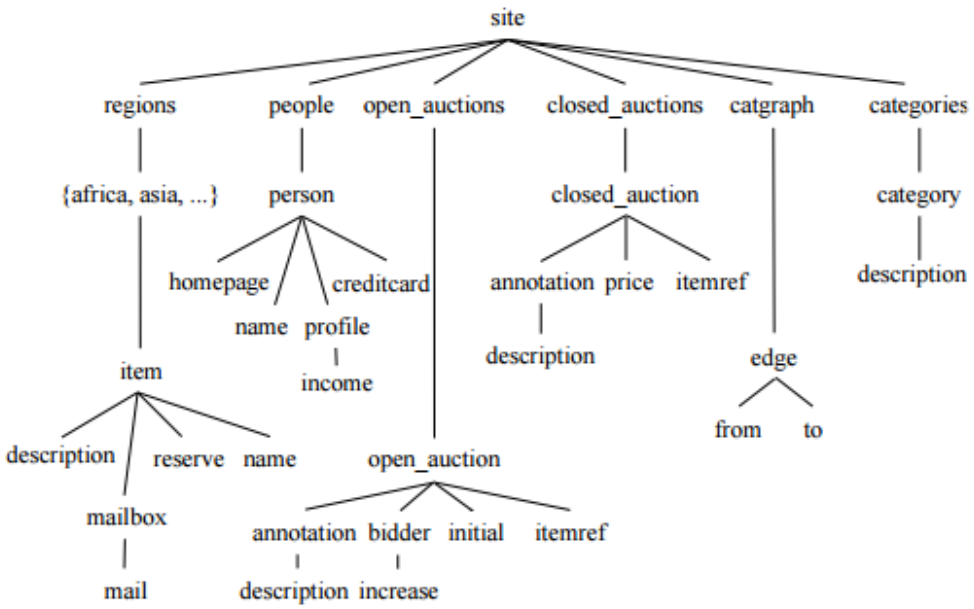

Figure 5.1: The nested data structure of the XMark benchmark dataset. From [41].

regard ordered access or sorting operations [41]. Though it would be interesting to see how the presented parallel sNRC systems perform when dealing with such challenges, it falls outside of the scope of this thesis because sNRC was not designed with ordering in mind. The remaining 15 queries are included as described in the original XMark publication. The full set of queries, and the used sNRC equivalents can be found in Appendix A.

Table 5.1 shows a list of all XMark queries that are used for the sNRC XMark benchmark. In this table you can find in the second column which parts of the input dataset are used for the query (see Figure 5.1). The third column shows the number of input elements and the complexity of their projection expressions. A *simple* path contains only direct sub-element relationships, while a *complex* path also contains non-direct sub-element relationships (see Section 3.2). The fourth column shows the number and types of Flink operators that are used in the Flink execution plan for that query. Each query uses the input and output operators, so these are omitted. A graphical representation of the evaluation plans for each query can be found in Appendix B. The fifth and final column shows the number of partitioning and broadcast steps used in the Flink execution plan. These steps serialize the data and then exchange it between Task Managers over the network. *Forward* operators are for forwarding data from one operator to the next within a same node of the cluster. Because these *Forward* operators do not apply serialization or exchange data over the network they are omitted from the table.

5.1.2 sNRC - XMark Benchmark Datasets

For the sNRC XMark local single node and SURFSara cluster experiments the same set of 8 datasets is used, ranging from 1 GB to 141 GB in size. These datasets are

Table 5.1: Overview of XMark queries used in the sNRC XMark Benchmark Suite and their complexity.

Query	Subsets	Input Elements	# Used Flink Operators	# Hash Partitionings
XMark 1	Persons	Simple (1)	Filter (1) FlatMap (1)	
XMark 5	Closed Auctions	Simple (1)		
XMark 6	Items (Full Dataset)	Complex (1)		
XMark 7	Prose (Full Dataset)	Complex (3)		
XMark 8	Persons Closed Auctions	Simple (2)	Filter (2) Map (2) CoGroup (1)	Hash Partition (2)
XMark 9a	Items (Europe) Persons Closed Auctions	Simple (3)	Filter (3) Map (5) CoGroup (2) Group Reduce (1)	Hash Partition (5)
XMark 9b	Items (Europe) Persons Closed Auctions	Simple (2) Complex (1)	Filter (3) Map (5) CoGroup (2) Group Reduce (1)	Hash Partition (5)
XMark 10	Persons	Simple (1)	Filter (1) FlatMap (1) Map (1) Group Reduce (1)	Hash Partition (1)
XMark 11	Persons Initial Bids (Open Auctions)	Simple (2)	Filter (2) Map (1) FlatMap (1)	Broadcast (1)
XMark 12	Persons Initial Bids (Open Auctions)	Simple (2)	Filter (3) Map (1) FlatMap (1)	Broadcast (1)
XMark 13	Items (Australia)	Simple (1)	Map (1)	
XMark 14	Items (Full Dataset)	Complex (1)	Filter (1) FlatMap (1)	
XMark 15	Closed Auctions	Simple (1)		
XMark 16	Closed Auctions	Simple (1)	Filter (1) FlatMap (1)	
XMark 17	Persons	Simple (1)	Filter (1) FlatMap (1)	
XMark 20	Persons	Simple (1)	Map (1) Group Reduce (1)	Hash Partition (1)

Table 5.2: Datasets used for the sNRC - XMark Benchmark

Name	Scaling factor	Elements per file	Size	# Files
Split 1GB	10	20000	1.1GB	35
Split 2GB	20	40000	2.2GB	35
Split 4GB	40	40000	4.4GB	69
Split 8GB	80	80000	8.8GB	69
Split 18GB	160	80000	18GB	138
Split 36GB	320	160000	36GB	138
Split 71GB	640	160000	71GB	275
Split 141GB	1280	320000	141GB	275

generated using the XMark dataset generation tool. The datasets and parameters that were used for generation are shown in Table 5.2.

5.2 Setups

In this section the configurations of the hardware and software that were used for running the experiments are described. First the local single node setup, and second the SURFSara cluster setup is discussed.

5.2.1 Local Single Node Setup

In this section we discuss the setup for the local single node experiments. Each of the XMark queries is executed 5 times and the runtimes are measured in milliseconds. For each query the fastest and slowest runtime are discarded, and the remaining 3 runtimes are averaged to obtain a more reliable estimation of the runtime performance. The local experiments are run on the datasets of 1GB, 2GB, 4GB and 8GB.

The local experiments were run on a single desktop PC. This desktop featured a 2.93Ghz 4 core CPU and 8 GB of RAM. The datasets are stored on a SSD drive. Flink snapshots are written to a regular harddrive.

For software the Java JDK 1.8.0_71 was used in conjuction with Apache Flink 0.10.1. The benchmarks were submitted to a local Flink cluster configured with a Flink parallelism parameter set to 4, 4096MB max memory for the Task Manager JVM and 512MB max memory for the Job Manager JVM.

5.2.2 SURFSara Cluster Setup

We now discuss the setup for our cluster experiments. These experiments use SURFSara's YARN cluster [1]. The nodes in this cluster feature a 2.6 Ghz 8 core CPU and 64 GB of RAM. The benchmarks are submitted to a Flink cluster running on top of YARN. The Flink Task Manager JVMs are configured to use a maximum of 16GB of memory, the Flink Job Manager JVM was configured to use at most 4GB of memory. The datasets are stored on the SURFSara YARN cluster Hadoop Filesystem (HDFS). The HDFS is configured to replicate files 3 times over the filesystem.

[1] `https://userinfo.surfsara.nl/systems/hadoop/description`

Table 5.3: SURFSara Cluster Configurations

# Worker Nodes	Datasets	Parallellism
1	1GB, 2GB, 4GB, 8GB	8
2	1GB, 2GB, 4GB, 8GB, 18GB	16
4	1GB, 2GB, 4GB, 8GB, 18GB, 36GB	32
8	1GB, 2GB, 4GB, 8GB, 18GB, 36GB, 71GB	64
16	1GB, 2GB, 4GB, 8GB, 18GB, 36GB, 71GB, 141GB	128

Table 5.4: Cluster Query Subset

Query	Input Type	Partitionings
XMark 1	Simple	No
XMark 7	Complex	No
XMark 8	Simple	Yes (2)
XMark 9a	Simple	Yes (5)
XMark 9b	Complex	Yes (5)
XMark 15	Simple	No

Table 5.3 shows the number of workers, used datasets and Flink parallelism parameter used for the different SURFSara cluster experiments.

Where for the local experiments the total set of sNRC XMark queries was used, we can not use the full set of queries for the cluster experiments due to time limitations. Instead a representative subset was chosen based on information listed in Table 5.1. From the queries in the table 4 categories can be distinguished:

- Simple input paths and no partitionings
- Simple input with partitionings
- Complex input without partitionings
- Complex input with partitionings

For each category one or multiple representative subqueries are selected. The selected queries are shown in Table 5.4.

XMark queries 1 and 15 are queries with simple projection expressions and without partitioning steps. XMark query 7 is a query with multiple complex projection expressions, but without partitioning steps. XMark queries 8 and 9a have input elements with simple projection expressions, and a varying amount of partitioning steps. Finally XMark query 9b is an adapted version of the original XMark query 9a with one input element modified to require a complex projection expression.

In addition to running the query benchmarks, a separate set of experiments is performed to determine the time required to read the input dataset. For these queries only the input dataset is loaded using the input projection algorithms. The actual queries are not evaluated on the input dataset.

5.3 Local Results

This section describes the results that were obtained when comparing the reference system to the improved system in a set of experiments on a single node. First we will compare the sizes of the input dataset after applying projection between the reference system, the improved system and the algorithm of Marian & Siméon [35]. In the second part the runtimes of the reference system and the improved system are compared for the full set of queries and datasets of 1GB, 2GB, 4GB and 8GB.

5.3.1 Projection Sizes

In this first experiment the effectiveness of the sNRC projection algorithm presented in Chapter 4 is assessed by comparing it to the reference algorithm and the work by Marian & Siméon [35]. The goal as defined at the beginning of Chapter 4 is to reduce the size of the input to: (1) speed up serialization of the nested values, (2) increase throughput for partitioning and broadcasting between workers nodes and (3) reduce memory footprint of the input data, resulting in less time spent on garbage collection by the JVM.

The relative reduction of the size of the input after applying the projection is measured by comparing the total size of the serialized data after projection to the total size of the serialized complete dataset. This experiment is performed for each query from the sNRC XMark Benchmark suite on the Split 1GB dataset.

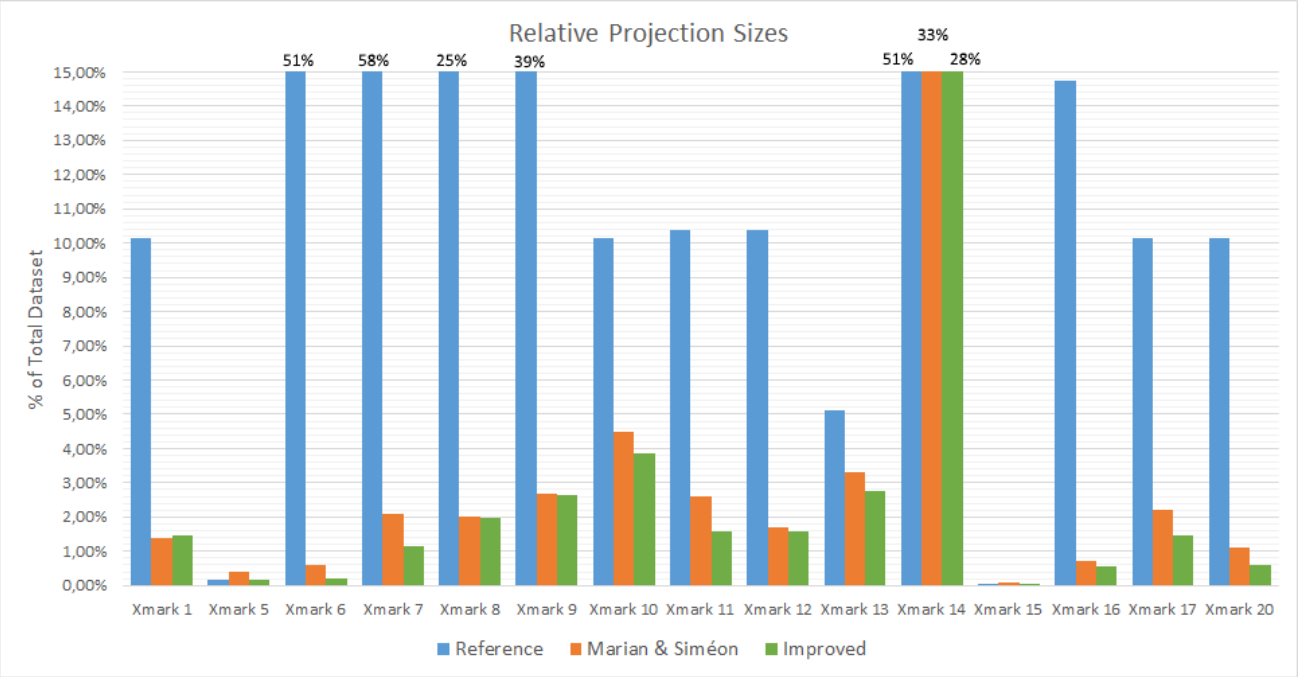

Figure 5.2: Relative projection sizes compared to the complete dataset.

Figure 5.2 shows the relative sizes of the input per query after projection for the reference system, Marian & Siméon and the improved system respectively. As ex-

pected, the improved projection performs much better than the reference system. We see this more clearly in Figure 5.3. An exception is XMark query 5, where the reference system uses the same projection expression as the improved system. This results in a projected dataset of the same size.

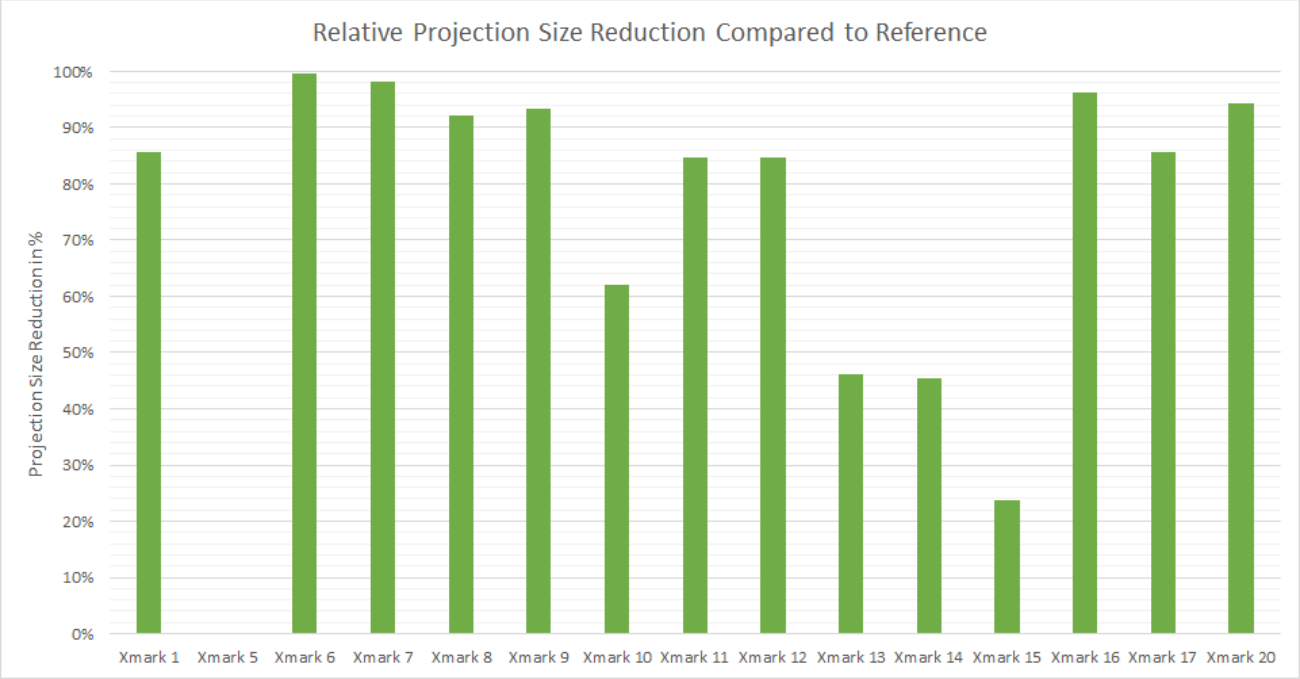

Figure 5.3: Relative projection size reduction of the snRC projection compared to the reference system.

From Figure 5.2 we see that the sNRC projection performs as good or better than Marian & Siméon on all XMark queries from the sNRC XMark benchmark suite: the system further reduces the size of the input dataset by 32% on average for all queries, and up to 92% for individual queries. In Figure 5.4 we can see the relative differences more clearly. The observed improvement can be explained by the fact that the input projection of the improved system does not include the nested structure from the root to the input elements. Most improvement is achieved where this nesting is deep (as can be seen for XMark query 15) or when the size of the nesting structure is relatively large compared to the input (for example XMark queries 5, 6, 7 and 11 when input elements are empty because the output contains a count operation).

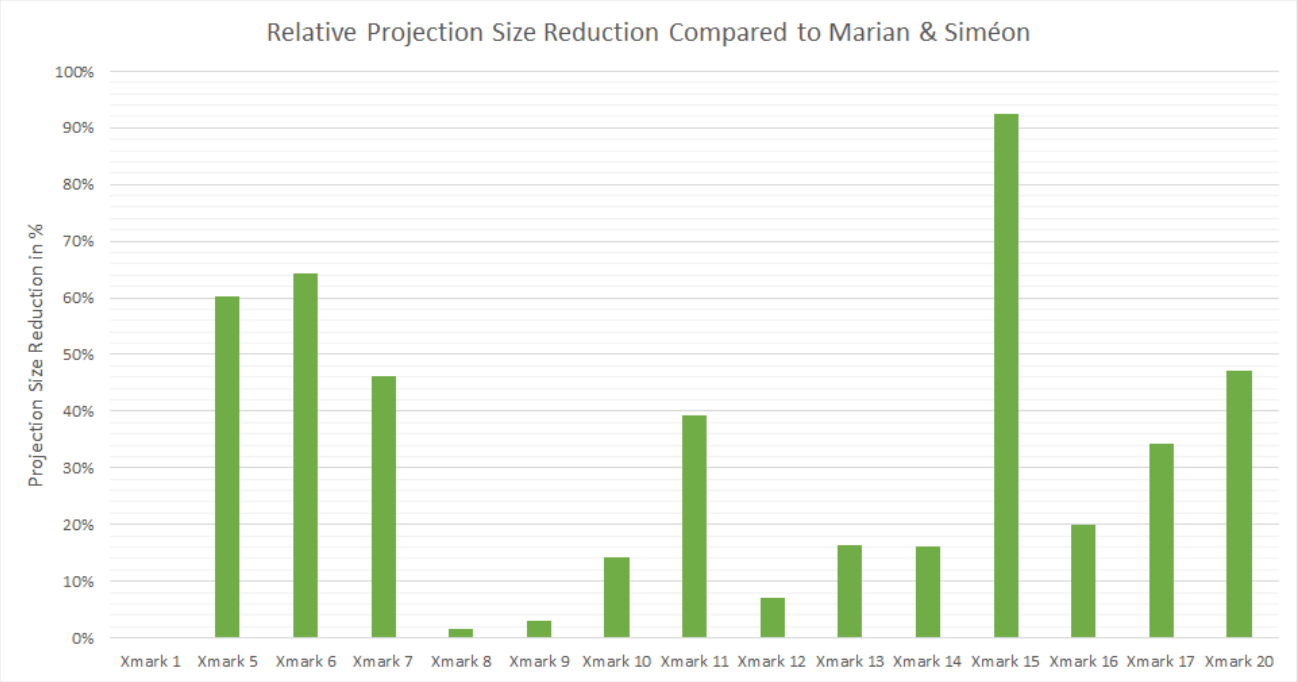

Figure 5.4: Relative projection size reduction of the snRC projection compared to Marian & Siméon.

5.3.2 Runtimes

In the previous section we have investigated how much the improved system reduces the size of the input dataset compared to the reference system. We expect the reduced size of the input to have three effects that reduce the time required to evaluate an sNRC XMark query on the parallel sNRC system. Tables E.1 and E.2 in Section E.1 of Appendix E show the complete list of runtime results for the local single node experiments of the reference system and improved system respectively.

From the runtime results a few observations can be made, especially when the characteristics of each query described in Table 5.1 are taken into account. Figure 5.5 provides insights by showing the relative difference in runtimes between the improved system and the reference system for the local single node experiments.

It can be seen that the performance for queries that have input elements with simple projection expression and no partitionings (queries 1, 5, 13, 15, 16, 17) have very similar performance between the reference and improved version for all dataset sizes. The runtimes of the reference algorithm are slightly better, but the difference is reduced as the dataset size increases.

For the queries that have input elements with complex projection expressions, the improved system shows greater runtimes than the reference system in this experiment. This is apparent in Figure 5.5 for queries 7 and 14. On query 9a the improved system shows smaller runtimes, but the equivalent with a complex projection expression shows a drop in relative runtime reduction. An exception is query 6, which has a complex

Figure 5.5: Relative improvement of the runtime for each for the sNRC XMark queries of the improved system compared to the reference system and for each of input the dataset sizes.

projection expression for its input element but performs better on the improved system regardless. The results for query 6 can be explained by two observations. Firstly, the improved system does not parse the sub-elements of the input elements, because the result of the query is a count operation which has been optimized in the improved system. Secondly, the complex projection expression for the input elements of query 6 is not as complex as the projection expressions used for queries 7 and 14. Queries 7 and 14 select from all sub-elements starting at the root of the dataset, while the projection expression for query 6 first has a simple component, allowing the loading algorithm to skip a large part of the input before processing all sub-elements.

The queries that have one or more partitionings (queries 8, 9a, 9b, 10, 11, 12 and 20) all perform better on the improved system, with an exception of query 10. This can be explained by the fact that the reduction of the input is not as significant as for the other queries with partitionings (see Figure 5.3) and the fact that the challenge of query 10 is formatting the output [41], for which no optimizations are introduced in the improved system. The Flink web interface timeline for query 10 shows that indeed more than half of the total time is spent on writing the output.

5.4 SURFSara Cluster Results

In this section the results of the experiments on a SURFSara cluster of up to 16 nodes and datasets of up to 141GB are shown. In Section E.2 of Appendix E the full result tables are shown. In this section plots are shown to investigate the differences in runtimes between the reference and improved system, as well as to investigate the scalability of the two systems.

5.4.1 Runtimes

In this section we observe the runtimes for the cluster experiments of both the reference system and improved system. It is discussed how the performance differs for the different types of queries identified in Table 5.4 earlier in this chapter, when the number of nodes changes, and when the size of the dataset changes.

Effect of Dataset Size on Runtimes

First the effect of the size of the dataset on the runtimes of both systems is investigated. Figures 5.6, 5.7, 5.8 and 5.9 show the runtimes of all queries for the cluster experiments for both systems for datasets of 141GB, 36GB, 4GB and 1GB.

Taking a closer look at these plots as well as the results shown in Section E.2 of Appendix E it can be seen that the improved system consistently has better runtimes for the queries with partitioning steps (queries 8, 9a, 9b). This gap in performance increases as the size of the dataset increases. At a dataset size of 141GB the reference system has a runtime of at most 3,34 larger than the runtime of the improved system. As the size of the dataset approaches 1GB, this gap approaches zero for all queries

with a partitioning step.

When we investigate query 9b which has partitioning steps as well as a complex projection expression for one of its input elements, we see that the runtimes are very similar to the runtimes of query 9a for both systems when the dataset is large. As the size of the input dataset decreases the runtime for the improved system approaches the runtime of the reference system faster than for queries without a complex input. This indicates that the impact of the complex input on the performance of the improved system becomes relatively small when the size of the input dataset increases. This same effect is observed for query 7 which is discussed below.

Queries without partitioning steps and with simple projection expressions for the input elements perform very similarly for both the reference system and the improved system. This observation complies with the results of the single node experiments discussed in the previous section. The results of the read-only experiments in Section E.2 show that indeed the times for reading the data are very similar between the two systems and as expected reducing the input dataset size does not have much effect on this type of queries.

Finally, the query without partitioning steps but with a complex projection expression for its input elements (query 7) is considered. In Figure 5.6 it can be seen that the performance is similar for the reference and improved system for a large dataset, slightly favouring the reference system. However, when the input dataset size decreases this gap increases as well as can be seen in Figures 5.7, 5.8 and 5.9. As stated above, this indicates that the impact of the complex input on the runtimes becomes smaller as the size of the input dataset increases.

Effect of Cluster Size on Runtimes

In the remainder of the section the effect of the number of nodes in the cluster on the runtimes of the queries for both the reference system and improved system is investigated. To do this we look at the results of both systems for the 8GB dataset, because this dataset was used for all cluster size configurations.

The runtimes and relative difference in runtimes of the reference system and improved system are as you would expect for an 8GB dataset based on our previous findings: Performances for both systems are similar for queries 1 and 15, which have no partitioning steps and simple projection expressions for the input elements. Query 7, which has no partitioning steps and complex projection expressions for the input elements, is evaluated between 1.84 and and 1.68 times faster on the reference system. Queries 8, 9 and 9a, which have partitioning steps, show between 1.31 and 3.11 times better runtimes on the improved system. For query 9a we see again that the effect of the complex projection expression supresses the performance gain from the improved input projection, but the improved system still shows runtimes that are between 1.31 and 2.21 times better.

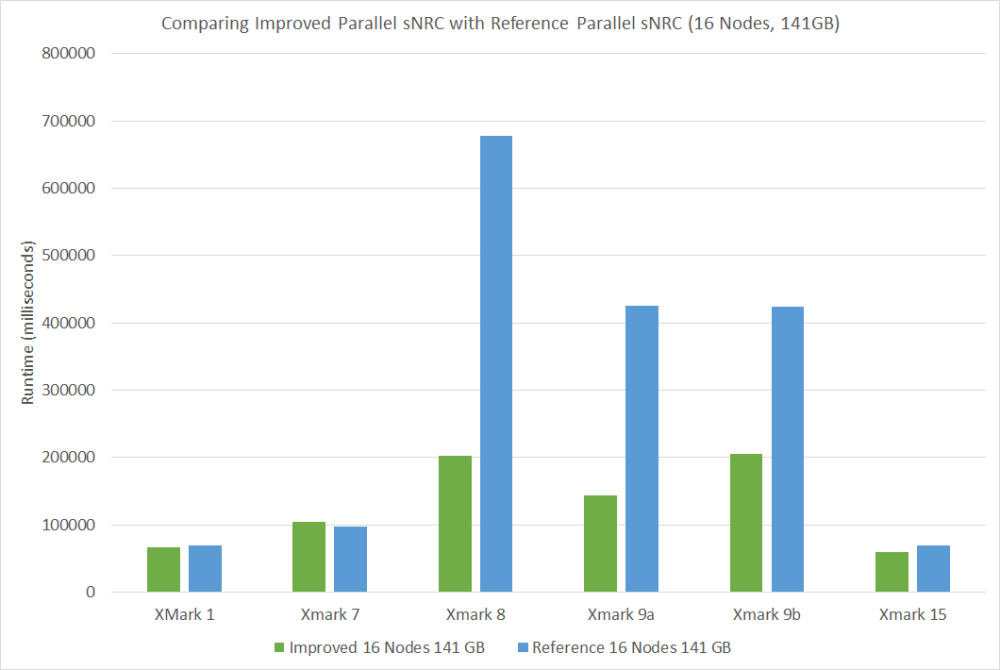

Figure 5.6: Runtimes of the reference system compared to the runtimes of the improved system on a cluster of 16 nodes and a dataset of 141GB.

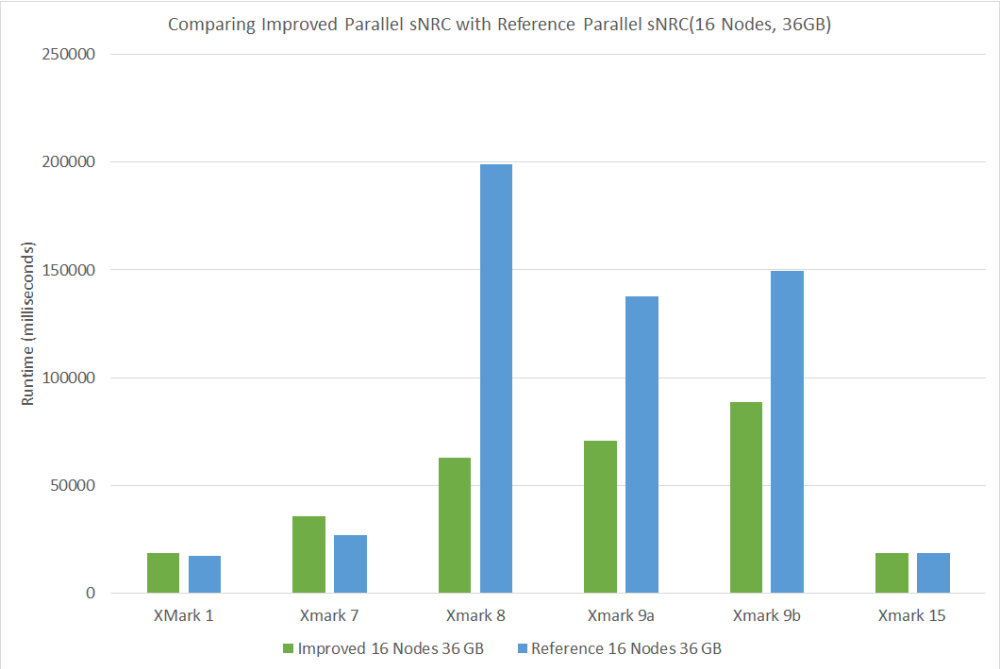

Figure 5.7: Runtimes of the reference system compared to the runtimes of the improved system on a cluster of 16 nodes and a dataset of 36GB.

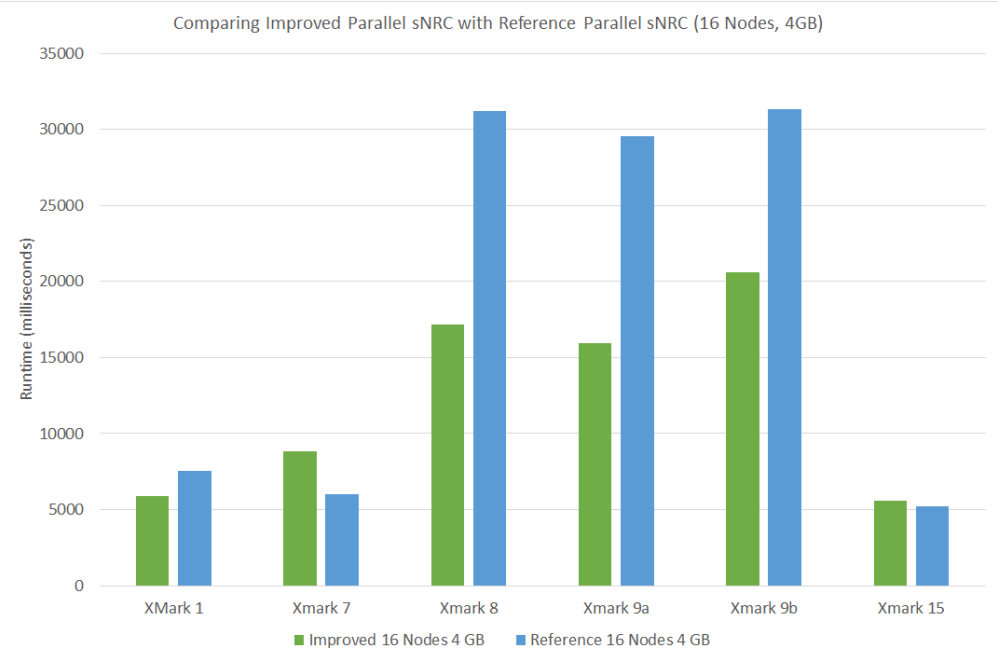

Figure 5.8: Runtimes of the reference system compared to the runtimes of the improved system on a cluster of 16 nodes and a dataset of 4GB.

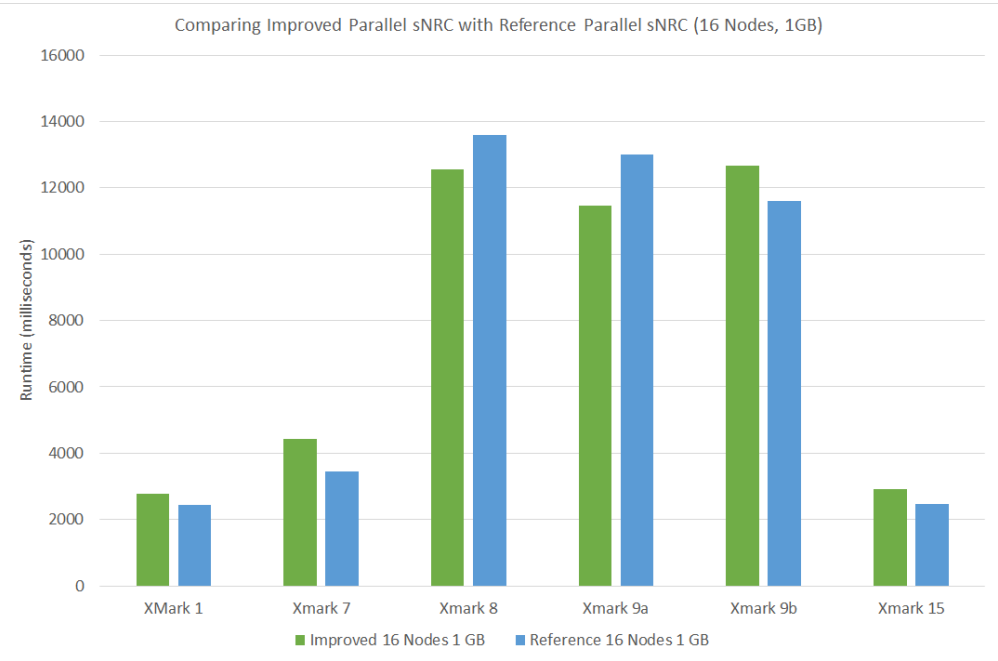

Figure 5.9: Runtimes of the reference system compared to the runtimes of the improved system on a cluster of 16 nodes and a dataset of 1GB.

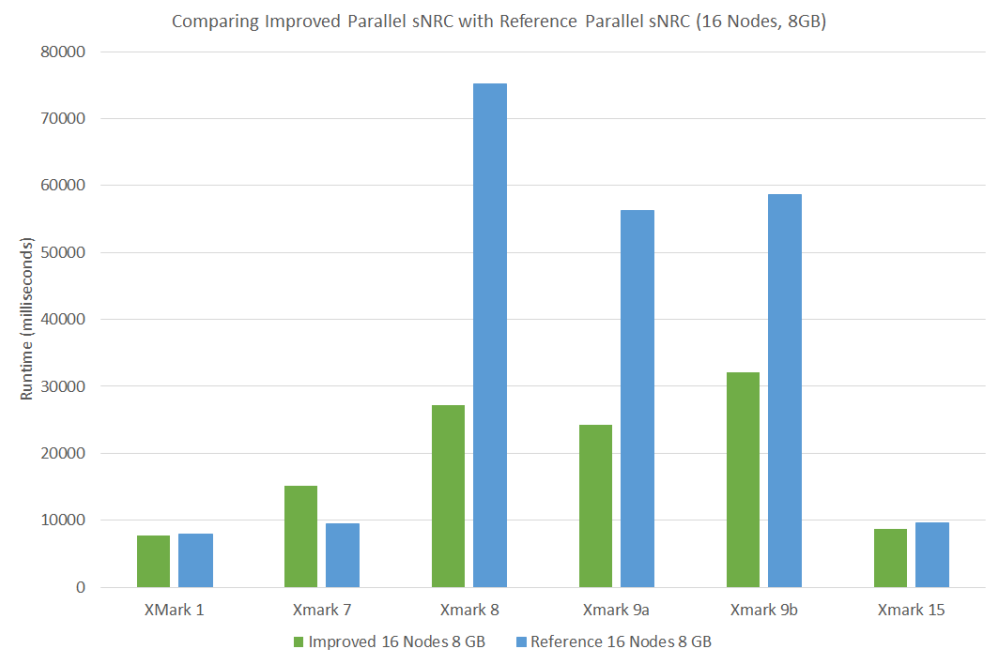

Figure 5.10: Runtimes of the reference system compared to the runtimes of the improved system on a cluster of 16 nodes and a dataset of 8GB.

These observations hold for all number of nodes. As we for example see in Figures 5.10 and 5.11 the runtimes stay relatively the same. An exception is the single node experiment on the cluster (see Figure 5.12, where we observe the same effect as reducing the dataset size: The performance for queries 1 and 15 remains the same, but reference system now performs much better on query 7 and the gap between the reference system and the improved system is reduced for queries 8, 9a and 9b. This effect is easily explained. In the case of a single node the partitioning steps need not serialize the data and transmit it over the network to other nodes. This means that two of the three expected results of the optimizations for the improved system stated at the beginning of this chapter lose their effect.

5.4.2 Scalability

In this section it is evaluated how well the reference system and improved system scale. To do this the number of nodes and size of the dataset are doubled each time, scaling from 1 node and an 8GB dataset to 16 nodes and a 141GB dataset. Figure 5.13 and 5.14 show the results for the reference system and improved system respectively.

The results indicate that both the reference system as well as the improved system queries 1, 7 and 15 scale linearly, with minor overhead. As we know from Table 5.1 these queries have operations that do not require partitioning, but solely rely on reading the input and applying map and filter operations. As was also also observed by Camacho-Rodríguez et al. [17], these operations are very well parallelizable by Flink.

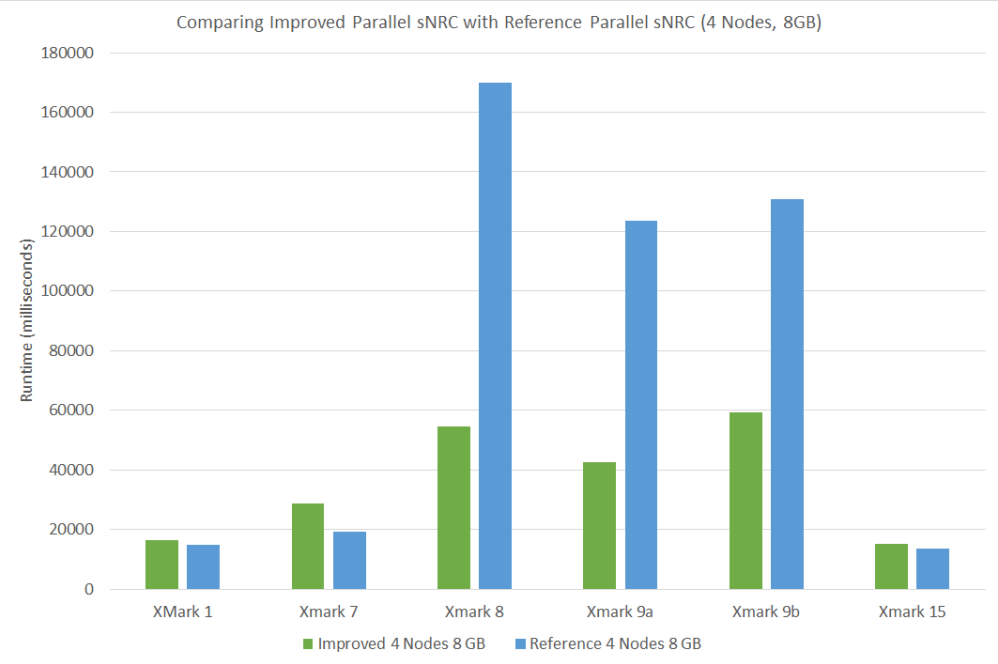

Figure 5.11: Runtimes of the reference system compared to the runtimes of the improved system on a cluster of 4 nodes and a dataset of 8GB.

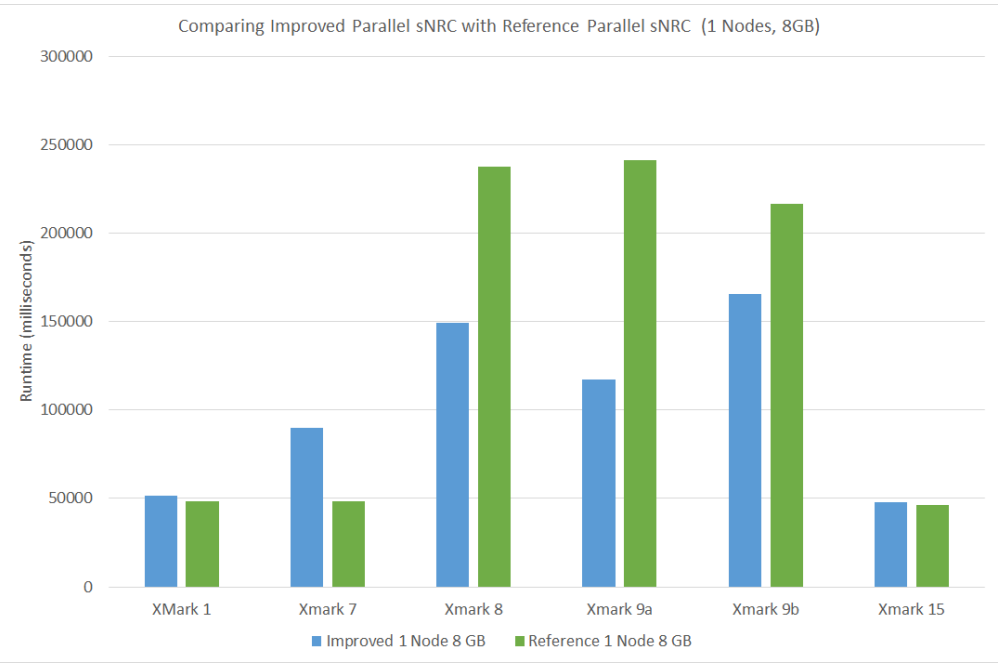

Figure 5.12: Runtimes of the reference system compared to the runtimes of the improved system on a cluster of 1 node and a dataset of 8GB.

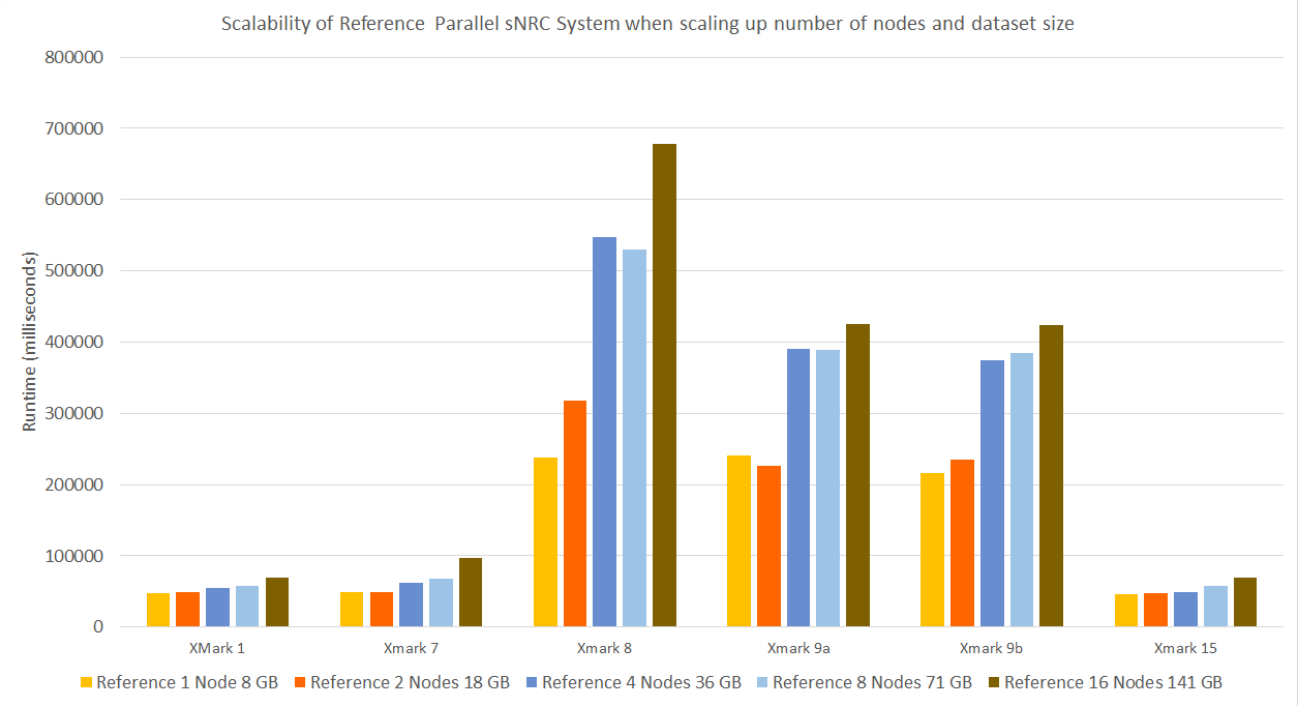

Figure 5.13: Runtimes for the reference system on a cluster when scaling up from a single node and an 8 GB dataset to 16 nodes and a 141 GB dataset.

Queries 8, 9a and 9b do require operations that use partitioning and these queries we see different results. Both systems suffer from more overhead when scaling up these queries than for queries that do not use partitioning. However, the improved system scales much better for these queries than the reference system. In the case of query 8, the reference system exhibits up 2.85 times greater runtimes when scaling up. In the case of the improved system this is at most 1.36 times greater.

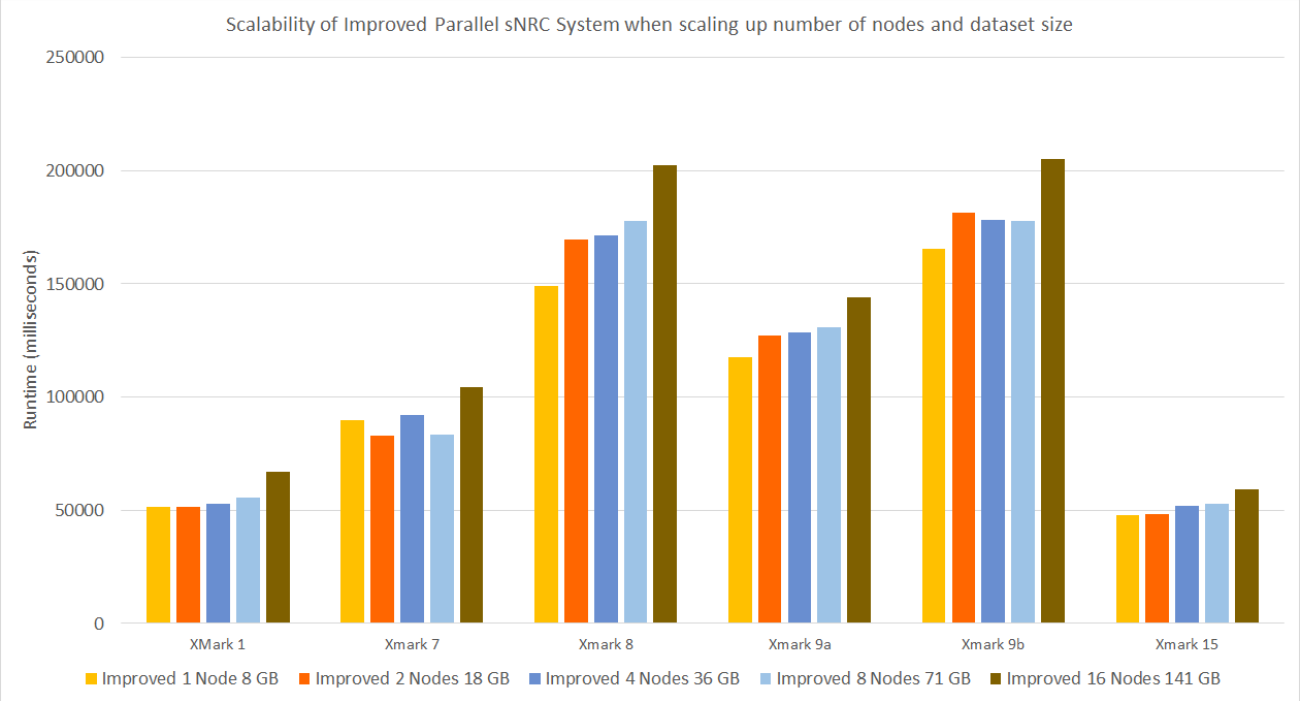

Figure 5.14: Runtimes for the improved system on a cluster when scaling up from a single node and an 8 GB dataset to 16 nodes and a 141 GB dataset.

Chapter 6

Conclusions and Future Work

In this chapter this thesis is concluded by summarizing and discussing the results, and reflecting on the work done to provide suggestions for future work. First our most important observations from the literature and experiments are summarized as answers to the the research questions that were stated in the introduction of this thesis. The process and work presented in this thesis are then discussed to identify strengths and points for improvement to finally present opportunities for future work.

6.1 Research Objectives

RQ1: *Which state-of-the-art parallel data processing platform is suitable for implicit parallelization of the Nested Relational Calculus for Semi-Structured Data?*

In Chapter 2 existing popular data processing platforms such as Hadoop, Spark and Flink were discussed, as well as query languages for nested data that are built on top of these platforms. In Chapter 3 these platforms are matched to the goal of querying nested datasets that are larger than main memory with sNRC in parallel. Flink and Spark are two modern data processing platforms that naturally support complex operations such as joins, which are required to allow maping sNRC queries to the data processing platform. Though Spark is a more mature platform, Flink is used for this thesis due to its truly streaming engine. This results in lower latencies by pipelining data between different operators. Flink features automatically managed memory, execution plan optimizations and efficient mechanisms for spilling data to disk when necessary. Moreover, its programming model allows it to easily be extended to support the sNRC data model.

RQ2: *How can an input semi-structured dataset be fragmentized to allow processing by a streaming data processing platform?*

In Chapter 2 a technique called *Input Projection* is discussed. Input projection is a popular approach for fragmentizing and trimming an input nested dataset before processing a specific query. These approaches were not designed with parallel systems in mind, but they are determined to be good fit to the problem in this thesis. In this work a set of *Projection Expressions* defines which parts of the input are needed to

evaluate a specific sNRC query. We refer to these parts as the input elements. The projection expressions are used to compute a projected dataset, which is a dataset that is smaller than the original dataset but still allows correct evaluation of an sNRC query. A loading algorithm is presented to compute the projected dataset based on the set of projection expressions. This answer, combined with the platform chosen at RQ1 resulted in a **Reference Parallel sNRC System**.

RQ3: *How can a reference input projection approach be improved upon to further reduce the size of the input semi-structured dataset?*

In Chapter 4 an **Improved Parallel sNRC System** is presented that uses static sNRC query analysis to determine a set of more detailed projection expressions. These projection expressions also describe which sub-elements of the input should be loaded. The static query analysis consists of a set of inference rules for bottom-up processing of an sNRC query to determine the set of projection expressions. A proof based on induction is provided to show that the set of determined projection expressions results in a valid projected dataset. By combining annotating the projection expressions and combining them into a projection tree several additional small optimizations are done. Finally, a more sophisticated loading algorithm is presented that uses the projection tree to compute the projected dataset.

RQ4: *What is the effect of the improved input projection on the performance in terms of runtime and scalability of the improved parallel sNRC system compared to the reference system?*

In Chapter 5 both the reference system and improved system are evaluated using the XMark benchmark. The improved input projection algorithm is shown to reduce the size of the input dataset by 73% on average and by up to 99.6% compared to the reference system and by 32% on average and up to 92% compared to the approach of Marian and Siméon [35].

The set of queries from the XMark benchmark is evaluated on a single node, as well as a cluster of up to 16 nodes and dataset sizes of up to 141 GB. On a single node both systems perform similarly for queries with simple input expressions and without operations that require partitioning. For queries with complex input expressions and without operations that require partitioning, the reference system performs better due to the more complex loading algorithm of the improved system. For queries that do require partitioning, the improved system consistently performs much better.

The results from the cluster experiments shows similar results to the single node experiment. For queries with simple input expressions and without operations that require partitioning both systems perform similarly. For queries with complex input expressions that do not require partitioning the reference system performs better. However, as the dataset size increases this gap is closed. For queries that have operations that do require partitioning the improved system performs much better. As the dataset size increases, the size of this gap increases.

Finally the scalability of the reference and improved system is investigated. It is found that both systems seem to scale linearly with minor overhead for queries that do not require partitioning. For queries that do require partitioning, the improved system

shows to scale much better than the reference system, showing a increase in runtime by a factor of 1,36 compared to 2,85 for the reference system.

6.2 Discussion and Reflection

In this section the process and results presented in this thesis are reflected on and discussed.

Firstly, sNRC is a generic query language for nested data. In this thesis the system is only evaluated using the XMark benchmark and a corresponding XML dataset. Despite this the parallel sNRC system is capable of supporting other nested data formats such as JSON. Supporting a wider range of data formats would also open the door to evaluate the system using a wider range of benchmarks or real-world datasets. The XMark dataset is a very popular benchmark with a well-designed set of queries, however using a wider variety of benchmarks would allow getting a better understanding of the performance of the system.

Secondly, the system presented in this thesis should be compared to other implicitly parallel systems for querying nested data. There is no previous work on sNRC itself to which could be compared. Other work has presents a paralellization of XQuery on Flink that is also evaluated on the XMark dataset. Even though their system focusses specifically on XML, while the system presented in this thesis is generic, a comparison would be valuable. Their work uses a modified set of queries, and their source code is unavailable. This made it it infeasible to perform a comparison within this thesis. Other systems such as Apache Pig or SparkSQL are very mature and utilize a large set of optimization techniques. Such a comparison would be unfair at this level of maturity of the parallel sNRC system. In order to get the system to a next level of maturity suggestions for future work are provided in the next section.

Finally, the number of repetitions for the experiments should be increased. Though I believe that given the time and resources available the decisions made in the experimental setup were the right ones, increasing the number of repetitions will further increase the reliability of the results. It is well known that shared clusters are not consistent in terms of performance. Our chosen approach was to filter the best and worst result, and to average the remaining results. This has increased the reliability, but some fluctuations in the results are still apparent. Increasing the number of repetitions would suppress these fluctuations further.

6.3 Future Work

In this thesis an implicitly parallel sNRC system built using Flink as a data processing platform is presented. It is valuable to have a first system to explore the possibilities of parallelized sNRC and to facilitate future work. In this section I will present the directions for future work that I believe will be most interesting and valuable.

Firstly, sophisticated input projection algorithm as optimization for the parallel sNRC system. It is shown that this stand-alone optimization is crucial to the system and yields good results in terms of scalability and runtimes for more complex queries.

Moreover, this input projection algorithm is shown to produce correct projections regardless of how an sNRC query is formulated. This opens doors for **developing a system of normalization and rewriting rules for sNRC**. Such a set of rules can then for example be used to investigate the possibilities of a cost-based rewriting engine for sNRC. it will be valuable to gain insights into the effects of different optimizations. Referring back this work, it would be interesting to define a set of rules that determine when apply the improved input projection algorithm and when to use a simple approach.

Secondly, as the parallel sNRC system gains more maturity in terms of supported data formats and optimizations it will be valuable to **compare the parallel sNRC system to existing systems for querying nested data**. In order to do this, a standard benchmark for comparing parallel nested data querying systems should be chosen or developed. This will provide more insights into the strengths and weaknesses of sNRC compared to existing systems. Additionally, it will be helpful to **determine how and in which situations the formalisms that underlie sNRC can be used to an advantage**.

Bibliography

[1] Apache flink vs apache spark as platforms for large-scale machine learning? http://stackoverflow.com/questions/29780747/apache-flink-vs-apache-spark-as-platforms-for-large-scale-machine-learning, April 2015. Accessed 17-02-2016.

[2] Spark vs flink low memory available. http://stackoverflow.com/questions/31935299/spark-vs-flink-low-memory-available, August 2015. Accessed 17-02-2016.

[3] What is the differences between apache spark and apache flink? http://stackoverflow.com/questions/28082581/what-is-the-differences-between-apache-spark-and-apache-flink, January 2015. Accessed 17-02-2016.

[4] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[5] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[6] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[7] Slim Baltagi. Flink vs spark. http://www.slideshare.net/sbaltagi/flink-vs-spark, October 2015. Slides for Flink Fordward 2015 conference. Accessed 17-02-2016.

[8] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/pacts: a programming model and execution framework

for web-scale analytical processing. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 119–130. ACM, 2010.

[9] Anders Berglund, Scott Boag, Don Chamberlin, Mary F Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. Xml path language (xpath). *World Wide Web Consortium (W3C)*, 2003.

[10] Kevin S Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh, Carl-Christian Kanne, Fatma Ozcan, and Eugene J Shekita. Jaql: A scripting language for large scale semistructured data analysis. In *Proceedings of VLDB Conference*, 2011.

[11] Scott Boag, Don Chamberlin, Mary F Fernández, Daniela Florescu, Jonathan Robie, Jérôme Siméon, and Mugur Stefanescu. Xquery 1.0: An xml query language, 2002.

[12] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, 16, 1998.

[13] Val Breazu-Tannen, Peter Buneman, and Limsoon Wong. Naturally embedded query languages. Technical report, University of Pennsylvania, 1992.

[14] Stéphane Bressan, Barbara Catania, Zoé Lacroix, Ying Guang Li, and Anna Maddalena. Accelerating queries by pruning xml documents. *Data & Knowledge Engineering*, 54(2):211–240, 2005.

[15] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal xml pattern matching. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 310–321. ACM, 2002.

[16] Peter Buneman, Mary Fernandez, and Dan Suciu. Unql: a query language and algebra for semistructured data based on structural recursion. *The VLDB Journal - The International Journal on Very Large Data Bases*, 9(1):76–110, 2000.

[17] Jesús Camacho-Rodríguez, Dario Colazzo, and Ioana Manolescu. Paxquery: Efficient parallel processing of complex xquery. *Knowledge and Data Engineering, IEEE Transactions on*, 27(7):1977–1991, 2015.

[18] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An xml query language for heterogeneous data sources. In *The World Wide Web and Databases*, pages 1–25. Springer, 2000.

[19] Songting Chen, Hua-Gang Li, Junichi 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*, pages 283–294. VLDB Endowment, 2006.

[20] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[21] Latha S Colby. A recursive algebra for nested relations. *Information Systems*, 15(5):567–582, 1990.

[22] Mick De Lange. Designing dawn: a data analytics workflow notation. Master's thesis, Delft Universitory of Technology, 2015.

[23] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[24] Stephan Ewen. Off-heap memory in apache flink and the curious jit compiler. `https://flink.apache.org/news/2015/09/16/off-heap-memory.html`, September 2015. Accessed: 15-02-2016.

[25] Leonidas Fegaras, Chengkai Li, Upa Gupta, and Jijo Philip. Xml query optimization in map-reduce. In *WebDB*. Citeseer, 2011.

[26] Mary Fernández, Daniela Florescu, Alon Levy, and Dan Suciu. Declarative specification of web sites with strudel. *The VLDB Journal*, 9(1):38–55, 2000.

[27] John Gantz and David Reinsel. Extracting value from chaos. 2011.

[28] Marek Grabowski, Jan Hidders, and Jacek Sroka. Representing mapreduce optimisations in the nested relational calculus. In *Big Data*, pages 175–188. Springer, 2013.

[29] Jan Hidders. sNRC: NRC as a data analytics workflow notation. Appendix C of this report, 2014.

[30] Jan Hidders, Natalia Kwasnikowska, Jacek Sroka, Jerzy Tyszkiewicz, and Jan Van den Bussche. Dfl: A dataflow language based on petri nets and nested relational calculus. *Information Systems*, 33(3):261–284, 2008.

[31] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[32] Hosagrahar Visvesvaraya Jagadish, Laks VS Lakshmanan, Divesh Srivastava, and Keith Thompson. Tax: A tree algebra for xml. In *Database Programming Languages*, pages 149–164. Springer, 2001.

[33] Leonid Libkin and Limsoon Wong. Query languages for bags and aggregate functions. *Journal of Computer and System sciences*, 55(2):241–272, 1997.

[34] Ioana Manolescu, Yannis Papakonstantinou, and Vasilis Vassalos. Xml tuple algebra. In *Encyclopedia of Database Systems*, pages 3640–3646. Springer, 2009.

[35] Amélie Marian and Jérôme Siméon. Projecting xml documents. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 213–224. VLDB Endowment, 2003.

[36] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[37] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pages 251–260. IEEE, 1995.

[38] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 165–178. ACM, 2009.

[39] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[40] Mark A Roth, Herry F Korth, and Abraham Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems (TODS)*, 13(4):389–417, 1988.

[41] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J Carey, Ioana Manolescu, and Ralph Busse. Xmark: A benchmark for xml data management. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 974–985. VLDB Endowment, 2002.

[42] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[43] Stan J. Thomas and Patrick C. Fischer. Nested relational structures. *Advances in Computing Research*, 3:269–307, 1986.

[44] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[45] Jan Van den Bussche, Dirk Van Gucht, and Stijn Vansummeren. Well-definedness and semantic type-checking for the nested relational calculus. *Theoretical computer science*, 371(3):183–199, 2007.

[46] Limsoon Wong. Normal forms and conservative properties for query languages over collection types. In *Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 26–36. ACM, 1993.

[47] Limsoon Wong. *Querying nested collections*. PhD thesis, University of Pennsylvania, 1994.

[48] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, volume 8, pages 1–14, 2008.

[49] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

[50] Roberto V. Zicari and Volker Markl. On apache flink. interview with volker markl. `http://www.odbms.org/blog/2015/06/on-apache-flink-interview-with-volker-markl/`, June 2015. Accessed: 15-02-2016.

Appendix A

Used sNRC equivalents of XMark benchmark queries

In this appendix the used sNRC query equivalents for the XMark queries are shown. These sNRC queries are evaluated on the XMark dataset converted to the sNRC data model. Moreover we present the projection expressions used by the reference system, and the projection expressions used by the improved system. The 'size' annotation is represented by a '#' and the input elements are annotated with a '*'.

XMark Query 1

Return the name of the person with ID 'person0".

XQuery

```
let $auction := doc("auction.xml") return
for $b in $auction/site/people/person[@id = "person0"]
       return $b/name/text()
```

sNRC

$\{[b.2.[name].2 \mid b \in \mathbf{in}.[site].2.[people].2.[person], b.2.[@id].2 \doteq \{\{person0\}\}]\}$

Reference Projection Expressions

```
<IN>.[site].2.[people].2.[person]
```

Improved Projection Expressions

```
<IN>.[site].2.[people].2.[person]*.2.[name].2
<IN>.[site].2.[people].2.[person]*.2.[@id].2
```

XMark Query 5

How many sold items cost more than 40?

XQuery

```
let $auction := doc("auction.xml") return
count(
  for $i in $auction/site/closed_auctions/closed_auction
  where $i/price/text() >= 40
  return $i/price
)
```

sNRC

$$\dot{\texttt{size}}(\{[i \mid i \in \texttt{in}.[site].2.[closed\_auctions].2.[closed\_auction], i.2.[price].2 \geq \{\{40\}\}]\})$$

Reference Projection Expressions

```
<IN>.[site].2.[closed_auctions].2.[{closed_auction}]
```

Improved Projection Expressions

```
<IN>.[site].2.[closed_auctions].2.[closed_auction].2.[price]*#
```

XMark Query 6

How many items are listed on all continents?

XQuery

```
let $auction := doc("auction.xml") return
for $b in $auction//site/regions
       return count($b//item)
```

sNRC

$$\dot{\texttt{size}}(\{[b..[item] \mid b \in \texttt{in}.[site].2.[regions]]\})$$

Reference Projection Expressions

```
<IN>.[site].2.[regions]..[item]
```

Improved Projection Expressions

```
<IN>.[site].2.[regions]..[item]*#
```

XMark Query 7

How many pieces of prose are in our database?

XQuery

```
let $auction := doc("auction.xml") return
for $p in $auction/site
return
  count($p//description)
  + count($p//annotation)
  + count($p//emailaddress)
```

sNRC

$$\{[\dot{\mathtt{size}}(p.2..[description]) + \dot{\mathtt{size}}(p.2..[annotation]) + \dot{\mathtt{size}}(p.2..[emailaddress]) \mid p \in \mathtt{in}.[site]]\}$$

Reference Projection Expressions

```
<IN>..[description]
<IN>..[emailaddress]
<IN>..[annotation]
```

Improved Projection Expressions

```
<IN>..[description]*#
<IN>..[emailaddress]*#
<IN>..[annotation]*#
```

XMark Query 8

List the names of persons and the number of items they bought.

XQuery

```
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
let $a :=
  for $t in $auction/site/closed_auctions/closed_auction
  where $t/buyer/@person = $p/@id
  return $t
return <item person="{$p/name/text()}">{count($a)}</item>
```

sNRC

$$\{[\langle p.2.[name].2, \dot{\mathtt{size}}(\{[t \mid t \in \mathtt{in}.[site].2.[closed\_auctions].2.[closed\_auction], t.2.[buyer].2.[@person].2 \doteq p.2.[@id].2]\})\rangle \mid p \in \mathtt{in}.[site].2.[people].2.[person]]\}$$

Reference Projection Expressions

```
<IN>.[site].2.[closed_auctions].2.[closed_auction]
<IN>.[site].2.[people].2.[person]
```

Improved Projection Expressions

```
<IN>.[site].2.[closed_auctions].2.[closed_auction]*#
```

```
<IN>.[site].2.[closed_auctions].2.[closed_auction]*#.2.[buyer].2.[@person].2
<IN>.[site].2.[people].2.[person]*.2.[name].2
<IN>.[site].2.[people].2.[person]*.2.[@id].2
```

XMark Query 9a

List the names of persons and the names of the items they bought in Europe.

XQuery

```
let $auction := doc("auction.xml") return
let $ca := $auction/site/closed_auctions/closed_auction return
let
    $ei := $auction/site/regions/europe/item
for $p in $auction/site/people/person
let $a :=
  for $t in $ca
  where $p/@id = $t/buyer/@person
  return
    let $n := for $t2 in $ei where $t/itemref/@item = $t2/@id return $t2
    return <item>{$n/name/text()}</item>
return <person name="{$p/name/text()}">{$a}</person>
```

sNRC

$$\{[\langle p.2.[name].2, \{[\{[t2.2.[name].2 \mid t2 \in \mathbf{in}.[site].2.[regions].2.[europe].2.[item], t2.2.[@id].2 \doteq t.2.[itemref].2.[@item].2]\} \mid t \in \mathbf{in}.[site].2.[closed\_auctions].2.[closed\_auction]]\} \mid p \in \mathbf{in}.[site].2.[people].2.[person]]\}$$

Reference Projection Expressions

```
<IN>.[site].2.[closed_auctions].2.[closed_auction]
<IN>.[site].2.[people].2.[person]
<IN>.[site].2.[regions].2.[europe].2.[item]
```

Improved Projection Expressions

```
<IN>.[site].2.[closed_auctions].2.[closed_auction]*.2.[itemref].2.[@item].2
<IN>.[site].2.[closed_auctions].2.[closed_auction]*.2.[buyer].2.[@person].2
<IN>.[site].2.[people].2.[person]*.2.[name].2
<IN>.[site].2.[people].2.[person]*.2.[@id].2
<IN>.[site].2.[regions].2.[europe].2.[item]*.2.[name].2
<IN>.[site].2.[regions].2.[europe].2.[item]*.2.[@id].2
```

XMark Query 9b

This is a variant of XMark Query 9a (the original XMark Query 9) which introduces a descendant (or sub-element) relationship for the closed auctions for additional com-

plexity.

sNRC

$$\{[\langle p.2.[name].2, \{[\{[t2.2.[name].2 \mid t2 \in \mathbf{in}.[site].2.[regions].2.[europe].2.[item] \\ t2.2.[@id].2 \doteq t.2.[itemref].2.[@item].2]\} \mid t \in \mathbf{in}.[site]..[closed\_auction]]\} \mid \\ p \in \mathbf{in}.[site].2.[people].2.[person]]\}$$

Reference Projection Expressions

```
<IN>.[site]..[closed_auction]
<IN>.[site].2.[people].2.[person]
<IN>.[site].2.[regions].2.[europe].2.[item]
```

Improved Projection Expressions

```
<IN>.[site]..[closed_auction]*.2.[itemref].2.[@item].2
<IN>.[site]..[closed_auction]*.2.[buyer].2.[@person].2
<IN>.[site].2.[people].2.[person]*.2.[name].2
<IN>.[site].2.[people].2.[person]*.2.[@id].2
<IN>.[site].2.[regions].2.[europe].2.[item]*.2.[name].2
<IN>.[site].2.[regions].2.[europe].2.[item]*.2.[@id].2
```

XMark Query 10

List all persons according to their interest and use French markup in the result.

XQuery

```
let $auction := doc("auction.xml") return
for $i in
  distinct-values($auction/site/people/person/profile/interest/@category)
let $p :=
  for $t in $auction/site/people/person
  where $t/profile/interest/@category = $i
  return
    <personne>
      <statistiques>
        <sexe>{$t/profile/gender/text()}</sexe>
        <age>{$t/profile/age/text()}</age>
        <education>{$t/profile/education/text()}</education>
        <revenu>{fn:data($t/profile/@income)}</revenu>
      </statistiques>
      <coordonnees>
        <nom>{$t/name/text()}</nom>
        <rue>{$t/address/street/text()}</rue>
        <ville>{$t/address/city/text()}</ville>
        <pays>{$t/address/country/text()}</pays>
        <reseau>
          <courrier>{$t/emailaddress/text()}</courrier>
          <pagePerso>{$t/homepage/text()}</pagePerso>
```

```
        </reseau>
      </coordonnees>
      <cartePaiement>{$t/creditcard/text()}</cartePaiement>
    </personne>
return <categorie>{<id>{$i}</id>, $p}</categorie>
```

sNRC

$$\{[\langle i, \{\langle\{personne\}, \langle\{statistiques\}, \{\langle\{sexe\}, t.2.[profile].2.[gender].2\rangle, \langle\{age\}, t.2.[profile].2.[age].2\rangle,$$
$$\langle\{education\}, t.2.[profile].2.[education].2\rangle, \langle\{revenu\}, t.2.[profile].2.[@income].2\rangle\}\rangle$$
$$\langle\{coordonnees\}, \{\langle\{nom\}, t.2.[name].2\rangle, \langle\{rue\}, t.2.[address].2.[street].2\rangle, \langle\{ville\}, t.2.[address].2.[city].2\rangle,$$
$$\langle\{pays\}, t.2.[address].2.[country].2\rangle, \langle\{reseau\}, \{\langle\{courrier\}, t.2.[emailaddress].2\rangle,$$
$$\langle\{pagePerso\}, t.2.[homepage].2\rangle\}\rangle\}\langle\{cartePaiement\}, t.2.[creditcard].2\rangle\}\rangle \mid$$
$$i \in \mathbf{set}(\mathbf{in}.[site].2.[people].2.[person].2.[profile].2.[interest].2.[@category].2),$$
$$t \in \mathbf{in}.[site].2.[people].2.[person], t.2.[profile].2.[interest].2.[@category].2 \doteq i]\}$$

Reference Projection Expressions

```
<IN>.[site].2.[people].2.[person]
```

Improved Projection Expressions

```
<IN>.[site].2.[people].2.[person]*.2.[profile].2.[interest].2.[@category].2
<IN>.[site].2.[people].2.[person]*.2.[profile].2.[gender].2
<IN>.[site].2.[people].2.[person]*.2.[profile].2.[education].2
<IN>.[site].2.[people].2.[person]*.2.[profile].2.[age].2
<IN>.[site].2.[people].2.[person]*.2.[profile].2.[@income].2
<IN>.[site].2.[people].2.[person]*.2.[address].2.[city].2
<IN>.[site].2.[people].2.[person]*.2.[address].2.[street].2
<IN>.[site].2.[people].2.[person]*.2.[address].2.[country].2
<IN>.[site].2.[people].2.[person]*.2.[name].2
<IN>.[site].2.[people].2.[person]*.2.[emailaddress].2
<IN>.[site].2.[people].2.[person]*.2.[homepage].2
<IN>.[site].2.[people].2.[person]*.2.[creditcard].2
```

XMark Query 11

For each person, list the number of items currently on sale whose price does not exceed 0.02% of the person's income.

XQuery

```
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
let $l :=
  for $i in $auction/site/open_auctions/open_auction/initial
  where $p/profile/@income > 5000 * exactly-one($i/text())
  return $i
return <items name="{$p/name/text()}">{count($l)}</items>
```

sNRC

$$\{[\langle p.2.[name].2, \dot{\texttt{size}}(\{[i \mid i \in \texttt{in}.[site].2.[open\_auctions].2.[open\_auction].2.[initial], \\ p.2.[profile].2.[@income].2 > \{5000 * i.2\}]\})\rangle \mid p \in \texttt{in}.[site].2.[people].2.[person]]\}$$

Reference Projection Expressions

```
<IN>.[site].2.[people].2.[person]
<IN>.[site].2.[open_auctions].2.[open_auction]
```

Improved Projection Expressions

```
<IN>.[site].2.[people].2.[person]*.2.[profile].2.[@income].2
<IN>.[site].2.[people].2.[person]*.2.[name].2
<IN>.[site].2.[open_auctions].2.[open_auction].2.[initial]*#
<IN>.[site].2.[open_auctions].2.[open_auction].2.[initial]*#.2
```

XMark Query 12

For each richer-than-average person, list the number of items currently on sale whose price does not exceed 0.02% of the person's income.

XQuery

```
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
let $l :=
  for $i in $auction/site/open_auctions/open_auction/initial
  where $p/profile/@income > 5000 * exactly-one($i/text())
  return $i
where $p/profile/@income > 50000
return <items person="{$p/profile/@income}">{count($l)}</items>
```

sNRC

$$\{[\langle p.2.[name].2, \dot{\texttt{size}}(\{[i \mid i \in \texttt{in}.[site].2.[open\_auctions].2.[open\_auction].2.[initial], \\ p.2.[profile].2.[@income].2 > \{5000 * i.2\}]\})\rangle \mid p \in \texttt{in}.[site].2.[people].2.[person], \\ p.2.[profile].2.[@income].2 > \{50000\}]\}$$

Reference Projection Expressions

```
<IN>.[site].2.[people].2.[person]
<IN>.[site].2.[open_auctions].2.[open_auction]
```

Improved Projection Expressions

```
<IN>.[site].2.[people].2.[person]*.2.[profile].2.[@income].2
<IN>.[site].2.[people].2.[person]*.2.[name].2
<IN>.[site].2.[open_auctions].2.[open_auction].2.[initial]*#
<IN>.[site].2.[open_auctions].2.[open_auction].2.[initial]*#.2
```

XMark Query 13

List the names of items registered in Australia along with their descriptions.

XQuery

```
let $auction := doc("auction.xml") return
for $i in $auction/site/regions/australia/item
return <item name="{$i/name/text()}">{$i/description}</item>
```

sNRC

$\{[\langle i.2.[name].2, i.2.[description].2\rangle \mid i \in \mathbf{in}.[site].2.[regions].2.[australia].2.[item]]\}$

Reference Projection Expressions

```
<IN>.[site].2.[regions].2.[australia].2.[item]
```

Improved Projection Expressions

```
<IN>.[site].2.[regions].2.[australia].2.[item]*.2.[name].2
<IN>.[site].2.[regions].2.[australia].2.[item]*.2.[description].2
```

XMark Query 14

Return the names of all items whose description contains the word ‘gold’.

XQuery

```
let $auction := doc("auction.xml") return
for $i in $auction/site//item
where contains(string(exactly-one($i/description)), "gold")
return $i/name/text()
```

sNRC

$\{[i.2.[name].2 \mid i \in \mathbf{in}.[site].2..[item], \mathbf{contains}(i.2.[description].2, \{gold\})]\}$

Reference Projection Expressions

```
<IN>.[site].2..[item]
```

Improved Projection Expressions

```
<IN>.[site].2..[item]*.2.[name].2
<IN>.[site].2..[item]*.2.[description].2
```

XMark Query 15

Print the keywords in emphasis in annotations of closed auctions.

XQuery

```
let $auction := doc("auction.xml") return
for $a in
  $auction/site/closed_auctions/closed_auction/annotation/
   description/
   parlist/
   listitem/
   parlist/
   listitem/
   text/
   emph/
   keyword/
   text()
return <text>{$a}</text>
```

sNRC

$\{[a \mid i \in \mathbf{in}.[site].2.[closed\_auctions].2.[closed\_auction].2.[annotation].2$
$.[description].2.[parlist].2.[listitem].2.[parlist].2.[listitem].2.[emph].2.[keyword].2]\}$

Reference Projection Expressions

```
<IN>.[site].2.[closed_auctions].2.[closed_auction].2.[annotation].2.[description].2
       .[parlist].2.[listitem].2.[parlist].2.[listitem].2.[text].2.[emph].2.[keyword]
```

Improved Projection Expressions

```
<IN>.[site].2.[closed_auctions].2.[closed_auction].2.[annotation].2.[description].2
       .[parlist].2.[listitem].2.[parlist].2.[listitem].2.[text].2.[emph].2.[keyword].2*
```

XMark Query 16

Return the IDs of those auctions that have one or more keywords in emphasis.

XQuery

```
let $auction := doc("auction.xml") return
for $a in $auction/site/closed_auctions/closed_auction
where
  not(
    empty(
      $a/annotation/description/parlist/listitem/
       parlist/listitem/text/emph/
       keyword/
       text()
    )
```

```
  )
return <person id="{$a/seller/@person}"/>
```

sNRC

$$\{[a.2.[seller].2.[@person] \mid a \in \mathbf{in}.[site].2.[closed\_auctions].2.[closed\_auction],\ a.2.[annotation].2.[description].2.[parlist].2.[listitem].2.[parlist].2.[listitem].2.[emph].2.[keyword].2 \doteq \emptyset]\}$$

Reference Projection Expressions

```
<IN>.[site].2.[closed_auctions].2.[closed_auction]
```

Improved Projection Expressions

```
<IN>.[site].2.[closed_auctions].2.[closed_auction]*.2.[annotation].2.[description]
       .2.[parlist].2.[listitem].2.[parlist].2.[listitem].2.[text].2.[emph].2.[keyword].2
<IN>.[site].2.[closed_auctions].2.[closed_auction]*.2.[seller].2.[@person]
```

XMark Query 17

Which persons don't have a homepage?

XQuery

```
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
where empty($p/homepage/text())
return <person name="{$p/name/text()}"/>
```

sNRC

$$\{[p.2.[name].2 \mid p \in \mathbf{in}.[site].2.[people].2.[person], p.2.[homepage].2 \doteq \emptyset]\}$$

Reference Projection Expressions

```
<IN>.[site].2.[people].2.[person]
```

Improved Projection Expressions

```
<IN>.[site].2.[people].2.[person]*.2.[name].2
<IN>.[site].2.[people].2.[person]*.2.[homepage].2
```

XMark Query 20

Group customers by their income and output the cardinality of each group.

XQuery

```
let $auction := doc("auction.xml") return
```

```
<result>
  <preferred>
    {count($auction/site/people/person/profile[@income >= 100000])}
  </preferred>
  <standard>
    {
      count(
        $auction/site/people/person/
         profile[@income < 100000 and @income >= 30000]
      )
    }
  </standard>
  <challenge>
    {count($auction/site/people/person/profile[@income < 30000])}
  </challenge>
  <na>
    {
      count(
        for $p in $auction/site/people/person
        where empty($p/profile/@income)
        return $p
      )
    }
  </na>
</result>
```

sNRC

$$
\begin{aligned}
&incomes = \mathbf{in}.site.2.people.2.person.2.profile.2.@income\\
&\quad \{\{\langle\{preferred\}, \dot{\mathbf{size}}(\{[a \mid a \in incomes.2, a \geq \{\{100000\}\}]\})\rangle\\
&\quad \langle\{standard\}, \dot{\mathbf{size}}(\{[b \mid b \in incomes.2, a < \{\{100000\}\}, a \geq \{\{30000\}\}]\})\rangle\\
&\quad \langle\{challenge\}, \dot{\mathbf{size}}(\{[c \mid c \in incomes.2, c < \{\{30000\}\}]\})\rangle\\
&\quad \langle\{na\}, \dot{\mathbf{size}}(\{[d \mid c \in incomes, c.2 \doteq \emptyset]\})\rangle\}\}
\end{aligned}
$$

Reference Projection Expressions

```
<IN>.[site].2.[people].2.[person]
```

Improved Projection Expressions

```
<IN>.[site].2.[people].2.[person]*#
<IN>.[site].2.[people].2.[person]*#.2.[profile].2.[@income].2
```

Appendix B

Flink Execution Plans for sNRC XMark Queries

In this Appendix the Apache Flink execution plans for the sNRC XMark queries from chapter A are shown. These plans are used by Flink to execute the sNRC queries in a distributed manner. The plans are generated by using Flink's Plan Visualizer on JSON exports of the Flink execution plans.

XMark Query 1

The Flink execution plan for the sNRC implementation of XMark Query 1 is shown in Figure B.1.

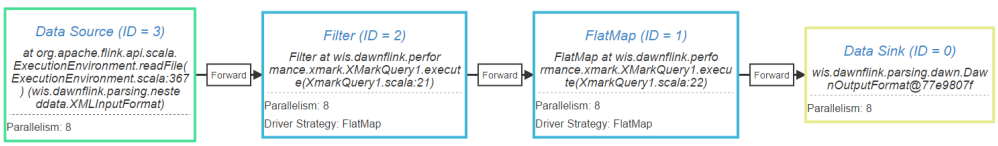

Figure B.1: Flink Execution Plan for sNRC XMark Query 1

XMark Query 5

The Flink execution plan for the sNRC implementation of XMark Query 5 is shown in Figure B.2.

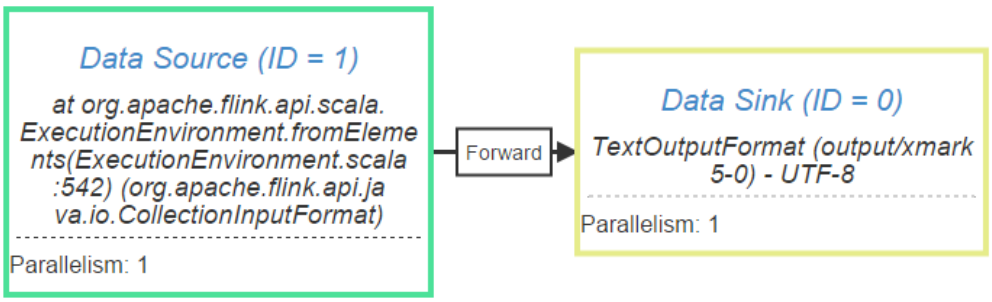

Figure B.2: Flink Execution Plan for sNRC XMark Query 5

XMark Query 6

The Flink execution plan for the sNRC implementation of XMark Query 6 is shown in Figure B.3.

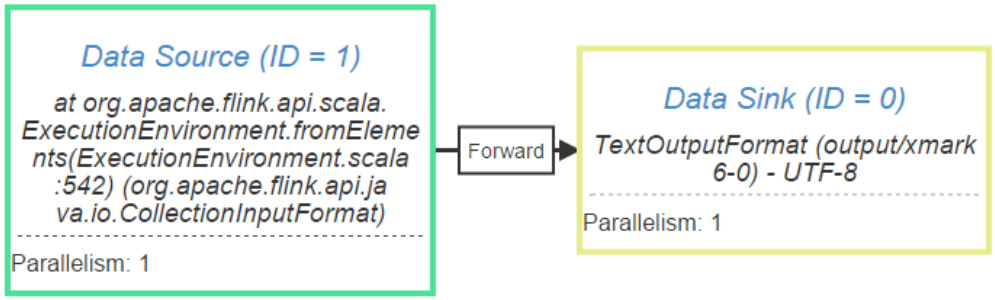

Figure B.3: Flink Execution Plan for sNRC XMark Query 6

XMark Query 7

The Flink execution plan for the sNRC implementation of XMark Query 7 is shown in Figure B.4.

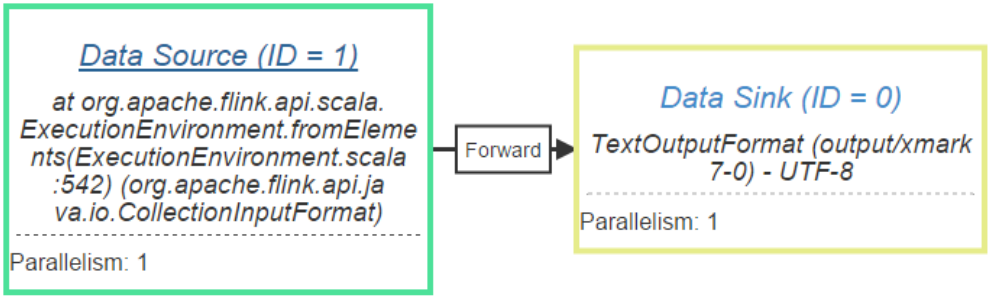

Figure B.4: Flink Execution Plan for sNRC XMark Query 7

XMark Query 8

The Flink execution plan for the sNRC implementation of XMark Query 8 is shown in Figure B.5.

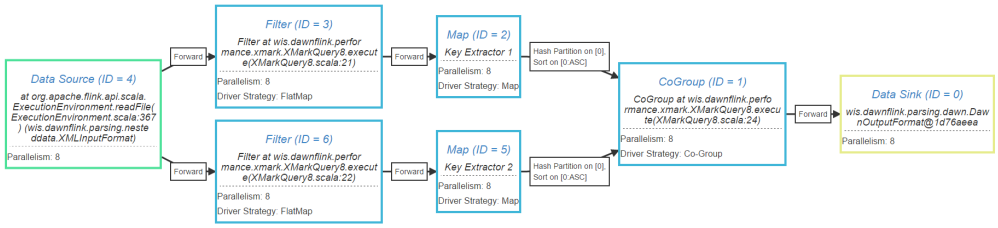

Figure B.5: Flink Execution Plan for sNRC XMark Query 8

XMark Query 9

The Flink execution plan for the sNRC implementation of XMark Query 9 is shown in Figure B.6.

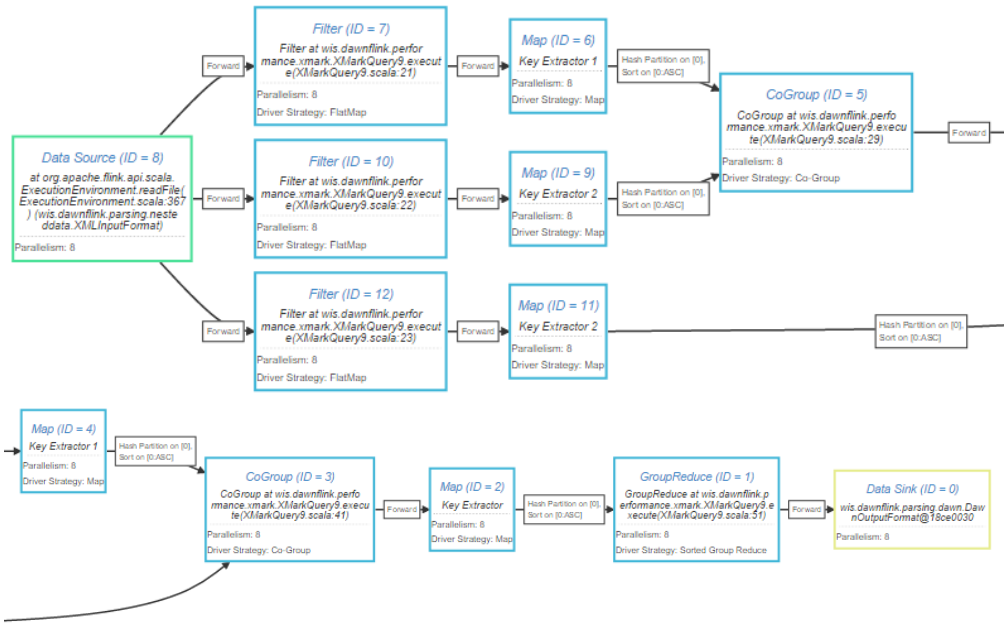

Figure B.6: Flink Execution Plan for sNRC XMark Query 9

XMark Query 10

The Flink execution plan for the sNRC implementation of XMark Query 10 is shown in Figure B.7.

Figure B.7: Flink Execution Plan for sNRC XMark Query 10

XMark Query 11

The Flink execution plan for the sNRC implementation of XMark Query 11 is shown in Figure B.8.

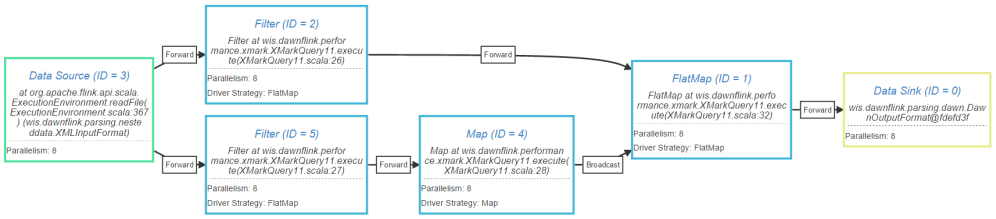

Figure B.8: Flink Execution Plan for sNRC XMark Query 11

XMark Query 12

The Flink execution plan for the sNRC implementation of XMark Query 12 is shown in Figure B.9.

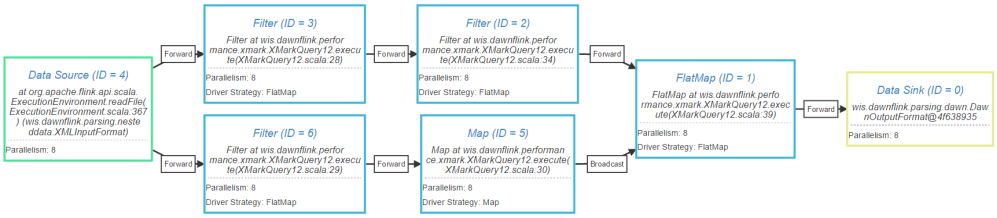

Figure B.9: Flink Execution Plan for sNRC XMark Query 12

XMark Query 13

The Flink execution plan for the sNRC implementation of XMark Query 13 is shown in Figure B.10.

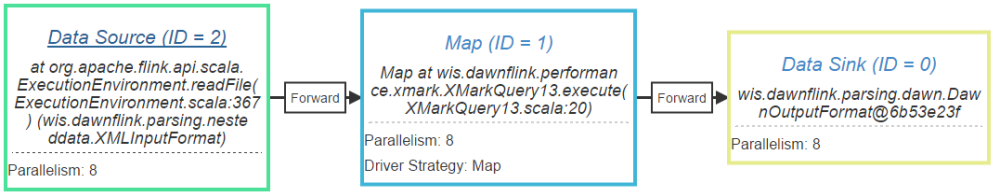

Figure B.10: Flink Execution Plan for sNRC XMark Query 13

XMark Query 14

The Flink execution plan for the sNRC implementation of XMark Query 14 is shown in Figure B.11.

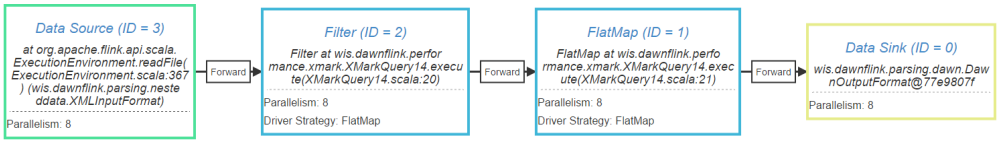

Figure B.11: Flink Execution Plan for sNRC XMark Query 14

XMark Query 15

The Flink execution plan for the sNRC implementation of XMark Query 15 is shown in Figure B.12.

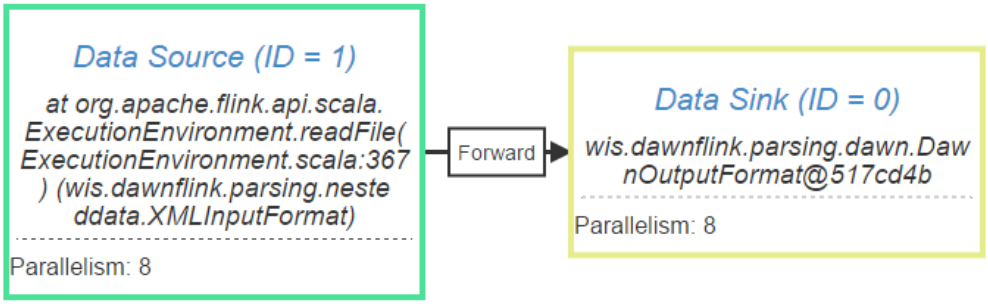

Figure B.12: Flink Execution Plan for sNRC XMark Query 15

XMark Query 16

The Flink execution plan for the sNRC implementation of XMark Query 16 is shown in Figure B.13.

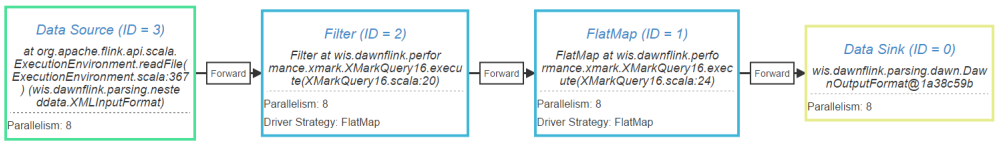

Figure B.13: Flink Execution Plan for sNRC XMark Query 16

XMark Query 17

The Flink execution plan for the sNRC implementation of XMark Query 17 is shown in Figure B.14.

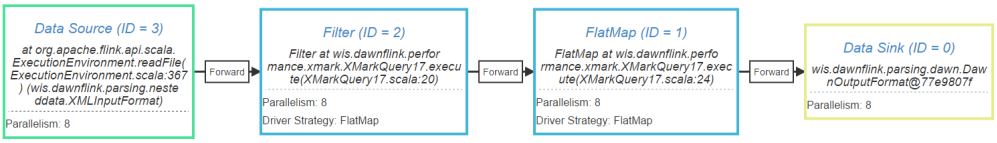

Figure B.14: Flink Execution Plan for sNRC XMark Query 17

XMark Query 20

The Flink execution plan for the sNRC implementation of XMark Query 20 is shown in Figure B.15.

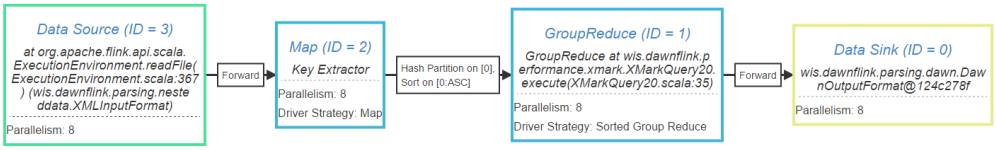

Figure B.15: Flink Execution Plan for sNRC XMark Query 20

Appendix C

sNRC: NRC as a Data Analytics Workflow Notation

NOTE: *The work that is shown in this appendix is property of Jan Hidders and his colleagues. This work on the Nested Relational Calculus for Semi-Structured data is used as a basis for this thesis. Because it is work in progress that is not yet published, this tidied up version is included as an appendix.*

Goal of document

This is a working document that describes our investigations into the design of a data analytics workflow language. The languages are based on well-known formalisms such as NRC and formal versions of XQuery. The intent is to design a language that (1) is well-understood in terms of semantics and expressive power, (2) can be efficiently implemented specifically on back-ends offering large scale parallelised processing and (3) allows the application of well-known optimisation techniques such as cost-based query rewriting. Moreover, the language comes in different textual and graphical notations that can be used for both theoretical analysis and workflow-like graphical presentations of the analytical workflows.

The Underlying Data Model: Nested Values

To define the data model on which we will operate, we postulate the following sets and basic concepts:

- C denotes the set of basic value constant denotations (like booleans, strings, integers, etc.), $\mathcal{B}$ denotes the set of basic values, we distinguish one special basic value constant denoted as $\langle\rangle$
- bags / multisets are denoted as $\{\{1,1,2\}\}$, the additive bag union is denoted as $\uplus$, we use $\{\{f(\bar{x}) \mid \varphi(\bar{x})\}\}$ to denote bag comprehension. The empty bag is denoted as $\emptyset$.
- ordered pairs containing values x and y are denoted as $\langle x, y\rangle$

- $\mathcal{V}$ denotes the set of *nested values*, which are bags of items, where *items* are either (1) a basic value in $\mathcal{B}$, (2) an ordered pair $\langle v_1, v_2 \rangle$ where $v_1, v_2 \in \mathcal{V}$. Note that we do not allow bags of bags, and that tuples can contain only bags. We explicitly allow heterogeneous bags, i.e., bags that contain elements of different types. We will allow only finitely nested values, i.e., we assume $\mathcal{V}$ is the smallest set that satisfies this definition.
  - The ordered pair $\langle x, y \rangle$ can also be understood as a key-value pair, rather then a small tuple. So JSON objects can be represented as bags of such pairs. That captures the idea that field names are first class citizens in the language.
  - It may seem odd that we allow only bags in ordered pairs, but it is a consequence of the "all expressions return a bag" principle that is followed in this language and simplifies its semantics and allows us to ignore typing.
  - As a shorthand we will let $\langle\rangle$ denote the pair $\langle \emptyset, \emptyset \rangle$, and $\langle v \rangle$ the value $\langle \emptyset, v \rangle$.

NRC for semistructured data: sNRC

We give here a formal definition of the dialect of NRC we will study here.

Preliminary notions

We postulate the following sets and basic concepts:

- X denotes the set of variable names
- B denoting the set of basic (user-defined) functions, with each $b \in B$ we associated a binary relation $[\![b]\!]$ that associates nested values with nested values

The syntax of sNRC

The syntax for the calculus over bags we intend to use:

$$\begin{aligned} E ::= {} & \mathbf{in} \mid X \mid C \mid \langle E, E \rangle \mid E.1 \mid E.2 \mid \\ & \emptyset \mid E \uplus E \mid \{\!\{ E \mid X \in E, \ldots, X \in E \}\!\} \mid \\ & B(E) \mid \dot{\mathbf{set}}(E) \mid E \doteq E. \end{aligned}$$

Here **in** denotes the input value, X denotes variables, C basic value constants, $\{\!\{e \mid \Delta\}\!\}$ denotes the flattening bag comprehension (i.e., it is a comprehension which additionally flattens the result to avoid bags of bags), B the basic user-defined functions. The function $\dot{\mathbf{set}}()$ eliminates duplicates and non-basic values. The expression $e_1 \doteq e_2$ compares basic values and returns a bag containing as many occurrences of $\langle\rangle$ as there are pairs of occurrences in e_1 and e_2, respectively, that represent the same basic value. E.g., comparing the value $\{\!\{1,2,2,3,3,4\}\!\}$ with $\{\!\{2,2,3\}\!\}$ using the $\doteq$ operator results in $\{\!\{\langle\rangle, \langle\rangle, \langle\rangle, \langle\rangle, \langle\rangle, \langle\rangle\}\!\}$.

We allow for denotations of values the same short-hands as for the values themselves: $\langle\rangle$ denotes the value $\langle\rangle = \langle \emptyset, \emptyset \rangle$ and $\langle e \rangle$ denotes $\langle e \rangle = \langle \emptyset, e \rangle$. We let $\{\!\{e \mid]\}\!\}$ also

be simply denoted as $\{[e]\}$. We will use in the right-hand side of comprehensions the short-hand $(e_1 \doteq e_2) \triangleq z \in (e_1 \doteq e_2)$ with z some fresh variable.

The $\dot{\mathbf{set}}()$ operator may seem weak, but does allow us to express duplicate elimination as it happens in the relational model. For example, if R_1 contains a bag of pairs containing singleton basic values then the corresponding set can be expressed as $\{[\langle x,y\rangle \mid x \in \dot{\mathbf{set}}(r.1), y \in \dot{\mathbf{set}}(\{[z.2 \mid z \in r, z.1 \doteq x]\})]\}$. Note that these expressions can also be used to simulate reasoning in settings where the input contains no duplicates and only singleton fields, by replacing the input relation R_1 with this expression. If this expression is called r' we can determine if $f(r) \equiv g(r)$ for such R_1 by determining if $f(r') \equiv f(r')$.

The semantics of sNRC

To ensure the definedness of the result of each expression in a semistructured and possibly untyped setting we will assume that all values, both inputs and outputs, are bags. This is similar to the approach taken in XQuery. In fact the language is similar to *XQuery core* as studied in terms of expressive power and evaluation complexity by Koch (2006) and Benedikt (2009). Note that this means that the expression 12 in fact denotes the bag $\{\{12\}\}$ rather then the number 12. The rule of thumb for operators that normally do not expect a bag is that they are mapped over the elements of the bag. So e.1 in fact constructs a bag by iterating over each element from the result of e and for each pair returning the first element. Also as in XQuery, the comprehension automatically flattens the result to avoid the construction of bags directly nested inside bags. So, for example $\{[\{[5]\} \mid x \in e]\}$ is equivalent to $\{[5 \mid x \in e]\}$ which returns a bag containing only the number 5 and is of the size of the result of e. Indeed $\{\{e\}\}$ is always equivalent to e. Consequently the expressions $\{[1]\}$ and 1 both denote the value $\{\{1\}\}$, and the expressions $1 \uplus 2$ and $\{[1]\} \uplus \{[2]\}$ both denote $\{\{1,2\}\}$.

The semantics is defined in terms of propositions of the form $\Gamma \vdash e \Rightarrow v$ where Γ is variable binding, i.e., a function that maps variable names to *items*, e an NRC expression and v a bag of nested values that represents the result of the evaluation of e under Γ. Note that Γ maps variable names to items, rather then nested values, i.e., variables are bound to basic values and ordered pairs but not to bags. This is done for the sake of simplicity as in this research we mostly use variables to iterate over the elements of a bag. Moreover, assigning a bag b to a variable x can be simulated by assigning the item $\langle \emptyset, b\rangle$ and everywhere that x occurs freely in the expression replacing it with $x.2$.

$$\frac{}{\Gamma \vdash x \Rightarrow \{\{\Gamma(x)\}\}} \qquad \frac{}{\Gamma \vdash c \Rightarrow \{\{c\}\}} \qquad \frac{\Gamma \vdash e_1 \Rightarrow v_2 \qquad \Gamma \vdash e_2 \Rightarrow v_2}{\Gamma \vdash \langle e_1, e_2 \rangle \Rightarrow \{\{\langle v_1, v_2 \rangle\}\}}$$

$$\frac{\Gamma \vdash e \Rightarrow v}{\Gamma \vdash e.i \Rightarrow \{\{u \mid \langle w_1, w_2 \rangle \in v, u \in w_i\}\}} \qquad \frac{}{\Gamma \vdash \emptyset \Rightarrow \emptyset} \qquad \frac{\Gamma \vdash e_1 \Rightarrow v \qquad \Gamma \vdash e_2 \Rightarrow w}{\Gamma \vdash e_1 \uplus e_2 \Rightarrow v \uplus w}$$

$$\frac{\Gamma \vdash e \Rightarrow v}{\Gamma \vdash \{[e \mid]\} \Rightarrow v} \qquad \frac{\Gamma \vdash e_2 \Rightarrow \{\{v_1, \ldots, v_m\}\} \qquad \forall_{i=1}^{m} (\Gamma_{[x \mapsto v_i]} \vdash \{[e_1 \mid \Delta]\} \Rightarrow w_i)}{\Gamma \vdash \{[e_1 \mid x \in e_2, \Delta]\} \Rightarrow \uplus_{i=1}^{m} w_i}$$

$$\frac{\Gamma \vdash e \Rightarrow v \qquad (v, w) \in [\![b]\!]}{\Gamma \vdash b(e) \Rightarrow w} \qquad \frac{\Gamma \vdash e \Rightarrow v}{\Gamma \vdash \dot{\mathbf{set}}(e) \Rightarrow \{x \mid x \in v, x \in \mathcal{B}\}}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow v \qquad \Gamma \vdash e_2 \Rightarrow w}{\Gamma \vdash e_1 \doteq e_2 \Rightarrow \{\{\langle\rangle \mid c_1 \in v, c_2 \in w, c_1 = c_2, c_1 \in \mathcal{B}\}\}}$$

Note that the iterators in the comprehension iterate over all the elements of a bag. The semantics of an sNRC expression can be interpreted as a total function that maps variable bindings to a nested values, presuming that all user-defined functions are also total functions that map nested values to nested values.

NRA for semistructured data: sNRA

We give here a formal definition of the dialect of NRA that is the algebraic counterpart of sNRC.

The syntax of sNRA

As an algebraic counterpart of sNRC we present sNRA. It has the following syntax:

$$F ::= \mathbf{id} \mid {}^{\lambda}C \mid F \circ F \mid {}^{\lambda}\langle F, F \rangle \mid \pi_1 \mid \pi_2 \mid$$
$${}^{\lambda}\emptyset \mid {}^{\lambda}\uplus \mid \mathbf{fmap}(F) \mid {}^{\lambda}\times \mid B \mid \dot{\mathbf{set}} \mid {}^{\lambda}\doteq.$$

Note that we annotate some constructs with λ to indicate they denote functions rather then values.

The semantics of sNRA

The semantics of the algebra is defined by the following rules. They define the proposition $(x, y) \in [\![e]\!]$ which denotes that the function associated with e maps the value x to the value y.

$$\frac{x \in \mathcal{V}}{(x,x) \in [\![\mathbf{id}]\!]} \qquad \frac{x \in \mathcal{V}}{(x,c) \in [\![\overset{\lambda}{c}]\!]} \qquad \frac{(x,y) \in [\![f]\!] \qquad (y,x) \in [\![g]\!]}{(x,z) \in [\![g \circ f]\!]}$$

$$\frac{(x,y) \in [\![f]\!] \qquad (x,z) \in [\![g]\!]}{(x,\{\!\{\langle y,z\rangle\}\!\}) \in [\![\overset{\lambda}{\langle f,g\rangle}]\!]} \qquad \frac{x \in \mathcal{V} \qquad y = \{\!\{v_i \mid \langle v_1,v_2\rangle \in x\}\!\}}{(x,y) \in [\![\pi_i]\!]} \qquad \frac{x \in \mathcal{V}}{(x,\emptyset) \in [\![\overset{\lambda}{\emptyset}]\!]}$$

$$\frac{x \in \mathcal{V} \qquad y = \{\!\{z \mid \langle u,v\rangle \in x, z \in (u \uplus v)\}\!\}}{(x,y) \in [\![\overset{\lambda}{\uplus}]\!]}$$

$$\frac{x \in \mathcal{V} \qquad y = \{\!\{v \mid z \in x, (\{\!\{z\}\!\},u) \in [\![f]\!], v \in u\}\!\}}{(x,y) \in [\![\mathbf{fmap}(f)]\!]}$$

$$\frac{x \in \mathcal{V} \qquad y = \{\!\{\langle \{\!\{s\}\!\}, \{\!\{t\}\!\}\rangle \mid \langle u,v\rangle \in x, s \in u, t \in v\}\!\}}{(x,y) \in [\![\overset{\lambda}{\times}]\!]}$$

$$\frac{x \in \mathcal{V} \qquad y = \{z \mid z \in x, z \in \mathcal{B}\}}{(x,y) \in [\![\dot{\mathbf{set}}]\!]}$$

$$\frac{x \in \mathcal{V} \qquad y = \{\!\{\langle\rangle \mid \langle u,v\rangle \in x, s \in u, t \in v, s = t, s \in \mathcal{B}\}\!\}}{(x,y) \in [\![\overset{\lambda}{\doteq}]\!]}$$

Note that no rule is specified for basic functions in B since their semantics was already postulated.

The relationship between sNRA and sNRC

There is a direct relationship between NRA and NRC in expressive power. To illustrate this we show that they can be mapped to each other.

Mapping sNRA to sNRC

Each sNRA expression can be represented by an sNRC expression with a single special free variable **in** that represents the input value, and vice versa. We let $e_{[x/e']}$ denote the expression e with all free occurrences of x replaced with e'.

$M(\mathbf{id}) = \mathbf{in}$
$M(\overset{\lambda}{c}) = c$
$M(g \circ f) = \{[M(g)_{[\mathbf{in}/x]} \mid x \in M(f)]\}$
$M(\overset{\lambda}{\langle f,g\rangle}) = \langle M(f), M(g)\rangle$
$M(\pi_i) = \mathbf{in}.i$
$M(\overset{\lambda}{\emptyset}) = \emptyset$
$M(\overset{\lambda}{\uplus}) = \mathbf{in}.1 \uplus \mathbf{in}.2$
$M(\mathbf{fmap}(f)) = \{[M(f)_{[\mathbf{in}/x]} \mid x \in \mathbf{in}]\}$
$M(\overset{\lambda}{\times}) = \{[\langle y,z\rangle \mid x \in \mathbf{in}, y \in x.1, z \in x.2]\}$
$M(b) = b(\mathbf{in})$

$M(\dot{\mathbf{set}}) = \dot{\mathbf{set}}(\mathbf{in})$
$M(^{\lambda}\doteq) = (\mathbf{in}.1 \doteq \mathbf{in}.2)$

Mapping sNRC to sNRA

We show that each sNRC expression with a single free variable **in** can be represented by an sNRA expression:

$M'(\mathbf{in}) = \mathbf{id}$
$M'(c) = {}^{\lambda}c$
$M'(\langle e_1, e_2\rangle) = {}^{\lambda}\langle M'(e_1), M'(e_2)\rangle$
$M'(e.i) = \pi_i \circ M'(e)$
$M'(\emptyset) = {}^{\lambda}\emptyset$
$M'(e_1 \uplus e_2) = {}^{\lambda}\uplus \circ {}^{\lambda}\langle M'(e_1), M'(e_2)\rangle$
$M'(\{[e\,|]\}) = M'(e)$
$M'(\{[e \mid x \in e']\}) = \mathbf{fmap}(M'(e_{[\mathbf{in}/\mathbf{in}.1, x/\mathbf{in}.2]})) \circ {}^{\lambda}\times \circ {}^{\lambda}\langle \mathbf{id}, M'(e')\rangle$
$M'(\{[e \mid x \in e', \Delta]\}) = M'(\{[\{[e \mid \Delta]\} \mid x \in e']\})$
$M'(b(e)) = b \circ M'(e)$
$M'(\dot{\mathbf{set}}(e)) = \dot{\mathbf{set}} \circ M'(e)$
$M'(e_1 \doteq e_2) = {}^{\lambda}\doteq \circ {}^{\lambda}\langle M'(e_1), M'(e_2)\rangle$

A note on n-ary functions

In the previous setting we used sNRA and sNRA to define unary functions, i.e., functions with one input parameter, but we can easily interpret these functions also as n-ary functions for some n as follows: if we interpret $f : \mathcal{V} \to \mathcal{V}$ as n-ary then we get the function $f^{[n]} : \mathcal{V}^n \to \mathcal{V}$ such that $f^{[n]}(v_1, \ldots, v_n) = f(\{\{\langle v_1, \{\{\langle v_2, \ldots \{\{\langle v_n, \emptyset\rangle\}\} \ldots\rangle\}\}\rangle\}\})$. And, vice versa, if we take an n-ary function $f^{[n]}$ then we can interpret it as a unary function f which is defined such that $f(v) = f^{[n]}(v.1, v.2.1, ..., v(.2)^{n-1}.1)$.

It follows that we can view sNRC and sNRA also as definition languages for n-ary functions. In that case the notion of semantical equivalence changes somewhat since the n-ary interpretation only considers some parts of the input. However, we can define it in terms of the original notion of semantical equivalence. For that we introduce a special short-hand in sNRC denoted as $\mathbf{in}^{[n]}$, which is defined by induction on n such that (1) $\mathbf{in}^{[0]} = \emptyset$ and (2) $\mathbf{in}^{[n+1]} = \langle \mathbf{in}.1, (\mathbf{in}^{[n]})_{[\mathbf{in}/\mathbf{in}.2]}\rangle$. For example, $\mathbf{in}^{[3]} = \langle \mathbf{in}.1, \langle \mathbf{in}.2.1, \langle \mathbf{in}.2.2.1, \emptyset\rangle\rangle\rangle$. Informally this function interprets the input as a tuple of n elements and projects on those. We now say that an sNRC expression e is an *n-ary expressions* if it holds that $e_{[\mathbf{in}/\mathbf{in}^{[n]}]} \equiv e$.

Likewise for sNRA we can define the corresponding projection function $\mathbf{id}^{[n]}$ which is defined by induction on n such that (1) $\mathbf{id}^{[0]} = {}^{\lambda}\emptyset$ and (2) $\mathbf{id}^{[n+1]} = {}^{\lambda}\langle \pi_1, \mathbf{id}^{[n]} \circ \pi_2\rangle$. For example, $\mathbf{id}^{[3]} = {}^{\lambda}\langle \pi_1, {}^{\lambda}\langle \pi_1, {}^{\lambda}\langle \pi_1, {}^{\lambda}\emptyset\rangle \circ \pi_2\rangle \circ \pi_2\rangle$. Then, for an sNRA expression f we say it is an *n-ary function* if it holds that $f \circ \mathbf{id}^{[n]} \equiv f$.

When defining n-ary functions in sNRC we will adopt the convention to let $\mathbf{in}_i$ with $i \leq n$ denote the i'th input value. This is essentially a syntactic short-hand for $\mathbf{in}(.2)^{i-1}.1$. We will from now on assume that all user-define functions are n-ary, and the notation $b(e_1, \ldots, e_n)$ will be used to denote the passing of n parameters to function b.

Theorem C.0.1 (n-ary functions in sNRC)**.** *For every n-ary function in sNRC there is an equivalent sNRC function that uses no* ***in*** *but only* $\mathbf{in}_i$ *for* $i \leq n$.

A graphical notation for sNRC/sNRA

We introduce a graphical notation called DAWN to represent the n-ary functions that can be defined by sNRC and sNRA. The general notation is the usual workflow style as is shown for example in Figure C.1. Every workflow has zero or more input ports, and exactly one output port. If a port has multiple outgoing edges it means the output is copied for each output edge. If an input port has multiple incoming edges, it means all the inputs are combined with an additive bag union. So the input of h is the bag union of the output of f and g. Every output port needs to have at least one outgoing edge, i.e., the result of every component must be used somewhere. The input ports of the whole workflow have zero or more outgoing edges. So not all inputs must be necessarily used. The output port of the whole workflow must have at least one incoming edge, or it will not have a defined value. The workflow must be acyclic, i.e., if we consider each component as a single node, then the connecting edges do not form a directed cycle.

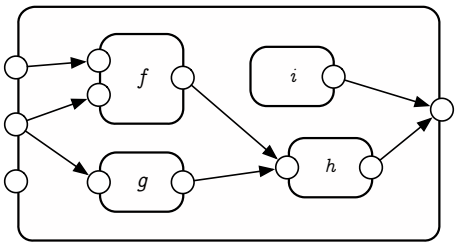

Figure C.1: **G**eneral workflow notation

We will define the semantics of workflows in terms of sNRC expressions denoting n-ary functions, and so do not use $\mathbf{in}$ but do use $\mathbf{in}_i$. Consider the example in Figure C.1. Assuming that we have sNRC expressions e_f, e_g, e_h and e_i for the corresponding components, then the semantics of the whole workflow is given by $\{[(x_i.2 \uplus x_h.2) \mid x_f \in \langle \emptyset, e_f(\mathbf{in}_1, \mathbf{in}_2)\rangle, x_g \in \langle \emptyset, e_g(\mathbf{in}_2)\rangle, x_i \in \langle \emptyset, \langle \emptyset, e_i()\rangle, x_h \in \langle \emptyset, e_h(x_f.2 \uplus x_g.2)\rangle\rangle]\}$. Here expressions of the form $e(e_1, \ldots, e_2)$ are a short-hand for $e_{[\mathbf{in}_1/e_1,\ldots,\mathbf{in}_n/e_n]}$. Note that if we do not wrap the results of the components in a pair, they will be iterated over rather then assigned as a whole to the iterating variable.

We of course allow the workflows to be recursively nested, so a component in the workflow can itself be again a complex workflows.

A special feature is that input ports of a component can be marked as *iterating*, which is indicated by a star, as is shown in Figure C.2. The meaning is that for these ports the incoming bags are iterated over, i.e., the function is applied to each element of the bag (wrapped as a singleton bag). If multiple ports are marked, then all combinations of the elements are taken. So in the figure the computed function is $\{[e_f(x, y, \mathbf{in}_3) \mid x \in \mathbf{in}_1, y \in \mathbf{in}_2]\}$.

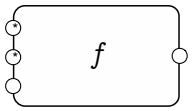

Figure C.2: **I**terating input ports

The primitive components we need to represent the functions in sNRC and sNRA are shown in Figure C.3. They consist of components for: (1) constants c, (2) the pair constructor, (3) the projection operators, (4) the empty bag, (5) the basic value equality operator, (6) user-defined functions and (7) the set operator that returns a set of basic values.

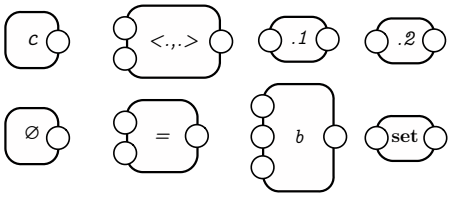

Figure C.3: **P**rimitive workflow components

From the previous the following will be clear.

Theorem C.0.2. *For every DAWN workflow with n input ports there is an equivalent n-ary sNRC expression.*

The converse also holds.

Theorem C.0.3. *For every n-ary sNRC expression there is an equivalent DAWN workflow with n input ports.*

The mapping is illustrated in Figure C.4.

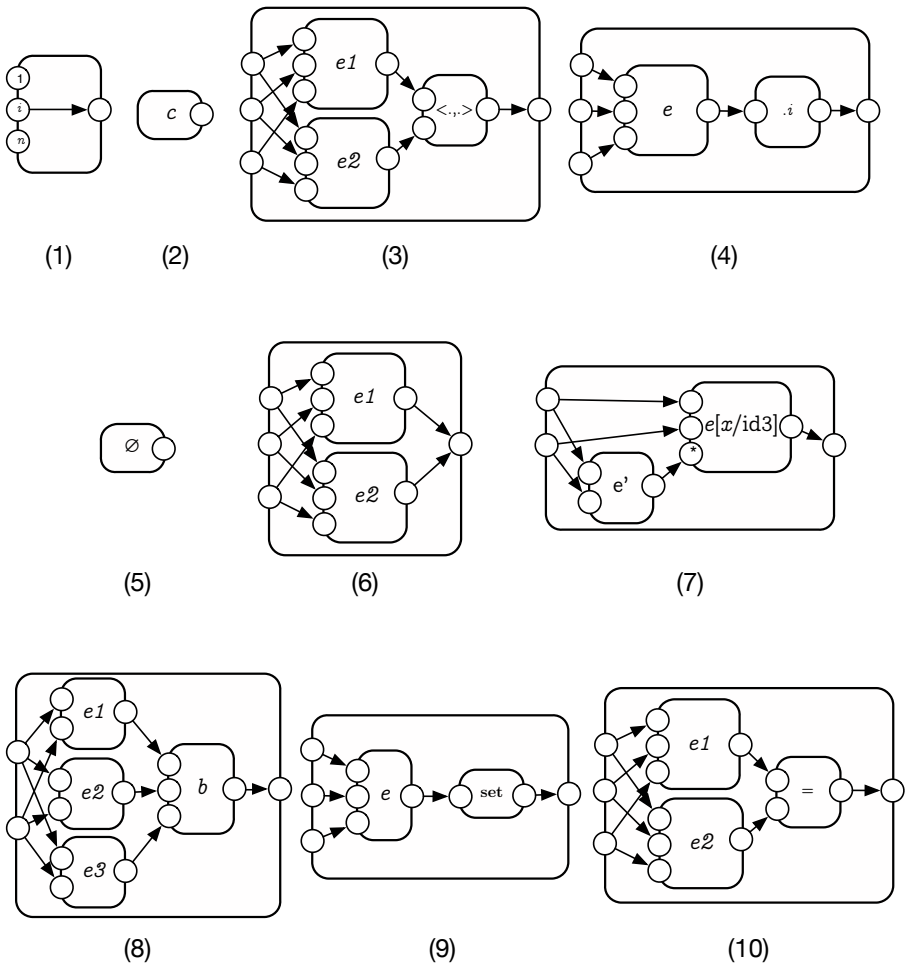

Figure C.4: **M**apping *n*-ary sNRC expressions to DAWN workflows

Appendix D

Used sNRC equivalents of XMark benchmark queries

In this appendix we show various UML diagrams of the sNRC - Flink reference and sNRC - Flink improved system.

Data Model

This UML describes the structure of the sNRC Data Model.

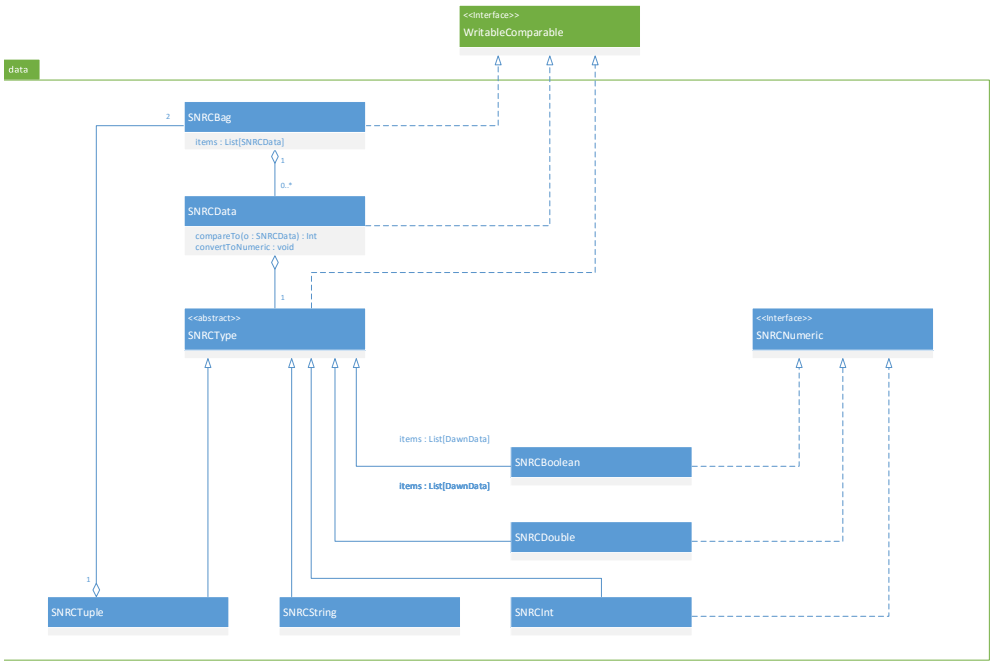

Figure D.1: UML Diagram of the Data Model package

Reference Parallel sNRC Architecture

This UML describes the most important classes of the reference parallel sNRC architecture that is used to execute the XMark Benchmark queries.

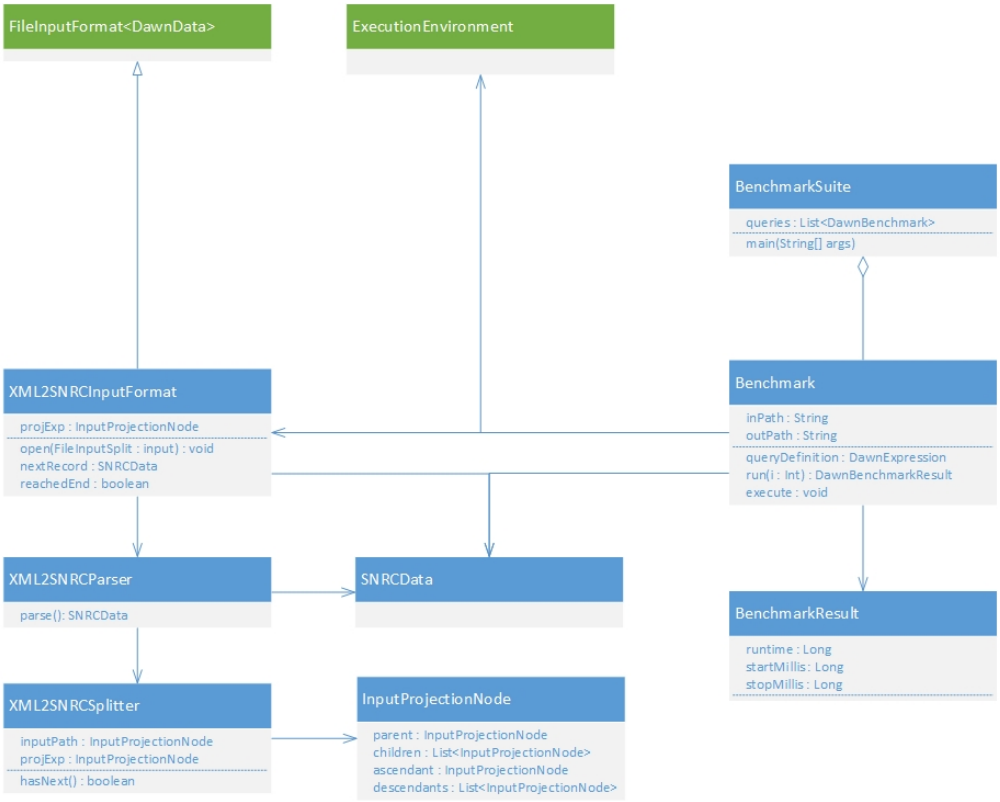

Figure D.2: UML Diagram of core classes of the reference parallel sNRC system

Improved Parallel sNRC Architecture

This UML describes the most important classes of the improved parallel sNRC architecture that is used to execute the XMark Benchmark queries. The static sNRC query analysis, improved loading algorithm and nested data parser interface and XML implementation have been added.

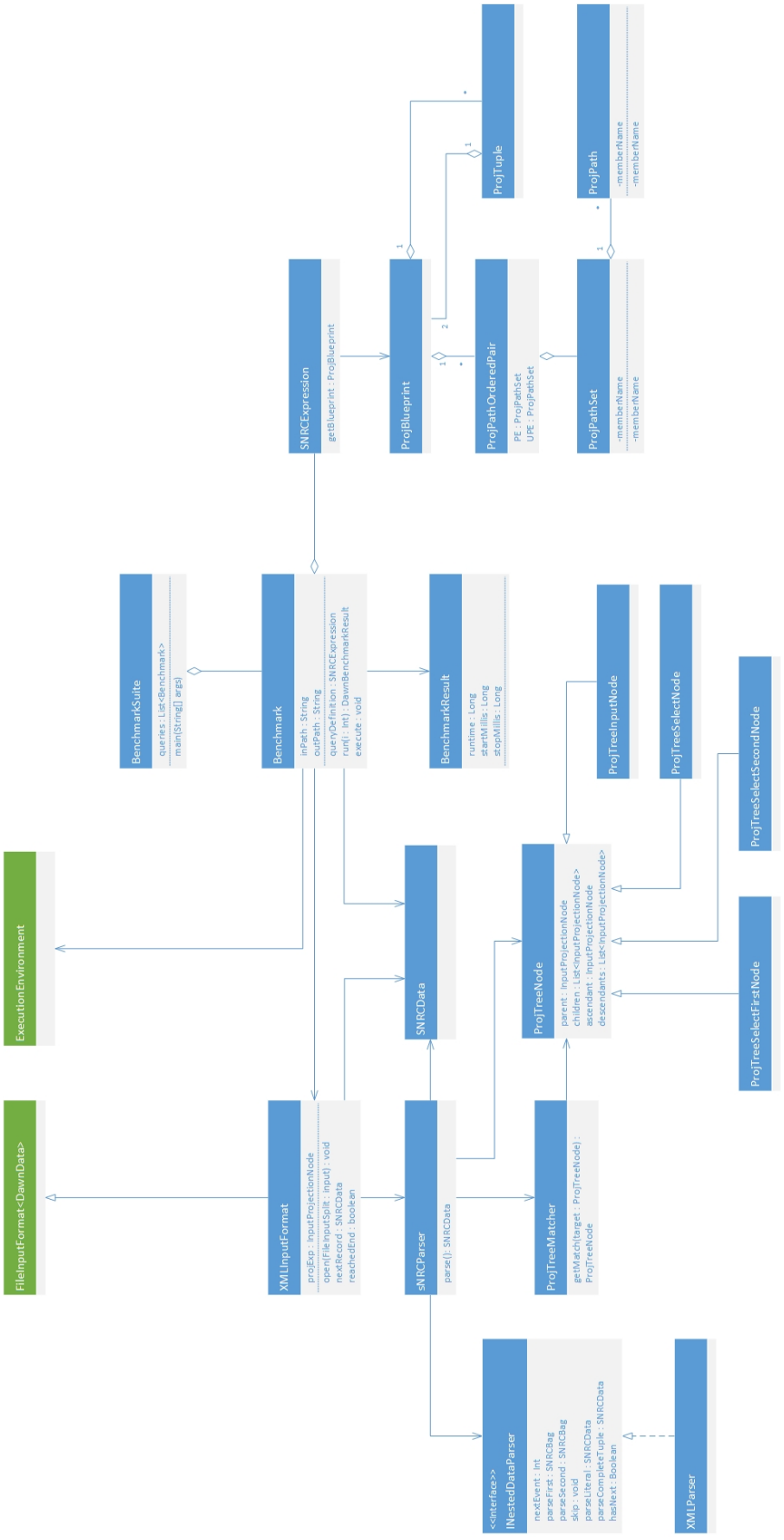

Figure D.3: UML Diagram of core classes of the improved parallel sNRC system

Appendix E

Complete Experiment Results

This appendix shows the complete list of results of the evaluation of the reference and improved parallel sNRC systems. This is the full list of results after pruning and averaging the results to increase reliability as described in Chapter 5. All displayed results are runtimes in milliseconds.

E.1 Local Single Node Experiment Results

E.1.1 Reference System Results

Table E.1: Local Single Node results for the reference system. The first column shows the dataset size. The other columns show the runtime in milliseconds for that specific query.

	XMark 1	Xmark 5	Xmark 6	Xmark 7	Xmark 8	Xmark 9a	Xmark 9b	Xmark 10
1GB	4042	3343	4862	4786	21187	43484	42301	67373
2GB	8337	6784	9972	9733	50245	84872	85751	151911
4GB	17695	15603	20116	20326	91559	159466	161287	288895
8GB	44718	37596	46016	46738	183460	334008	339600	588333

	Xmark 11	Xmark 12	Xmark 13	Xmark 14	Xmark 15	Xmark 16	Xmark 17	Xmark 20
1GB	11643	8615	6082	12468	3102	4389	4616	12226
2GB	24114	19306	11333	24166	6017	8576	8770	22455
4GB	59788	46074	22591	45805	14443	18720	18600	56160
8GB	173551	128133	49248	89094	38750	45973	45338	103397

E.1.2 Improved System Results

Table E.2: Local Single Node results for the improved system. The first column shows the dataset size. The other columns show the runtime in milliseconds for that specific query.

	XMark 1	Xmark 5	Xmark 6	Xmark 7	Xmark 8	Xmark 9a	Xmark 9b	Xmark 10
1GB	4692	4107	3295	17006	14471	18269	33636	67870
2GB	9465	7928	6302	33952	26321	33462	62108	145606
4GB	20390	18137	14832	61584	53829	67846	122786	287615
8GB	46426	42139	40068	117313	109583	142949	243515	573438

	Xmark 11	Xmark 12	Xmark 13	Xmark 14	Xmark 15	Xmark 16	Xmark 17	Xmark 20
1GB	10724	9654	6023	30852	4097	5016	5242	6626
2GB	20831	17972	11964	57167	7710	9009	9534	12070
4GB	42786	37713	21529	101601	17465	19926	19965	25239
8GB	98735	82379	53135	199721	43876	45147	47228	53795

E.2 Cluster Experiment Results

E.2.1 Reference System Results

Table E.3: Cluster results for the reference system. The first column shows the number of nodes and dataset size. The other columns show the runtime in milliseconds for that specific query and configuration.

	XMark 1	Xmark 7	Xmark 8	Xmark 9a	Xmark 9b	Xmark 15
1 Node - 1GB	6835	8923	17946	16636	17104	6719
1 Node - 2GB	13861	16777	32686	29958	28623	13118
1 Node - 4GB	24616	41178	76656	61443	88272	23549
1 Node - 8GB	51430	89719	149222	117384	165534	47961
2 Nodes - 1GB	4742	7210	13595	14007	17224	4349
2 Nodes - 2GB	7378	11601	28985	24864	29884	7319
2 Nodes - 4GB	13406	17653	35542	34740	30653	13716
2 Nodes - 8GB	26295	43750	88442	82340	92195	24478
2 Nodes - 18GB	51395	83078	169541	126964	181264	48426
4 Nodes - 1GB	3381	5998	13055	12738	16105	3052
4 Nodes - 2GB	6112	10550	17327	20811	25442	5372
4 Nodes - 4GB	8180	15334	29222	26834	36885	7597
4 Nodes - 8GB	16428	28734	54597	42598	59144	15024
4 Nodes - 18GB	26171	39273	86007	61716	81434	26510
4 Nodes - 36GB	52651	92128	171340	128514	178411	51885
8 Nodes - 1GB	3911	5914	15243	13249	14825	3002
8 Nodes - 2GB	5468	7870	16123	16697	19675	5452
8 Nodes - 4 GB	6368	9455	18822	19632	22842	5274
8 Nodes - 8 GB	10592	18018	35916	35331	45748	9910
8 Nodes - 18 GB	15018	26100	88156	61706	69081	16666
8 Nodes - 36 GB	29903	57243	120306	100575	107613	27637
8 Nodes - 71 GB	55458	83370	177746	130676	177944	53044
16 Nodes - 1 GB	2782	4429	12549	11460	12673	2921
16 Nodes - 2 GB	4160	9352	16970	15161	18821	5135
16 Nodes- 4 GB	5863	8847	17158	15912	20592	5605
16 Nodes - 8 GB	7777	15100	27200	24286	32119	8711
16 Nodes - 18 GB	9880	17296	36242	36679	42786	9961
16 Nodes - 36 GB	18768	35890	62919	70939	88734	18832
16 Nodes - 71 GB	33424	54625	112265	114826	115224	31015
16 Nodes - 141 GB	66989	104189	202409	144248	205242	59066

Table E.4: Cluster results for the reference system when only reading the input data. The first column shows the number of nodes and dataset size. The other columns show the runtime in milliseconds for that specific query and configuration.

	XMark 1	Xmark 7	Xmark 8	Xmark 9a	Xmark 9b	Xmark 15
1 Node 1	6731	7760	14201	21424	21264	6730
1 Node 2 GB	13235	14647	27941	41649	42103	12175
1 Node 4 GB	25918	27477	52561	78470	79334	25578
1 Node 8 GB	47165	50984	98222	140735	150299	47520
2 Nodes - 1 GB	4987	4962	8787	13848	13684	3654
2 Nodes - 2 GB	7420	9628	14590	22043	20750	6078
2 Nodes - 4 GB	14395	16045	28344	42431	41424	12249
2 Nodes - 8 GB	24158	30760	53461	78541	82280	24650
2 Nodes - 18 GB	49421	55848	102026	150290	149461	47343
4 Nodes - 1 GB	4172	4236	5805	9369	9899	2601
4 Nodes - 2 GB	5205	5940	10571	18734	18396	5241
4 Nodes - 4 GB	7829	8593	14942	22345	22629	7410
4 Nodes - 8 GB	14870	16967	30060	43530	45893	13517
4 Nodes - 18 GB	29366	32672	62387	93165	92181	28446
4 Nodes - 36 GB	50896	59799	107314	160175	161022	51321
8 Nodes - 1 GB	3565	3455	4825	7431	8027	2360
8 Nodes - 2 GB	5882	5043	8828	16908	16510	4426
8 Nodes - 4 GB	5351	6125	9747	15234	15890	4560
8 Nodes - 8 GB	9735	10602	19455	29932	30263	9188
8 Nodes - 18 GB	14413	17093	29856	46050	47461	14851
8 Nodes - 36 GB	28348	35965	60819	96126	96266	26419
8 Nodes - 71 GB	47210	56919	98607	147355	146021	46258
16 Nodes - 1 GB	3101	4808	4529	7536	7735	2318
16 Nodes - 2 GB	4225	5461	6815	11793	13365	3768
16 Nodes - 4 GB	4800	5331	8279	14337	16060	4611
16 Nodes - 8 GB	8568	8840	17203	29478	28709	7811
16 Nodes - 18 GB	9779	11513	18393	30300	31552	8555
16 Nodes - 36 GB	18946	22571	42280	63204	63876	17551
16 Nodes - 71 GB	29009	45042	66156	93867	92595	29241
16 Nodes - 141 GB	66514	67143	134152	159053	163847	64737

E.2.2 Improved System Results

Table E.5: Cluster Results for the Improved system. The first column shows the number of nodes and dataset size. The other columns show the runtime in milliseconds for that specific query and configuration.

	XMark 1	Xmark 7	Xmark 8	Xmark 9a	Xmark 9b	Xmark 15
1 Node - 1GB	7483	8591	75767	59466	60666	6440
1 Node - 2GB	13086	16868	163870	117447	93435	13110
1 Node - 4GB	27239	29513	239098	137480	110573	24687
1 Node - 8GB	48257	48567	237444	241430	216246	46190
2 Nodes - 1GB	4357	4904	29504	26741	25948	3700
2 Nodes - 2GB	7661	8442	56015	48062	47299	6807
2 Nodes - 4GB	14116	17995	172470	122030	121151	13044
2 Nodes - 8GB	25495	30123	312230	204755	180421	23887
2 Nodes - 18GB	49567	49378	318017	226881	234393	47879
4 Nodes - 1GB	3115	3571	16794	15643	15787	2694
4 Nodes - 2GB	5996	6507	47422	38458	38845	5423
4 Nodes - 4GB	8104	9406	57316	45713	46879	7633
4 Nodes - 8GB	14924	19161	169870	123493	130968	13470
4 Nodes - 18GB	26362	29577	243453	173020	171645	26195
4 Nodes - 36GB	54629	62432	547970	389840	373796	48949
8 Nodes - 1GB	3672	4107	16032	15592	17848	2880
8 Nodes - 2GB	4692	5629	26955	28212	28160	4723
8 Nodes - 4 GB	5888	6915	47761	41917	41506	4947
8 Nodes - 8 GB	8769	10705	77807	65350	62008	9862
8 Nodes - 18 GB	14641	21290	185936	134958	133790	14848
8 Nodes - 36 GB	28562	35710	294355	195261	210531	29790
8 Nodes - 71 GB	58407	67898	530046	389219	385334	58118
16 Nodes - 1 GB	2450	3442	13605	13006	11598	2454
16 Nodes - 2 GB	3837	6019	24231	22191	23659	4051
16 Nodes- 4 GB	7528	6003	31192	29525	31333	5236
16 Nodes - 8 GB	7884	9439	75124	56287	58596	9589
16 Nodes - 18 GB	12872	12923	99406	86769	86823	11174
16 Nodes - 36 GB	17394	27052	199082	137588	149391	18436
16 Nodes - 71 GB	31628	41952	293738	191831	192626	29981
16 Nodes - 141 GB	70057	97524	678003	425385	423332	70002

Table E.6: Cluster results for the improved system when only reading the input data. The first column shows the number of nodes and dataset size. The other columns show the runtime in milliseconds for that specific query and configuration.

	XMark 1	Xmark 7	Xmark 8	Xmark 9a	Xmark 9b	Xmark 15
1 Node - 1 GB	9315	13401	16861	24625	44664	9852
1 Node - 2 GB	13974	18075	29110	42985	58736	13787
1 Node - 4 GB	25067	42739	50629	81433	161987	25330
1 Node - 8 GB	48995	57055	96840	149685	192287	47010
2 Nodes - 1 GB	4459	5298	8016	12080	18747	3924
2 Nodes - 2 GB	7814	14567	15444	24324	56356	7120
2 Nodes - 4 GB	13774	22770	26644	42795	87305	12858
2 Nodes - 8 GB	25823	43851	49603	76879	177537	24868
2 Nodes - 18 GB	50111	69535	101404	150758	251645	49724
4 Nodes - 1 GB	5652	5359	5971	9807	18419	2832
4 Nodes - 2 GB	5424	7471	9605	14709	26789	5056
4 Nodes - 4 GB	8618	14946	17342	26630	52369	7866
4 Nodes - 8 GB	15319	21927	30483	47823	74155	15733
4 Nodes - 18 GB	25884	33404	50616	77667	109401	23946
4 Nodes - 36 GB	50521	73277	98059	151046	262492	47824
8 Nodes - 1 GB	5294	4745	5424	9534	16234	2661
8 Nodes - 2 GB	8565	7875	8640	15088	25526	4350
8 Nodes - 4 GB	6532	8944	10253	16480	29598	5495
8 Nodes - 8 GB	9715	17069	19793	31407	57385	9607
8 Nodes - 18 GB	14032	28237	28909	44257	101685	14239
8 Nodes - 36 GB	27515	49949	55859	86551	184481	27426
8 Nodes - 71 GB	52739	101699	99441	164553	356284	56792
16 Nodes - 1 GB	5619	6363	5001	8178	15682	2527
16 Nodes - 2 GB	6849	8819	8853	13310	27492	3909
16 Nodes - 4 GB	6667	8440	9529	15595	30059	5004
16 Nodes - 8 GB	8950	15901	17623	26999	55282	8655
16 Nodes - 18 GB	10408	20815	20683	32217	70340	10444
16 Nodes - 36 GB	18424	30808	36978	53658	105083	16762
16 Nodes - 71 GB	29999	50670	62744	90165	179554	30104
16 Nodes - 141 GB	67142	98450	137025	195959	349166	63314