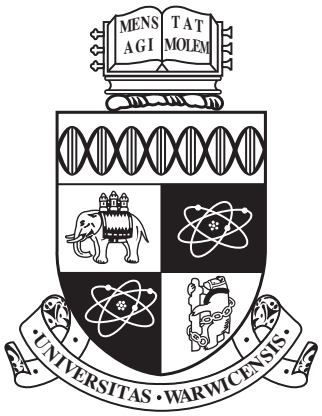


Platforms for Deployment of Scalable On- and Off-line Data Analytics

by

Peter Coetzee

A thesis submitted to The University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

Doctor of Philosophy

Department of Computer Science

The University of Warwick

April 2017

Abstract

The ability to exploit the intelligence concealed in bulk data to generate actionable insights is increasingly providing competitive advantages to businesses, government agencies, and charitable organisations. The burgeoning field of Data Science, and its related applications in the field of Data Analytics, finds broader applicability with each passing year. This expansion of users and applications is matched by an explosion in tools, platforms, and techniques designed to exploit more types of data in larger volumes, with more techniques, and at higher frequencies than ever before.

This diversity in platforms and tools presents a new challenge for organisations aiming to integrate Data Science into their daily operations. Designing an analytic for a particular platform necessarily involves "lock-in" to that specific implementation – there are few opportunities for algorithmic portability. It is increasingly challenging to find engineers with experience in the diverse suite of tools available as well as understanding the precise details of the domain in which they work: the semantics of the data, the nature of queries and analyses to be executed, and the interpretation and presentation of results.

The work presented in this thesis addresses these challenges by introducing a number of techniques to facilitate the creation of analytics for equivalent deployment across a variety of runtime frameworks and capabilities. In the first instance, this capability is demonstrated using the first Domain Specific Language and associated runtime environments to target multiple best-in-class frameworks for data analysis from the streaming and off-line paradigms.

This capability is extended with a new approach to modelling analytics based around a semantically rich type system. An analytic planner using this model is detailed, thus empowering domain experts to build their own scalable analyses, without any specific programming or distributed systems knowledge. This planning technique is used to assemble complex ensembles of hybrid analytics: automatically applying multiple frameworks in a single workflow.

Finally, this thesis demonstrates a novel approach to the speculative construction, compilation, and deployment of analytic jobs based around the observation of user interactions with an analytic planning system.

Acknowledgements

In the first instance, a debt of gratitude goes to my supervisor, Prof. Stephen Jarvis, for affording me the opportunity to both undertake the research described in this thesis and for introducing me to the cast of colleagues and friends who have so wonderfully coloured the years of my Ph. D.

None of this work would have been possible without the support, discussions, and coffee-breaks offered by these colleagues and friends. I am particularly keen to thank James Archbold, Dr. David Beckingsale, Dr. Robert Bird, Richard Bunt, Dr. Adam Chester, Prof. Graham Cormode, James Davis, James Dickson, Dr. Nathan Griffiths, Tim Law, Danielle Lloyd, Andy Mallinson, James Marchant, Dr. John Pennycook, Caroline Player, Stephen Roberts, Faiz Sayyid, Phil Taylor, and Dr. Steven Wright. Thank you all for your open ears and minds, for proof-reading reams of draft writing, and for countless much-needed diversions.

Further, I have greatly enjoyed the professionalism and assistance of a number of support staff in the Department of Computer Science, including Jane Clarke, Richard Cunningham, Sharon Howard, Dr. Christine Leigh, Lynn McLean, Dr. Roger Packwood, Catherine Pillet, and Gill Reeves-Brown. Your tireless efforts go unrecognised all too often; we would all be stranded without you.

Beyond the University of Warwick, particular thanks go to my mentor at IBM Research, Octavian Udrea, as well as the whole ACAM team: Mark Feblowitz, Anton Riabov, and Shirin Sohrabi. You provided me with a fascinating and stimulating summer of research and a wonderful insight into one of the world's great research organisations. I am also grateful to the wider Streams department for all their support to the Speculative MARIO project. My sincere gratitude goes to all of those involved in this research within my sponsors: my supervisors and contacts have been helpful and supportive throughout. They will never be thanked enough for all they do.

Finally, but by no means least, my love and boundless gratitude to my whole family for their unwavering support; in particular to Mum, Dad, Chris, and Amy. For safe haven, last-minute getaways, and words of encouragement and love, a heartfelt thanks I will never be able to put into words, and will never forget.

Declarations

This thesis is submitted to the University of Warwick in support of the author's application for the degree of Doctor of Philosophy. It has been composed by the author and has not been submitted in any previous application for any degree. The work presented (including data generated and data analysis) was carried out by the author except in the following case:

- Execution times for Speculative MARIO (Chapter 6) were collected with the assistance of Dr. Octavian Udrea and Dr. Anton Riabov (IBM Research, T.J. Watson)

Parts of this thesis have been previously published by the author in the following:

[23] P. Coetzee and S. Jarvis. CRUCIBLE: Towards unified secure on- and off-line analytics at scale. In *Proceedings of the 2013 International Workshop on Data-Intensive Scalable Computing Systems*, pages 43–48, Denver, CO, USA, 2013. ACM.
(The work presented in this paper appears in Chapter 4).

[26] P. Coetzee, M. Leeke, and S. Jarvis. Towards unified secure on-and off-line analytics at scale. *Parallel Computing*, 40(10):738–753, 2014.
(The work presented in this paper appears in Chapter 4).

[24] P. Coetzee and S. Jarvis. Goal-based analytic composition for on- and off-line execution at scale. In *Proceedings of IEEE Trustcom/BigDataSE/ISPA, 2015*, volume 2, pages 56–65, Helsinki, Finland, 2015. IEEE.
(The work presented in this paper appears in Chapter 5).

[25] P. Coetzee and S. A. Jarvis. Goal-based composition of scalable hybrid analytics for heterogeneous architectures. *Journal of Parallel and Distributed Computing*, 2016. URL `http://doi.org/10.1016/j.jpdc.2016.11.009`.
(The work presented in this paper appears in Chapter 5).

[27] P. L. Coetzee, A. V. Riabov, and O. Udrea. Methods and systems for improving responsiveness of analytical workflow runtimes, November 2016. US Patent 9,495,137.

(The work on which this patent is based appears in Chapter 6).

Sponsorship and Grants

The research presented in this thesis was made possible by the support of the following benefactors and sources:

- The University of Warwick, United Kingdom:
  Engineering and Physical Sciences Research Council CASE Studentship (2012–2016; K503204)
- IBM T.J. Watson Research, Yorktown Heights:
  Summer Internship (2015)
- EPSRC Capital Equipment Grant:
  "Provision of a Portfolio of Massively Parallel, Data-intensive Analytics Platforms" (2014–2015; K011618)

Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
CDR	Call Data Records
CSV	Comma Separate Values
CURIE	Compact URI
DAG	Directed Acyclic Graph
DFS	Depth First Search
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DSL	Domain Specific Language
ECG	Electrocardiogram
ETL	Extract Transform and Load
FATE	Fault Tolerant Execution
FFT	Fast Fourier Transform
GFS	Google File System
HDFS	Hadoop Distributed File System
HPC	High Performance Computing
HTN	Hierarchical Task Network
HTTP	HyperText Transfer Protocol
IBM	International Business Machines
IDE	Integrated Development Environment
IP	Internet Protocol
IPC	Inter-Process Communication
JAR	Java Archive
JNI	Java Native Interface
JSON	JavaScript Object Notation

JVM	Java Virtual Machine
MAC	Media Access Control
MARIO	Mashup Automation, Runtime Instrumentation, and Orchestration
OLAP	Online Application Processing
OWL	Web Ontology Language
PE	Processing Element
POSIX	Portable Operating System Interface
RAM	Random Access Memory
RDD	Resilient Distributed Dataset
RDF	Resource Description Framework
RDFS	RDF Schema
RPC	Remote Procedure Call
SAMOA	Scalable Advanced Massive Online Analysis
SOAP	Simple Object Access Protocol
SPADE	Stream Processing Application Declarative Engine
SPL	Stream Processing Language
SPLMM	SPL Mixed Mode
SPPL	Streaming Processing Planning Language
SQL	Structured Query Language
SSD	Solid State Disk
TCP	Transmission Control Protocol
UI	User Interface
URL	Uniform Resource Locator
URN	Uniform Resource Name
UUID	Universally Unique Identifier
WSDL	Web Services Description Language
XML	Extensible Markup Language
ZK	ZooKeeper

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

J. Lyons and Co.'s LEO I computer [13] represented an early shift in the application of calculating machines: extracting insights from data faster and more accurately than any human could reasonably achieve. In such early systems, data were typically homogeneous, structured, and predictable in nature; applications were restricted to well specified areas such as stock tracking and payroll. Within a decade, these simple applications gave rise to the notion of *Business Intelligence* [70]; the automated extraction and aggregation of content from the existing documents and metadata available to an organisation, in order to provide actionable insights. In many respects, these goals are an extension of the concept of *Scientific Management* [104] popularised by Taylor at the turn of the 20^{th} century: applying the scientific method (in the form of measurement, analysis, hypothesis generation, and experimentation) to business processes.

For many years Business Intelligence applications focused on structured data in OLAP databases, perhaps integrating data from two disparate sources to summarise and report on business activities, or to provide evidence to support complex decision making. Much of the work during this period was on ETL (Extract, Transform, Load) applications, and reporting front-ends. Some effort was put into data integration, and basic statistical summaries.

It was not until the turn of the 21^{st} century that Thomas Davenport fused his seminal work on enterprise knowledge management [32] with principles of data-driven computation and the science of statistics, formulating the principle of *data exploitation* as a competitive advantage for businesses [31]. In a brief few years, adoption of such analytical techniques rose rapidly, alongside the burgeoning field of *Data Science* [69]. Data Scientists specialise in the application of complex

statistical and machine learning models, extracting valuable information from a sea of data. These skills have found a natural home in a tremendous breadth of application areas, including cybersecurity, manufacturing, customer relationship management, advertising, and digesting data from the Internet of Things. The unique value offered to an organisation by data science is well presented by Hal Varian, chief economist at Google [109];

> “The ability to take that data – to be able to understand it, to process it, to extract value from it, to visualise it, to communicate it – that's going to be a hugely important skill in the next decades, not only at the professional level but even at the educational level for elementary school kids, for high school kids, for college kids. Because now we really do have essentially free and ubiquitous data, so the complementary scarce factor is the ability to understand that data and extract value from it.
>
> I think statisticians are part of it, but it's just a part. You also want to be able to visualise the data, communicate the data, and utilise it effectively. But I do think those skills – of being able to access, understand, and communicate the insights you get from data analysis – are going to be extremely important. Managers need to be able to access and understand the data themselves. ”

The scale of recent adoption of large-scale data analysis platforms and the principles of data science has led to a matching growth in the ecosystem of supporting technologies. This ecosystem was initially launched by the publication of Google's MapReduce [34], but has quickly grown to include novel products and research from the likes of The Apache Foundation, Hortonworks, Cloudera, MapR, IBM, UC Berkeley's AMPLab, and a multitude of others [16, 33, 41, 50, 54, 82, 89, 94, 123]. These supporting technologies, each with their own strengths and capabilities, have elevated the simple process reporting enabled by early business intelligence techniques to a scale, complexity, and utility previously unheard of: all under the umbrella of *Data Analytics*.

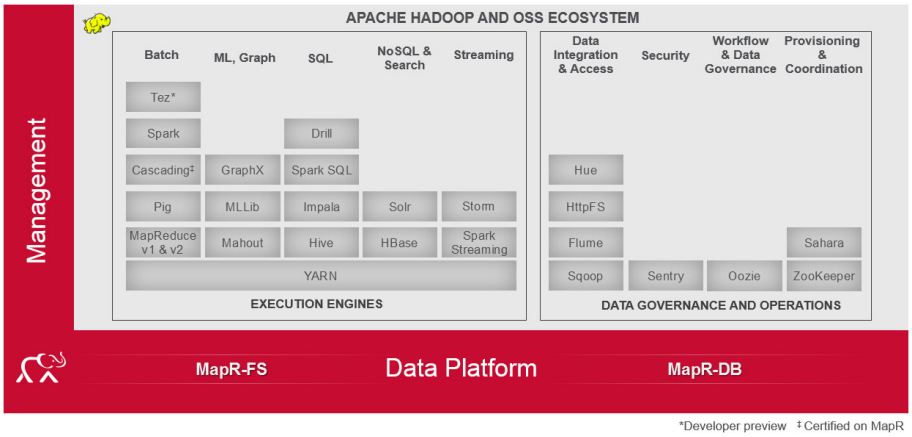


Figure 1.1: MapR Hadoop Technology Stack[1].

1.1 Motivation

This explosion of analytical techniques, programming frameworks, and runtimes has stemmed from a recognition that different classes of problem require different analytical approaches. Selecting the approach to use for a given problem (and completing a successful implementation) is non-trivial, requiring both expertise and experience in an enormous breadth of systems (see Figure 1.1 for an overview of just some of the tools available from a single vendor) as well as a strong theoretical grounding in the computer science principles behind them.

Furthermore, specialists with all of the relevant skills must typically make a vital engineering decision before beginning to craft their analytic: do they wish to use a bulk analytic paradigm which permits enormous historical analyses but may struggle to deliver timely insight? Or should they instead target a streaming runtime, making use of a more challenging programming model but with the ability to achieve continuous or near-real-time insight? Often, such a specialist will select whichever runtime seems the most natural to the problem they are trying to express: should later requirements emerge which result in this choice being sub-optimal, or a need for multiple paradigms arises, the specialist engineer(s) must adopt the burden of maintaining multiple implementations of

[1]Reproduced from `https://www.mapr.com/support/overview`

the same analytic whilst ensuring their analyses are both correct and equivalent.

Chapter 4 aims to lighten this burden by implementing a domain specific language (DSL) which describes analytics at a high level as communicating sequential processes. This DSL targets execution through a common runtime model in multiple streamed and bulk analytic environments, and includes an implementation of automated cell-level security labelling.

This implementation challenge is further compounded by a split in knowledge present in many organisations. Typically, an organisation contains domain experts who understand their data, relevant queries, and business requirements – but lack understanding of programming analytic frameworks, concurrency, and so forth. The software engineers responsible for implementing these requirements represent a separate group of stakeholders altogether. This split in expertise often results in many iterations of development, and sometimes failure of an analytics project altogether [53]. It is challenging to recruit individuals with both sets of skills, and so they either accept the possibility of failure through a traditional iterative model, or they attempt to empower their domain experts to express their analyses themselves – often concealing the power of the underlying analytic framework through high-level abstractions.

Addressing the challenges faced by domain experts in formulating their queries requires a different approach. Given the range of analytic platforms and paradigms already discussed, and an ever-present need for domain experts to deploy analytics which exploit the increasing heterogeneity in their environments rapidly enough to deliver results in time, programming-based solutions are no longer sufficient. Chapter 5 examines the use of an abstract analytic model to enable goal-based planning of analytics, handing control of analytic design and execution to the aforementioned domain experts, and permitting software engineers to concentrate on small, composable analytic components. This thesis investigates the use of this planning technique, in association with platform-specific code generation, to a number of case studies – including a variety of runtime platforms, as well as hybrid analytics that are deployed across multiple

platforms simultaneously on a heterogeneous platform.

A further issue that becomes apparent when enabling domain experts to compose their own analytics is the latency between completion of the design of their analytic and the availability of their results. In some instances, this delay is a result of platform start-up costs (as in MapReduce [84]), while elsewhere it results from the complexity of compiling the analytic (as in InfoSphere Streams). Almost all analytics will ultimately suffer from delays in the actual processing of the results, delivering their first insights sometimes minutes or hours after an analytic is actually launched.

Traditional solutions to this problem have sought to optimise the underlying platform, or design faster implementations of the algorithms powering their analysis. These optimisations are time-consuming and difficult to implement, and are only able to deliver a limited performance improvement. Furthermore, when used in a multi-tenancy analytic environment, the bursty nature of queries often results in poor system utilisation [60, 122].

Chapter 6 demonstrates an alternative approach to reducing this so-called *time to insight*, based around the use of heuristics to speculatively compile and deploy arbitrary analytics, making use of spare capacity in the cluster. A variety of approaches to this speculative execution are discussed, and evaluated using a selection of real-world analytics.

1.2 Methodology

The research described in this thesis has been motivated by the goal of improving the ability of expert users to express their analytics to the underlying framework in a manner that enables them to achieve timely insight. In order to accomplish this, a three-phased approach has been used for each contribution described;

1. Background reading into the problem and research into the current state of the art in the literature, helping to inform;

2. Design, implementation, and iterative improvement of a novel approach to

the problem, which is then used to;

3. Evaluate the resulting solution using exemplar workloads.

The nature of the evaluation phase changes depending on the aim of the contribution; it includes a range of benchmarks, profiling tools, and qualitative analyses of the system in use. The selection and design of these tests has been informed by the author's industrial experience as a practitioner, in addition to private validation with the work's industrial sponsors.

1.3 Thesis Contributions

The research presented in this thesis makes the following contributions:

- We develop the first reported high-level Domain Specific Language (DSL) and suite of runtime environments, adhering to a common runtime model, that provide consistent execution semantics across on- and off-line data, called Crucible. This is the first DSL designed specifically to target the execution of on- and off-line analytics with equal precedence. This DSL permits a single analytic to be run equivalently over multiple data sources: locally, over Accumulo data, and over files in the Hadoop Distributed File System (HDFS). It includes a novel framework for the semi-automated management of cell-level security, applied consistently across runtime environments, enabling the management of data visibility in on- and off-line analysis. We additionally present an evaluation of the performance of Crucible on a set of best-in-class runtime environments, demonstrating framework optimisations that result in an average performance gap of just 14× when compared to a suite of native implementations

- We use a new abstract model of assembly and execution for arbitrary analytics, centred around a semantically rich type system to enable a novel solution for goal-based planning of on- and off-line hybrid analytic applications, requiring little programming ability or prior knowledge of

available analytic components by the user. We demonstrate automatic code generation for the planned analytic across scalable compute architectures, integrating heterogeneous on- and off-line runtime environments, and validate its use through application to four case studies taken from the domains of telecommunications and image analysis. Our results include an exploration of the performance and scalability of the planning engine as well as the resulting analytics in both on- and off-line runtime environments, demonstrating comparable performance with equivalent hand-written alternatives.

- We present the first reported modular, generalised approach to speculative composition, compilation, and execution of data analytics which makes decisions in an on-line fashion, without requiring any *a priori* knowledge of analytical components or configuration. This approach includes a collection of policies which configure its decision-making behaviours, as well as a detailed exploration of real-world deployment considerations for such a system informed by both streaming and batch real-world customer applications. We demonstrate how this approach to speculative execution is used to make successful predictions in these applications about the analytics a user will compose, how this improves response times in both streaming and offline analysis, and include a rigorous evaluation of how the above policies work together to compile, deploy, and in some cases being collecting results before the user completes the specification of their analytic. Within these applications, we show how speculative execution can deliver over 100× improvements in time-to-results by exploiting the spare compute capacity in production environments.

1.4 Thesis Overview

The remainder of the thesis is structured as follows:

Chapter 2 presents an overview of the architectures, concepts, and terminology

currently used in creating scalable data analytics. It discusses the foundational research and development in this field as well as the basic techniques used for distributed computation with, and analysis of, large-scale data.

Chapter 3 details the current state-of-the-art in tools and techniques for composing data analytics for deployment on the scalable architectures discussed in Chapter 2, including a survey of related work in the field. There are a variety of techniques for designing data analytics, which are discussed in detail here, along with the key types of framework in which they run. This chapter also covers high level non-programming based approaches, and discusses their strengths and limitations.

Chapter 4 describes Crucible, a first-in-class framework for the analysis of large-scale datasets that exploits both streaming and batch paradigms in a unified manner. The Crucible framework includes a domain specific language for describing analyses as a set of communicating sequential processes, a common runtime model for analytic execution in multiple streamed and batch environments, and an approach to automating the management of cell-level security labelling that is applied uniformly across runtimes. This chapter shows the applicability of Crucible to a variety of state-of-the-art analytic frameworks, and discusses detailed optimisation considerations for these frameworks.

Chapter 5 proposes a novel semi-automated approach to the composition, planning, and code generation of scalable hybrid analytics, using a semantically rich type system which requires little programming expertise from the user. This approach is the first of its kind to permit domain experts with little or no technical expertise to assemble complex and scalable analytics, for hybrid on- and off-line analytic environments, with no additional requirement for low-level engineering support. This chapter includes an analysis of the performance of the planning engine, and shows that the performance of its generated code is comparable with that of hand-written analytics.

Chapter 6 demonstrates a novel approach to speculatively compiling and deploying analytics using statistics-based heuristics and automated reuse of deployed code, as well as a set of policies to be used within this speculative execution framework and explores deployment considerations arising from a set of real-world customer analytics. This chapter explores how this approach is used to make successful predictions in real-world streaming and batch customer applications about the analytics a user will compose, as well as detailing a rigorous evaluation of how the available policies work together to compile, deploy, and in some cases collect the user's results before they complete their analytic specification.

Chapter 7 concludes the thesis, including a summary of the research contributions and discusses alternative applications and future avenues for ongoing research in this field.

CHAPTER 2

Architectures for Data Analytics

Enabling data scientists and domain experts to analyse their data at scale presents a unique set of challenges to the software engineers responsible for crafting scalable systems for analysis. It is no longer sufficient to simply procure a single large system for analysis of many datasets – the magnitude of the data and complexity of analysis involved often cannot be processed in a reasonable timeframe on a single compute node. As a result, data scientists require a distributed systems approach to the deployment of their analytics: a collection of (often commodity class) compute nodes, which are responsible for the storage and processing of data according to the end users' requirements.

Prior to the dawn of the field of data science, much research went into distributed processing environments. Typically, this was in one of two areas: either Grid Computing or High Performance Computing (HPC). Grid Computing largely focuses on how to schedule collections of tasks to individual compute nodes, with little emphasis on performance or co-ordination across the logical cluster; instead, it favours loosely coupled (possibly geographically distributed) nodes working in concert to accomplish a larger task. HPC, by comparison, is concerned primarily with highly compute intensive simulation problems requiring maximum floating point performance, with a secondary focus on the memory hierarchy, interconnects, and I/O; eking out maximum performance for a limited set of problems through tight integration of the complete hardware and software stack.

By contrast, Cloud Computing typically uses a large number of commodity-class nodes to serve the compute requirements of remote customers. This can be by concealing the complexity of managing the hardware estate through multi-

tenancy virtualisation, or by offering a set of higher-level services which further conceal the compute fabric beneath. As such, much of the research in cloud computing has investigated [1, 29, 35, 49, 56] problems which are applicable to systems for data analysis at scale; scheduling, security, and performance.

Problems which are *data* rather than *compute* intensive reflect novel challenges over and above traditional distributed systems:

Volume: The sheer quantity of data to be analysed, often multiple petabytes (10^{15} bytes) in size, exceeds the reasonable capability of a compute-intensive architecture. Simply storing the raw data, let alone performing complex analysis, can be a challenge.

Velocity: Many of these data sets are not static; the rate of arrival of new data is a challenge for both the ingest pipeline and to maintain the freshness of analytical results.

Variety: Often, extracting insight from data requires the integration of a number of disparate sources of data, often encoded in different formats (semi-structured or structured), or with subtly different semantics on their fields.

Veracity: Every data source has some form of uncertainty, whether resulting from sensor deviation (e.g., in Internet of Things sensor packages) or outright misinformation (e.g., as is often seen on Twitter [17, 86]). Analytics consumers must be careful when using such information to be sensitive to this within their domain, lest predictions are skewed, compound errors inflate, or worse.

As a result, these problems require a different approach – new architectures, new programming models, and new supporting frameworks. Broadly, there are two categories of approaches to this problem: *offline*, or bulk analysis, and *online*, also known as streaming analysis.

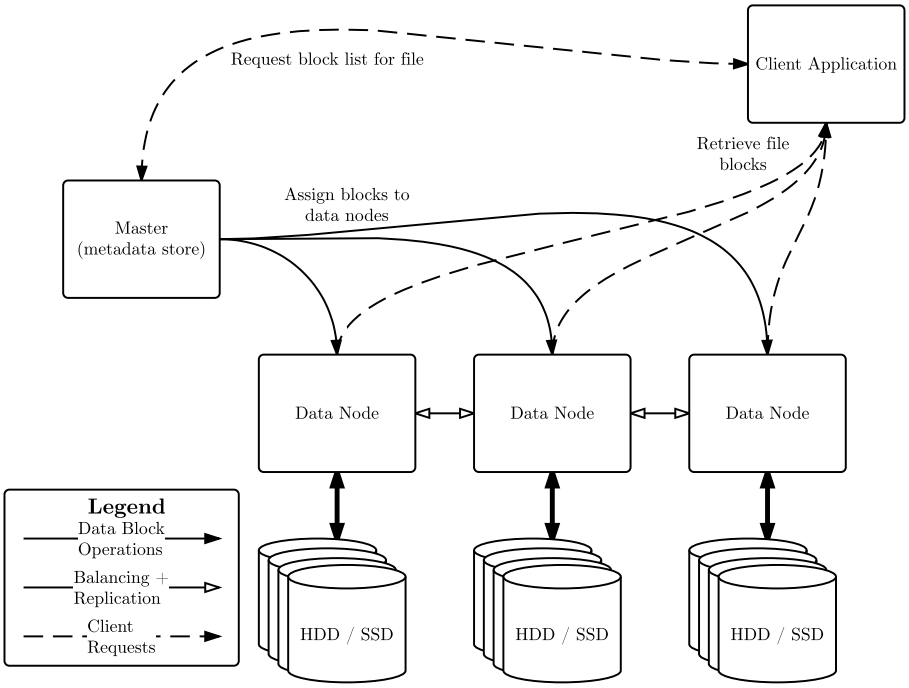


Figure 2.1: Offline Data Warehousing Architecture.

2.1 Offline Data Analytics

A typical bulk analytical system consists of a large number of heterogeneous nodes, each of which contains multiple local hard disks (typically magnetic disk, but more recently systems using SSD or large banks of volatile storage are starting to emerge [59, 117]). A Distributed File System (DFS) is then deployed atop this cluster of nodes, responsible for co-ordinating their behaviour into a single coherent file system. Figure 2.1 illustrates this model; each *Data Node* contains a set of hard disks, and are coordinated by the *Master*. When a *Client Application* wishes to read or write a file, it communicates with the Master to request a *block list* for the file. This block list describes a mapping from file blocks to Data Nodes: the Client then communicates directly with each Data Node to store/retrieve the file block-by-block. Once the file has been stored once, the Master directs the Data Nodes to exchange blocks in order to maintain data redundancy and level out the load across the cluster's disks.

Google's distributed file system, GoogleFS [43] was one early DFS technology of this type, giving rise to the open source Hadoop Distributed File System (HDFS) [97]. As hardware failure is a near-certainty for systems at this scale [48, 96, 120], the Master Node also manages replication of these file blocks, ensuring that duplicate copies of each block are stored in multiple nodes, racks, and even data centres. Crucially, all of this replication complexity, in addition to the awareness of the physical layout of the cluster (racks, network topology, and geographic distribution), is managed by the Master Node and concealed entirely from applications running over the DFS. Note that no data blocks ever pass through the Master Node: it is solely used for metadata operations. A client requests a *block list* for a given file (either for storage or retrieval), and communicates directly with the Data Nodes responsible for holding those blocks. Block-level replication happens directly between Data Nodes on the most local network interconnect available.

Unlike traditional relational databases, once data is stored to a node, it is rarely retrieved by a client: data is not split into readily retrieved records, but rather recorded in enormous flat files – too large for any client to retrieve. Instead, queries or analytical tasks are submitted to the cluster, which will schedule them across as many nodes as are required to complete the jobs in a timely fashion, such that the analysis of a given chunk of data occurs on a node which already holds that data. Results are gathered over the network and aggregated centrally. Such jobs can take anywhere from seconds to hours to execute, depending on their complexity.

The DFS architecture in Figure 2.1 is readily extended to encompass this style of distributed data analysis, as in Figure 2.2. Here, instead of a client application simply retrieving or storing blocks of a file, it submits a job to the *Resource Manager* – the only architectural difference between this and the model in Figure 2.1. Note here that the Master still supports direct access to file block lists, and manages the assignment of file blocks to the Data Nodes. The Resource Manager uses the block metadata to determine where a job should be

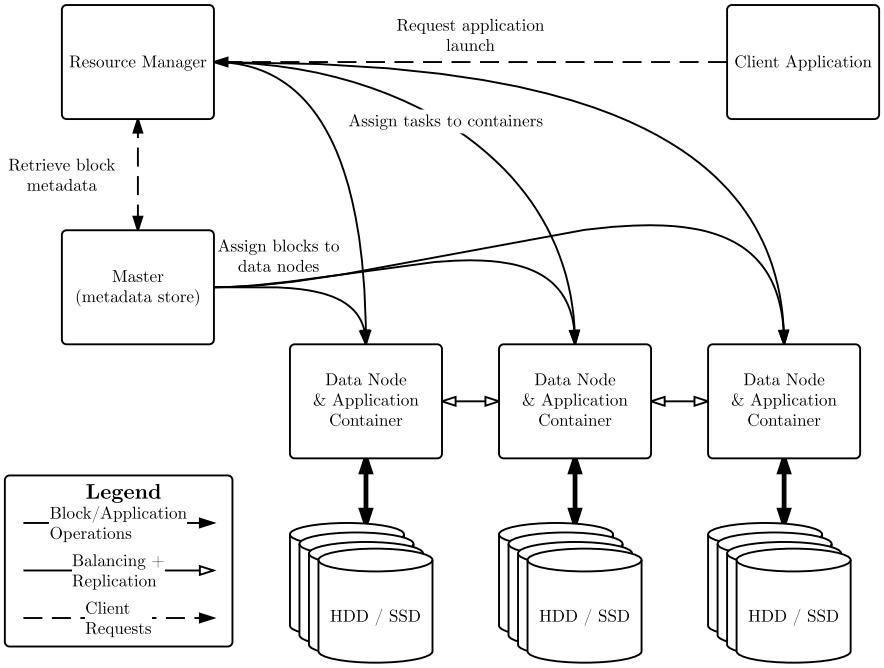


Figure 2.2: Extending the Offline Data Warehouse for Analytics.

executed, taking into account the current load on the cluster. The job request issued by the Client includes a description of the job's inputs and of the analysis to be performed (encoded in a runtime-specified fashion), which the Resource Manager uses alongside the block metadata to determine how to parallelise and schedule the job. This set of scheduling decisions result in a collection of tasks and a partial ordering for them. The Resource Manager then assigns tasks to application containers on the Data Nodes themselves based on the location of the input blocks each task uses.

The processing capability of this kind of cluster depends heavily on the nature of the data and the applications to be run: typically, cluster capabilities are analysed in terms of the ratio of CPU cores to hard disk spindles, of spindles to GB of storage, and of CPU cores to memory for highly iterative workloads (e.g., offline model building for machine learning).

2.1.1 Data Management

A number of projects build data management systems on top of the underlying file-based interface of the DFS. Google originally proposed Bigtable [19] as a key-value store on top of GFS. Apache HBase [6] is a common open source implementation of this principle, built on top of HDFS. Instead of simply splitting a flat file across the DFS, these schema-less stores assign key-value pairs to tables, and then split the data in each table into *shards*; continuous subsets of the key-space, written to the DFS as separate files. Each of these systems add a server process alongside each Data Node, which manages the shard of a table assigned to that server. Different implementations also refer to these shards as *regions* or *tablets*.

A third project (privately developed in parallel with HBase and later made available as open source), Apache Accumulo [5], adds cell-level security, increased fault tolerance (through its FAult Tolerant Execution framework, FATE), and a novel server-side processing paradigm [39] to the existing Bigtable infrastructure[1]. An Accumulo key is split into a number of fields, as illustrated in Figure 2.3. Keys are sorted in lexicographical order, with no constraints on the format of the Column Family, Column Qualifier, or Row ID: these are defined by the application. Column Visibilities specify a boolean cell-level security expression, supporting arbitrary labels and a syntax including AND (&) and OR (|) specifiers, as well as parentheses to override the natural order of these operators. The inclusion of a timestamp for a given Key permits the server to efficiently write a stream of *mutations* to the table, and in a later procedure (called *compaction*) remove mutations which are overwritten by a later change. As the table is sorted by the key, when a client scans the table the Accumulo server may simply skip repeated mutations for a given key, taking the first (most recent) timestamp it encounters.

[1] As Apache Accumulo is used for experiments later in this thesis, further explanations of these Bigtable-based stores refer to the specific details of Accumulo's implementation.

<table>
<tr><td colspan="4">Key</td><td rowspan="2">Value</td></tr>
<tr><td rowspan="2">Row ID</td><td colspan="3">Column</td><td rowspan="2">Timestamp</td></tr>
<tr><td>Family</td><td>Qualifier</td><td>Visibility</td></tr>
</table>

Figure 2.3: Accumulo Key-Value Store field structure.

Accumulo is optimised for random insertion and retrieval of massive amounts of structured data (e.g., tens of trillions of records, multiple petabytes of data, and ingest of 100,000,000 entries per second [61]), as well as large scans across a table. It supports the MapReduce programming model, in addition to a server-side processing paradigm called *iterators*. These iterators may alter the stream of key-value pairs on the tablet server before they are returned to the client, or before mutations are written to disk during compaction. This model is described as particularly valuable for maintaining statistical measures or summations over a dataset.

2.2 Online Data Analytics

An alternative approach to data analysis, known as *online* or *streaming* analysis, sees the data source treated as a potentially infinite stream of values. In this model, it is not feasible to store all of the raw data for later analysis: instead, nodes in the cluster store queries or analytic tasks, executing them over each datum as it arrives. In addition to ameliorating the cost of storing data, this model of analysis need not wait for the entire dataset to be processed before results are delivered. This facilitates near-real-time (or "as soon as possible") analysis for some problem domains.

While it is not necessarily feasible to create a streaming analytic for all types of data or all algorithms, often a new online algorithm can be created to achieve similar results to its offline counterpart. Sometimes, significant research effort is required to uplift an offline analytic onto an online platform – in some cases, trading off the speed of result generation with accuracy of those results. One significant example of this is in SAMOA [33], which aims to enable Machine

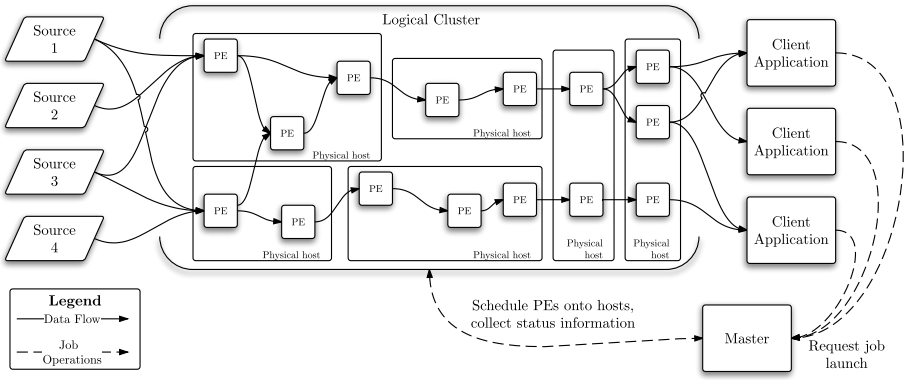
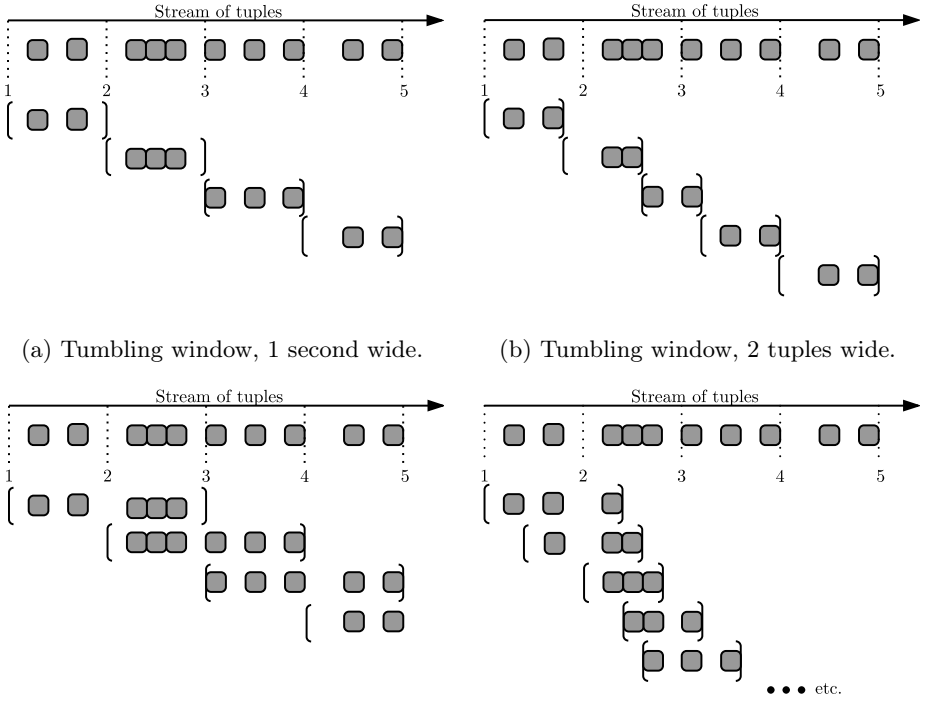


Figure 2.4: Online Analytics Architecture.

Learning on streaming processing platforms to both validate and update models in near-real-time.

Most streaming frameworks express their analytics as workflows, passing tuples of data from one Processing Element (PE) to another. The framework is then responsible for scheduling PEs onto the available hardware, ensuring maximal throughput and minimal latency for each job, as seen in Figure 2.4. Optionally, a framework may also offer placement constraints, such as to ensure two PEs are always/never scheduled onto the same node, or to partition a cluster to reflect variation in classes of hardware or user. Fault tolerance in such an environment typically involves re-scheduling PEs from the failed node onto other available nodes in the cluster, potentially shuffling PEs from other jobs in the process. More advanced fault tolerance can also maintain state about which PEs a tuple has passed through, buffering tuples until they are acknowledged as having been processed by the job. This necessarily adds overhead to each PE (particularly in memory use), but offers a valuable capability in environments where tuple loss is unacceptable.

These PEs each express an atomic operation on a tuple of data. Most frameworks offer a library of standard PEs, as well as a facility for defining arbitrarily complex PEs using either a standard imperative programming language, or a Domain Specific Language (DSL). Some examples of reusable PEs include



(a) Tumbling window, 1 second wide.

(b) Tumbling window, 2 tuples wide.

(c) Sliding window, 2 seconds wide, evicts every 1 second.

(d) Sliding window, 3 tuples wide, evicts after each tuple.

Figure 2.5: Examples of windowing configurations.

facilities for splitting apart and union-joining flows of tuples, for load shedding, reading and writing common data formats, network I/O, or filtering a stream.

More advanced operations such as aggregations and identity joins typically require *windowing*. Streaming windows can be modelled as lists of tuples, with an initially empty state. Tuples are added to the window as they arrive on the stream. Each window is configured with an *emit policy* and an *eviction policy* dictating its behaviour. The emit policy determines when the contents of the window are emitted to the PE on which the window is applied; for example, this could be based on time ("emit every second"), or on count of tuples ("emit every 10 tuples"). The eviction policy dictates when old tuples are removed from the window: again, these are determined by elapsed time since the start of the window or the count of tuples in the window. Windows which evict all tuples after each emit are called *tumbling* windows; examples of these can be seen in

Figures 2.5(a) and 2.5(b). *Sliding* windows, as seen in Figures 2.5(c) and 2.5(d), evict only a part of the window at a time. This technique allows for operations which must otherwise scan an entire dataset to return a partial or approximate result in a timely fashion.

A typical cluster procured for streaming analysis differs from the hardware deployed for offline analytics: systems often lack local storage, and emphasis is placed on minimising the latency of their interconnect and memory hierarchy. The ratio of CPU cores to both memory and network bandwidth is a valuable metric for the processing capability of such a system.

2.3 Hybrid Analytic Architectures

Most of the above technologies facilitate execution of an analytic over a single paradigm, be it online or offline. AT&T Research, as part of their Darkstar project [58], have constructed a hybrid stream data warehouse, DataDepot [45]. This uses online techniques to perform analysis on data as it arrives at the data warehouse, updating the contents of the bulk data store in the process. The trade-off between result latency and accuracy has led Marz et al.to propose the Lambda Architecture [73], in which a streaming platform is used to maintain an approximate set results (e.g., by sampling a random subset of the input values to generate a near-real-time summary of data), and longer-running offline jobs are used to correct the error in this system over time.

A small body of research has examined the use of a single language to target both streaming and offline runtimes. For example, IBM DEDUCE [64] defines code for MapReduce using SPADE (Stream Processing Application Declarative Engine), the programming language used in early versions of InfoSphere Streams. This permits a unified programming model and syntax, but does not offer any direct execution equivalence between a MapReduce PE and a PE written for Streams. Furthermore, SPADE is now deprecated, as it has been replaced by SPL (Stream Processing Language).

2.4 Summary

The particular set of challenges facing data scientists in extracting high value, low volume information from high volume, low value data necessitates a novel set of software approaches to the distributed storage and analysis of their data. A number of common approaches to solving these problems have been explored in this chapter, building on the early research and development presented in Google's GFS [43] and MapReduce [34] as well as IBM's InfoSphere Streams [89].

Although an increasing number of research and engineering organisations are starting to examine the convergence and integration of online and offline analytic techniques, at the time of writing this research is still in its infancy. This is partly a result of the diversity of programming models used in these analytical platforms (examined in further detail in Chapter 3), and the corresponding diversity in runtime models. The research presented in this thesis examines the application of a number of best-in-class platforms for online and offline analysis (Apache Hadoop, Accumulo, and InfoSphere Streams) in *unified* and *hybrid* analytic applications. It will also address the question of how to permit domain experts and data scientists to interact with their datasets and analyses *without* first requiring that they learn these diverse programming and runtime models.

CHAPTER 3

Composition of Data Analytics

The frameworks discussed in Chapter 2 impose a low-level model of runtime execution on analytics. They do not, necessarily, dictate the programming model that must be used to interact with that execution model. There is considerable further work in the literature on techniques and tools to enable various types of user to interact with their data in a natural fashion. In this chapter, we examine some of these approaches, and the trade-offs they incur.

3.1 Programming Frameworks

Since the early implementations of scalable data analytics using a DFS and MapReduce, much work has gone into models for storing and analysing data. While MapReduce makes it much simpler for an engineer to write an analytic to be distributed over a dataset, as described in Section 3.1.1 its expressivity is somewhat constrained. This has led to a number of alternative approaches being proposed, described further in Sections 3.1.2 and 3.1.3.

3.1.1 MapReduce

The MapReduce programming model (see Figure 3.1) begins with a *map* phase, in which data is read from disk, parsed, and turned into key-value pairs: a programmer-supplied Mapper defines how to turn a datum from the input into a set of keys and values. After the map phase, data is *shuffled* by the framework: it is written to disk, sorted by key, and potentially redistributed onto the nodes that will perform the next phase of the computation. The final phase is the *reduce*: a programmer-supplied Reducer is supplied with a key and a list of values for each key produced by the Mapper. It emits a (usually reduced) set of key-value pairs

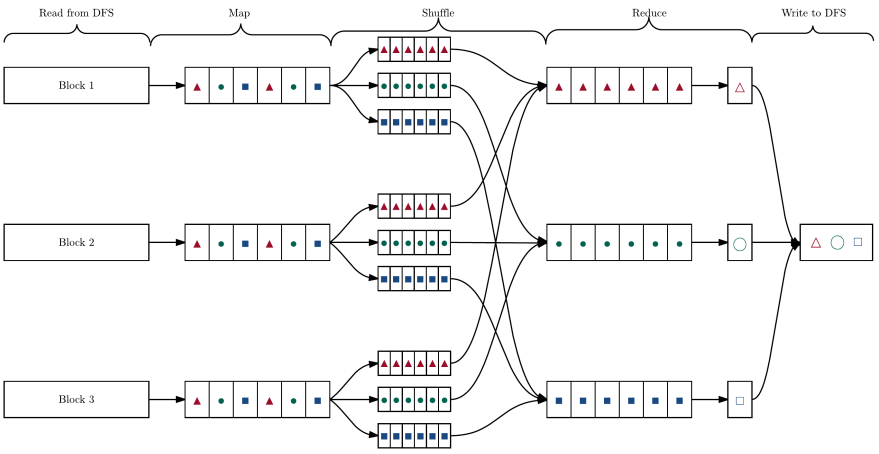


Figure 3.1: Phases of MapReduce Execution.

for the input. Depending on the job configuration, the MapReduce framework may insert an optional *combine* phase between the map and reduce phases. This phase is supplied with a separate Reducer which is both commutative and associative, which the framework can apply to subsets of the output of the map phase (potentially repeatedly, as needed). This optimisation is necessary for jobs generating particularly large value lists for each key, as the full set of values need not be collected and processed in a single Reducer operation. The results of the computation are written back to the DFS.

Apache Pig [82] builds on top of the MapReduce engine, adding a scripting language (Pig Latin) for defining steps to be executed in a workflow. These steps are compiled into a series of Mappers and Reducers, which are executed in a Bulk Synchronous Parallel fashion [20]. Pig Latin adds the ability to specify simple iterative algorithms, filter using arbitrary predicates, as well as performing simple aggregations, grouping, and joins.

While Pig improves the expressivity of MapReduce, there is a significant latency in the startup of jobs on the Hadoop MapReduce framework [84]. As Pig Latin scripts are executed as a series of MapReduce jobs, the impact of this startup latency (as well as the cost of disk reads and writes for each phase of the computation) quickly multiplies.

3.1.2 DAG Runtimes

An alternative use of the MapReduce paradigm is to encode more complex analytical workflows as Directed Acyclic Graphs (DAGs) of low-level components. In such DAG runtimes, nodes represent the analytic components, and edges the dataflows between these components. For example, Cascading [16] uses the Java programming language to define a DAG, with each component assigned to Mappers or Reducers in the MapReduce framework. Of particular value here is the breadth of the standard library which Cascading is distributed with: many classes of problem can leverage this library of standard functions to reduce development time.

Apache Spark [123] offers a notable extension of this model: it facilitates the use of a cluster, which may optionally be running Hadoop, for in-memory analytics. Spark is sensitive to HDFS data locality, but does not depend on HDFS directly. It can operate equally effectively over local storage or a custom storage layer built by the Spark team, Tachyon [67]. Spark implements its own execution framework on top of the base operating system, handling the scheduling of its atomic operations onto available hardware. These atomic operations are inspired by the functional programming paradigm: they include builtins such as `map`, `reduce`, `flatMap`, `filter`, `join`, `take`, `count`, etc.. However, unlike the declarative graph definitions like those employed by Cascading or Apache Storm (see Section 3.1.3), Spark wraps its operations in procedural Java code which results in a lazily generated and evaluated graph of operations in the Spark runtime. This offers the user more flexibility, defining the precise flow of operations at run-time rather than design-time.

Spark operations are represented as transformations of *Resilient Distributed Datasets* (RDDs) [124]. Each Spark operation transforms an RDD of one type to an RDD of another: these RDD definitions are constructed and maintained by the Spark runtime. While an operation may be applied to a single RDD, the execution framework will map that operation to a number of *partitions* (for example, one per block of data in HDFS) executing on any number of nodes in

the cluster, optimised for data locality. In Spark's nomenclature, each operation is executed as a set of *stages*, which are mapped to *workers*. A given stage may have dependencies on the execution of prior stages in the workflow. A stage either executes in the *driver* (the machine hosting the application), or is distributed to a worker node which hosts the referenced partition of an RDD (Spark may move or shuffle RDD partitions during execution of a job to facilitate data parallelism; a Spark application may also request re-partitioning of an RDD as needed).

If at any time in a computation the node hosting a partition of an RDD fails, Spark can recover that subset of the analysis from the definition of the RDD. Spark may additionally speculatively "race" nodes to completion of a given operation on an RDD, if it suspects a node is running slowly.

There are a variety of extensions to the base Spark programming model. GraphX and Bagel [116] are both graph processing engines built on top of the base Spark runtime, based on Google's Pregel [71] research. There are a collection of machine learning algorithms built on Spark ready for reuse in Spark applications, called MLlib [38]. Finally, Spark Streaming [125] is a solution for implementing streaming analytics on the Spark runtime. It does not permit direct portability of offline analytics to an online environment (the RDD definitions differ), but reuses many of the atomic operations and concepts in its programming model.

3.1.3 Streaming Frameworks

In addition to the offering from Apache Spark described above, a number of online analytic frameworks offer their own programming models for describing streaming analysis. These often consist of an API or language to declaratively describe the *topology* of an analytic as a collection of Processing Elements (PEs), and connections between these PEs (tuples traverse these edges during execution). This directed graph representation then uses an imperative language to define the behaviour of each PE. This behaviour can be purely reactive (tuples are produced in response to an input tuple), or in a multi-threaded PE tuples may

be generated asynchronously (e.g., in a time-based window operator).

One of the earliest streaming data analytics platforms to be deployed at scale was IBM's InfoSphere Streams [89] (marketed initially as System S), followed shortly by the open source Apache Storm [7], originally developed by BackType. Others include Yahoo!'s S4 [78] (also in the Apache Incubator), which offers an agent-based programming model rather than the typical workflow-based model. This makes deployment scenarios and performance prediction somewhat more challenging than Storm and Streams, which offer a lower-level abstraction, but permits analytics to be designed in a more loosely coupled fashion. Alternatives include Esper [37], which provides a cross-platform streaming analysis API for Java and .NET, and Microsoft's StreamInsight [3] product, which offers tight integration with Microsoft SQL Server.

Apache Storm is notable for its popularity and short learning curve. It offers a number of models through which to design a Storm topology, the simplest of which behaves as described above: a Java API is used to declare the PEs in a topology ("Bolts" in the Storm model; data sources are referred to as "Spouts") and their connections. Configuration is available to manually define the level of parallelism of a PE, as well as how data should be shuffled and distributed to PEs in these parallel regions. Storm additionally offers guarantees about message processing: when a message is assigned an identifier, Storm tracks this identifier through the topology to ensure it is acknowledged as processed. If it is not processed within a timeout window, PEs which may have processed it are informed that the message has failed, and offered the chance to re-process that message. A second API, called Trident, offers a higher-level abstraction over the Storm topology with an API akin to that used by Apache Spark. It uses primitive operations such as `project`, `join`, `partitionAggregate`, and `each` to describe the transformations that should be applied to a stream. The Trident abstraction is used to generate a standard Storm topology, which is ultimately compiled into Java bytecode and deployed on a Java Virtual Machine (JVM) for execution within the Storm framework.

IBM InfoSphere Streams uses a Domain Specific Language (DSL) to model the processing graph, which it then translates into C++ code (a process called *transpilation*), which is compiled against the Streams libraries for high-performance execution. It additionally offers declarative annotations to describe the parallelism of PEs, as well as partitioning requirements, to ensure host co-location (the given PEs must be grouped on the same host) and ex-location (the given PEs may never be scheduled onto the same host) where needed. In addition to these language-level capabilities, the Streams compiler reasons about a topology in order to perform *fusion* of PEs: combining two streaming tasks into a single PE, such that it is ultimately compiled to one C++ operator, and all message passing between these PEs is performed in-memory, without having to enter the operating system's network stack. Where available, the optimality of fusion can be improved using sample workload data, which the compiler uses to simulate the flow of tuples through the topology. These advanced optimisations result in considerable performance gains over the pure JVM implementations offered by Apache Storm [76].

3.2 SQL and SQL-like Interfaces

Some vendors offer solutions for authoring analytics that do not employ complete programming languages. SQL provides one such vehicle for this; Apache Spark SQL [115] and Cloudera Impala [62] both offer an SQL-style interface onto NoSQL data stores. Apache Hive [107] offers an API to describe the structure of data already stored in HDFS, treating flat files as virtual database tables. It then permits arbitrary queries to be executed against these pseudo-tables, using a derivative of SQL called HiveQL. Tools such as Google's Dremel [74], and the Apache Software Foundation implementation Drill [50], promise SQL-like interactive querying over data stored in a variety of NoSQL data stores, from flat files (CSV, JSON, etc.) to the likes of Bigtable [19] and HBase [6].

The work of Jain et al.[55] aims to standardise the use of SQL for streaming

analysis, but its techniques have not been applied to both on- and off-line analytics. Furthermore, other than through the introduction of User Defined Functions or syntax extensions, there exist entire classes of analytics that cannot be represented in SQL [66].

3.3 Visual Workflow Languages

A variety of approaches allowing less technical users to compose analytics have been reported. Research in this area is often in the context of web-based mashups, however many of the requirements for consuming data at "web scale" are equally applicable to data analytics. Yu et al. [119] provide a rich overview of a number of different approaches, including Yahoo! Pipes [87]; one of the first in a number of recent dataflow-based visual programming paradigms for mashups and analytics. Such solutions require sufficient technical knowledge from their users so that they can navigate, select and compose components of a processing pipeline. Knowledge of a supporting programming language is not required, which removes the challenge of learning programming syntax, but this does not obviate the need for a detailed understanding of the available components, their semantics and their use.

Pipes has inspired a number of extensions and improvements, such as Damia [4], PopFly [68] and Marmite [114]. The work of Daniel et al. [30] aims to simplify the use of tools like Pipes by providing recommendations to a non-expert on how to compose their workflows. Others, such as Google's (discontinued) Mashup Editor [46] take a more technical approach, requiring an in-depth knowledge of XML, JavaScript, and related technologies, but in so doing permit a greater degree of flexibility.

3.4 Automated Planning & Composition

Often, subject-matter domain experts lack the technical skills to make use of the approaches outlined above. As a result, a number of research projects have investigated the automated composition of analytics, using techniques from AI planning. Whitehouse et al. [112] propose a semantic approach to composing queries over streams of sensor data, employing a declarative mechanism to drive a backward-chaining reasoner and solving for possible plans at execution time. Sirin et al. [98] introduce the use of OWL-S [72] for query component descriptions in the SHOP2 [77] planner (a hierarchical task network planner). OWL-S extends the purely syntactic composition of services afforded by WSDL by adding a semantic model of the inputs and outputs to a web service. Another common approach, taken by Pistore et al.in BPEL4WS [85], uses transition systems as a basis for planning. A recurring theme in these approaches is that of composing queries by satisfying the preconditions for executing composable components. The runtime composition approach is flexible, but has implications for performance at scale.

There has been considerable work in the area of web service composition for bioinformatics; BioMOBY [113] specifies a software interface to which services must adhere, then permits a user to perform discovery of a single service based on their available inputs and desired outputs; it does not manage the planning and composition of an entire workflow. Taverna [81] offers a traditional "search" interface (making use of full-text and tag-based search) to locate web services which a user can manually compose in the Taverna interface. This form of manual search and assembly requires considerable user expertise, and an understanding of the art of the possible.

Research in Software Engineering has examined analogous problems to this. Stolee et al. [102] examined the use of semantic models of source code as an indexing strategy to help identify blocks of code that will pass a set of test cases, presenting the user with a collection of existing candidate solutions to

their problem. Such semantic searches have additionally been trialled in web service composition [10, 28]. However, the complexity of the semantic model and inherent uncertainty in retrieval accuracy make assembly of multiple blocks of code somewhat risky – there is a considerable probability that the retrieved code samples are not composable.

These web-services-based systems typically involve considerable user training (whether in the composition interface or in the formal specification of their query language), and at their core aim to answer single questions through service-oriented protocols such as WSDL and SOAP. Often, large-scale data analytic workflows aim instead to analyse significant amounts of data in parallel – an execution model which is closer to that found in high-performance computing simulations than in web mashups. In addition to the complexity of WSDL and SOAP definitions, the services offered must often be written specifically for use with such a system: their implementation depends directly on, e.g., a SOAP implementation. There are many existing libraries of components in the data analytics space which cannot be reasonably re-written to enable integration with a composition system: instead, it is desirable for such a system to interface with the existing APIs of the target runtime directly.

One noteworthy solution to the composition problem is that taken by IBM's research prototype, MARIO [92], which builds on SPPL, the Streaming Processing Planning Language [90, 91]. The authors characterise MARIO as offering *wishful search*, which a user drives by entering a set of goal tags. The MARIO planning engine then aims to construct a sequence of analytical components that will satisfy those goals. Tags correspond to those applied to flows of components within engineer-defined code templates. In practice, due to the tight coupling between the engineer-created *tagsonomy* and the actions available to the end user (components are often manually tagged as compatible), it is rare for MARIO to create a novel or unforeseen solution to a problem.

In addition to being a standalone planner, MARIO is integrated into the IBM Automated Analytics Composer. This solution provides the user interface onto

the MARIO planner, as well as orchestrating the compilation and deployment of the resulting jobs onto the correct runtime framework. It will additionally collect results for presentation to the user using a framework called WebViz [126]. As a result of this orchestration and deployment engineering, MARIO and the IBM Automated Analytics Composer are particularly well-suited for integration into experimental analytic systems, such as for automated data exploration [11, 93] and hypothesis generation [101]. It is in this context which MARIO is used later in this thesis (Chapter 6) to demonstrate and evaluate an approach to speculative compilation and deployment of analytic workflows.

The principle of speculative execution has been widely studied in Computer Science. There is a long history of branch and value prediction in CPU architecture to enable instruction-level parallelism, pipelining, and speculative execution [40, 57, 100]. Exploitation of such fine-grained techniques enable considerable performance improvements in production codes [88, 95] by making sub-millisecond performance gains many times over millions of instructions. More coarse-grained speculative execution is used to hide latency in expensive operations, such as in hard disk controller software [18, 22] or network clients [63, 75, 83]. Speculative execution has additionally been used in data analytics workflows before: Apache Hadoop uses it as a mechanism for mitigating the impact of faults [21, 121] by executing single tasks on multiple nodes when a cluster has spare capacity.

3.5 Summary

Crafting scalable analytics for deployment either on- or off-line requires a mastery of an enormous variety of runtimes and programming models. Some of these are based on bulk synchronous runtimes, while others treat analytics as workflows of communicating sequential processes. Each offers its own advantages, optimisation potential, and has its own degree of suitability for a given problem – few implementations permit portability between these runtimes.

In addition to the differences in programming model, a number of the implementations explored in this chapter are targeted at different levels of user ability. From SQL dialects to point-and-click assembly interfaces, these various interaction models each abstract away the complexity of planning and optimising code in detail, relying instead on advanced code analysis and optimising compilers.

Most of these approaches aim to model the *nature* of an analytic in an abstract form, before compiling it for execution in a specific framework. Few, if any, of these implementations attempt to use this model of an analytic to target more than one runtime. Apache Spark makes some steps towards this with its RDD abstraction, but it requires the use of separate APIs for dealing with streaming data. As discussed in Section 2.3, there is an increasing appetite for combining the low-latency processing capabilities of streaming analytical engines with the bulk analysis capabilities of offline data stores. This thesis aims to address the challenge of programming these diverse systems, as well as bridging the gap between those with the knowledge of *how* to program these systems and those with the knowledge of *what* analysis to perform.

CHAPTER 4

Unified Secure On- and Off-Line Analytics

To derive insight and provide value to organisations, data scientists must make sense of a greater volume and variety of data than ever before. In recent years this challenge has motivated significant advances in data analytics, ranging from streaming analysis engines such as IBM's InfoSphere Streams to an ecosystem of products built on the MapReduce framework.

When data specialists set out to perform analysis they are typically faced with a decision: they can opt to receive continuous insight but limit analytic capabilities to a functional or agent-oriented streaming architecture, or make use of a bulk data paradigm but risk batch analyses taking hours or even days to complete. It is, of course, possible to maintain systems that target streamed and batch paradigms separately, though this is less desirable and more costly than having a single system with the semantics to account for those paradigms in a unified manner. The need to support multiple methodologies presents a further challenge: ensuring analyses are correct and equivalent across platforms. These issues are complicated further by deployment scenarios involving multi-tenant cloud systems or environments with complex access control requirements.

The research described in this chapter seeks to alleviate many of these issues through the development of Crucible, a framework consisting of a domain specific language (DSL) for describing analyses as a set of communicating sequential processes, a common runtime model for analytic execution in multiple streamed and batch environments, and an approach which automates the management of cell-level security labelling uniformly across runtimes. In particular, this chapter demonstrates how Crucible (named after the containers used in chemistry for high-energy reactions) can be used across multiple data sources to perform

highly parallel distributed analyses of data simultaneously in streaming and batch paradigms, efficiently delivering integrated results whilst making best use of existing cloud infrastructure.

The remainder of this chapter is structured as follows: Section 4.1 introduces the Crucible system and describes its abstract execution model; Section 4.2 presents a performance analysis and discussion of the three key Crucible runtimes; Section 4.3 details their associated optimisations. Finally, Section 4.4 summarises this research.

4.1 Crucible System

Crucible builds on the most desirable attributes of existing analytic approaches in order to offer a single framework for developing secure analytics to be deployed at scale on state of the art multi-tenancy on- and off-line data processing platforms. It employs a similar programming model and approach to task parallelism as the likes of InfoSphere Streams, while offering consistent execution semantics across both on- and off-line data.

Software applications written for bulk analysis in a high security environment must maintain annotations on their data, also known as *security labels*. Crucible facilitates this through the inclusion of a semi-automated framework for the management of these labels, and permits the application of them equivalently across data sources and runtimes (as discussed in Section 4.1.3). In order to ease the creation of analytics at scale, Crucible requires support for synchronisation across components deployed on a given runtime (discussed further in Sections 4.1.4 and 4.1.5), to ensure the integrity of shared state. Finally, to realise its aim of easing the creation of scalable analytics, support for a standard library of broadly applicable cross-platform components is important: this is discussed in Section 4.1.6.

In order to facilitate the creation of advanced analytics for on- and off-line distributed execution, the Crucible DSL makes use of a higher level

language abstraction than typical analytic frameworks, such as those discussed in Chapter 3. This enables a degree of portability that is not typically achievable under other schemes; an engineer may write their analytic once, in a concise high-level language, and execute across a variety of paradigms without knowledge of runtime-specific implementation details. In addition, the user is afforded the ability to exploit an array of best-in-class runtime models for the execution of Crucible code.

Furthermore, this approach seeks to free domain specialists from concerns about the portability of an analytic's correctness and security. Each Crucible runtime is responsible for ensuring that analytics are run with equivalent execution semantics, through adherence to Crucible's execution model. This is the foundation on which Crucible's assurances of cross-platform correctness are built. The high level nature of the Crucible language permits the user greater confidence that the analytic they *intend* is the analytic they have *written*. As well as providing assurances regarding functional correctness, automated application of security labelling frees the user from having to ensure they have not violated the security constraints associated with the data they are using.

A risk organisations face when integrating a suite of analytics into their operations is the constantly evolving state-of-the-art in analytic frameworks. Crucible can help to mitigate this risk, as the "porting" of an entire suite of analytics becomes a matter of introducing a new Crucible runtime for the new framework; provided the runtime adheres to Crucible's execution model, portability of correctness is assured.

4.1.1 Crucible DSL

As the vast majority of analytic frameworks are built on the Java Virtual Machine, Crucible must target the JVM in the first instance; support for other languages and interfaces is secondary. By targeting the JVM, Crucible additionally gains the use of the vast library of existing open-source Java code. It would be possible to design Crucible as a set of Java interfaces to the runtimes discussed later in

this chapter, however in our experience this results in extremely verbose code: it is the goal of Crucible to move the expression of an analytic to be as close as possible to the user's intended analysis, with a minimum of "scaffolding". It is therefore important that the design of the DSL facilitates the integration of both JVM primitives and other Java libraries.

Instead of designing the language semantics for a novel language from scratch, Crucible's DSL is built on the XText [36] language framework. Through XText's use of XBase, an embeddable version of the XTend Java Virtual Machine (JVM) language, Crucible avoids the need to implement a new parser and design a Turing-complete language implementation. Crucially, the XBase syntax is not dissimilar to Java (with some higher level primitives and syntactic sugar). Crucible's extension to XBase provides a syntactic framework for modelling Processing Elements (PEs), while the syntax and semantics of standard XBase code are reused for each PE's processing logic.

At a high level, a Crucible analytic (such as in Listing 4.1, a topology of three linearly connected PEs) is structured similarly to a Java code file; it consists of a package declaration for code organisation (line 1), a set of Java/Crucible imports (lines 3-4), and then one or more *process* declarations, each describing a PE (lines 6, 14, 29). In the Crucible DSL, each PE is modelled by a Java class, with a name and an optional superclass. The body of a PE is divided into a set of unordered blocks:

- `config` – Compile-time configuration constants. The initialisation of these may involve an arbitrary expression. These are transpiled to `const` fields in the Java class. (Lines 7, 15, 33);
- `state` – Runtime mutable state; shared globally between instances of this PE. These variables may be declared `local`, in which case their values are stored only locally on instances of the PE. Section 4.1.4 discusses the use of global state in Crucible. These are transpiled as instance variables on the Java class. (Lines 16, 31);

```
1  package eg.counter
2
3  import crucible.lib.pe.FileSource
4  import crucible.lib.pe.FileSink
5
6  process Source extends FileSource {
7     config : {
8       Filename = '/usr/share/dict/words'
9       ReadLines = false // Read chars, not lines
10    }
11    outputs : [FileLine, FileCharacter]
12 }
13
14 process Filter {
15    config : int N = 1500000 // For TopN calculation
16    state : int seen = 0
17    output : Keys
18    input : Source.FileCharacter -> {
19       if ((seen) >= N) {
20          Keys.emit('done' -> true, 'key' -> Character::MIN_VALUE)
21       } else if (Character::isLetter(character)) {
22          seen = seen + 1
23          Keys.emit('key' -> Character::toUpperCase(character),
24                    'done' -> false, 'total' -> seen)
25       }
26    }
27 }
28
29 process CountingWriter extends FileSink {
30    output : Results
31    state : counts = ('A'.charAt(0) .. 'Z'.charAt(0))
32              .toInvertedMap[ new AtomicInteger ]
33    config : Filename = 'counts.txt'
34    input : Filter.Keys -> {
35       if (done) {
36          log.info(counts.toString)
37          Results.emit('total' -> total, 'counts' -> counts as Map,
38                       'tstamp' -> System::currentTimeMillis)
39       }
40       counts.get(key.charValue as int)?.incrementAndGet
41    }
42    input : CountingWriter.Results -> super
43 }
```

Listing 4.1: An Example CRUCIBLE Topology fragment, counting the frequency of characters in the input.

- `output(s)` – Declaration of the named output ports from the `process`. Each output is represented in the transpiled code as an instance variable of the CRUCIBLE library `Output`. (Lines 11, 17);

- `input` – A block which maps the qualified name of an output (in the form `ProcessName.OutputName`) to a block of code to execute upon arrival of a tuple from that port. The keys inferred to be present on the input tuple are present as variables in this code block. In the transpiled Java class, each input is generated as a `receive` method, based on the qualified name of the output to which it subscribes (Lines 18, 34, 42).

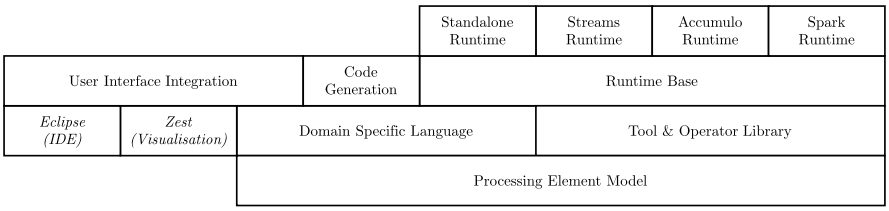


Figure 4.1: Components of the Crucible System. Entries in *italics* are external dependencies.

Crucible transpiles a topology described in the DSL into idiomatic Java based on the Crucible PE Model (the bottom layer of Figure 4.1). This is in contrast to many other JVM languages, such as Scala [79], which directly compile into unreadable bytecode. Compiler support is used to provide syntactic sugar for accessing global shared state and the security labelling mechanism, which are discussed in more detail in Section 4.1.3. The Code Generation component noted in Figure 4.1 is responsible for this transpilation process. It is built on the XText *Java Model Inferrer*, which uses the syntax description (as listed in Appendix A) to generate a parser and abstract syntax tree (AST) generator. This AST is supplied to code implemented in Crucible for gathering tuple types through XBase's type inference, and generating Java classes in accordance with the description above. The result of this process is a series of Java classes which inherit from `PEDefinition` and interact with the internal Crucible Java API, shown in Figure 4.2. These classes turn the various Crucible keywords described above into class fields and methods – effectively generating for the user the verbose Java "scaffolding" which Crucible avoids. For example, the 87-line sample Crucible file in Listing 4.1 is transpiled to 560 lines of Java across four separate classes; to give a sense of the complexity of these classes, the class representing the `Filter` PE consists of 10 fields and 14 `public` methods.

4.1.2 Message Passing

Crucible PEs communicate using message passing; a call to *`Output`*`.emit(...)` causes all subscribers to that output to receive the same message. No guarantees

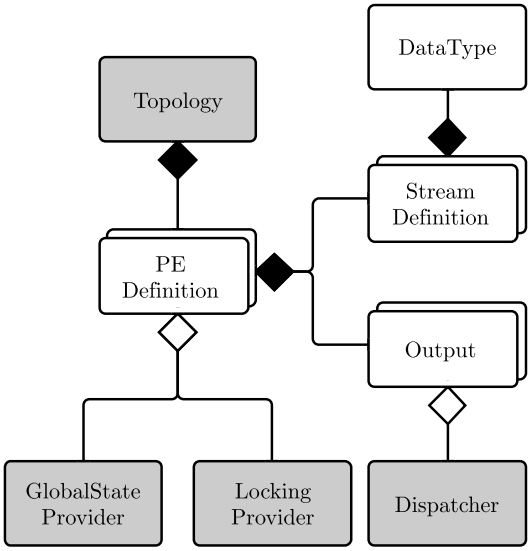


Figure 4.2: Crucible Model Composition diagram, showing the composition of the core model (white) and the runtime injectable components (grey).

are given about the ordering of messages interleaved from different sources. Messages are emitted as a set of key-value pairs, encoded as a single tuple. At compile time Crucible performs type inference on all of the `emit` calls in the topology to generate a correctly typed and named `receive` method interface on each subscriber; the key of an item in the tuple is used as the parameter name on the method. This type inference is based on the semantics of the XBase language; in order to support implicit type declaration (akin to `val` and `var` in Scala, or `var` in C#), XBase supports introspection-based type inference. The algorithm used in Crucible is based on this implementation, with an extension to trace the origin of a particular variable (and thus its type) across not just method calls, but across PE subscriptions in the Crucible execution model. These subscriptions are modelled as method calls for the purposes of this type inference, thus preventing the need to extend any formal verification of the XBase type system.

4.1.3 Security Labelling

Crucible's Security Labelling system is motivated by the need to cope with complex access control requirements in multi-tenancy environments. For example, the provenance or classification of data may need to be tracked on a cell level in order to determine the visibility of a datum for a user. Ensuring that these visibilities are tracked consistently is a challenge that requires a great deal of attention to detail throughout the evolution of an analytic system. Reasoning *a priori* about these labels in a consistent manner is impossible when labels must be determined at runtime based on attributes of the data or the data source: a use case which it is important for Crucible to support.

Crucible's labelling protocol is built on the concept of *cell-level visibility expressions*, similar to those described by Bell and LaPadula [9]. As in the Bell and LaPadula security model (BLM), Crucible uses the Star property to arrange that a given user (in this case, their agent in the form of an analytic) may not `write down` in terms of security level: in Crucible this is implemented by accumulating an expanding set of *labels* for each cell (datum) in the form of a *visibility expression*.

This expression is given as a conjunction of disjunctions across named labels. For example, the expression "$Marketing$ & ($Administrator$ | $Manager$)" requires that a user is authorised to read the $Marketing$ label, as well as either $Administrator$ or $Manager$. If they lack sufficient authorisation, they are not permitted awareness of the existence of that cell. Crucially, Crucible lacks the notion that one label (or level, in the BLM) is inherently "lower" than another; instead, expanding a set of labels to require more authorisations is considered equivalent to requiring a higher security level.

In practise, this principle is implemented by declaring an empty security label for every instance of a variable in the system. This label is accessible to a developer by calling the `label` extension method on an object reference. A user may manually add a conjunction to a label using the `+=` operator – this label may be statically defined or generated at runtime. For example, the label

associated with the `x` variable is expanded through either literal expansion; calling `x.label += "A | B"`, or expansion by label reference; `x.label += y.label`. More formally, consider a function λ which returns the visibility expression for the datum held by a given identifier, and a label expansion function $\epsilon(a, b)$ which re-assigns the visibility expression held by identifier a to include the visibility expression ($lambda$) of b:

$$\begin{aligned} &\lambda(a) : \text{Expression for identifier } a \\ &\epsilon(a, b) : \lambda_1(a) \ = \ \{\lambda_0(a), \ \lambda(b)\} \end{aligned} \tag{4.1}$$

Note that here the syntax λ_{n+1} is used to denote the "next expression" for a given identifier: each time expansion occurs, n is incremented.

Labelling of object-oriented method invocation makes the conservative assumption, in the interests of correctness, that the receiver's state may be mutated by the supplied arguments. Therefore:

$$\begin{gathered} c.foo(d, e, f) \ \Rightarrow \ \begin{cases} \epsilon(c, d) \\ \epsilon(c, e) \\ \epsilon(c, f) \end{cases} \\ \lambda_1(c) \ = \ \{\lambda_0(c), \ \lambda(d), \ \lambda(e), \ \lambda(f)\} \end{gathered} \tag{4.2}$$

Assignment of a value to a non-final Java variable (e.g., as in `g = h`, where `h` is any expression; not to be confused with `g.label = h.label`) requires clearing the contents of its label prior to expansion, as accumulated state is discarded. If the right expression (`h`) contains any identifiers, expansion must occur;

$$g = h \ \Rightarrow \ \begin{cases} \lambda_1(g) = \emptyset \\ \forall(i) \in h, \ identifier(i) \Rightarrow \epsilon(g, i) \\ \lambda_2(g) \ = \ \{\lambda(i_0) \ .. \ \lambda(i_n)\} \end{cases} \tag{4.3}$$

As objects may contain mutable state, when a label for `x` expands to encompass the label for `y`, and `y`'s label is later expanded, `x`'s label must include these

additions:

1	# Ass: y.label == ''	$\lambda_0(y) = \emptyset$
2	x.label += 'foo'	$\lambda_0(x) = \{\text{“}foo\text{”}\}$
3	x.doSomething(y)	$\epsilon(x, y)$
4	y.label += 'bar'	$\lambda_1(y) = \{\text{“}bar\text{”}\}$
5	x.label == 'bar&foo'	$\lambda_2(x) = \{\text{“}bar\text{”}, \text{“}foo\text{”}\}$

(4.4)

As a result of this system of label manipulations, CRUCIBLE is able to ensure that `write up` semantics, as used in the BLM, are applied throughout code written in the DSL. In the process of ensuring this is the case, in particular in the case highlighted in Equation 4.4, the model errs on the side of *over*-protection of information. If desired, a CRUCIBLE development environment can be configured to allow a user direct access to the label `clear` mechanism to reset a label, effectively empowering them (and the PE) as Trusted Subjects in the BLM.

Application of Labelling

This labelling requires support from the CRUCIBLE compiler to transform invocations of the tuple emission method, `emit(Pair<String,?> ... tuple)`, into invocations of the form `emit(Pair<SecurityLabel, Pair<String,?>> ... tuple)`. Note that in the Java type system this has the same type erasure as the original method, allowing the signature replacement to be made transparently. The API seen by the user does not present the requirement for a `SecurityLabel`; the user only expects to provide their `emit` method a varargs input of `Pair<String,?>`. However, during transpilation each of these parameters is wrapped in another `Pair`, this time with generic parameters `SecurityLabel` and `Pair<String,?>`, to hold the generated label and the original parameter element respectively. Therefore, when the user's attempt to invoke `emit(Pair<String,?> ... tuple)` instead calls `emit( Pair<SecurityLabel, Pair<String,?>> ... tuple)`, Java's generated bytecode considers both to be an invocation of the same `emit(Pair[] tuple)` method.

Concordantly, when generating the signature for a `receive` method, the compiler interleaves parameters with their labels: an interface of ⟨`String, Integer`⟩

```
1  process Mean {
2   state : int sum = 0
3   output : RunningAverage
4   input : Filter.Keys -> {
5      if (seen % 100 == 0) {
6         RunningAverage.emit('mean' -> sum / seen)
7         sum = 0
8      }
9      sum = sum + key.charValue as int
10  }
11 }
```

Listing 4.2: CRUCIBLE fragment for calculating the mean of results from Listing 4.1.

```
1  protected int $_sum = 0;
2  protected final SecurityLabel sum$label = new SecurityLabel();
3
4  public void receive$Filter$Keys(
5          final SecurityLabel done$label, final boolean done,
6          final SecurityLabel key$label,  final char key,
7          final SecurityLabel seen$label, final int seen) {
8    if (((seen % 100) == 0)) {
9      int _divide = (this.getSum() / seen);
10     Pair<Object,Object> _mappedTo = Pair.<Object, Object>of(
11         "mean", Integer.valueOf(_divide));
12     this.RunningAverage.emit(Pair.<SecurityLabel, Object>of(
13         new SecurityLabel(sum$label, seen$label), _mappedTo));
14     this.setSum(0);
15     sum$label.$clear();
16   }
17   char _charValue = Character.valueOf(key).charValue();
18   this.setSum(this.getSum() + ((int) _charValue));
19   sum$label.expand(key$label);
20 }
```

Listing 4.3: Fragment of transpiled Java code from Listing 4.2.

instead becomes ⟨`SecurityLabel, String, SecurityLabel, Integer`⟩. Listing 4.2 shows a simple CRUCIBLE fragment, designed to illustrate the automated application of security labelling in practice, while Listing 4.3 shows what this code transpiles to after processing by the CRUCIBLE compiler. Note in this listing how the declared `state` variable, `sum`, is given an instance variable, `$_sum`, and a `SecurityLabel`, `sum$label` (The `$` character is reserved for use in CRUCIBLE identifiers, but permitted in Java source; as such, it is used throughout the generated code to create identifiers which will not risk naming collisions). Furthermore, the labelling rules described above are applied consistently in the code: as `sum` is modified using the value of `key`, the `sum$label` is expanded to encompass the instance of `key$label` passed from the upstream PE (Listing 4.3 Line 19, per Rule 4.2). Similarly, when the value of `sum` is cleared (Listing 4.2 Line 7), `sum$label` is also cleared (Listing 4.3 Line 15), per Rule 4.3. As the assignment is to an absolute value with no associated label, no further expansion

is required at this point.

Thus, if the sequence of tuples in Table 4.1 (each containing a `key`, a `seen` count, and a field to indicate if the stream is `done`, as emitted on lines 20 and 23 of Listing 4.1) were emitted to the `Mean` PE from `Filter.Keys`, the given output would occur on `Mean.RunningAverage`. The table uses [..] notation to denote the security label associated with a value. In this example, keys are labelled as to whether they are a consonant ("C"), or a vowel ("V"). The `seen` total has the label "S", for Seen, added. The addition of these annotations is not shown in Listing 4.1 in the interests of simplicity. The annotations on `key` and `seen` are added manually by the author, using the *`var`*`.label += "X"` syntax, while the `mean` label is accumulated automatically.

`Filter.Keys`			→	`Mean.RunningAverage`
`done`	`key`	`seen`		`mean`
false []	G [C]	98 [S]		0.72 [C & S]
false []	H [C]	99 [S]		1.44 [C & S]
false []	I [V]	100 [S]		0.73 [V & S]
false []	J [C]	101 [S]		1.46 [V & C & S]

Table 4.1: Worked example of security label application

It is important to note that due to Crucible's integration with the JVM, this mechanism should not be considered secure for arbitrary untrusted code; it aims only to assist the security-conscious engineer by making it easier for them to comply with security protocols and audit requirements.

4.1.4 Global Synchronisation & State

Figure 4.2 shows how classes in the model interact; instances of many of these classes (shaded) are injected at runtime using Google Guice [108], permitting the behaviour of the runtime to be integrated with the relevant platform without changes or specialisation in the user code.

Crucible's global synchronisation and shared state components make use of `GlobalStateProvider` and `LockingProvider` implementations which are injected at runtime, based on the configured runtime environment. As discussed

previously, if not marked `local`, `state` variables are globally scoped. Thus, if multiple instances of a PE are run simultaneously, they will share any updates to their state; these changes are made automatically. This mechanism is applied without any guarantees about transactional integrity or serialisability, which in limited circumstances is acceptable, e.g., when sampling for 'a recent value'.

In those circumstances which require serialisability, an `atomic` extension method is provided to take an exclusive reentrant lock on a field, and apply the given closure to the locked state in a form of distributed locking. For example, to take a lock on `myObject` and invoke method `f` on it, one may write `myObject.atomic[ myObject.f() ]`. The behaviour of this is similar to Java's `synchronized` keyword, with two key distinctions: the locking is guaranteed across multiple instances of a PE within a job, even across multiple hosts; and the `atomic` method may be applied to multiple objects by locking a list of variables e.g., `#[x, y, z].atomic[...`*`code block`*`...]`, in which case all locks are acquired before invoking the closure. A consistent ordering of locking and unlocking is applied, as well as a protocol lock, to ensure that interleaved requests across critical regions do not deadlock. This locking protocol obeys the strong strict two-phase commit rule, by expanding all locks before entering the critical region (the code block in the closure), and releasing all locks thereafter.

The runtimes described in Section 4.1.5 make use of two possible synchronisation implementations. The first of these is entirely in-memory and suitable only for single-JVM deployments – the shared state implementation assumes that only a single PE of each type is running at a time. The locking provider employs the Java library's `java.util.concurrent.locks.ReentrantLock` to implement the protocol lock and per-variable locks as described above.

The distributed implementation is slightly more involved, using an Apache ZooKeeper [8] (ZK) quorum for inter-process and inter-host synchronisation. Global locking is based on a per-job ZK path for each named lock. This path is created when the lock is first instantiated – when a client wishes to take the lock, they perform the following sequence:

1. Create a node under the lock path with the `ephemeral` and `sequence` flags set; retain the node sequence number
2. Enumerate the children of the lock node (without setting a watcher)
3. If the lowest sequence number in Step 2 is equal to that of the node created in Step 1, exit the protocol as the lock is held by this client
4. Maintain a watcher on the next-lowest sequence number: when that node is deleted, this client holds the lock and exits the protocol

When a client wishes to release a lock, it simply deletes the relevant node from the ZK quorum. Note that this protocol involves no polling or timeouts – and each deletion only notifies (and thus wakes) the client which owns the lock next.

The ZK-backed Global shared state provider involves a simpler protocol, since it does not offer any protection against race conditions. All state operations for a given PE are performed within a per-job per-class ZK path. Any global fields within the PE are created as nodes within the relevant ZK path on PE instantiation: serialised data is stored in these nodes when a field is updated. Each client maintains a watcher on the nodes associated with its fields – when the watcher is triggered, the client updates the state on the PE with the new value. By loading data from ZK asynchronously, reads of global state variables are efficient (potentially at the cost of unnecessary network traffic, if a variable is updated often but rarely read).

4.1.5 Crucible Runtimes

Crucible offers three key runtime environments for the execution of analytics transpiled from the DSL source. This model, in which an executable compiled to Java source is integrated with existing environments by a runtime shim, is similar to that used in COMPSs [105], the componentised superscalar programming model and runtime system. COMPSs uses the EMOTIVE [44] middleware to enable execution on a variety of runtime environments; in a similar vein,

Crucible uses its own novel library of runtime middleware to convert each framework's native runtime behaviour to integrate with the Crucible message passing model. The key advancement of Crucible's implementations over the likes of COMPSs is that instead of simply converting between APIs with essentially similar runtime models (e.g., for the deployment of virtual machines on a cloud and scheduling of jobs on those instances), each of the Crucible runtimes described below must integrate fundamentally different execution models.

Standalone Processing

The first, and simplest, runtime environment is designed for readily testing a Crucible topology locally, without any need for a distributed infrastructure. This Standalone environment executes a given topology in a single JVM, relying heavily on Java's multithreading capabilities. Simple in-memory locking and global state are provided as the topology will always be located in a single JVM.

Message passing is performed entirely in-memory, using a shared `Dispatcher` instance with a blocking concurrent queue providing synchronisation. This queue, an instance of `java.util.concurrent.LinkedBlockingQueue`, maintains a queue of `java.lang.Runnable` tasks which are passed to a `java.util.concurrent.ExecutorService` for execution in a thread-pool. These tasks are used for registering/de-registering subscribers, and submitting tuples to the list of subscribers for a given PE. Multiple Reader Single Writer semantics for the subscriber mapping are ensured using a series of `java.util.concurrent.Semaphore` mutexes.

On-Line Processing

IBM's InfoSphere Streams provides the platform for Crucible's streaming (on-line) runtime engine. An extension to the Crucible DSL compiler generates a complete SPL (IBM's Streams Processing Language) project from the given topology. This project can be imported directly into InfoSphere Streams Studio; it consists of the required project infrastructure (including toolkit and classpath

dependency references), and a single SPL Main Composite describing the topology. Each SPL PE in Streams is an instance of a Streams-specific wrapper class, `CruciblePE`. This class handles invocation of the `receive$...` tuple methods, dispatch between Streams and the `CruciblePE`s, and tuple serialisation.

There is a one-to-one mapping between tuples emitted in Crucible and tuples emitted in Streams. Each key in a Crucible tuple has a fixed field in a Streams tuple type, and values for all keys are transmitted with each emission. These values are interleaved with their associated security labels, such that the label for a given key immediately precedes it. Tuple values must be converted between Streams and Crucible using a serialisation framework of some kind: this framework is injected into the PE at runtime. Kryo is a runtime library for Java which serialises an arbitrary Java `Serializable` into a buffer of bytes with a similar contract to the built-in Java `ObjectOutputStream` and `ObjectInputStream`, only with superior time and space efficiency [2, 99]. On these strengths, the default serialiser for Crucible is Kryo – but it would be feasible to add, for example, a Protocol Buffers [47] based implementation if interoperability with external systems were required. Security Labels are *not* serialised through Kryo; to facilitate their inspection by debug tooling on the Streams instance, as well as easing their consumption in a non-Crucible analytic workflow, they are encoded as strings. Security Labels are written as `rstring` values, while all others are serialised as an immutable `list<int8>` (representing an array of the serialised bytes).

Each of these `CruciblePE` instances can be scheduled into separate JVMs running on different hosts, according to the behaviour of the Streams deployment manager. Manual editing of the generated SPL, e.g., to use SPLMM (SPL Mixed Mode, using Perl as a preprocessor), can be used to parallelise a single PE across multiple hosts. The injectable global synchronisation primitives discussed in Section 4.1.4 may be used to ensure correctness in this form of data-parallelism.

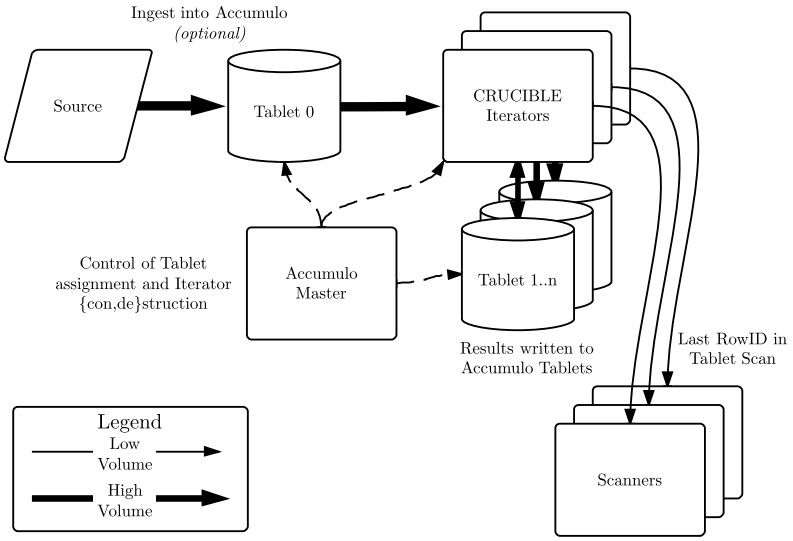


Figure 4.4: Crucible Accumulo Runtime Message Dispatch, demonstrating how Scanners are used to pull data through a collection of custom Iterators to analyse data sharded across Accumulo Tablets.

Off-Line Processing

The mapping from Crucible's execution model to Accumulo for off-line processing is more involved. In order to exploit the data locality and inherent parallelism available in HDFS, while maintaining the event-driven programming model employed in the Crucible DSL, the Accumulo runtime makes use of Accumulo Iterators [39]. An `Iterator` may scan multiple tablets in parallel, and will stream ordered results to the `Scanner` which invoked the iterator. Crucible makes use of this paradigm by spawning a `CrucibleIterator` for each PE in the topology, along with a multithreaded `Scanner` to consume results. Each `CrucibleIterator` may be instantiated and destroyed repeatedly as the scan progresses through the data store.

Each `CrucibleIterator` is assigned to its own table, named after the `UUID` of the Job and the PE to which it refers. Values map onto an Accumulo Key by using a timestamp for the Row ID, the Source PE of a tuple as Column Family, and the emitted item's key as Column Qualifier. Column Visibility is used to

encode Security Labels, making efficient use of Accumulo's native support for cell-level security.

In this way, the `CrucibleIterator` can invoke the correct `receive` method on a PE, by collating all $\langle key, value, label \rangle$ triples of a given `RowID`. By mapping Crucible Security Labels onto Accumulo Visibilities, all message passing data (and final results) are persisted to HDFS with their correct labels: external Accumulo clients may read that state, provided they possess the correct set of authorizations: ensuring cell-level security well beyond the Crucible system boundary.

Crucible's `AccumuloDispatcher` takes tuples emitted by a PE, and writes them to the tables of each subscriber to that stream, for the relevant `Crucible Iterator` to process in parallel. The final component is the multithreaded `Scanner`, which continually consumes from the iterator stack, restarting from the last key scanned when the stack exhausts available input, thus ensuring that the job fully processes all tuples in all tables.

This flow is presented in Figure 4.4: the *Accumulo Master* schedules Crucible's Iterators onto *Tablet Servers* as a result of requests from the client-side *Scanners*. There is one Scanner present for each PE in the system: in Figure 4.4 there are therefore three PEs shown – there could be a many-to-many mapping of PEs to Tablets, as Accumulo distributes data for each PE's table across the available Tablet Servers. Note here that only the final results are returned to the client-side Scanners: all intermediate data is written across the Accumulo cluster's internal network.

4.1.6 Standard Library

The last Crucible component is the standard library. This includes the components necessary for the operation of the aforementioned runtimes, along with a set of base PE implementations to simplify the creation of Crucible topologies. These provide examples of data ingest from a variety of sources, such as from the APIs of Flickr and Twitter, along with primitives to read and write file data.

An XPath PE is valuable for extracting data from XML. This library additionally includes operators for bloom filters and serialisation to/from common data formats such as JSON.

This library is implemented in standard Java, and no special infrastructure is required to extend it. It is intended that users of Crucible may extend this library with custom PEs, or publish their own, simply by writing Java which conforms to a given interface, and making it available on the classpath. For single-use Java operators this may additionally be done within the analytic's Crucible IDE project – the compiler will load and integrate the operator automatically.

4.2 Crucible Runtime Performance

Crucible's *functional* correctness has been validated using a suite of JUnit unit tests, integration testing of analytics against known-good results, and through user acceptance testing. However, beyond this functional correctness, it is necessary to validate the performance of the various Crucible runtimes. This initial analysis of Crucible's performance presents results from the pre-optimisation codebase, with a focus on comparing the scaling behaviour of each Crucible runtime against a functionally equivalent native implementation.

4.2.1 Experimental Setup

In order to analyse the scaling behaviour and accurately compare the performance of Crucible against native implementations, a cross-platform benchmark is required. Most data-intensive benchmark suites in the literature [29, 42, 52, 111] focus on a single type of runtime: OLAP queries, streaming analysis, MapReduce, etc. As a result, this thesis uses its own simple benchmark design, counting the frequency of letter occurrence in a dictionary (akin to Listing 4.1), limited to the top N results, where N is the configured problem size. This design has been selected to maximise the impact of the software design on the wall-time of

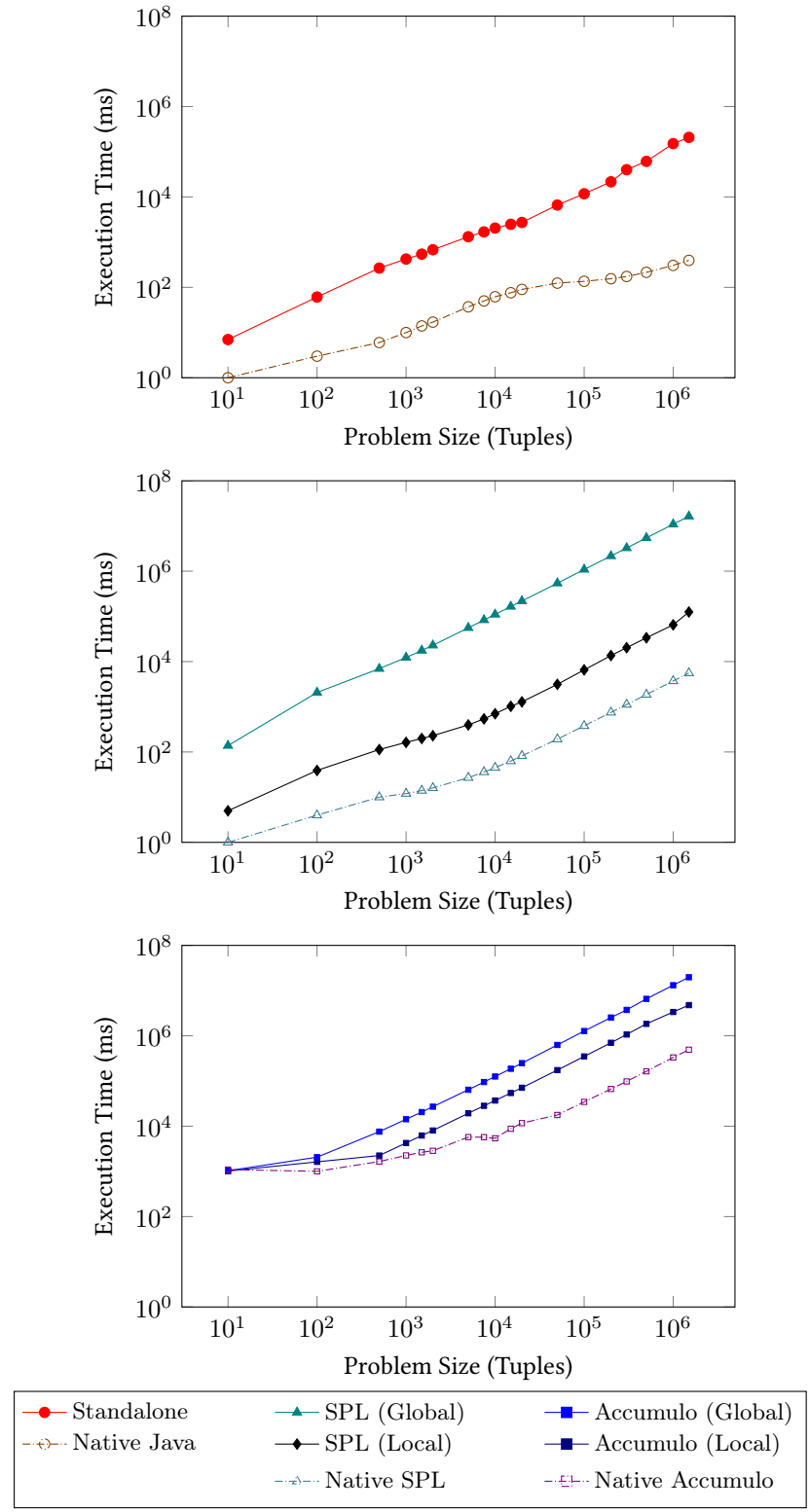


Figure 4.5: Scalability comparison of Crucible Runtimes against hand-written Native Implementations.

the analysis: a more compute-intensive benchmark would be better suited to testing the performance of the underlying hardware architecture. As it stands, this benchmark should spend most of its execution time in the various runtimes under test, highlighting their performance characteristics as much as possible – as shown in the subsequent analysis (Section 4.2.2), this benchmarking task sufficiently demonstrates the performance gap between the Crucible framework and native implementations. There is a key distinction between Crucible and the native implementations here; as the native environments lack support for security labelling, only the Crucible runtimes track the per-cell security labels.

These results were collected on a small development cluster, consisting of three Tablet Servers, one Master, and three Streams nodes. Each node hosts two dual-core 3.0 GHz Intel Xeon 5160 CPUs, 8 GB RAM, and 2×1GbE interfaces.

4.2.2 Analysis

The three graphs in Figure 4.5 shows the results of this testing across the Standalone, Streams SPL, and Accumulo runtimes respectively. It is clear from these results that the Crucible runtimes, in the main, scale proportionally to their native equivalents. There is a noticeable performance gap for each of the runtimes in this basic implementation of Crucible, demonstrating the need for further optimisation to enhance the per-tuple processing delay in all Crucible runtimes (to be discussed in Section 4.3).

The "Global" and "Local" data series are worth noting, as they highlight the performance difference between Global (ZooKeeper-based) and Local (in-memory) shared state providers (see Section 4.1.4, and the discussion at the end of Section 4.1.5). Some analytic jobs do not require all of the features of Crucible at all times, and thus it is valuable to be able to disable performance-hampering features such as these.

While comparing the absolute performance of Crucible and the native implementations, it is important to consider the engineering implications of the approach in Crucible. Removing much of the "scaffolding" of other solutions

has enhanced the expressivity of the Crucible DSL to the point where the above benchmark was implemented in ∼40 lines of the Crucible DSL, as opposed to ∼260 for the three native implementations. Furthermore, the Crucible implementation can be executed across multiple runtimes, whereas the native implementations are each specific to either on- or off-line environments. In our experience, the two to three days taken to write and debug the suite of native analytics was reduced to under a day with Crucible.

4.3 Crucible Runtime Optimisation

Section 4.1.5 described the implementation of the three core Crucible runtimes. Section 4.2 demonstrated near linear scaling of their performance over growing input sizes. However, it also showed that they lacked sufficiently strong performance when compared to hand-written implementations. A series of significant enhancements and optimisations have been implemented in each of these runtimes, in order to improve time-to-solution performance. The experimental results described in Sections 4.3.1– 4.3.3 were collected on the same specification of system described for the original Crucible experimental results, using the same benchmark – no code optimisations have been applied to the benchmark itself. These results are, therefore, directly comparable.

4.3.1 Standalone Processing

The standalone runtime provides an ideal test environment for general optimisations to the Crucible framework, due to its lack of complex inter-process communication (IPC). Figure 4.6 shows the proportion of the runtime spent in various functional sections of the code for a variety of standalone runtime models discussed in further detail below. Figure 4.6 (and later 4.9) were generated using the YourKit instrumenting Java profiler [118] on the maximum problem size. The profiler was used to find method hotspots, and measure the call tree time (the total time spent executing code in the method body, and the cost of its

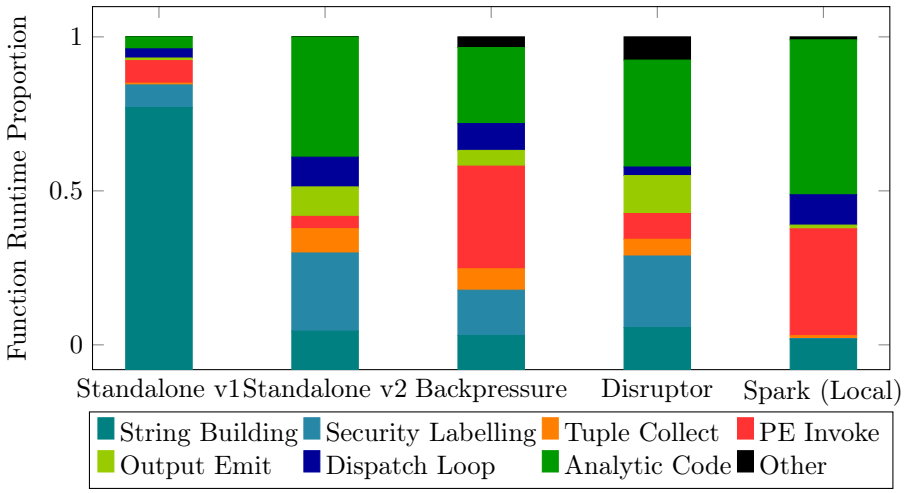


Figure 4.6: Function runtime breakdown across Standalone Dispatchers.

method calls, recursively) from those hotspots: these are the functions broken out in the figures. The profiler was configured to omit the analytic's warm-up time, in order to illustrate the proportional function runtimes for an analytic during the bulk of its execution. It is important to note that these data do not capture the proportion of time spent blocked or context switching, and thus are not sufficient on their own to compare the *absolute* performance of the runtimes.

The original version of the standalone runtime (*Standalone v1*) spends over 75% of its wall time building strings, either naïvely for logging purposes or for analytic output. The *Standalone v2* entry avoids building descriptions of data structures and components for logging if the log message is not going to be emitted. In addition to this, the profiles in Figure 4.6 show that approximately 7.5% of the runtime is spent examining PE configuration and building data structures to invoke the topology's PEs. The *Standalone v2* entry addresses this by introducing new code generation to enforce a set of guarantees to the Crucible runtime that permit accurate compile-time reasoning about the ordering of keys in tuples. As a result, tuple keys (parameters) on the `emit` and `receive` interfaces of a given PE are equivalently ordered when the code is generated, which allows the runtime to avoid reordering and validating tuples when invoking a PE. These changes result in this entry spending a significantly

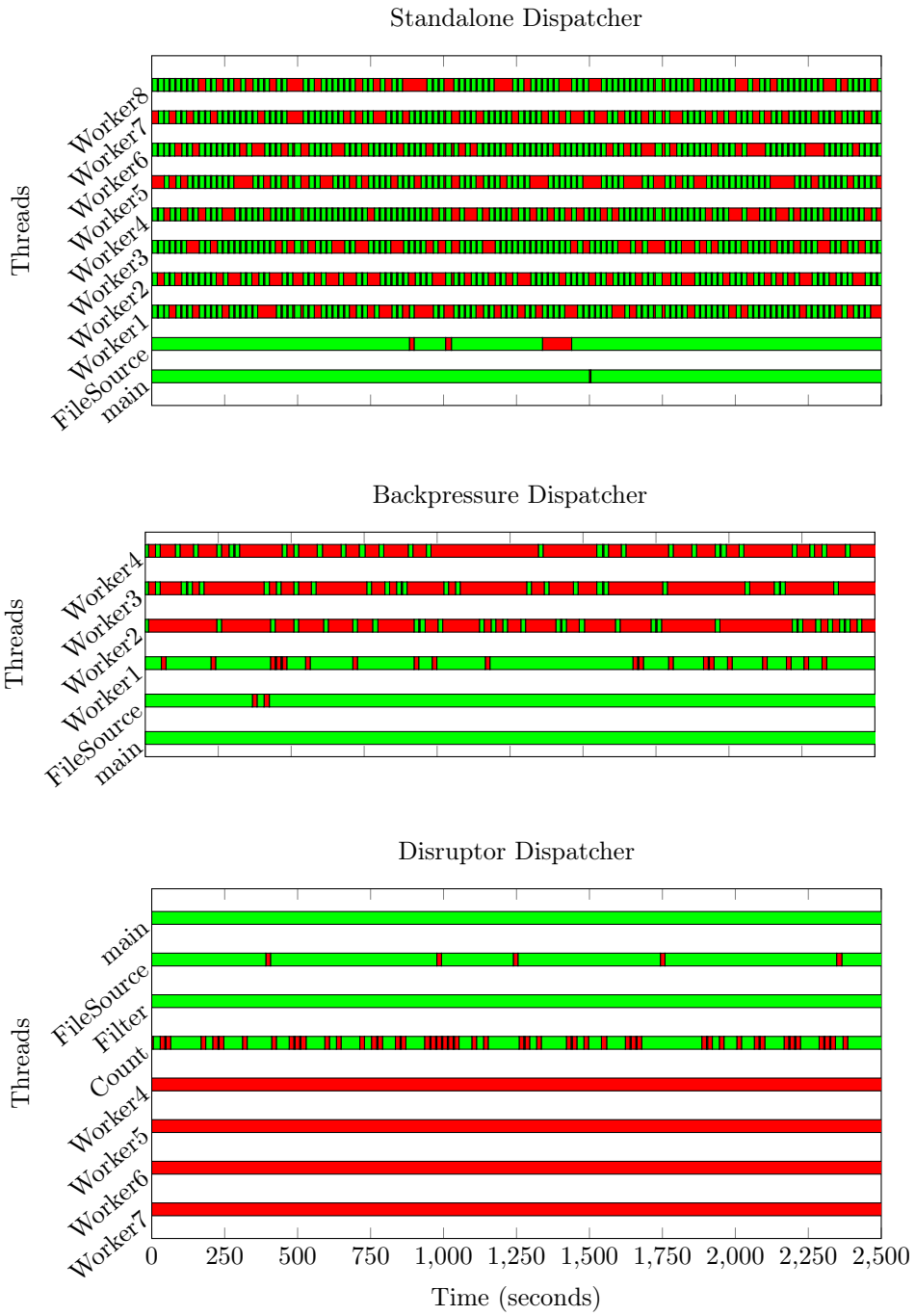


Figure 4.7: Thread utilisation in the Standalone, Backpressure, and Disruptor Dispatchers respectively. **Green** blocks represent the periods when the thread was in the *Running* state; **Red** blocks represent periods the thread was not running.

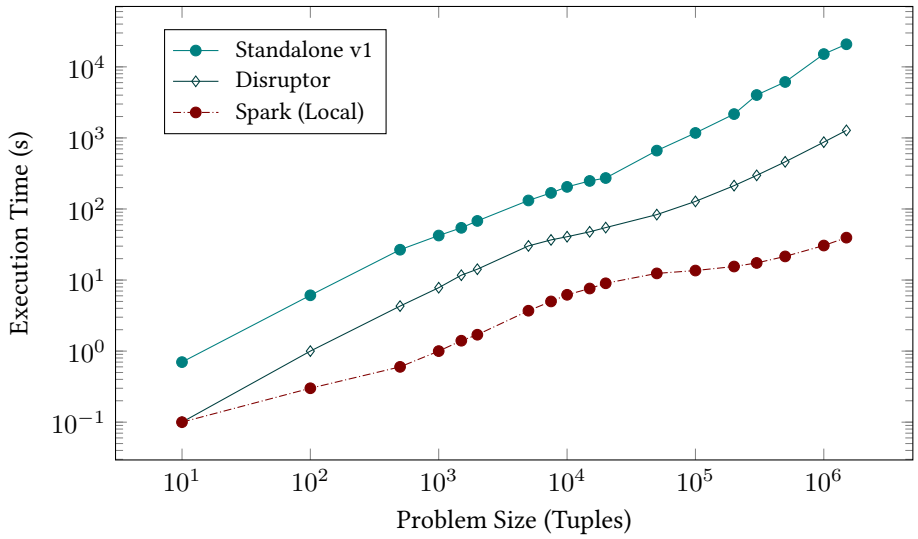


Figure 4.8: Scalability Comparison of Crucible Standalone Runtimes and Native Implementations.

higher proportion of its time ($\sim 40\%$ rather than $\sim 4\%$) performing the actual analysis.

With a better optimised Standalone environment, the importance of thread utilisation becomes increasingly apparent. Figure 4.7 has been derived from the same YourKit Profiler traces as Figures 4.6 and 4.9; it gives a description of when threads in each dispatcher are actually executing (they may be scheduled off the CPU by a lack of work to be done, or being switched out of the `Runnable` state for other reasons, such as being blocked waiting on I/O or a lock). Each line corresponds to a worker thread – where possible, these have been labelled with the work done by that thread, if a single task was consistently scheduled onto it. The top chart of this figure shows that the set of workers in the Standalone v2 dispatcher spend a significant amount of time switching in and out of a runnable state; none of the Worker threads (responsible for receiving submitted tuples and invoking the relevant PE) show full CPU utilisation. Two further experiments were conducted based on these results. The first of these, illustrated in the middle chart of Figure 4.7, introduced the use of backpressure [103] to slow down PEs that were producing tuples faster than downstream PEs could consume

them (thus reducing the probability of resource starvation).

In this arrangement, the queues which form the message passing buffers between PEs are given a fixed upper bound in size (512 elements). When a PE generates more than this many elements without any being processed downstream, a semaphore controlling access to the queue is exhausted and blocks awaiting a permit. Permits are released into the semaphore when elements are removed from the end of the queue; in this way, if PEs downstream are unable to process tuples at the rate they are being produced, the upstream PEs are blocked from executing, permitting the downstream PEs more scheduled CPU time to "catch up". This shows much better thread utilisation, but does not make adequate use of the multi-core architecture on which it runs.

The final, and best performing, standalone Dispatcher implementation makes use of the LMAX Disruptor [106], detailed in the bottom chart of Figure 4.7. The LMAX Disruptor is a lock-free (avoiding the cost of managing locks, even through the relatively efficient CAS (compare-and-swap) mechanism used in modern locking protocols) thread-safe implementation of a ring-buffer, which maintains two pointers: one for the current write-position of the buffer, and one for the last "committed" entry – slots in the buffer are "claimed" by a thread before being written to, and "committed" when the write is complete. The Disruptor is designed specifically for high-throughput producer/consumer operations, and as such is ideally suited to use in a Dispatcher. It maintains its own thread pool for consumers (downstream PEs) – as a result of its pointer arrangement, messages (in Crucible's use, tuples) are automatically batched.

The Disruptor can either be configured to overwrite elements in the ring-buffer when it is full (which would result in lost messages, in what is typically referred to as *load shedding*), or to reject requests to write if the buffer is full: in which case the calling thread may block until the buffer has space. The latter semantics are used in Crucible, thus extending the notion of backpressure discussed above into the Disruptor Dispatcher. It is noteworthy that the Disruptor Dispatcher scheduled each PE to its own thread consistently, and significantly

reduced the amount of context switching by permitting the threads to run truly concurrently whenever there was data available. The superior thread utilisation of the Disruptor Dispatcher is borne out in the runtime results of Figure 4.8, demonstrating a speedup of over 16× of a Disruptor-based runtime model over the original Standalone model. Whereas the original model suffered a performance penalty of 526× over the native implementation, the Disruptor model is only 32× slower, with no code changes to the analytic itself.

There is no significant difference in the actual analytical code executed in the Crucible implementation than the hand-written implementation; this relatively simple analytical task exposes the costs of running the Crucible framework (maintaining threads, dispatch mechanisms, etc.). The single-threaded hand-written implementation suffers none of these costs – it simply executes the core analytic. Comparing the "Analytic Code" segment of the Disruptor chart in Figure 4.6 to the absolute runtime of the Native Java implementation in Figure 4.5 bears this out; at the maximum problem size, Native Java took 395 ms to execute, whereas the Disruptor's Analytic Code executed in 347ms. The difference between these is due to the startup time of the Native Java implementation, which is not accounted for in the Disruptor runtime.

4.3.2 On-Line Processing

Figure 4.9 details how the framework optimisations already described have impacted the breakdown of function runtime in Streams `CruciblePE`s. In order to better understand the costs involved in the existing Crucible SPL execution model, an SPL topology was instrumented to measure the latency introduced by tuple I/O. The Native SPL results show the latency in passing a message into or out of a PE written entirely in SPL. JNI (the Java Native Interface) does not involve any Crucible code; it simply measures the latency introduced by causing tuples to be passed from the Streams SPL interface into the Streams Java interface. This is a necessary precondition for execution of Crucible's Java target code. The final results include the SPL and JNI latencies, as well

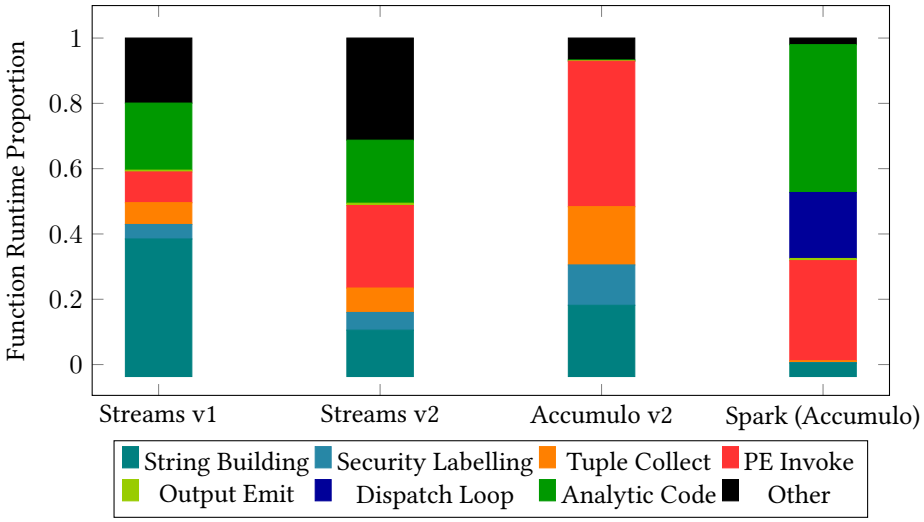


Figure 4.9: Function runtime breakdown across On- and Off-line Dispatchers.

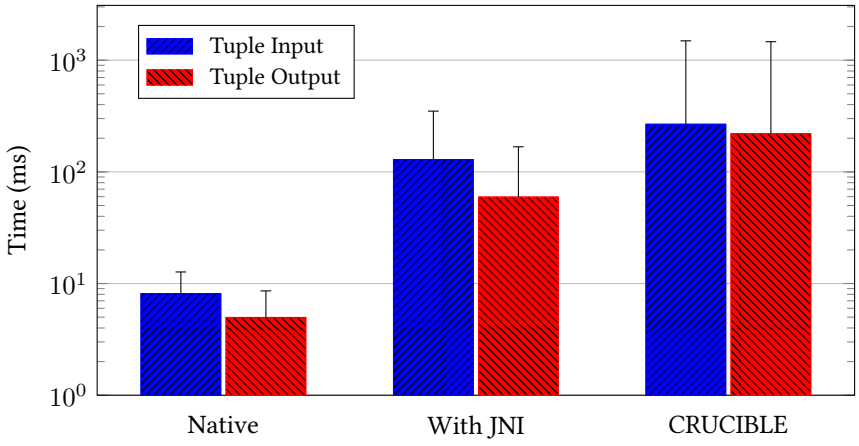


Figure 4.10: SPL Tuple I/O Instrumentation.

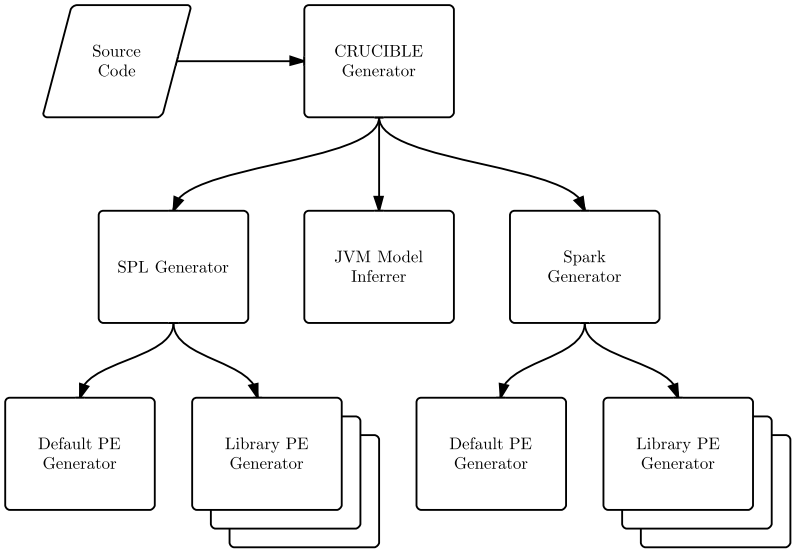


Figure 4.11: Crucible Code Generation Hierarchy.

as those introduced by transcoding and converting data types between Streams Java tuples and tuples in Crucible, as well as execution of the PE invocation logic (but no PE logic). These results are presented in Figure 4.10; the JNI interface is responsible for a considerable proportion of the latency in invoking a Crucible PE.

In order to minimise the impact of Streams' JNI latency, it is necessary to improve Crucible's generation of the SPL target to make better use of native SPL operators. For example, many of the Crucible library functions exist natively within SPL (e.g., file sources and sinks, or timed "beacon" emitters) as higher performance variants of the Java code. By introducing pluggable code generators, Crucible enables library developers to override the default generation engine and create *runtime specific* variants of a given PE.

This pluggable code generation system offers each generator for a pair of ⟨PE class, Runtime environment⟩ the opportunity to generate code for a given instance of a PE by simply being loaded on the classpath for the IDE. The generators are sought using their annotation: `@Generate(target=SomePE.class,`

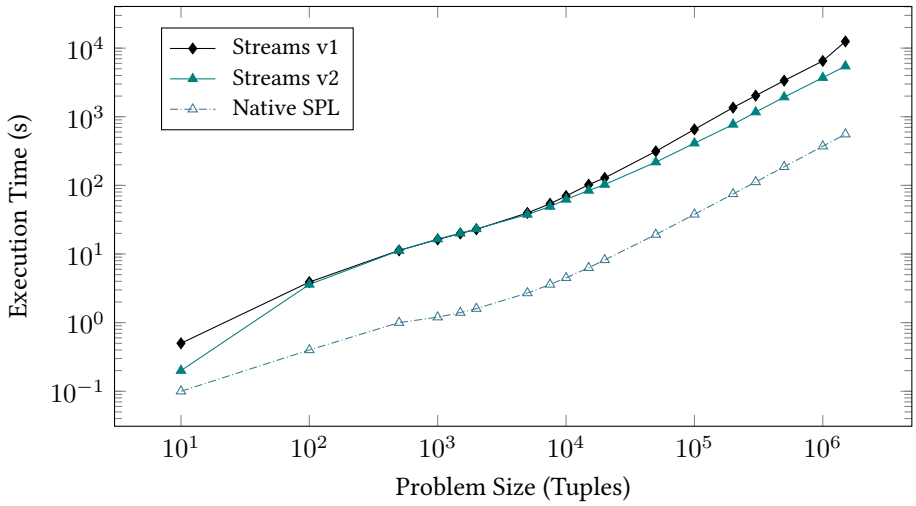


Figure 4.12: Scalability Comparison of Crucible Online Runtimes and Native Implementations.

`type="SPL")` indicates that the annotated class is a generator for `SomePE` in the `SPL` (Streams) runtime. The integration of the various code generation components in the new architecture is highlighted in Figure 4.11. The existing Crucible Java code generation is performed by the same JVM Model Inferrer as before; the extended architecture simply adds the pluggable generators on the bottom layer. For example, Listing 4.4 shows the original output of the Crucible SPL generator; note the application of a `CruciblePE` instance to each PE in the SPL graph. Listing 4.5 shows the same analytic compiled under the new code generation mechanism. The type signatures are identical, but native code is instead generated to support the file source and sink PEs from the Crucible topology without altering the semantics of the tuple processing.

Figure 4.12 illustrates the 2.3× speedup that more advanced SPL generation has allowed on the existing Crucible benchmark – the performance of Crucible transpiled for InfoSphere Streams, once 22× slower than a native implementation, is now under 10× slower. The nature of these improvements is such that they can offer even greater speedups as the topology becomes more complex, making use of more SPL library functions.

```
composite Process {
    type
        FilterCount__Results__Type = tuple<rstring counts__label,
            list<int8> counts, rstring total__label, list<int8> total,
            rstring tstamp__label, list<int8> tstamp>;
        Source__FileLine__Type = tuple<rstring done__label,
            list<int8> done, rstring line__label, list<int8> line>;
        Source__FileCharacter__Type = tuple<rstring character__label,
            list<int8> character, rstring done__label, list<int8> done>;
    graph
        (stream<Source__FileLine__Type> Source__FileLine;
         stream<Source__FileCharacter__Type> Source__FileCharacter
        ) = CruciblePE() {
            param
                peClass       : 'freq.Source';
                configModule : 'freq.ProcessConfigurationModule';
        }

        (stream<FilterCount__Results__Type> FilterCount__Results) =
            CruciblePE(Source__FileCharacter) {
            param
                peClass       : 'freq.FilterCount';
                configModule : 'freq.ProcessConfigurationModule';
        }

        () as Write = CruciblePE(FilterCount__Results) {
            param
                peClass       : 'freq.Write';
                configModule : 'freq.ProcessConfigurationModule';
        }
}
```

Listing 4.4: Crucible Streams v1 SPL Generation.

```
composite Process {
    type
        // Repeated types omitted for brevity
    graph
        stream<blob value> Source__File__Source = FileSource() {
            param
                file       : '/usr/share/dict/words';
                format     : block;
                blockSize : 1u;
        }

        stream<Source__FileCharacter__Type> Source__FileCharacter =
            Functor(Source__File__Source) {
            output Source__FileCharacter :
                character__label = '',
                character = (ustring)convertFromBlob(value),
                done__label = '',
                done = false;
        }

        (stream<FilterCount__Results__Type> FilterCount__Results) =
            CruciblePE(Source__FileCharacter) {
            param
                peClass       : 'freq.FilterCount';
                configModule : 'freq.ProcessConfigurationModule';
        }

        () as Write = FileSink(Count__Results) {
            param
               file    : '/nfs/tmp/count.txt';
               append : true;
               format : txt;
        }
}
```

Listing 4.5: Crucible Streams v2 SPL Generation.

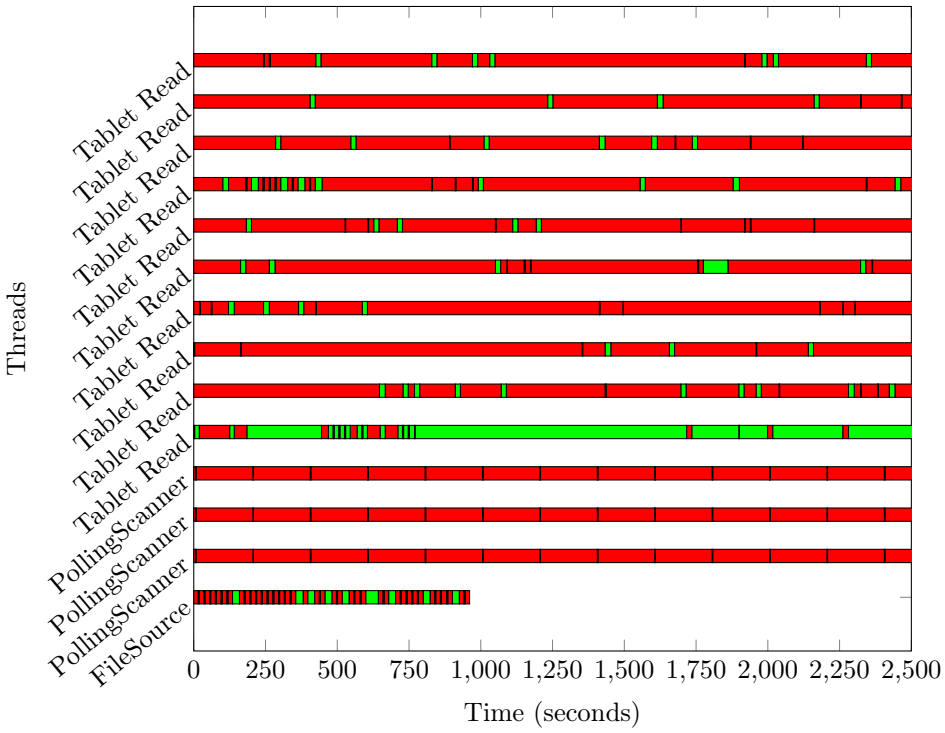


Figure 4.13: Thread utilisation in the Accumulo (v2) Dispatcher.

4.3.3 Off-Line Processing

Figure 4.13 reveals problems with interfacing Accumulo with the original runtime model: even after the optimisations described in Section 4.1.5, the large number of Tablet Read-Ahead threads show drastic under-utilisation for the workload. These threads are an Accumulo optimisation, which predict the data to will be read next in a table scan, and buffer that data in memory. As data is read, it is passed through the Iterator stack configured on that table and that connection (see Figure 4.4 for an overview of how the PollingScanner instances interact with Accumulo's tablet servers; these Tablet Read threads host the custom Crucible Iterators described in the figure). There is no client control over how these threads are spawned and scheduled – a single datum can be read many times by these threads; if an Iterator in the stack performs significant computation, it will slow down the whole read-ahead thread. Furthermore, the process of swapping an Accumulo Iterator out of a read-ahead thread is such that it forces

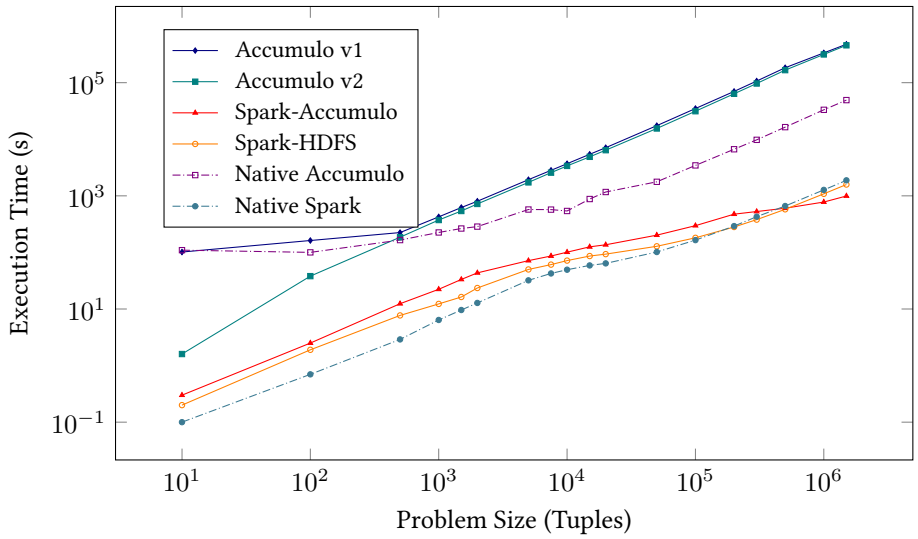


Figure 4.14: Scalability Comparison of Crucible Offline Runtimes and Native Implementations.

a rebuild of the iterator's state when it is swapped back in. The impact of this can be clearly seen the *Accumulo v2* column of Figure 4.9; a vanishingly small proportion of time is spent processing the actual analytic, with the vast majority being spent constructing and configuring Iterators (the "PE Invoke" segment).

These results reveal that the Accumulo Iterator model is incompatible with performing heavy computation and message passing using the Accumulo table interfaces at scale. Instead, these optimisations make use of Apache Spark for execution of Crucible analytics over data in either Accumulo or native HDFS. In support of this, we add a new `DataSource` PE which is closely integrated with the Spark Code Generator. A Crucible `DataSource` is an abstraction of the concept of a source PE, identified by a URN, with a fixed set of outputs per tuple. The precise code used to retrieve tuples from the source are determined by the runtime that is loaded; it may be an Accumulo table, a file in HDFS, or a streaming source, e.g., a network socket. This abstraction can be seen as analogous to the protocol segment of a URI (e.g., `http://` or `hdfs://`), only with a greater degree of flexibility in the parsing of the URN and its transformation into a data source PE specification.

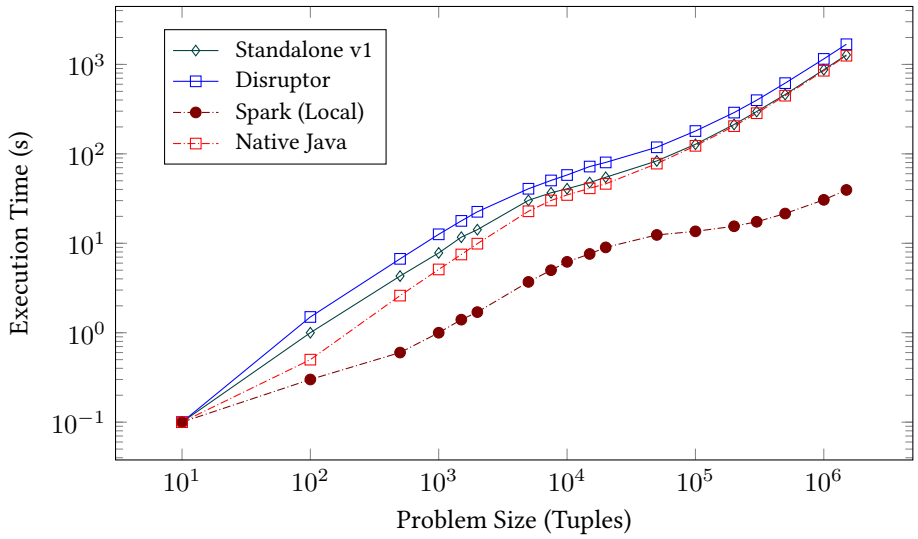


Figure 4.15: Scalability Comparison of Crucible Standalone Runtimes and Native Implementations, including Apache Spark (Local Mode).

The Crucible Spark runtime schedules a graph of `map` operations, starting with the relevant `DataSource`, applying the directed graph of Crucible PEs to the full dataset. The precise scheduling of these operations is determined and optimised by the Spark runtime engine. Each `map` operation emits an RDD (Resilient Distributed Dataset; an abstraction for a collection of data managed by Spark) of pairs ⟨`Output Name, Tuple Data`⟩, which is split along the key to create the relevant source of tuples for the next stage(s) of the analytic.

Figure 4.9 demonstrates how superior the function breakdown is for the Accumulo-Spark runtime model compared to the original Accumulo Iterator interface, and Figure 4.14 shows the significant performance enhancement in terms of time-to-solution that this offers; over 480× from the original Accumulo Iterator model to running Spark over Accumulo. Furthermore, the Crucible Spark runtime with the `DataSource` abstraction for the first time enables the processing of arbitrary Hadoop files and text files in an equivalent and scalable fashion using Crucible.

At higher scales, the performance of Crucible's Spark-HDFS environment converges on that of the native implementation. In practice, the higher-level

Spark builtins used in this native implementation come with a small performance penalty at scale. This somewhat surprising result bears out the idea that while layers of abstraction increase expressivity, they always come with a performance cost: Crucible uses only low-level Spark primitives, and the cost of *its* abstraction is similar to that of the high-level Spark primitives at scale. Performing bulk analysis through the use of Accumulo Iterators with Crucible was approximately 10× slower than the equivalent native implementation; with Spark on HDFS files, this is now almost 1.2× faster than the native implementation used.

This Spark-based approach has the added advantage of providing an alternative execution paradigm for Standalone mode (the functional runtime breakdown for this mode is detailed in Figure 4.6, and the relative performance in Figure 4.15), as Spark may be run over in-memory datasets without the backing of a Hadoop RDD.

4.4 Summary

This chapter has detailed the development and optimisation of Crucible, a framework consisting of a DSL for describing analyses as a set of communicating sequential processes, a common runtime model for analytic execution in multiple streamed and batch environments, and an approach to automating the management of cell-level security labelling that is applied uniformly across runtimes. This research has served to validate the approach that Crucible takes in transpiling a DSL for execution in a unified manner across a range on- and off-line runtime environments, as well as forming an investigation into techniques for deployment across these architectures. The work has demonstrated the application of analysis written in Crucible to data sources including HDFS files, Accumulo tables, and traditional flat files. The results presented demonstrate that the selection of runtime model for execution of Crucible topologies is critical; making a difference of up to 480×. The net result of these optimisations is a suite of best-in-class runtime models with equivalent

execution semantics and a 14× performance penalty over the equivalent native hand-written implementations.

This research has demonstrated a number of valuable capabilities as a result of being based on a DSL; the tight integration of key language features, particularly security labelling and atomic operations, enables the implementation of sample analytics in one sixth the amount of code as the native implementations – a substantial improvement in engineering time, cost, and risk.

Contributions such as the cell-level security labelling framework may be applied to other DSL-based frameworks, such as InfoSphere Streams; it may be possible to plug a labelling system into another JVM language's compiler (e.g., Scala or Jython), however this approach has not been tested here. Frameworks which do not use a DSL, such as Spark or COMPSs, may be able to make use of some of the middleware design concepts in this chapter. For example, typically when a framework such as Spark wants to target a new runtime mode of operation it implements a new runtime framework for that mode of operation (such as with the introduction of Spark Streaming). Instead, a Crucible-style middleware approach could allow analytics to be executed on existing scalable runtimes, thereby making use of existing work in the scalability and fault-tolerance of these systems without having to reimplement such concepts from scratch.

CHAPTER 5

Composition of Hybrid Analytics for Heterogeneous Architectures

Large organisations rely on the craft of both systems engineers and domain experts to create specialist analytics which provide actionable business intelligence. In many cases their knowledge is complementary; the engineer has knowledge of concurrency, parallel architectures and engineering scalable systems, and the domain expert understands detailed semantics of their data and appropriate queries on that data.

Recruiting individuals with both sets of knowledge is challenging, particularly in a growing market, so organisations are typically left with two options: (i) They make use of traditional (often iterative) development models, in which engineers elucidate requirements from stakeholders, develop a solution to meet those requirements, and then seek approval from the stakeholders; or (ii) Engineers empower domain experts by offering high-level abstract interfaces to their execution environments, thus concealing the difficulty (and often the potential for high performance and scalability) of developing a hand-tuned analytic.

Consider the Flickr[1] analytic depicted in Figure 5.1. Each component of the analysis is represented by a box, with arrows indicating the flow of data from one component to another. There are many runtime environments in which the components of this analytic could be deployed, depending on the wider system context. If user data is being crawled, for example, a streaming (on-line) analytic engine such as IBM InfoSphere Streams might be employed for subset **A**, while person data in subset **B** might reside in an HDFS (Hadoop Distributed File System) data store. Each of these runtime environments specify their own

[1] `http://www.flickr.com/`, a photo sharing website

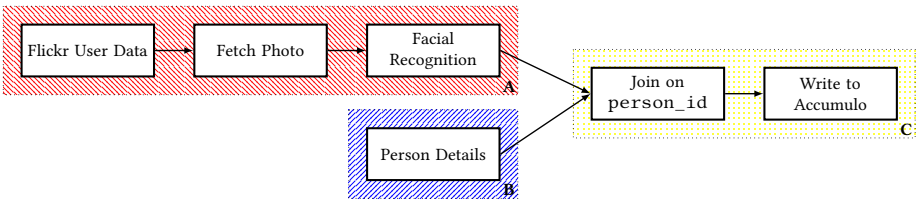


Figure 5.1: A sample analytic, reading profile pictures from Flickr and using facial recognition to populate an Accumulo table.

programming model, optimisation constraints and engineering best practices. This complexity is increased when constructing a hybrid analytic which makes use of data from multiple runtimes: should subset **C** of this Flickr analytic be executed in an on- or off-line runtime environment, and which configuration would be most performant and scalable?

The divide between engineering expertise and domain knowledge has led researchers to consider approaches which make best use of available skills, without the drawbacks inherent in traditional models of cooperation, as discussed in Chapter 3. This chapter presents a new approach to this problem, in providing a framework through which domain experts can compose and deploy efficient and scalable hybrid analytics without prior engineering knowledge. This approach removes the need for the user to understand the variety of runtime frameworks on which their analytics may be deployed, the specifics of how to transform data for processing across multiple heterogeneous frameworks, or even an *a priori* understanding of the components available to them. For example, in the Flickr analytic described above, the user might understand that they wish to use both Flickr and their Person Details database, but not the specifics of how to turn a stream of Flickr crawl data into photographs for facial recognition, how to perform the join, and how to make Accumulo and Streams interact. A suitable planning framework allows for a system to fill in such gaps with feasible analytics, without the user needing training in these engineering concepts.

The research described in this chapter directly targets the challenges of delivering on-demand results for novel analytics, in the face of ever increasing complexity and heterogeneity of both large networked data sources and the

systems used to analyse these data at scale. As the range of software models and hardware platforms increases apace, new models for creating fast data analytics must target not only the engineers with experience of these systems, but specialists with domain knowledge to craft the right analytics, rapidly enough to deliver results in time. Traditional languages are not sufficient for this: automated composition presents the best opportunity to enable non-technical specialists to interact with the analytic platforms crafted by expert engineers.

The remainder of this chapter is structured as follows: Section 5.1 outlines the high-level approach adopted in this research and the implications of design choices; Sections 5.2 and 5.3 detail our approach to modelling analytics and planning their execution respectively; Section 5.4 describes the process of efficient code generation; Section 5.5 illustrates the application of this approach through four case studies. Finally, Section 5.6 provides a performance evaluation of this framework, before summarising the research in Section 5.7.

5.1 High-Level Overview

To compose an analytic from a user's goals, the approach presented here employs the components outlined in Figure 5.2. An abstract Analytic Model (detailed in Section 5.2) is used to create a knowledge-base of processing elements (PEs). This knowledge-base encodes information about the types available in the planning system, the PEs which produce and consume these data types, and a collection of pre- and post-conditions attached to these PEs. It is important to note that the creation of this knowledge-base is beyond the scope of this research: it is assumed that engineers in organisations with a need for an analytic planning system are willing to undertake the manual annotation of the PEs they make available to their users.

This knowledge-base provides a semantically precise description of the information encoded in the data both required and produced by the available PEs. It is the contention of this research that this metadata is sufficient to facilitate the

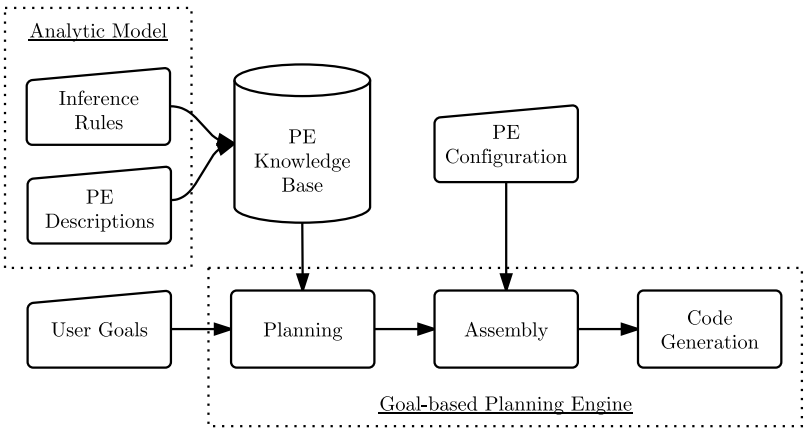


Figure 5.2: Steps in composing an analytic.

automated composition and deployment of complex analytics across multiple runtime platforms in a heterogeneous data-intensive compute environment.

In order to do this, the system collects goals from the user as a second input to the planning process. There are three types of goals that the user may supply to constrain the planning process (see Section 5.3):

- The output types that the analytic must produce;
- The datasource with which the analytic must begin;
- Post-conditions, including those concerned with the state of the runtime environment in which the analytic executed.

For example, to create the sample analytic described in Figure 5.1, the user might specify:

- Types: `person_id`, `person_name`, `postal_address`, `email_address`
- Source: `FlickrUserData`
- Post-condition: `AccumuloSink` PE Used

These constraints are provided to a planning process (Section 5.3), which uses a bidirectional search strategy to traverse the graph of possible PE connections. It

aims to satisfy the given constraints using a minimal number of PEs, producing a set of possible analytics which can be presented to the user. A user-friendly rendering of the analytic can be provided along with textual descriptions of the PEs in the analytic to help the user select which version to deploy. Any unbound configuration options are then supplied by the user to the assembly process (e.g., which Accumulo table to write to, or tunable parameters for the facial recognition), which makes the abstract plan concrete and resolves any ambiguities (e.g., which of the available URL fields to fetch). Finally, code generation (Section 5.4) is invoked on the plan to create an executable analytic.

5.1.1 Methodology

The approach described in this chapter is applicable to a number of runtime models and analytic frameworks. We have implemented and tested it using real analytics in a system called Mendeleev, named after the scientist responsible for composing and organising the periodic table as we know it today. We use a library of real PEs and customer problems to test the scalability of the code generation, and a synthetically generated representative PE library to test the scalability of the planning approach. This, coupled with a qualitative investigation of the use of the planner to generate solutions to these customer problems, forms the basis of the rigorous evaluation in Sections 5.5 and 5.6.

5.1.2 Impact of Design Choices

One of the key assumptions made in this research is a workflow-style execution model. This model pervades the literature on data analytics [7, 16, 34, 50, 54, 82, 89, 123]: while it places a limitation on the range of frameworks that can be used (particularly outside of the real of scalable data analysis), it enables high performance execution across the most common data analytic platforms.

The use of an RDF model [65] to encode the PE knowledge-base slightly increases the set of skills required by engineers to annotate their PEs. However, as discussed in Section 5.3, the strong semantics behind an RDF ontology enable

the use of both system- and engineer-defined inference rules (along with special predicates, as in Section 5.3.2) to enrich the knowledge-base, ultimately reducing the effort required to describe all aspects of the PEs.

In order to make best use of the applicability of the message-passing model, no further assumptions are made during the planning process as to the suite of runtime frameworks which are available or in use. These frameworks are encoded in two places only: the customer-specific model of library PEs, and in a set of pluggable code generation modules. This prevents the planning process from using runtime-specific knowledge (which must be encoded in inference rules or special predicates), but makes it simple to add further runtime frameworks to the Mendeleev implementation.

Approaches which do not use Mendeleev's semantically rich model, or its planning mechanism based around path-finding through candidate analytic space, suffer from the need to manually annotate PE compatibility rules, or to manually assemble elements together after using another discovery mechanism. Furthermore, existing approaches do not attempt to assemble hybrid analytics for execution on heterogeneous architectures; the separation in Mendeleev's model between the abstract concept of an analytic and the specific implementation details of code generation etc. is key in facilitating this.

5.2 Modelling Analytics

This research employs a novel abstraction by which the planning and the concrete implementation of an analytic can be logically separated. There are two components to this model: a semantically rich type system, and a set of analytic components which reference these types. This research models an analytic as a set of parallel-composed communicating sequential processes [51], called Processing Elements (PEs). These pass tuples of data (consisting of a set of named, strongly typed elements) from one PE to the next. When a PE receives a tuple, it causes a computation to occur, and zero or more tuples are emitted on its output

based on the results of that computation at some point thereafter. Data source PEs may emit tuples spontaneously, without any input occurring. Nothing in the model is specific to the planning process – it is an abstract representation of the concrete implementation of a collection of composable components.

The model is encoded in an RDF graph describing the available types and PEs[2]. Types may exhibit polymorphic inheritance, as in a typical second-order type system, indicated using the `mlv:parent` relationship. The statement `:x mlv:parent :y` asserts that `:y` is the super-type of (and therefore subsumes) `:x`. These inheritance relationships may form an acyclic graph, provided a single type name is specified for the target runtime (for example, a Java class or SPL primitive type). Each type declares this using a single `mlv:nativeCode` statement per runtime somewhere in its hierarchy. For example, a buffer of bytes might represent more than one type of information (e.g., a PDF file or an image), even though the data underlying it is the same type, as in Listing 5.1.

Listing 5.1: RDF graph for a simple type hierarchy.

```
# The "raw" ByteBuffer parent type
type:byteBuffer rdf:type mlv:type ;
  mlv:nativeCode  [ rdfs:label "java.nio.ByteBuffer" ;
    mlv:runtime mlv:crucible ] ;
  mlv:nativeCode  [ rdfs:label "list<uint8>" ;
    mlv:runtime mlv:streams ] .
# An image encoded in a ByteBuffer
type:image rdf:type mlv:type ;
  mlv:parent type:byteBuffer .
# A PDF file encoded in a ByteBuffer
type:pdfFile rdf:type mlv:type ;
  mlv:parent type:byteBuffer .
```

In addition to this basic polymorphism, a type may contain an *unbound variable* with an optional parameter (akin to a generic type in Java [14], or a template in C++ [110]). This is used to describe PEs which transform an input type to an output without requiring precise knowledge about interpretation of the information encoded in the data. This information is instead passed using the

[2]RDF types are given in this chapter using W3C CURIE [12] syntax. The following RDF namespaces are used:

```
rdf  http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs  http://www.w3.org/2000/01/rdf-schema#
mlv  http://go.warwick.ac.uk/crucible/mendeleev/ns#
type  http://go.warwick.ac.uk/crucible/mendeleev/types#
```

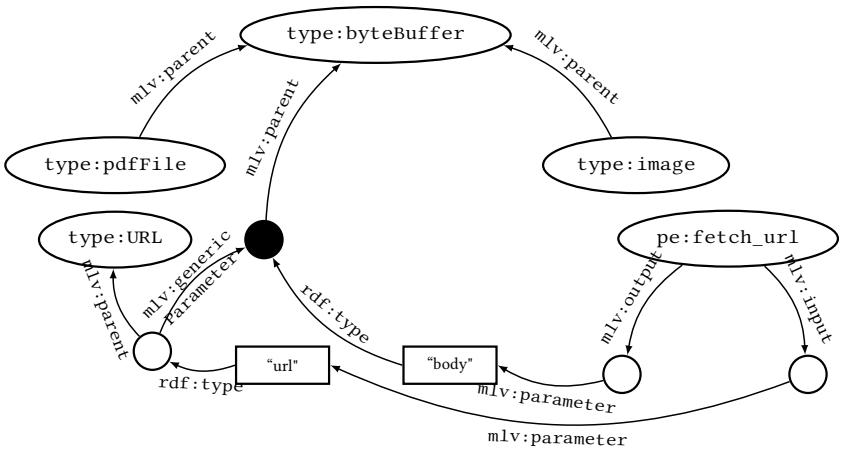


Figure 5.3: Graph visualisation of the RDF description of a portion of the example model. `_:urlType` bnode represented by ●.

unbound variable specified as a generic parameter. For brevity, these generic parameters are described here as `type⟨genericParameter type⟩`. For example (see Listing 5.2), a PE for fetching data over HTTP might take an input type of `type:URL`, which has been parameterised with the generic parameter `⟨_:urlType mlv:parent type:byteBuffer⟩`. It would then output data with the type of the variable `_:urlType`, a subtype of `type:byteBuffer` which is bound to a specific type (e.g., `type:image` in the Flickr analytic described above) during the planning process.

Listing 5.2: Modelling unbound type variables in RDF.

```
# Declaration of a generic type
type:URL rdf:type mlv:genericType ;
  mlv:nativeCode [ rdfs:label "java.net.URL" ;
    mlv:runtime mlv:crucible, mlv:accumulo ] ;
  mlv:nativeCode [ rdfs:label "rstring" ; mlv:runtime mlv:streams ]
      .
# PE input declaration for url<_:urlType>
#    (bnode _:urlType represents variable)
_:sampleInput rdf:type [
  mlv:parent type:URL ;
  mlv:genericParameter _:urlType
] .
# Variable for the type parameter to URL
_:urlType rdf:type type:byteBuffer .
# PE output parameter using the variable
_:sampleParameter rdf:type _:urlType .
```

A visualisation of the RDF graph resulting from this type hierarchy can be seen (along with a subset of the PE model described in Listing 5.3 later) in Figure 5.3. The unbound variable `_:urlType` is highlighted as a filled black circle in this figure.

As suggested by the types used above, the engineers who describe their PEs are encouraged to do so using the most specific types possible. For example, the more precise semantics of `type:image` are to be preferred to `type:byteBuffer`, even though both result in the same `mlv:nativeCode`.

5.2.1 PE Formalism

MENDELEEV's model of execution makes assumptions about the behaviour of PEs described to the MENDELEEV system, and the manner in which they are assembled. These assumptions are encoded in a formalism describing the MENDELEEV model implemented in the planner (detailed in Section 5.3). PEs which do not fully conform with these assumptions may still be described to the system – these require the use of *constraints*, described in Section 5.3.2. However, the manner in which PEs are assembled may not be directly influenced by the PE descriptions: it is this which is defined below.

In this model, we consider a PE χ_n to have a set of declared input types μ_n, and a set of declared output types ν_n. For a data source, $\mu_n = \emptyset$ (it produces data without any inputs being present), while for a sink $\nu_n = \emptyset$ (it receives inputs of data, but produces no output). Tuple data generally accumulates as it passes through each PE, treating it as an enrichment process on the data it receives; the model assumes that the inputs to a given PE's computation are not discarded during computation, meaning these data are available for use in PEs later in the chain. No specific knowledge about the processing performed is encoded in the model. More formally, a PE χ_n has an accumulated output type (denoted as τ_n) based on the type of the tuple received on its input, τ_{n-1}. Thus,

to determine τ_n for a given PE, the entire enrichment chain must be known:

$$\tau_n = \nu_0 \cup \nu_1 \cup ... \cup \nu_{n-1} \cup \nu_n \tag{5.1}$$

Or, inductively:

$$\tau_n = \tau_{n-1} \cup \nu_n \tag{5.2}$$

This model can be extended to include PEs (e.g., complex aggregations) that clear the accumulated data in a tuple declaration before emitting their outputs; this extension is considered in greater detail as part of the planning process in Section 5.3.2. This extension allows the introduction of PEs which are *not* enrichment operators.

One important extension to this model is in support of operators which require inputs on more than one port, such as join operators (discussed in further detail in Section 5.3). These receive two or more discrete sets of input types, and by default emit the union of their accumulated inputs. Thus, for an operator χ_n with inputs χ_i and χ_j, τ_n is given as follows:

$$\tau_n = \tau_i \cup \tau_j \tag{5.3}$$

The ability for two PEs to connect relies upon reasoning about a form of subsumption compatible with the type model described above. A type u can be said to be subsumed by a type v ($u \triangleleft v$) if one of the following cases hold true:

$$u \triangleleft v \Leftarrow \begin{cases} u \texttt{ mlv:parent } v \\ u \texttt{ mlv:parent } t, t \triangleleft v \end{cases} \tag{5.4}$$

$$u\langle t \rangle \triangleleft v\langle s \rangle \Leftarrow u \triangleleft v \wedge t \triangleleft s \tag{5.5}$$

A PE χ_x is considered *fully compatible* with χ_y, and is thus able to satisfy the

inputs of PE χ_y, if the following holds true:

$$\forall t \in \mu_y, \exists u \in \tau_x \mid u \triangleleft t \tag{5.6}$$

Types on PEs are therefore compared by equality and subsumption; no other comparisons are possible in the MENDELEEV system. In the RDF model, each PE definition includes the native type name associated with the PE, as well as the set of (typed) configuration parameters, and input and output ports. Additionally, the model may include user-friendly labels and descriptions for each of these definitions. Unlike other planning engines (particularly HTN style planners such as MARIO), which require the engineer to additionally implement prototype code templates, this RDF model is the only integration that is required between a PE and the MENDELEEV system. For example, a more complete version of the HTTP fetching PE described above is shown in Listing 5.3. Appendix B details the results of applying inference to this model. These inference steps make it possible to use a SPARQL query such as that shown in Listing 5.4 to retrieve all PEs which may emit a given type.

Listing 5.3: Modelling an SPL (IBM's Streams Processing Language) HTTP Fetch PE in RDF.

```
pe:fetch_url rdf:type mlv:spl_pe ;
  mlv:nativeCode "lib.web::FetchURL" ;
  mlv:input [
    mlv:parameter [  # url is a URL<?T>
      rdfs:label "url" ;
      rdf:type [
        mlv:parent type:URL ;
        mlv:genericParameter _:fetch_type
      ]
    ]
  ] ;  # End input declaration
  mlv:output [
    rdfs:label "HttpOut" ;
    mlv:parameter [  # httpHeaders is a header_list
      rdfs:label "httpHeaders" ;
      rdf:type type:header_list
    ] ;
    mlv:parameter [  # body is a ?T
      rdfs:label "body" ;
      rdf:type _:fetch_type
    ]
  ] .  # End output declaration
```

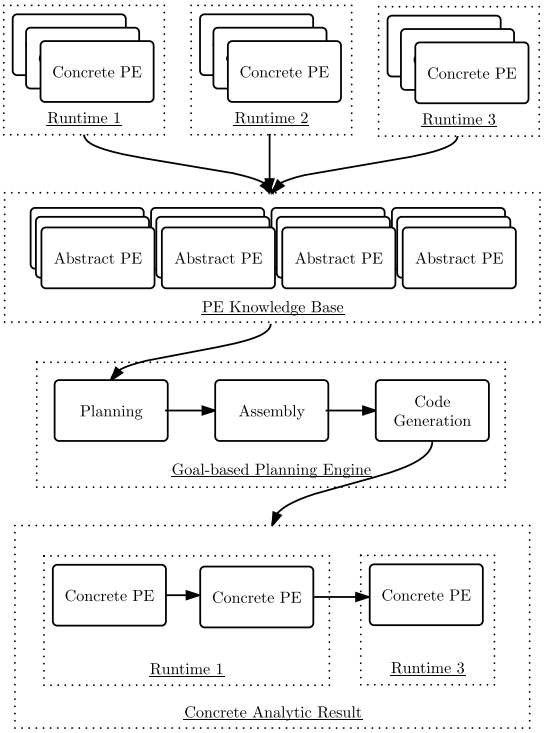


Figure 5.4: Using the PE Model abstraction to separate planning and concrete PE implementations.

```
# byteBuffer is the parent type of ?T
_:fetch_type rdf:type type:byteBuffer .
```

Listing 5.4: Querying the RDF knowledge-base for PEs which emit a given type

```
SELECT DISTINCT ?pe
WHERE {
        ?pe rdf:type mlv:pe ;
                mlv:output ?output .
        ?output mlv:parameter/rdf:type/rdfs:subClassOf ?type .
}
```

5.2.2 PE Model Abstraction

This model abstracts the concrete implementation of an analytic away from runtime framework-specific details. This is vital to enable hybrid planning, as all runtime frameworks may be treated equally: as seen in Section 5.3, PEs from any framework may be assembled in a workflow. The PEs represented by this abstraction are later made concrete by the code generation process (Section 5.4),

which translates from the execution model assumed in the PE model to runtime primitives, invoking the user-defined components the model describes, as shown in Figure 5.4.

5.3 Goal-Based Planning

Building on the semantically rich type system described above, this approach implements a goal-based planner. The aim of this planner is to explore the graph of possible connections between PEs using heuristics to direct the search, accumulating types in the τ set until the user-supplied constraints have all been satisfied, or the planner determines that no solution exists. This custom planner utilises an off-the-shelf forward chaining reasoner to perform initial inference over the RDF graph, and then materialises possible analytic subgraphs as exploration occurs. In this way, the potential explosion of the complexity of the graph is kept to a minimum: only those parts of the graph required for the plan exploration are created in memory.

5.3.1 Type Closure

Given the RDF model of the PE knowledge-base, a suite of forward inference rules are pre-computed before any planning may occur. These rules are applied using a forward chaining reasoner (the Rete-UL algorithm implemented in FuXi [80]), and compute three key types of closure. First, RDFS [15] reasoning is applied to the types in the knowledge-base (primarily to compute the closure over second-order types). Next, unbound type variables are compared, to compute potential subsumption. Finally, candidate PE matches are inferred based on rules derived from the *full compatibility* specification in Section 5.2, Equation 5.6. A PE χ_x is considered *partially compatible* with χ_y, and is thus a potential candidate for

sending tuples to PE χ_y, if one of the following holds:

$$\exists t \in \mu_y, u \in \nu_x \mid u \triangleleft t \tag{5.7}$$

$$\exists t \in \mu_y, u{<}v{>} \in \mu_x, v \in \nu_x \mid v \triangleleft t \tag{5.8}$$

$$\exists s{<}t{>} \in \mu_y, u{<}v{>} \in \nu_x \mid u \triangleleft s, v \triangleleft t \tag{5.9}$$

Less formally; if there is a type in the output signature of PE χ_x which subsumes a type in the output of PE χ_y, then there is a candidate connection between χ_x and χ_y (from the rule in 5.7). The rule in 5.8 unpacks generically parameterised types on the input to χ_x, such that if $u\langle v\rangle$ is in the input set of χ_x and v is in the output set, then any type t on the inputs to χ_y allows for a candidate connection between these PEs. Finally the rule in 5.9 allows for generically typed parameters in the output set of χ_x which are subsumed by parameters in the input set of χ_y to also allow for a candidate connection between these PEs.

For example, consider `pe:fetch_url` described in Listing 5.3; it requires a URL parameterised with any `type:byteBuffer`. Consider also a PE called `pe:exif`, which (for the sake of this example) requires a `type:image` on its input (where `type:image` $\triangleleft$ `type:byteBuffer`), and outputs a number of Exif[3] facts:

$$\mu_{\text{fetch\_url}} = \{\texttt{type:url}\langle \texttt{\_:T} \triangleleft \texttt{type:byteBuffer}\rangle\} \tag{5.10}$$

$$\nu_{\text{fetch\_url}} = \{\texttt{\_:T}\} \tag{5.11}$$

$$\mu_{\text{exif}} = \{\texttt{type:image}\} \tag{5.12}$$

$$\nu_{\text{exif}} = \{\texttt{type:camera}, \texttt{type:lat}, \texttt{type:lon}, \texttt{type:fstop}, ...\} \tag{5.13}$$

Through Equation 5.8 above, the `_:T` output by `pe:fetch_url` can potentially be used to satisfy the input to `pe:exif`. In this case, `pe:fetch_url` is considered partially compatible with `pe:exif`, and is marked as a candidate connection when `_:T` is bound to `type:image`.

[3]Exchangeable image file format; image file metadata

5.3.2 Conditions

Once the type closure is computed, a further suite of rules annotates each PE in the knowledge-base with a set of pre- and post-conditions, derived from the input and output specification. A pre-condition is automatically applied specifying the runtime environment for each PE: this is derived from the `rdf:type` specified for the PE. These inferred conditions can be augmented in the RDF PE model with two further types of condition.

The first of these condition types are used to alter the behaviour of the inference or the search process. For example, a `mlv:clearPreConditions` statement is used when modelling PEs which do not automatically pass on the data received on their inputs. Such PEs may include aggregation operations (grouping etc.), windowing operators, or those which apply complex non-enrichment algorithms to their inputs. Another special condition, `mlv:clearRuntime` is implemented to remove post-conditions from the τ set which specify the current runtime environment. For example, Listing 5.5 models a PE which aggregates input data into a Gaussian Mixture Model using an Expectation Maximisation algorithm.

Listing 5.5: RDF Model for a Gaussian Mixture Model implemented on Apache Spark

```
pe:gmm2d a mlv:spark_pe ;
  rdfs:label "Apply EM to generate a 2D Mixture of Gaussians
      modelling the input" ;
  mlv:nativeCode "mendeleev.pe.GMM2D" ;
  mlv:input [
    mlv:parameter [ rdfs:label "x" ; rdf:type type:double ] ;
    mlv:parameter [ rdfs:label "y" ; rdf:type type:double ]
  ] ;  # Clear existing pre-conditions
  mlv:postCondition [ mlv:clearPreConditions pe:gmm2d ] ;
  mlv:output [  # Emit a collection of weighted 2D Gaussians
    rdfs:label "Gaussians" ;
    mlv:parameter [ rdfs:label "weight" ;
      rdf:type type:gmm_weight ] ;
    mlv:parameter [ rdfs:label "x" ;
      rdf:type type:gaussian_x ] ;
    mlv:parameter [ rdfs:label "y" ;
      rdf:type type:gaussian_y ] ;
    mlv:parameter [ rdfs:label "theta" ;
      rdf:type type:gaussian_rotation ] ;
    mlv:parameter [ rdfs:label "A" ;
      rdf:type type:gaussian_magnitude ]
  ] .
```

The second type of user-specified condition is one which has no special meaning to the planner, but makes an assertion about the state of the analytic. These are employed, in conjunction with the `mlv:clearRuntime` condition above, to manage the transition between runtimes. Listing 5.6 gives an example of how synthetic runtimes are used (in this case, `mlv:accumulo_to_streams`) to constrain the planner, so that an Export node from one runtime is followed immediately by an Import node for the next. These provide the necessary hooks for the code generators (discussed in Section 5.4) to create suitable code for managing the inter-runtime transport of data.

Listing 5.6: RDF Model for an Import and Export transport from the Accumulo Iterator paradigm into IBM InfoSphere Streams.

```
pe:accumulo_to_streams_export a mlv:accumulo_pe ;
  rdfs:label "Export Accumulo->Streams" ;
  mlv:nativeCode "mendeleev.pe.StreamsExportIterator" ;
  mlv:input [ rdfs:label "Data" ] ;
  # Clear existing runtime; reset to the synthetic
  #   mlv:accumulo_to_streams runtime
  mlv:postCondition [ mlv:clearRuntime
    pe:accumulo_to_streams_export ] ;
  mlv:postCondition [ mlv:runtime mlv:accumulo_to_streams ] .

pe:accumulo_to_streams_import a mlv:spl_import_pe ;
  rdfs:label "Import Accumulo->Streams" ;
  mlv:nativeCode "mendeleev.pe::AccumuloImport" ;
  mlv:output [ rdfs:label "Data"] ;
  # Require the synthetic mlv:accumulo_to_streams runtime
  mlv:preCondition [ mlv:runtime mlv:accumulo_to_streams ] ;
  # Replace the mlv:accumulo_to_streams runtime with mlv:streams
  mlv:postCondition [ mlv:clearRuntime
    pe:accumulo_to_streams_import ] ;
  mlv:postCondition [ mlv:runtime mlv:streams ] .
```

In practice, this inference closure is calculated offline and the resultant graph is stored in order to ensure interactive performance.

5.3.3 Search & Assembly

The search through the graph of partially compatible PEs is outlined in Algorithm 1. This algorithm finds a set of pathways through the graph of candidate PE connections which will generate the required set of post-conditions, while fulfilling the pre-condition requirements of each PE. In order to minimise the

search-space explosion and minimise memory consumption, the search is performed bi-directionally with iterative deepening, using an empirically selected heuristic to expand the search space backwards for every three levels of forward search. This setting may be configured in the MENDELEEV implementation – other settings were tested, but in practice a ratio of 1:3 was found to perform well. Similarly, if a source or a sink constraint is specified, it is used to optimise the search process. The algorithm proceeds in six stages:

L2-4: Every 3 levels of forward search, expand the set of backward search candidates by one more step;

L5-11: If the call to SOLVE does not provide a bound on the source, launch a solver to generate results for all sources in the model;

L12-17: If the current PE has more than one input, launch a new SOLVE to satisfy the pre-conditions of each input;

L18-21: Update the sets of accumulated conditions (τ), and test to see if all required post-conditions are satisfied; if so, this branch of the search terminates;

L22-26: Attempt to search the next level (recursively), using only the set of backwards candidates;

L27-29: If the above step did not yield any new paths, repeat the search with PEs not in the set of backwards candidates.

A simple heuristic ranking may be applied to this set of candidate pathways e.g., based on the number of PEs in the path (if two paths accumulate the same post-conditions, it can be considered that their results are similar, and thus the shorter, "simpler" path should be preferred). It is not sufficient to automatically select and assemble one of the available paths arbitrarily: some user interaction is required to validate that the correct analytic is selected.

Once the user selects an execution plan from the generated options, it must be assembled into a concrete plan. This process involves binding keys from

Algorithm 1 Bidirectional Planning, searching for a given set of target conditions (ϕ), source PE (σ), accumulated conditions (τ), and backwards search set (β).

```
1:  procedure SOLVE(φ, σ, τ, β)
    ▷ Every 3 levels of forward search, advance backwards
2:      if search_level % 3 == 0 then
3:          β ← β ∪ providers_of(φ)
4:      end if
5:      if σ not given then
6:          results ← ∅
7:          for all source s in model do
8:              results ← solve(φ, s, τ, β)
9:          end for
10:         return results
11:     end if
12:     results ← ∅
    ▷ Check σ for secondary inputs
13:     for all input i in inputs(σ) do
14:         if i not satisfied by τ then
15:             results ← results ∪ solve(preConditions(i), σ, ∅, ∅)
16:         end if
17:     end for
    ▷ Update τ with postConditions of σ, and check for completion
18:     τ ← τ ∪ postConditions(σ)
19:     if τ satisfies φ then
20:         return [σ]
21:     end if
    ▷ Depth-first search of PEs in β
22:     forward ← consumers_of(τ)
23:     candidates ← dfs_search(forward ∩ β, φ, σ, τ)
24:     for all candidate in candidates do
25:         results ← results ∪ [σ, candidate]
26:     end for
    ▷ Depth-first search of remaining candidates
27:     if results == ∅ then
28:         results ← dfs_search(forward − β, φ, σ, τ)
29:     end if
30:     return results
31: end procedure
```

Algorithm 2 Type Pruning.

```
1: procedure PRUNE_TYPES(pe, φ)
   ▷ Remove types from τ_pe that are not in the φ set
2:     τ_pe ← τ_pe ∩ φ
   ▷ Add types to the φ set that are required by this PE
3:     φ ← φ ∪ μ_pe
   ▷ Recurse to all publishers of data to this PE
4:     for all σ in publishers(pe) do
5:         prune_types(σ, φ)
6:     end for
7: end procedure
```

each tuple to the required output types. For example, if a tuple of Flickr user data contained two `type:url`⟨`type:image`⟩ parameters, a profile background and a user avatar, and it was passed to the aforementioned `pe:fetch_url`, the assembly process must bind one of these parameters on its input. In practice, no reliable heuristic is available for this, and user configuration is required. For a domain expert this should not present a difficulty, as they can be expected to understand both the nature of the fields in their data and (with the brief descriptions of PEs in the RDF knowledge-base) how the PEs will operate on the fields they configure.

This planning and assembly process generates an acyclic graph of PEs as its output, with a single goal-state node and one or more source nodes. It can also, therefore, be considered a tree rooted on the goal node. The goal node will have a τ which includes all types passed forward to that node – however, many of the types specified in the post-conditions may not be needed in order to correctly complete the computation. As a result, the assembly process takes a second pass across the topology to prevent it from passing unnecessary data forwards. This type pruning algorithm is outlined in Algorithm 2; it makes a single breadth-first traversal over the topology backwards from the goal node, computing this set of required types and simultaneously removing any types which are not required later in the topology. This helps to control the otherwise unlimited expansion of tuple width, improving the space, time and message passing complexity of the resultant analytic.

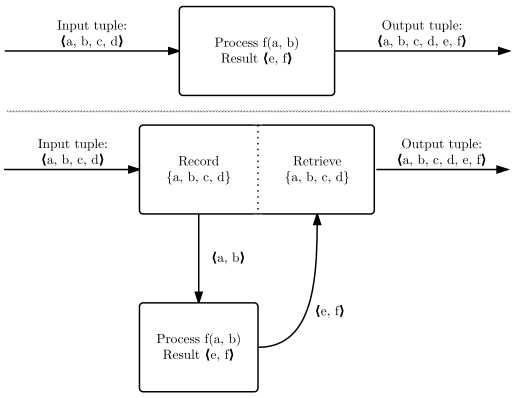


Figure 5.5: Top: Mendeleev message passing model for a process `f`. Bottom: Crucible wrapper-based model of field copying semantics.

5.4 Code Generation

Once the concrete execution plan is assembled, it is passed to a pluggable code generator. Mendeleev's planner produces a concrete plan, which the code generator must turn into native code for execution on a mixture of on- and off-line runtimes. The only form of optimisation performed at this stage is the type pruning described above; other forms of task-level parallelism, scheduling, shuffle strategies, etc. are determined by the runtime framework on which the generated analytic is executed. To deploy an analytic on a given framework, Mendeleev may either be used to generate native code for each runtime directly, or an intermediate representation which manages the differences in runtime models.

5.4.1 DSL Code Generation

Mendeleev has been designed to generate code using the Crucible DSL (Chapter 4) as an intermediate representation. Crucible's suite of runtime environments, adhering to a common runtime model, help provide Mendeleev-generated analytics with consistent execution semantics across on- and off-line runtimes.

There is one key difference between the Mendeleev and Crucible execution models: whereas Mendeleev assumes that all keys in the input tuple are

passed through on the output, Crucible does not perform this pass-through automatically. It is possible to implement these semantics in Crucible, however. Figure 5.5 illustrates how this might be achieved in the basic Crucible execution model. Mendeleev's conceptual model (the top of Figure 5.5) shows a PE `f(a, b)` which generates the tuple ⟨`e, f`⟩ as its results, passing through the full input tuple along with those results. At the bottom of Figure 5.5, an implementation of the Mendeleev tuple field copying semantics in the basic Crucible model shows how each functional PE is wrapped in one which stores the input tuple fields, and appends them to the output of each tuple from that functional PE.

While this theoretical approach produces correct results, the extra message passing it involves would slow topologies down considerably (as discussed in Chapter 4, minor changes in message passing patterns or costs can have a significant impact on scalability in a Crucible topology). Instead, Mendeleev generates a synthetic parent PE in Java for each PE in the Crucible topology, overriding a small portion of the base Crucible runtime on a per-PE basis with generated code. This parent is responsible for intercepting received and emitted tuples, recording the inputs in local state, and appending the relevant outputs of that PE's pruned accumulated type on tuple output. To use the example of `pe:fetch_url` in the Flickr analytic above, this synthetic parent might record the `type:profile_image` URL on its input, and append it to the output tuple. Note that this synthetic parent must be aware of tuple fields which have been pruned from the output in Algorithm 2.

5.4.2 Native Code Generation

When an analytic does not require the flexibility or features of the Crucible DSL (or PEs are only available in a native implementation, not a Crucible library), direct native code generation may be a more performant option. This code generation option relies on the accuracy of both the input and output specifications, and the manually entered pre- and post-conditions of PEs to generate the correct code. Four native code generators are implemented in

MENDELEEV: two for Accumulo, one for IBM InfoSphere Streams SPL, and one for the Meteor.js reactive web presentation framework.

Accumulo requires two separate code generators: one for the base Accumulo table (consisting of heterogeneous rows of Key-Value pairs), and one for an Iterator stack which may be applied on top of this. These generators simply create a pair of Java classes which configure an Accumulo connection, set up the requisite Iterator stack, and return a Scanner of rows to the calling site. This Iterator stack executes on the server-side as the Scanner is used on the client which consumes the results.

MENDELEEV includes a set of inter-runtime transports, enabling the motion of data from one analytic to another. Modelling these with pre- and post-conditions (Section 5.3.2) affords the ability for each transport to have a distinct implementation designed for optimal performance (such as sending JSON to Meteor.js, but writing Key-Vaue Mutations to an Accumulo table). The implemented transports, and their Export / Import mechanisms, are outlined in Table 5.1.

Source Runtime	Export Behaviour	Import Behaviour	Destination Runtime
Accumulo	No-Op	Scanner; SPL type conversion	Streams
Accumulo	JSON Serialisation	Scanner; JSON parse	Meteor.js
Streams	Convert SPL types to Java; Kryo Serialise; write to Accumulo table	No-Op	Accumulo
Streams	JSON Serialisation; TCP socket server	TCP socket client; JSON parse	Meteor.js
Meteor.js	TCP socket client	TCP socket server; JSON parse; SPL type conversion	Streams
CRUCIBLE	TCP socket server	TCP socket client; SPL type conversion	Streams
CRUCIBLE	Kryo Serialise; write to Accumulo table	No-Op	Accumulo

Table 5.1: MENDELEEV Import/Export implementations.

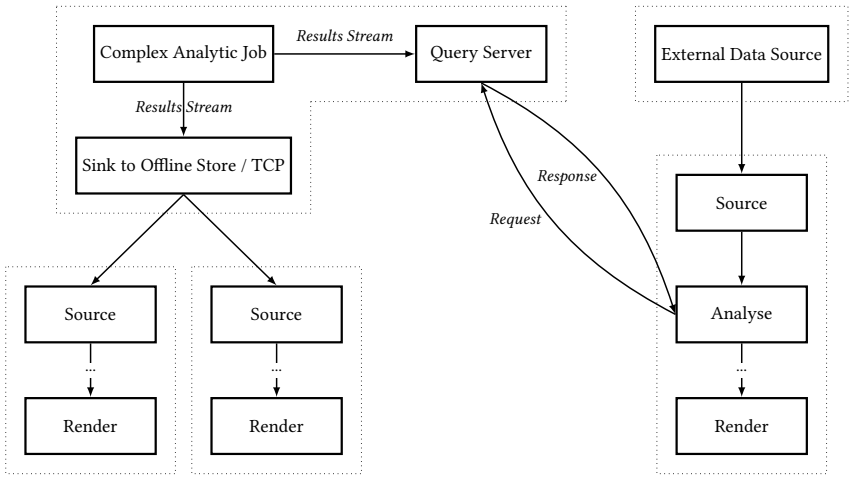


Figure 5.6: Deployment scenarios for complex analytics.

5.4.3 Integrating Complex Analytics

Some users of a system such as MENDELEEV require complex carefully engineered analytics (e.g., to build up state about a set of identifiers, or for performance-tuned machine learning algorithms). In the interests of efficient system utilisation, it is often desirable to run these types of analytic as a central job to which other analytics may subscribe. Several patterns can be used to expose this behaviour transparently to a user in MENDELEEV, as illustrated in Figure 5.6.

First, it is possible to simply write all results from the "Complex Analytic Job" to a persistent store, as a results cache (shown below the complex job in Figure 5.6). This approach results in treating the output as an offline data source for each new MENDELEEV analytic. It is also possible to achieve a streaming equivalent by exporting results on a TCP Socket Server. This approach has a relatively low implementation overhead, but depending on the use cases for the complex analytic may result in more complex MENDELEEV plans (e.g., due to a frequent need to join this data with other sources). An alternative for analytics which use the complex job as a source of enrichment is that of an RPC-style model (shown to the right of the job in Figure 5.6). This is suitable for large stateful analytics, although it requires the maintenance of an RPC query server

and associated infrastructure for distributed configuration. This infrastructure is outside the scope of Mendeleev; systems such as Apache ZooKeeper have been found to fulfil this distributed configuration requirement in practice.

As well as complex analytic jobs, some organisations will have complex sources of data (e.g., a large relational database with many views). The complexity of extracting data from such a system need not be reflected by equivalent complexity in Mendeleev; each potential view can be considered a different source PE in the knowledge-base, with its own tuple type information entered accordingly. Note that Mendeleev has no restriction on the number of times a single target PE type (a given value of `mlv:nativeCode`) may appear in a knowledge-base (e.g., with different configuration parameters to turn on or off features of that PE). This is a useful design pattern; engineers can prepare a single general-purpose accessor PE which is configured in the knowledge-base to represent many different data sources. As the PE knowledge-base is RDF-based, it is additionally possible to extend the set of inference rules to include generators for permutations and combinations of different PE parameters, rather than entering them by hand.

5.5 Case Studies

To better understand the process of composing analytics in Mendeleev, this section presents a series of case studies and an evaluation of this technique. These analytics have been generated with Mendeleev, using a small shared library (see Appendix C for a full listing of the PEs in this library) of general-purpose PEs. Figure 5.7 illustrates the generated analytics for each case study below; each figure shows the PEs in an analytic (as boxes), the tuple subscriptions between those PEs (arrows indicate the direction of flow), and the runtime for each subset of PEs (shaded outer boxes). Note that, for brevity, explicit Import/Export nodes have been omitted from these representations.

Each section below describes a new case study; the set of constraints for each

case study describes the full specification that was given to the MENDELEEV planner to return the analytic or set of analytics described. No further tweaking of the knowledge-base or planner were required.

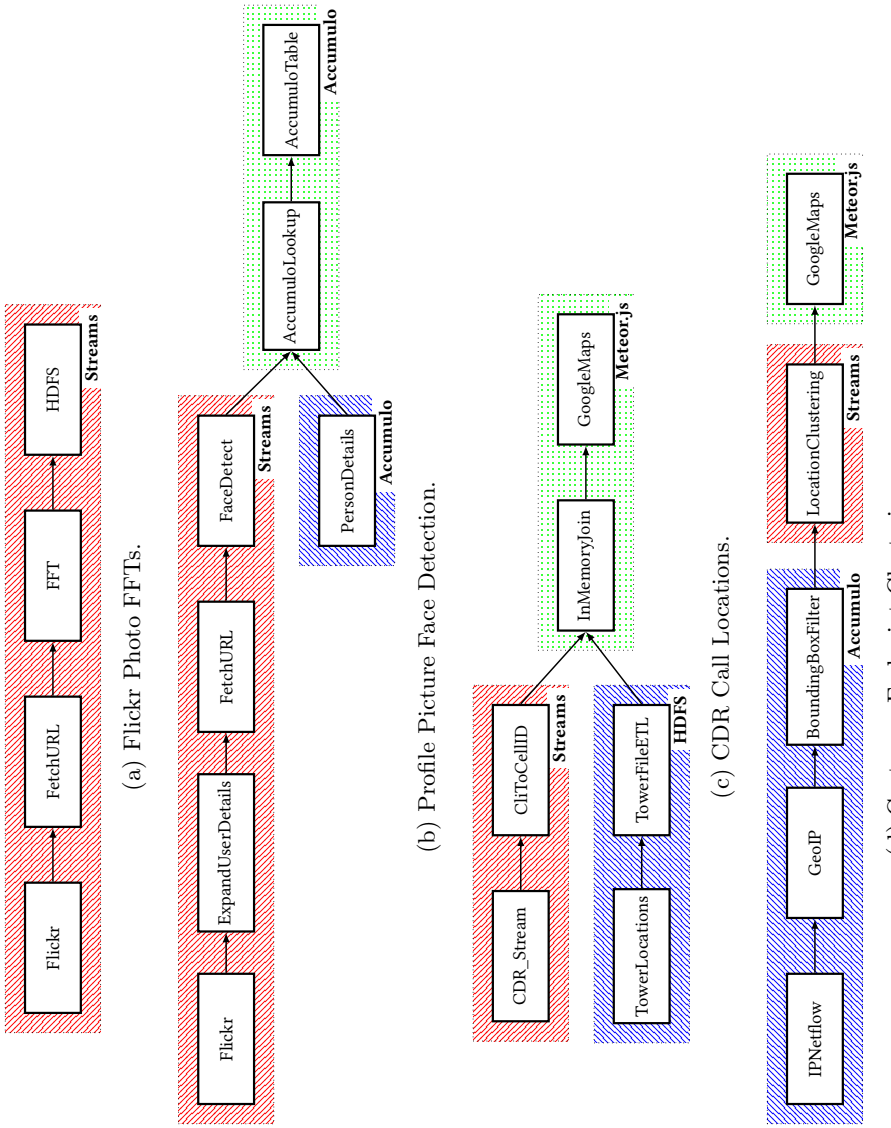


Figure 5.7: Planned analytics for Flickr Image and Telecommunications Data analysis.

5.5.1 Flickr FFT Workflow

The user wishes to compute and store the Fourier transform of images from Flickr, and store those results in HDFS for use later in their workflow. Engineers have exposed a crawl operator which emits Flickr photo metadata to the MENDELEEV system. The user selects the following bounds from the user interface:

1. PE Used: HDFS
2. Types: `image`, `fft2d`[4]

With each refinement of a bound, the MENDELEEV UI plans a new set of plausible analytics to answer that query. It is interesting to note here, that the query does not explicitly require data from Flickr; any data sources in the knowledge-base which can be used to return an `image` may be offered to complete this query. In this instance, MENDELEEV produces a single result: the analytic shown in Figure 5.7(a).

5.5.2 Case Study: Flickr Facial Recognition

A different analyst has an interest in annotating Flickr images with the email addresses of the people in them using a facial recognition system, sending their results to an Accumulo table (as described in the original example in Figure 5.1. They configure MENDELEEV to search as follows:

1. PE Used: AccumuloTable
2. Types: `person`, `emailaddress`

The user is presented with a single analytic, but closer inspection shows that it does not use Flickr as a datasource. They refine their query interactively to bind the source to "Flickr". This returns four candidate analytics; the user selects the version which crawls Flickr for new results using the Streams runtime (shown in Figure 5.7(b)), writing results to an Accumulo table. This data is

[4]The output of a Fourier transform on 2-D input data

used to look up Person Details from an Accumulo table in a compaction-time Accumulo Iterator. During the assembly stage, there are two image URLs to choose between; the Flickr photo and the user's profile picture. They configure the FetchURL PE to use the latter and complete their assembly.

5.5.3 Case Study: Telecommunications Call Events

An analyst for a mobile telecommunications company wishes to display a live map of call events for a video wall in their Network Operations Centre. They configure the following query, which results in the analytic in Figure 5.7(c):

1. PE Used: GoogleMaps
2. Types: `msisdn`[5], `tower_latitude`, `tower_longitude`

5.5.4 Case Study: Telecommunications IP Endpoints

A further analyst, with an interest in IP traffic and routing, wishes to determine hotspots with which their customers communicate, for both network layout purposes and to check the telecommunications company has the right peering agreements in place. They configure a query:

1. PE Used: BoundingBoxFilter, GoogleMaps
2. Types: `ipaddress`, `cluster_latitude`, `cluster_longitude`

Their resulting analytic is shown in Figure 5.7(d) – this has been selected from the three analytics returned by the query. However, their analytic is not fully assembled until the GeoIP PE has its `ipaddress` parameter bound to the source or destination IP. As the analyst is interested in determining the locations their connections terminate, they select the destination IP, and complete the analytic assembly. Note here that plans were additionally generated for deployment against streaming IP Netflow data, as well as this historical database of events. This is an ideal use case for a Crucible-based solution: the generated code can

[5]A unique telecoms subscriber identifier

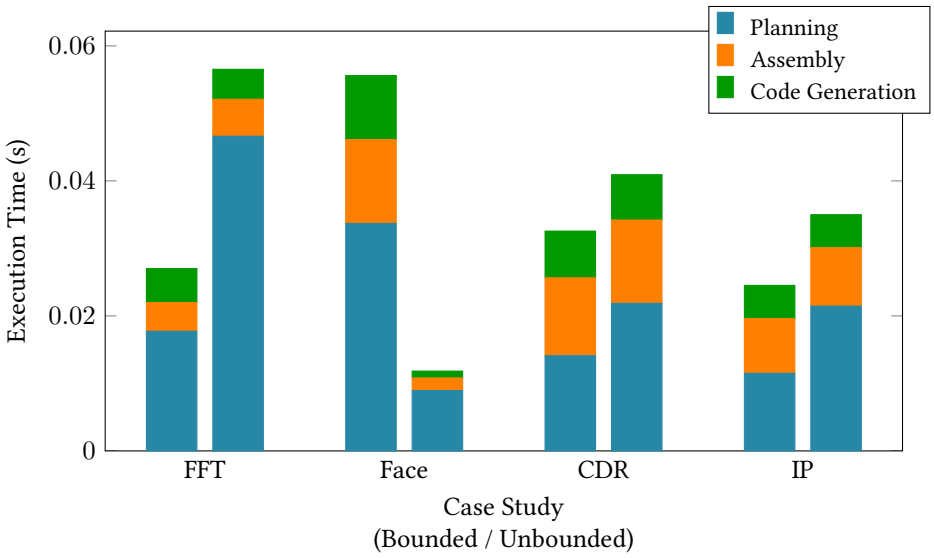


Figure 5.8: Benchmark results for the MENDELEEV planner when applied to the case studies.

then simply be deployed to either their streaming or their offline platform, and the CRUCIBLE framework will select the relevant instance of the datasource.

5.6 Performance Evaluation

In order to better understand the performance characteristics of the MENDELEEV implementation, and thus demonstrate its viability for real-world use, two key aspects of performance are examined: (i) the time taken for the planning and assembly process; and (ii) the runtime performance of the resulting analytics.

5.6.1 Planner Performance

In order to examine the performance of the planning process, the four case studies discussed above are again used. Each case study has been benchmarked as a *bounded* query (with a data source specified) and as an *unbounded* query (no source specified, forcing the planner to attempt to infer possible sources). The performance of the planner against a test knowledge-base of 20 PEs can be seen in Figure 5.8. This test knowledge-base describes the real PEs used in the

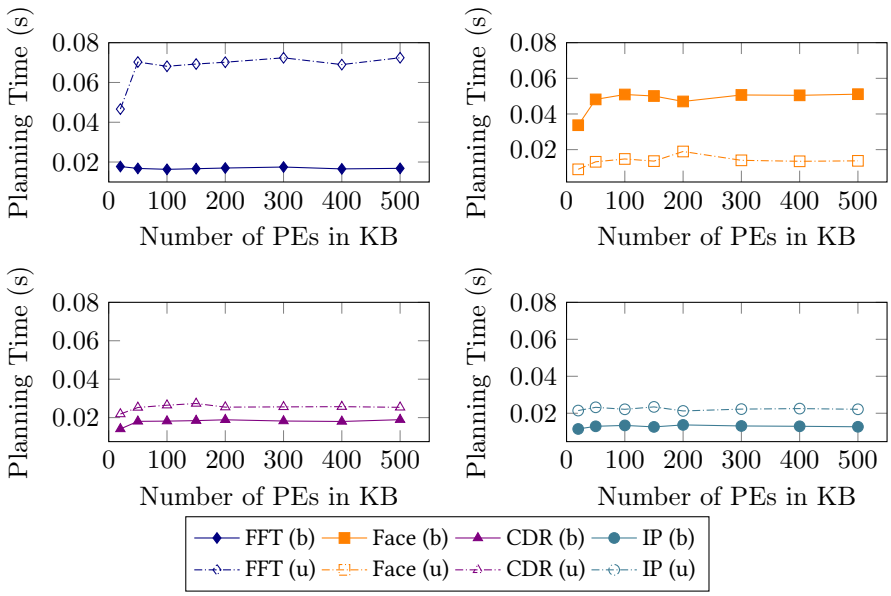


Figure 5.9: Scaling of the MENDELEEV planner with knowledge-base size for both (b)ounded and (u)nbounded case studies.

case studies described in Section 5.7. On average, each PE is described with 11 RDF statements, and there are 75 types described in the model. To highlight the accessibility of this approach, these planner experiments are performed on a typical workstation class machine – containing a 4-core Intel Core i7 CPU with 8 GB of RAM.

The backwards search optimisations used in the planning algorithm prevent many of the unbounded queries from taking significantly longer than their bounded equivalents. The two notable exceptions to this are in the FFT query (which does not list any grounded types in its goal to inform the choice of source), and the Face Detection query, which, in its unbounded form, altogether fails to generate a correct solution (but does so quickly). The bounded Face Detection query is the longest-running assembly and generation process, due to the complexity of the resulting analytic; both in terms of the number of tuple fields to be processed in the pruning analysis, and the number of PEs in the resulting analytic. In this small knowledge-base, planning takes consistently less than 60 milliseconds across all of these tests.

In order to better understand how the bidirectional search in the planning phase scales as the knowledge-base expands, a further set of planner benchmarks are presented in Figure 5.9 for knowledge-bases of varying size over both the bounded and unbounded query variants above. The PEs in this expanded knowledge-base were synthetically generated, using the following strategy:

- Types in the existing knowledge-base were manually classified into four categories: General (17 types), IP (5 types), Web (24 types), and Telephony (13 types)
- Each synthetic PE selects a primary category at random. There is a 30% chance that a PE also selects a secondary category
- There is a 90% chance a PE has an input description, and a 90% chance it has an output
- Each input/output generates a set of parameters (the count of which is Gaussian distributed with $\mu = 5$ and $\sigma = 2$) from the set of types belonging to its category/categories

All synthetic PEs are "reachable" in the graph search, and as such have an impact on planning time. They show that in scaling the size of the knowledge-base from 20 to 50 PEs there is a noticeable performance impact. However, due to the bidirectional optimisation in the search, beyond this scale there is little negative impact on the search time. At no point does the planning take longer than 80 milliseconds in the case studies tested, regardless of knowledge-base size. More complete information about the number of plans considered in the search, and the number found and returned, can be seen in Table 5.2.

5.6.2 Runtime Performance

It is valuable to compare the performance of Mendeleev's generated code to hand-written analytics in both the Crucible DSL and in native code. For this, hand-written native and Crucible code for each runtime is compared

Query	Plans Considered	Plans Returned	Planning Time (s)
FFT (b)	53	9	0.017
FFT (u)	126	14	0.072
Face (b)	16	4	0.051
Face (u)	1	1	0.013
CDR (b)	40	31	0.019
CDR (u)	40	31	0.025
IP (b)	8	3	0.012
IP (u)	9	3	0.022

Table 5.2: Number of plans considered and returned in the 500 PE stress test knowledge-base for both (b)ounded and (u)nbounded queries.

to Mendeleev, using a shared library of basic Java operations to implement two variants of the "IP Communications Endpoints" case study described above (Figure 5.7(d)). In the first set of experiments, Crucible is used as the target for comparison, comparing the performance of Mendeleev-generated Crucible code to both hand-written Crucible and native implementations. For these experiments, the full un-filtered dataset is explored. The second set of experiments compare the performance of Mendeleev's native code generation to hand-written native code for the bounding-box filtered version of the analytic.

These analytics were all executed against 194 offline packet capture files, corresponding to 100 Gb of raw capture data (5.8 GB of packet headers). Results were collected on a test cluster consisting of three Hadoop Data Nodes / Accumulo Tablet Servers, one NameNode / Accumulo Master, and two Streams nodes. Each node hosts two 3.0 GHz Intel Xeon 5160 CPUs, 8 GB RAM and 2×1GbE interfaces: the same specification of system used for benchmarking in Chapter 4.

		Records Processed (millions)											
		5		10		20		30		40		50	
	Code Type	Time	Latency	Time	Latency	Time	Latency	Time	Latency	Time	Latency	Time	Latency
Standalone Runtime	Auto-generated DSL	296.73	0.13	591.69	0.11	1179.93	0.13	1770.09	0.11	2359.72	0.10	2948.60	0.12
	Hand-written DSL	333.53	0.16	664.23	0.16	1324.30	0.17	1983.13	0.16	2644.04	0.16	3305.23	0.16
	Hand-written Java	227.52	0.40	453.88	0.38	906.48	0.40	1360.02	0.39	1813.83	0.40	2265.44	0.38
Spark Runtime	Auto-generated DSL	131.69	0.14	208.44	0.14	326.59	0.16	444.34	0.14	561.01	0.14	677.45	0.13
	Hand-written DSL	177.22	1.52	268.73	0.24	442.72	0.29	608.24	0.29	768.83	0.24	939.39	0.40
	Hand-written Spark	117.75	1.24	186.86	1.56	286.40	1.58	384.19	1.38	482.51	1.93	579.88	1.54
Streams Runtime	Auto-generated DSL	1274.68	1.03	2509.74	1.09	4977.64	1.06	7443.37	1.08	9906.07	1.04	12369.67	1.00
	Hand-written DSL	1401.68	1.20	2762.88	1.18	5476.11	1.20	8181.20	1.15	10886.18	1.15	13595.48	1.14
	Hand-written SPL	1041.24	1.00	2063.17	0.98	4103.68	0.97	6143.75	1.00	8173.90	1.01	10195.93	1.01
Streams Only	Auto-generated	3382.75	0.25	5007.45	0.12	7385.10	0.98	9616.30	0.19	11018.85	0.18	12338.60	0.25
	Hand-written	2691.5	0.13	3776.9	0.10	5887.65	0.12	7995.05	0.17	9677.60	0.15	11369.15	0.20
Streams + Iterators	Auto-generated	957.75	0.12	1788.00	0.08	3404.00	0.10	5395.80	0.28	7046.48	0.59	8761.28	0.13
	Hand-written	774.45	0.10	1455.70	0.07	2846.55	0.09	4569.50	0.27	6014.65	0.34	7814.35	0.11

Table 5.3: Benchmarking results (makespan wall time and per-tuple latency) for each runtime mode and code type.

Unfiltered Crucible Analysis

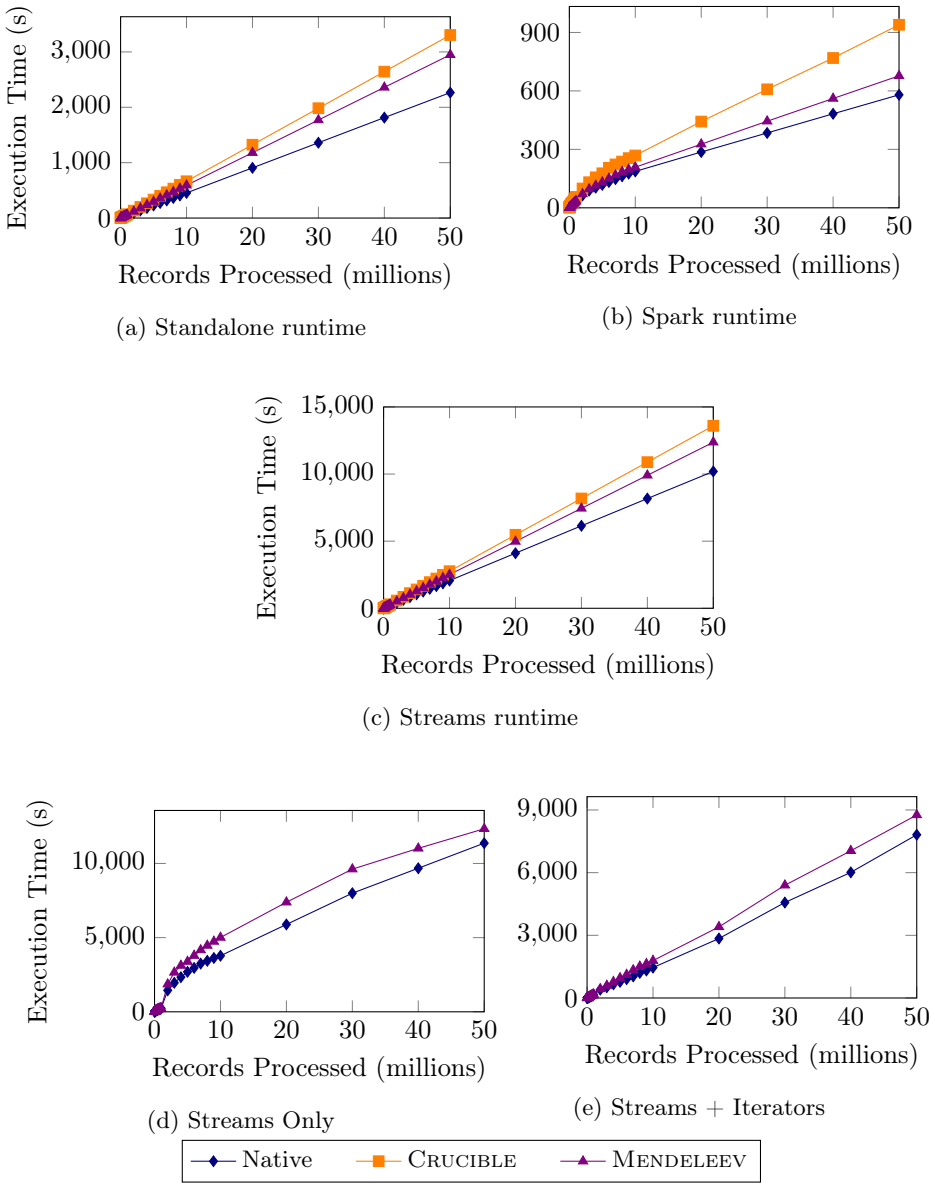
Five equivalent variants of the unfiltered analytic were created: (i) Mendeleev-generated Crucible; (ii) hand-written Crucible; (iii) a multi-threaded Java analytic; (iv) a Spark topology written in Java; and (v) an SPL topology, with associated Java primitive operators. The upper half of Table 5.3 shows the performance and scalability (makespan time for a given input size and latency per tuple) of the analytic on each runtime type in turn; Standalone, Apache Spark (HDFS mode) and on IBM InfoSphere Streams. These data are additionally presented graphically in Figures 5.10 and 5.11.

These benchmark results show that Mendeleev's auto-generated code consistently outperforms the hand-written Crucible topology by as much as 1.4×, without any programming or engineering expertise from the user. This somewhat counter-intuitive result is a side-effect of the additional compile-time knowledge that Mendeleev infers about the input and output tuples. Hand-written Crucible code has to pass and validate much wider tuples, containing all of the fields generated by the base PE being used. Mendeleev, by contrast, is able to make stronger assumptions about the fields required at each stage; the synthetic parent Crucible PE that Mendeleev generates (described in Section 5.4.1) avoids much of the tuple validation that Crucible must perform on hand crafted PEs, and passes fewer fields at each stage.

An equivalent analytic, hand-written and hand-tuned for each runtime, outperforms Mendeleev by a maximum of 1.3× in these experiments. Furthermore, the latency on a per-tuple basis remains low, with a variance of between 10^{-3} and 10^{-5}. The relative speedup of Mendeleev to Crucible and a manually written topology on each runtime environment is detailed in Table 5.4.

Filtered Native Analysis

This final set of experiments examines the performance of the Mendeleev-generated native code executing across all three supported runtimes simultaneously. Four variants of the filtered analytic are used: (i) Mendeleev-generated



(a) Standalone runtime

(b) Spark runtime

(c) Streams runtime

(d) Streams Only

(e) Streams + Iterators

Figure 5.10: Execution time for each runtime mode and code type. NB: Charts (d) and (e) have no Crucible implementation.

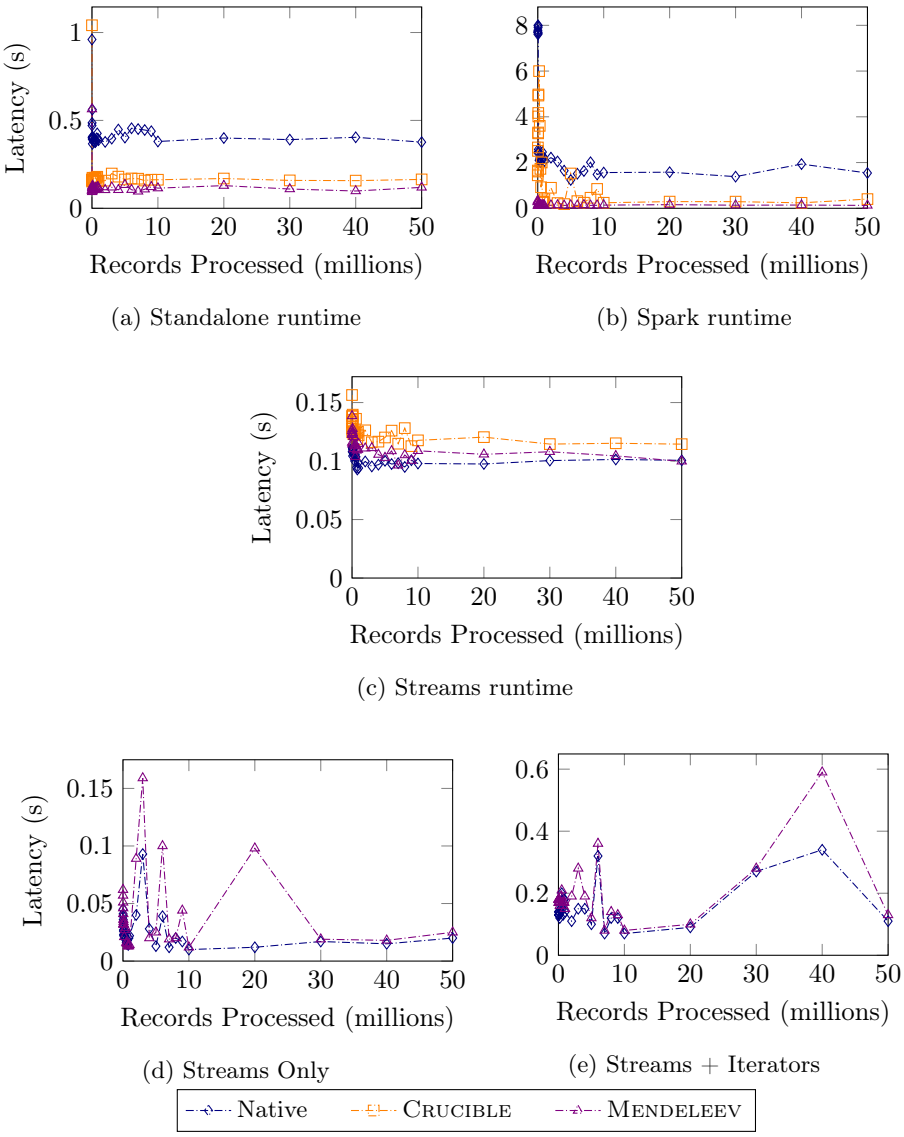


Figure 5.11: Execution latency for each runtime mode and code type. NB: Charts (d) and (e) have no Crucible implementation.

Environment	Mendeleev vs Crucible	Manual vs Mendeleev
Standalone	1.12×	1.30×
Spark	1.39×	1.15×
Streams	1.10×	1.22×

Table 5.4: Relative speedup of Mendeleev to Crucible and hand-written code over Mendeleev.

SPL, pulling all data out of Accumulo and processing it entirely in InfoSphere Streams; (ii) An equivalent Streams-only hand-written analytic; (iii) Mendeleev-generated SPL with Accumulo Iterators to perform the GeoIP and BoundingBoxFilter steps; and (iv) An equivalent hand-written Streams with Accumulo Iterators implementation. The first two variants perform the entire work of the analytic in Streams, while the latter two implementations push the GeoIP and bounding box filtering work into the Accumulo Iterator, and perform the clustering calculations in Streams.

The performance gap between the auto-generated and hand-written code is smaller here than when Crucible is used; on average, Mendeleev's code is only 1.1× slower than the equivalent hand-written implementation. The full results for both makespan and per-tuple latency are shown in the latter half of Table 5.3 and Figures 5.10 and 5.11. These results are summarised in the relative speedup of Mendeleev to these hand-written implementations in Table 5.5.

Environment	Native vs Mendeleev
Streams Only	1.12×
Streams + Iterators	1.09×

Table 5.5: Relative speedup of hand-implemented native runtimes over Mendeleev.

In addition to assessing the performance of Mendeleev, these results also highlight the value of a hybrid approach to analytic execution: the hybrid Streams-Iterator approach is at least 1.5× faster than a pure streaming solution. This performance increase is not as a result of Accumulo Iterators being inherently

faster than Streams, but rather through the reduction in data passed over the network, and the extra parallelism in Accumulo's Iterator execution model.

5.7 Summary

This chapter has documented: (i) A new abstract model for the assembly and execution of hybrid analytics, based on a semantically rich type system; (ii) A novel approach to goal-based planning using this model, which requires little engineering expertise from the user; (iii) A mechanism for performant, scalable code generation for these analytics, integrating data across heterogeneous on- and off-line platforms; (iv) An implementation through a system called Mendeleev; (v) demonstration of the applicability of this technique through a series of case studies, where a single interface is used to create analytics that can be run simultaneously over on- and off-line environments; and (vi) Performance benchmarking that shows that Mendeleev-generated analytics offer runtime performance comparable with hand-written code.

Crafting scalable analytics in order to extract actionable business intelligence is challenging. It requires both domain-level and technical expertise; experience of tuning and scaling, and supporting tools for analytic composition, planning, code-generation and effective deployment. Few frameworks exist that provide end-to-end solutions that address these challenges.

The research presented in this chapter builds on the wishful-search concept behind MARIO (introduced in Section 3.4), yet at the same time allows the discovery and composition of novel analytics. It is the first documented approach to target the execution of automatically generated hybrid analytics in heterogeneous compute environments. The performance penalty over hand-written and tuned analytics has been shown to be a maximum of 1.3× in the included experiments; an acceptable cost for an automated framework of this type.

CHAPTER 6

Speculative Execution of Analytic Workflows

Chapters 3-5 have introduced a variety of techniques, both novel and from existing literature, which permit different types of user to compose analytics for deployment on a number of scalable data-intensive compute architectures. These techniques often suffer a considerable latency between the completion of the user's design process and delivery of the first set of results from the composed application. In some cases this is due to start-up costs associated with the analytic framework [84], while in others the complexity of the analytic itself is to blame. This latency between composition of an analytic and delivery of its results can be minutes or even hours – a considerable delay for users attempting to explore their data through analysis, or who require interactive results.

One common approach to mitigating this latency is through traditional software optimisation; whether of the user-generated code, or of the underlying framework. This form of optimisation delivers varying degrees of improvement, but will always have its limits – and costs. Such optimisations are time consuming to implement, and typically significantly increase the complexity of the optimised codebase.

The research presented in this chapter makes use of a high-level analytic composition tool, backed by a catalogue of composable analytic components (as in Figure 6.1). Many existing approaches aim to improve cluster utilisation through code optimisation and improved job scheduling: however this rarely results in complete utilisation, and production environments often have unused compute capacity [60, 122]. Instead of attempting to make an individual analytic execute faster, this chapter describes a novel approach to speculatively compiling and deploying analytics in order to make use of spare cluster capacity. This

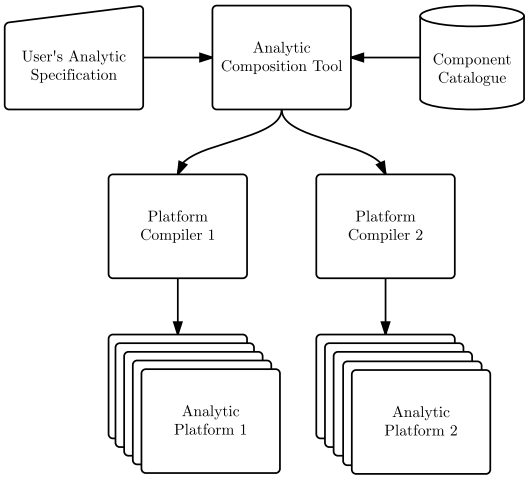


Figure 6.1: Architecture of an analytic workflow composition tool.

approach employs statistics-based heuristics and automated reuse of deployed analytic components in order to improve the response time in deployment of complex analytic workflows.

The remainder of this chapter is structured as follows: Sections 6.1 and 6.3 detail the approach taken in this research and the decision-making policies tested, while Section 6.4 discusses a number of real-world deployment considerations arising from customer analytics. Section 6.5 describes a detailed evaluation of the speculative execution system in this chapter. Finally, Section 6.6 summarises the research presented in this chapter.

6.1 Approach

In this research, we present a novel approach to speculatively compiling and deploying dynamically assembled analytic workflows in order to reduce the latency between a user's request and the delivery of the first results from their analysis. This research collects statistics on the analytics users create, and uses these in a set of heuristic policies to predict the analytic a user is intending to create, while they are still designing that analytic. Once a prediction is made, code may be generated; compiled; and deployed to start generating results

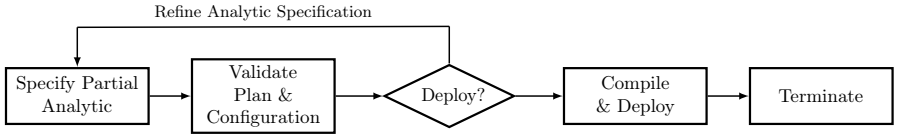


Figure 6.2: Model control flow of an existing analytic assembly system

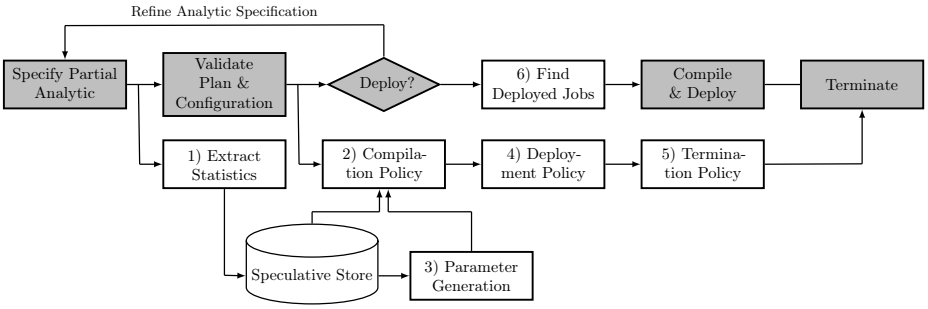


Figure 6.3: New analytic assembly control flow with Speculative Plugin (existing components shaded)

for that analytic on an entirely automated basis. Common sub-components of analyses are reused where possible to further reduce time-to-insight.

This middleware-based approach is agnostic to the runtime framework in use, and is demonstrated in this chapter using both IBM InfoSphere Streams and Apache Pig. This research requires no modifications to the runtime framework, nor changes to the analytic components themselves: all modifications are performed within the analytic assembly system. This chapter refers to the speculative extensions to such a system as a "Speculative Plugin".

The approach taken by this research makes certain assumptions about the control flow in the target assembly system. This flow, illustrated in Figure 6.2, sees the user provide the system with a specification for a (not necessarily complete) analytic. This specification is validated by the assembly system, potentially collecting further user-supplied configuration parameters. If the user is not satisfied with the analytic's current state, they may iteratively refine the specification they supply to the assembly system. When the user requests deployment of this plan, this research assumes that a directed acyclic graph of components is generated, and passed on to a compilation and deployment

step. The nature of the work done during the compilation step depends on the framework the job in question targets. For example, Apache Pig requires no pre-compilation step (the Pig script is compiled upon job submission), while InfoSphere Streams invokes a complex optimising compiler to transpile SPL into C++ and compile that C++ to object code. After deployment, the resulting job will run until user cancel or natural termination of the job.

This research extends the base model to include (Figure 6.3):

1. A component to observe all user interactions with the system and *Extract Statistics* on that basis;

2. A further observer of all valid plans for a given specification, which passes them into a *Compilation Policy*. This policy works to predict the set of possible analytics the user might deploy (Section 6.3.1);

3. A *Parameter Generation Policy*, which generates compile-time parameters for jobs (Section 6.3.2);

4. A *Deployment Policy*, responsible for considering which of these compiled jobs would be valuable to deploy (Section 6.3.3);

5. A *Termination Policy*, which works to ensure that old, unused, and less-valuable sub-jobs are terminated (Section 6.3.4); and

6. A modification to the deployment process which finds and reuses existing jobs or sub-jobs for a user's deployment request (Section 6.3.5).

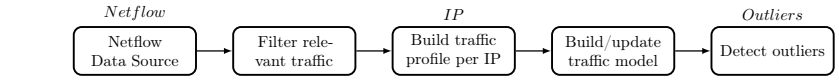
This model results in user-visible performance improvements provided either (i) the Speculative Plugin successfully begins compilation or deployment of the user's target job while they are still iterating on their specification; or (ii) the job they wish to deploy contains common sub-flow(s) with existing compiled or deployed job(s), in which case sub-component reuse will occur. Note that this model starts "cold"; without any prior knowledge of the plan space or user preferences. It must learn all relevant details about parameters and plans whilst it attempts to generate speculative jobs on-line.

Statistics are updated on-line, each time a user submits a specification for a partial analytic or deploys a job. These statistics include the frequency with which features of specifications are requested, the frequency with which particular components or sub-graphs appear in an assembled analytic, as well as the configuration parameters that jobs are ultimately deployed with. The model of analytic composition used in this research makes the conservative assumption that parameters may alter the generated source code for a given workflow. As a result, the compilation policy outlined above additionally references the parameter generation policy to decide which sets of parameters to compile for a given plan. This on-line approach to learning analytic patterns complicates the design and implementation of policies, but ensures that the Speculative Plugin remains flexible in the face of new usage patterns and unseen workflow graphs.

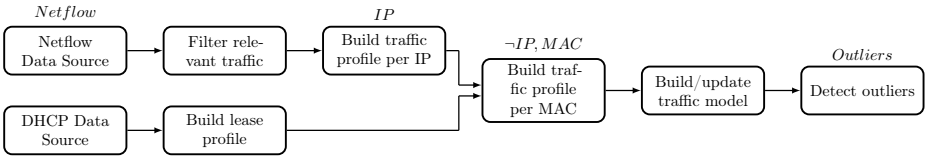
Both the base and extended models described above are compatible with a number of existing analytic assembly systems in the literature. Taverna, for example, offers a flexible model for assembling workflows for bioinformatics, performing *in silico* experiments and analysis using a variety of web services. It incorporates a tag-based search capability and a visual workflow assembly tool, which users employ to specify and compose their analytic. The resulting code (in a language called SCUFL2) is executed on a Taverna server.

Alternatively, some approaches use component identifiers (URIs or tags) and an AI planner or reasoner to assemble workflow components on an automated basis; as in MARIO and the Mendeleev planner detailed in Chapter 5. Such approaches may include multiple possible plans for a given specification: the plan validation stage described above may therefore include user input to select the desired plan from a range of options.

To illustrate the assembly of analytics as a workflow of components, Figure 6.4 depicts two possible cybersecurity analytics inspired by the evaluation analytics used in Section 6.5, as assembled by the MARIO system. In this example, the analytic components are depicted by rectangles and the dataflow connections between them by arrows. The analytic in Figure 6.4(a) uses Netflow data, filtered



(a) Requested tags: $Netflow$, $Outliers$, IP



(b) Requested tags: $Netflow$, $Outliers$, MAC

Figure 6.4: Sample analytic workflows

of irrelevant and special traffic (e.g., from DNS servers) to profile the amount of traffic of every IP on the network; this information is used to continuously update a model of normal traffic (e.g., by using a Gaussian Mixture Model) and detect outlier IPs according to that model. The analytic in Figure 6.4(b) does something very similar, but also brings DHCP lease data into the picture, thus building profiles on a per-MAC-address rather than per-IP-address basis.

Note that the analytic components and the resulting workflows in Figure 6.4 are platform agnostic. In a streaming environment, the *Netflow Data Source* represents a live connection to the Netflow packet stream produced by a network switch, whereas in an offline environment such as Apache Pig, this component can be implemented as a Pig language fragment that `LOAD`s a recorded dataset from disk. In Figure 6.4, the outputs of the analytic components are annotated with *tags* depicted above the analytic components for convenience[1]. In this context, a user's specification of an analytic is a set of tags together with any parameters required by the analytic components assembled into the workflow matching the user tags. Partial specifications such as $Netflow$, $Outliers$ can be matched by multiple analytics – in this case, both analytics in Figure 6.4. The assembly system proposes the "best" combination according to the combined cost and quality of analytic components, or lacking that information, by simply choosing the shortest workflow.

[1]Note here that one component uses a negated tag, $\neg IP$. The full tag algebra in the MARIO system is described in [92]

Lastly, it is useful to note the potential for computation reuse exhibited in this example – the sub-flow ⟨*Netflow Data Source*, *Filter relevant traffic*, *Build traffic profile per IP*⟩ is common to both analytics. If the analytic in Figure 6.4(a) is already running in a streaming environment or has already completed execution in an offline environment, the intermediate data streams or data sets created by this sequence could be reused when the analytic in Figure 6.4(b) is requested by the user.

6.2 Implementation

This approach is demonstrated in this chapter using the IBM research prototype MARIO, which assembles, compiles, deploys, and manages results for analytics for a variety of runtime frameworks annotated using the CASCADE language. While it would be possible to apply this approach to Mendeleev, there would be a significant amount of non-research engineering required to automatically compile and deploy the code generated by Mendeleev, as well as manage the jobs it deploys. These features have been integrated and well tested in MARIO already.

CASCADE annotations include a set of tags for each analytic component, which a user selects from in order to specify the behaviour of the analytic they wish to deploy. MARIO leverages an AI planner for assembly of these analytics, although the research described in this chapter requires only that a catalogue of analytics is available, searchable by tags. Components in MARIO are represented as code fragments that can be assembled and deployed using platform-specific plug-in extensions, which were developed for IBM InfoSphere Streams, Apache Pig, shell scripts and other platforms. Compilation and deployment are multi-threaded, with a priority queue for each ensuring that the newest and highest scoring plans are compiled first. The remainder of this chapter refers to the application of the Speculative Plugin to MARIO as “Speculative MARIO”.

6.3 Policies

Each of the policies described in the model above are first considered in isolation in order to understand how they work individually. Section 6.5 considers the interaction of configurations and combinations of these policies together.

6.3.1 Compilation Policy

When a user submits their partial specification, and the plan search returns its results, the compilation policy is able to start recommending plans for compilation. This is a three stage process. First, the policy enumerates all possible plans for this specification, and all *rooted subgraphs* (subgraphs which start at a data source, and perform zero or more analytical steps on that data) within each plan, accumulating the frequency with which each subgraph appears in the set of possible plans. These subgraphs and frequencies are then sorted such that subgraphs with the highest frequency of occurrence appear first in the output. Subgraphs with equal frequency are sorted in descending order of the number of components in the graph; longer subgraphs are preferred, as they are more likely to result in an (at least partially) successful match when the user finalises their design. This sorted set of subgraphs will naturally contain many plans which are *covered* by (are strict subsets of) others in the enumeration. The final stage of the compilation policy thus discards any so-called *covered* plans which are sorted later in the output, so as to minimise redundant compilation effort.

6.3.2 Parameter Generation Policies

Each of the plans generated above makes use of a parameter generation policy before being added to a compilation queue. This research examines three different parameter generation policies, each of which is limited to producing a fixed number of parameter sets for each plan – this number is governed by a system-level parameter. If a parameter has not yet been observed by the Speculative Plugin, a default value is generated or retrieved from that

component's specification (if possible). If no default value can be generated or retrieved, speculative compilation of this candidate plan is cancelled – the next time a plan with this parameter is deployed by a user, there will be an observation to inform future parameter generation.

The first of these policies, *Random*, simply selects combinations of parameters at random, based on those which have been deployed before. A refinement of this, *Frequency Weighted*, selects parameters randomly with a weight derived from the frequency with which a given parameter set has been observed before. Finally, the *Top Frequency* generation policy generates a list of parameters based purely on the frequency with which they have been deployed, without any stochastic component.

6.3.3 Deployment Policies

Deployment decisions are taken on an on-line basis, as plans for a given search event are generated. This research tests four configurations of deployment. The first two configurations simply *Disable* all speculative deployment, or *Deploy Everything* that is compiled. This provides a set of baseline figures for a naïve approach. A third policy tests deployment of a *Random* selection of compiled jobs, with a system-level tuneable parameter for the probability of deploying a given plan. Finally, the *Top N* policy uses the ordering of plan scores created in the compilation policy to deploy only top-rated plans, limited to a number defined in a system-level parameter.

6.3.4 Termination Policy

In order to ensure that speculatively executed jobs do not overwhelm available cluster resources, a termination policy is used. This employs the concept of *server ticks* as a proxy for time which takes into account the level of activity in the Speculative Plugin: a server tick occurs every time a job is compiled or deployed by the Speculative Plugin. A job is described as having been "used" if either a user has requested its deployment (and not yet requested its termination),

or another job has used data which it publishes. If a job has not been "used", therefore, within the last T server ticks, it is *terminated* immediately and marked as unavailable for reuse. This approach can be considered similar to a Least Recently Used cache invalidation policy.

The behaviour of termination varies depending on the platform on which the given job is deployed. In most environments, termination should cancel any running tasks associated with the job (e.g., streaming processors, MapReduce tasks, Yarn containers, etc.). An extension to the termination policy may further be used to "clean up" unused data stored to shared filesystems (such as the output files from an Apache Pig job, or debug logging from a streaming analytic). The precise number of server ticks to use for these timeouts depends on the size and capacity of the cluster to which jobs are submitted. As such, it is dictated by a system-level tuning parameter.

6.3.5 Sub-Flow Identification & Sharing

In order to facilitate job sharing and sub-flow reuse, it is necessary to define a stable scheme for identifying a subgraph within a flow. This research achieves this by traversing the graph in topological order, accumulating a textual description of the generated code and input / output connections from each component. In the event of a tie in the topological ordering, the traversal uses a stable solution for tie-breaking. For each intermediate output in the plan's flow, the accumulated description is hashed using SHA-256: the resulting digest is used as an identifier for that flow.

Once a job is generated in the Compilation Policy, an *Export Selection Policy* adds nodes to the flow for platform-specific export implementations. These exports indicate that results from this portion of the analytic should be made available to future subscribers. The specific implementation of this loosely framed requirement depends on the runtime model employed in the target framework. For example, in IBM InfoSphere Streams, an `Export` processing element is included in the graph, to export a live stream of results; no caching occurs,

and the cost of this export operation is near-zero. In Apache Pig, this export operation must persist the full result-set of this portion of the analytic to the underlying filesystem (typically HDFS). This is not without cost, but has the advantage of making the full set of partial analysis available for future re-use – not just results for the data analysed while both the "publisher" and "subscriber" job are running.

When a plan is passed to the deployment process, it seeks the longest rooted subgraph which has results available, or failing that the longest which has already been compiled. These existing nodes are then cut from the new plan, and replaced with a matching Import node, to make use of the results from the speculatively prepared flow. When deployment for such a cut flow is requested, each of its transitive dependencies must first be deployed if results are not already available.

6.4 Deployment Considerations

In real-world deployments, there are a number of engineering-related considerations not addressed above. Often, some components of an analytic have side effects on external systems; e.g., writing results to an external store, or altering the configuration of connected hardware. It is not desirable that these external systems should be affected by speculatively executed analytics, however it is not plausible to reliably isolate them in the presence of arbitrary analytic code. Furthermore, some components may alter their behaviour (e.g., to change database authorisations) based on the user who launches it, and thus must not be shared between jobs for multiple users.

Speculative MARIO proposes an engineering solution to these problems based around CASCADE annotations. When components are added to the MARIO system, they may be annotated to indicate that no sharing of this component may take place. These annotations add a constraint to Speculative MARIO, indicating that these components may be present in speculatively *compiled* jobs,

but that jobs containing these components may not be speculatively *deployed*. If Speculative MARIO encounters such a component, it will be able to both compile and deploy the rooted subgraph of any components up to that point in the plan graph. Listing 6.1 below gives an example of a component annotated in CASCADE to indicate it may not be speculatively deployed.

```
/#*
@type           "spl"
@title          "Send user alerts by email"
@tags           UserAlert Output Email
@speculative    "no-deploy"
*#/
component EMailUserAlerts(input AlertStream) {
    // Component SPL code omitted for brevity
}
```

Listing 6.1: CASCADE annotation specifying that a component may not be speculatively deployed.

An alternative solution to this problem would be to add *gateway* components to the deployed job. These could be configured to only enable a flow of data into a given component (whether by disabling the stream in a Streams topology, or delaying a processing stage in Pig or Bash) when the user requests its deployment. In this way, a Speculative Plugin may submit the entire speculative job to the runtime, and begin its processing, without impacting external systems.

6.4.1 Alternative Deployment Scenarios

While this research has been demonstrated as an optimisation of the performance of the interactive MARIO user experience, there are other deployment scenarios in which it may be valuable. For example, if early speculative results can be obtained for an analytic before the user completes their planning, it is possible to present them with a live results view alongside their planning session: as they modify parameters, or add tags to their search, the impact of these changes can be demonstrated interactively. In an exploratory data analysis context, this could significantly improve a user's ability to gain understanding of the nature of their data and the analytic they are designing. For example, the case study

presented in Section 5.5.2 sees a user inspecting the analytic workflow generated by the MENDELEEV planner, and deciding to refine their bounds. With this interactive result presentation in place, the user's decision can be informed by the actual results created by the analytic, not just MENDELEEV's visualisation of the generated workflow. In an exploratory data analysis context, this could significantly improve a user's ability to gain understanding of the nature of their data and the analytic they are designing.

A second context where this research applies, which is not directly explored in this thesis, is for hypothesis testing (as described by Riabov, Sohrabi, et al. [93, 101]), which would see Speculative MARIO used instead by a non-human agent – this has the potential for deeper collaboration between the AI systems. For example, the scoring of plans in the compilation policy may be informed by the hypotheses under test – or the testing agent may use information about currently available speculative jobs to decide which hypothesis to test next.

6.4.2 Policy Design

When deploying this speculative approach against real-world systems, policy design may be influenced by differences in the behaviour and capabilities of the target framework. For example, when Speculative MARIO is deployed against the Apache Pig framework, there is almost no compilation time to account for: many more Pig scripts can be generated and speculatively prepared during the users' planning session than for a compiled language like SPL. By the same token, a Streams job may begin processing and delivering results, once deployed, far quicker than a Pig script which must process its dataset in full before delivering any results at all. Deployments in homogeneous systems, e.g. a purely Apache Pig environment, may find that some policies can be tuned to better suit that single platform. This possibility has not been explored in this thesis: as demonstrated in Section 6.5, the set of policies and heuristics detailed in this research can be applied successfully to both streaming and offline analysis.

6.5 Performance Evaluation

In order to evaluate both the general approach and the specific policies presented above, this research makes use of a *user simulator*. In each iteration of the simulator, it decides on a *target analytic* which it intends to compose. It then generates a series of requests to the MARIO web interface in order to emulate the behaviour of a user, refining the requirements of their analytic specification. Once the simulator generates the target analytic, it requests deployment of the job and awaits the return of results, multiple times. This process is instrumented in order to collect timing information on the code generation, compilation, job launch time, and the time for the first results to return.

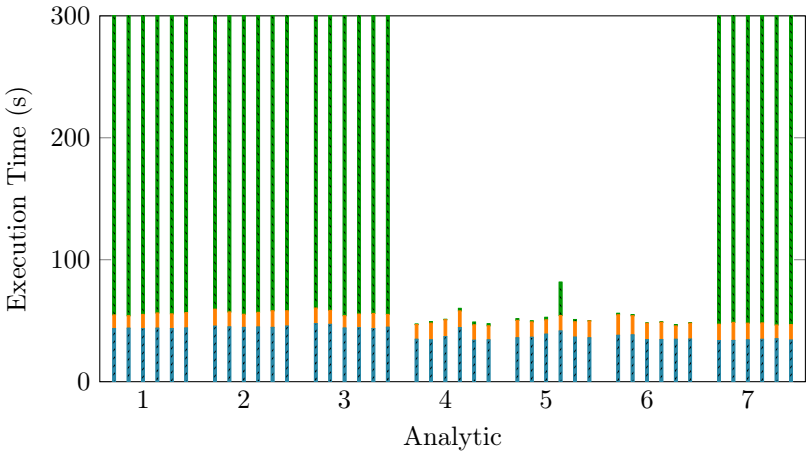
This simulator was used to benchmark MARIO without the Speculative Plugin in place. A collection of 32 customer analytics, from three different problem domains (healthcare, cyber security, and manufacturing) and with two different target platforms (IBM InfoSphere Streams and Apache Pig), were used in the benchmark. The healthcare analytics are provided with ECG (electrocardiograph) data, and other sensors from hospital beds, in order to predict critical care incidents before they happen. The cyber security analytics use network probe data to model gaussian mixture models of "normal" activity on the network (DHCP probes, DNS resolution, etc.), and detect hosts which behave outside of this normal, reporting on them for further investigation (akin to the sample analytics in Figure 6.4). Finally, the manufacturing data comes from a CPU fabrication plant, using quality control metrics from various stages on the production line to predict the yield of CPUs on a given silicon wafer – these data can be used to recycle wafers which will likely have a high failure rate before completing manufacture and test.

These results can be seen in Figure 6.5, showing the times for code generation and compilation; deployment; and collection of the first results from the analytic (if such results were generated within a 5 minute timeout). Each analytic was composed and deployed six times. As the existing implementation does not reuse

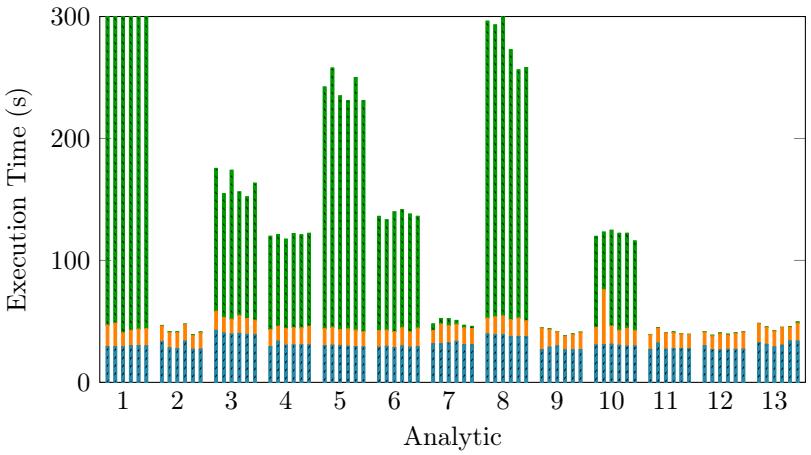
either sub-flows or complete analytics, no improvement in deployment time is evident across repeated runs.

These non-speculative results additionally demonstrate that, in spite of considerable variety in details of the analysis and problem domain, there is a consistently high cost to compilation of the generated streaming analytics; typically more than 40 seconds. Depending on the details of the analysis performed, around 25% of the jobs returned results in under a second: the rest either exceeded the timeout, or returned results within minutes of the launch. Results differed noticeably in the Apache Pig tests; there is no compilation step to speak of, only code generation for the Pig script. Each of the analytics in this test suite returned results in around 5–11 seconds.

The same suite of instrumented customer analytics and user simulator were used to collect timing information for each of the policies discussed in Section 6.3. In order to compare the performance of these policies, the Speculative Plugin was started from a cold state, and a trace of 60 analytic composition sessions launched against it sequentially. Each test uses the compilation policy described in Section 6.3.1, one parameter generation policy from Section 6.3.2, and one deployment policy from Section 6.3.3. These configurations are described in these results as *Deployment Policy/Parameter Policy*.



(a) Critical care ECG analysis (InfoSphere Streams).



(b) Cyber security attack detection (InfoSphere Streams).

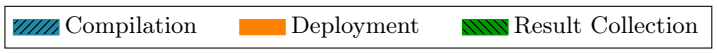
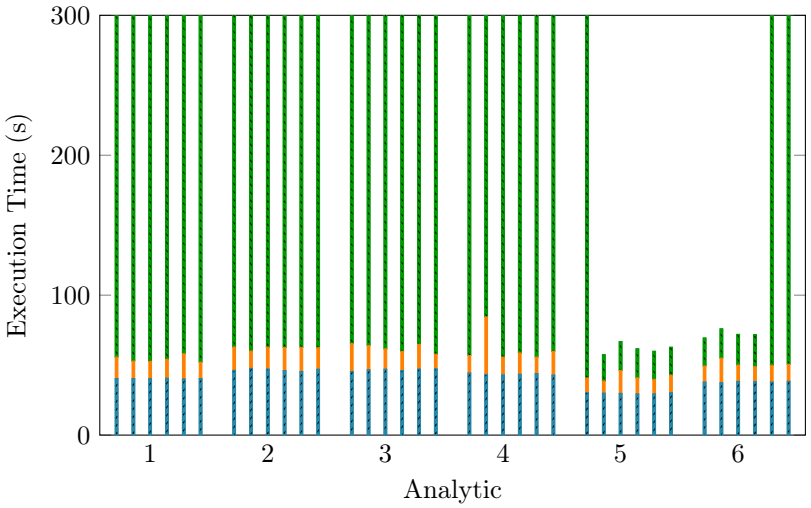
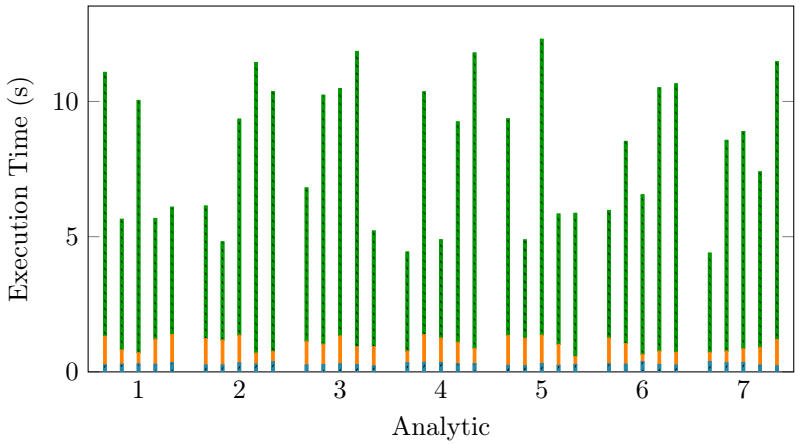


Figure 6.5: Time taken for repeated compilation, deployment, and collection of first results for real-world analytics through MARIO (5 minute timeout on results collection)



(c) CPU fabrication defect rate analysis (InfoSphere Streams).



(d) Offline cyber security attack detection (Apache Pig).

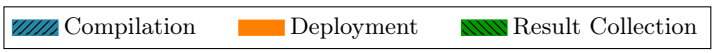


Figure 6.5: *(contd.)* Time taken for repeated compilation, deployment, and collection of first results for real-world analytics through MARIO (5 minute timeout on results collection)

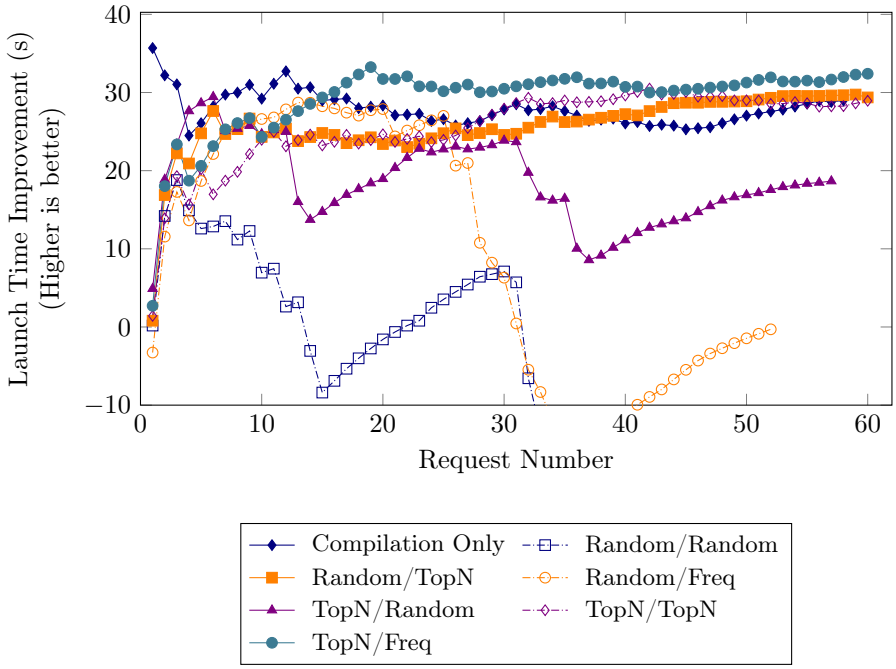


Figure 6.6: Moving average of improvement in job launch times with the Speculative Plugin. Y-Axis clamped at -10 seconds.

Figure 6.6 shows a moving average of the *improvement* in job launch time for these analytics (that is, the difference between the time taken to launch each analytic with and without the Speculative Plugin) as the policies warm up. These results show a rapid warm-up to a steady state (in around 10 requests) for most of the policies. The Random deployment policy offers highly unpredictable results, often resulting in actually increased job launch times (the Y axis of this chart is clamped to -10 seconds for clarity: the Random policies have been observed causing as much as an 80 second increase in launch times over the base results presented in Figure 6.5). This is a result of the extra load this implementation of the Speculative Plugin puts on the MARIO job deployer: there is an internal deployment queue which is saturated with less useful jobs. The most consistently successful configuration of policies here appears to be Top N Deployment, and Frequency-Weighted Random Parameter Generation, providing some validation of the decision process described in Section 6.3.1.

Figure 6.7 summarises these improvements for each policy, and additionally

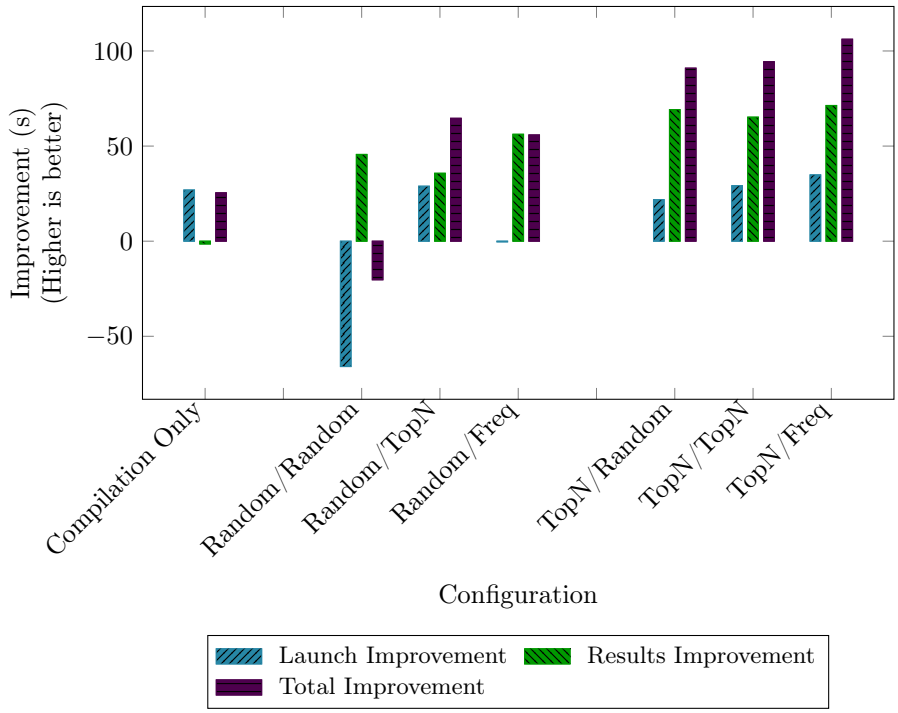


Figure 6.7: Average improvement in launch times and result collection times for each policy

presents the average improvement in results collection time for each configuration. These aggregated results demonstrate that while most policy combinations resulted in a net positive "Total Improvement", the selection of parameter generation strategy has a noticeable impact on the size of this improvement. The deployment strategy appears to be less significant, provided some form of speculative deployment is enabled. The heuristic approaches presented in this chapter consistently outperform random selection: the best results are obtained by using the Frequency Weighted parameter generation policy, and the Top N deployment policy (TopN/Freq).

		Application Suite									
		CPU Defects		Critical Care		Cyber Security		Offline Cyber Security		**Averages**	
		Count	Percentage	Count	Percentage	Count	Percentage	Count	Percentage	**Count**	**Percentage**
Total Requests	Requested Flows	150	N/A	150	N/A	280	N/A	213	N/A	193.3	N/A
	Requested Components	870	N/A	450	N/A	1045	N/A	975	N/A	788.3	N/A
Compilation	Full Hit	95	63.3%	92	61.3%	194	69.3%	N/A	N/A	127.0	65.7%
	Miss (In Progress)	47	31.3%	51	34%	67	23.9%	N/A	N/A	55.0	28.4%
	Miss (Complete)	8	5.3%	7	4.7%	19	6.8%	N/A	N/A	11.3	5.9%
Deployment – Full Flow	Full Hit	45	30%	49	32.7%	105	37.5%	91	42.7%	72.5	36.6%
	Partial Hit	43	28.7%	37	24.7%	70	25%	74	34.7%	56.0	28.2%
	Miss (In Progress)	53	35.3%	55	36.7%	84	30%	35	16.4%	56.8	28.6%
	Miss (Complete)	9	6%	9	6%	21	7.5%	13	6.1%	13.0	6.6%
Deployment – Components	Full Hit	306	35.2%	144	32%	372	35.6%	489	50.2%	327.8	39.3%
	Partial Hit	65 / 126	7.5% / 14.5%	48 / 106	10.7% / 23.6%	93 / 274	8.9% / 26.2%	121 / 292	12.4% / 29.9%	81.75 / 199.5	9.8% / 23.9%
	Miss (In Progress)	380	43.7%	171	38%	321	30.7%	141	14.5%	253.3	30.3%
	Miss (Complete)	58	6.7%	29	6.4%	78	7.5%	53	5.4%	54.5	6.5%

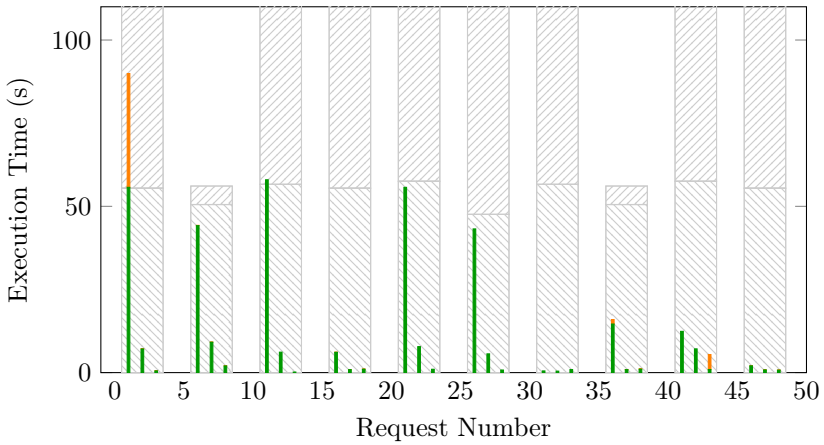
Table 6.1: Hit rate for each application suite, detailing full hits, partial hits, and misses for both full flows and on a per-component basis.

Table 6.1 presents the results of a hit-rate analysis over each of these application suites for this optimal set of policies. Each set of results (with the exception of Compilation, which has no concept of a Partial Hit) is divided into up to four categories:

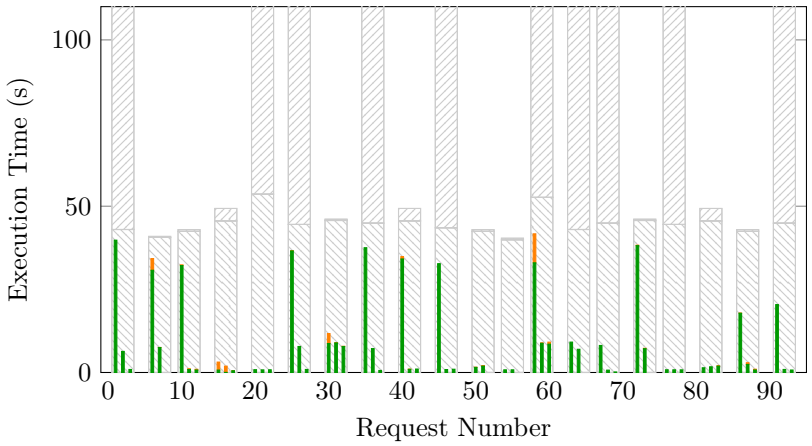
- **Full Hit**: The proportion of requests which were speculatively computed in their entirety;
- **Partial Hit**: Those requests in which a subgraph within the flow had been speculatively computed, but the remainder must be computed;
- **Miss (In Progress)**: Indicates requests for which speculative computation had begun, but which were not complete by the time the user submitted their deployment request;
- **Miss (Complete)**: Requests which the Speculative Plugin failed to predict in time.

The compilation results indicate that approximately 2/3 of requests hit the compilation cache successfully. With additional compilation resources available, results suggest this could go as high as 94%, due to the proportion of compilation requests successfully predicted but not completed. The hit rates for deployment are somewhat lower, due to constraints on the amount of cluster resource available to Speculative MARIO. These indicate a little over a 1/3 full hit rate, with a further 1/4 of requests partially hit.

In order to better understand these partial hits, the final set of data in Table 6.1 shows per-component hit rates rather than per-flow hit rates. The partial hit data in this set of results indicates both the number of components in partial hits which were speculatively deployed, and the number of requested components in these flows which resulted in a partial hit. Across the full set of experiments, there is a consistently under 7% complete miss rate: over 93% of jobs were speculatively predicted and had at least begun to be compiled or deployed prior to the users' request.

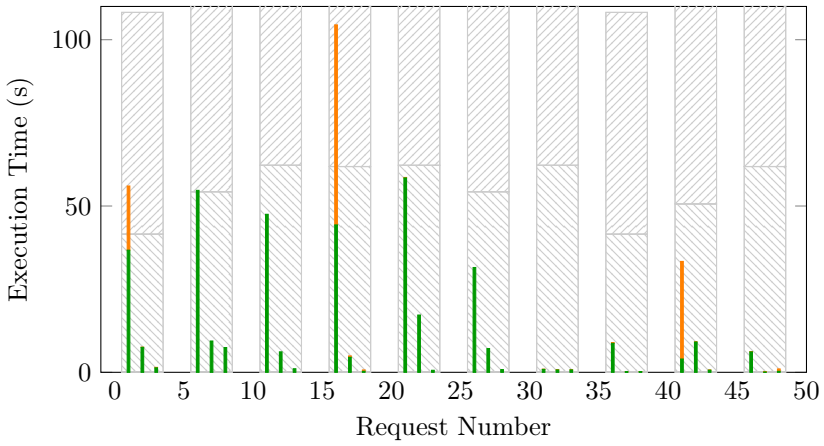


(a) Critical care ECG analysis (InfoSphere Streams).

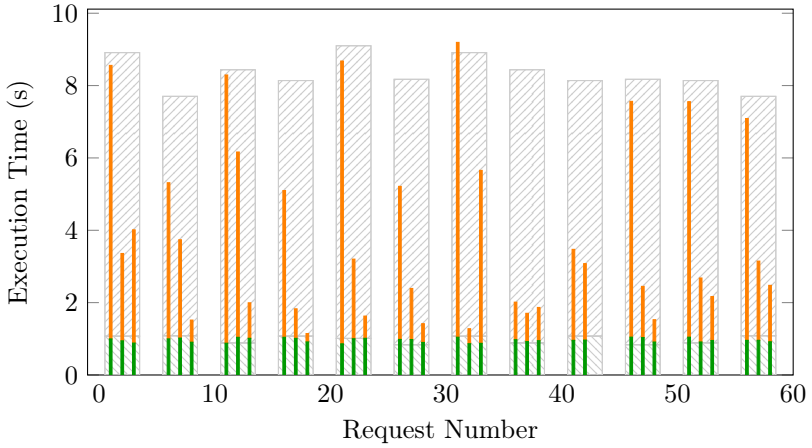


(b) Cyber security attack detection (InfoSphere Streams).

Figure 6.8: Time taken for repeated compilation, deployment, and collection of first results for real-world analytics through Speculative MARIO (5 minute timeout on results collection)



(c) CPU fabrication defect rate analysis (InfoSphere Streams).



(d) Offline cyber security attack detection (Apache Pig).

Figure 6.8: *(contd.)* Time taken for repeated compilation, deployment, and collection of first results for real-world analytics through Speculative MARIO (5 minute timeout on results collection)

Finally, the results of applying the optimal set of policies over these applications are presented in Figure 6.8. The original job launch and results collection times are given for each analytic in the background: in many cases, the Speculative Plugin causes results to be collected before even the Base Launch is complete, representing a considerable improvement in user experience. The

longer the Speculative Plugin is active for, the better its results become: in later runs, even when the system is presented with a novel analytic that it has to do some compilation for, results are still returned in under 0.1 seconds. This is a result of the reuse of common sub-jobs: as partial processing has already occurred for these workflows, the time taken to finish processing is far lower.

6.6 Conclusions

This chapter has detailed (i) the first reported generalised approach to on-line speculative composition, compilation, and execution of data analytics; (ii) A novel collection of modular policies to be used within this framework to alter how its decisions are made. It has described (iii) how real-world deployment considerations inform the implementation of this speculative framework in practice. This implementation is (iv) used to demonstrate the application of this speculative execution model to real customer analytics, considerably improving response times for users of both streaming and batch analytic systems, as well as (v) informing a rigorous evaluation of how each of these policies influence the performance improvement afforded by the system.

This research has demonstrated that existing systems for composing analytic workflows are capable of predicting the analytics a user is attempting to generate based on only partial specifications of their target analytic. This predictability can be exploited to speculatively generate, compile, and begin execution of such workflows without additional user input. This speculative execution has been shown to significantly reduce the users' perceived latency in such a workflow composition system by exploiting spare compute capacity in production environments: at worst, runtimes are not negatively impacted, and at best results are available with sub-0.1 second latencies.

CHAPTER 7

Discussion and Conclusions

This thesis has explored a number of issues arising out of the explosive growth of analytical techniques, requirements, and frameworks in recent years. Much of the existing literature in this area has focused on the optimisation of a single class of algorithms on a single runtime framework, either by improving the underlying framework, or through the introduction of a novel abstraction, programming model, or domain specific language (DSL). This thesis has described and proposed solutions to some of the challenges in navigating the available architectures for a particular analytic workload, as well as in finding and training expert users to employ these systems.

Specifically, Chapter 4 introduced the first reported DSL to target execution on both on-line (streaming) and off-line (bulk) analytical frameworks with equal precedence. There has been prior work in this area to take offline analytics and re-engineer them for execution in streaming environments [33, 45, 58]. Other approaches define frameworks from the ground up for analytic execution (e.g. Spark [123] or Cascading [16]), adding support for a streaming execution model. IBM's DEDUCE [64] is the closest approach in existing literature to that taken in Chapter 4; it used SPADE (a predecessor to SPL) to define MapReduce jobs, providing some commonality in the language used to write analytics. This chapter instead described the first DSL with a common execution model for analytics which is applied to provide consistent semantics across a collection of runtime implementations. The DSL, called Crucible, permits a user to craft a single analytic, and execute it equivalently over a number of data sources and runtime models. In the described implementation, these analytics may be executed using IBM InfoSphere Streams, Apache Spark, and Apache Accumulo, as well

as in a local testing mode – wherever the user's data is available. It includes a novel framework for managing cell-level security labels, and automatically propagating labels through the execution path taken by each datum. This chapter additionally presented an evaluation of Crucible's performance across the suite of runtime implementations, and discussed framework optimisations that resulted in a typical 14× performance gap when compared to hand-tuned native code.

In Chapter 5, this thesis explored an alternative approach to composing analytics for execution both on- and off-line. It described Mendeleev, a goal-based planning engine using a model of analytic behaviour based on transformations of a semantically rich type system. Mendeleev was applied in this thesis to on-line, off-line, and hybrid analytic planning using a collection of case studies taken from the domains of telecommunications and image analysis. These case studies demonstrate automatic analytic code generation for Crucible, IBM InfoSphere Streams, Apache Accumulo Iterators, and Apache Spark (which may be executed in local or distributed mode), in addition to visualisation code based around the JavaScript frameworks Meteor and D3. The results presented in this chapter demonstrated performance of the generated analytics which is comparable to hand-tuned native code, as well as interactive performance and scalability of the planning engine itself. Existing approaches to the automated composition of code prior to Mendeleev were often in the realm of web mashups [30, 119]. Some use hierarchical planning approaches to generate code [85, 98, 112] – much like Mendeleev, many such systems answer queries by satisfying the preconditions for executing composable components. Of particular relevance to Mendeleev is IBM's MARIO [92], which builds on SPPL, the Streaming Processing Planning Language [90, 91]. IBM characterises MARIO as offering *wishful search*, which a user drives by entering a set of goal tags. In practice it is rare for MARIO to create a novel or unforeseen solution to a problem. Mendeleev builds on the wishful-search concept behind MARIO while allowing for the discovery and composition of novel complex analytics, using a higher-level granular model of

analytic behaviour, utilising existing techniques from AI planning.

Finally, Chapter 6 described the first reported use of automatic speculative composition, compilation and execution of analytic workflows. It made use of the flexibility of a goal-based planning engine to significantly reduce the time to insight; that is, the perceived latency between the user submitting a job for execution, and that job beginning to return useful results. This chapter made use of IBM's Automated Analytics Composer (built on the MARIO planner) to demonstrate its abstract model for speculative execution, due to the maturity of the engineering behind its runtime orchestration capability. Within the framework of this implementation, this thesis has evaluated a collection of strategies and configurations for each of the decision points in Speculative MARIO. These strategies were shown to improve the apparent performance of the IBM Automated Analytics Composer by over 100× using spare production cluster capacity – in many cases, delivering results for a user's analytic before they complete its design and deployment. In spite of the breadth of previous study of speculative execution [21, 40, 57, 100, 121], no attempt has been made to use it to improve the performance of analytic workloads by utilising spare cluster capacity. Unlike existing approaches in the literature, the research in Chapter 6 speculatively generates coarse-grained tasks for execution based on predictions of the analytic a user *intends* to deploy. Further, it aggressively caches and shares results of sub-components in the workflow in a platform-sensitive manner. These capabilities combine to create considerable improvements in the perceived performance of the user's data analytics platform.

7.1 Limitations

The work in Chapters 4 and 5 of this thesis has been benchmarked and demonstrated using a limited set of exemplar applications. These applications are designed to be similar to real-world workloads, although do not come from actual enterprise deployments: unfortunately, such applications are typically

commercially sensitive and not suitable for publication. There is generally a lack of high-grade analytic benchmark suites with representative workloads for execution in on-, off-line, *and* hybrid contexts in the literature. Most existing benchmarks [29, 42, 52] target bulk analysis on Hadoop only (with the notable exception of the recently published BigDataBench [111], which offers multiple analytics, each with an associated runtime), aiming to compare implementations of the Hadoop runtime. None of these benchmarks compare the execution of the analysis encoded in the benchmark on different paradigms. As a result, this thesis was compelled to use its own exemplar applications and benchmarks, based on the author's experience of analytics "in the wild". These exemplar applications are selected to cover a sufficiently diverse range of analytic requirements to be able to support the claims made in this thesis.

Even with these benchmarks, it is challenging to objectively and holistically evaluate Crucible and Mendeleev against hand-written code: runtime is only one aspect of the value of an analytic framework. These runtime-based evaluations fail to capture the ease with which an analytic may be expressed, understood by an outsider, and debugged. They do not capture the learning curve of the resulting system, nor its expressivity – some of the evaluated approaches may not be able to express all possible analytics, or the difficulty of doing so may be prohibitive for a user.

Another potential limitation of the work in this thesis is the scope of the implementation of Mendeleev in Chapter 5. While the work correctly generates hybrid analytics for execution on a variety of platforms, it does not attempt to solve associated scheduling problems. As the concrete plan describes a directed acyclic graph of processing elements, there is no ambiguity in the job dependencies. This scheduling is therefore considered an engineering problem, as opposed to a research challenge: frameworks exist in the literature to manage scheduling of workflows within a single runtime (e.g., Apache Oozie [54]). However, no such frameworks are described for scheduling workflows across multiple such runtimes (e.g., for hybrid applications on InfoSphere Streams and using Accumulo

Iterators). This engineering problem should be readily solvable within an existing cross-platform deployment orchestrator such as IBM's Automated Analytics Composer.

Finally, there is a potential limitation in the manner in which Speculative MARIO was benchmarked. While every effort has been made to generate realistic workloads, the benchmark employs a synthetic user simulator, not actual traces of user activity. As existing deployments of the IBM Analytics Composer are on private, protected (often classified) networks, obtaining actual traces of user activity is not possible. The simulator used was designed to stress the system as much as possible: it subjected MARIO to a higher load than any of these existing deployments of the technology. It is therefore believed that the results are sufficiently representative to support the conclusions presented in Chapter 6.

7.2 Applications

The research in this thesis has been demonstrated and tested on a limited but varied set of application areas. However, the frameworks and technologies on which it builds, and which it enhances, are applicable (and in use) across a wide range of industries – some of these include;

- Reconfigurable analysis for the complex variety of sensor configurations and analyses required to enable oil exploration
- Low latency hybrid batch and stream processing for hedge funds
- Analysis of high-dimensional high-variety data for national security applications across both offline and streaming data sources
- Analysis of user profiles across historical and real-time click streams for marketing applications
- Time-series analysis using workflows of custom tools for neuroscience and research.

These applications barely scratch the surface of current deployments of tools like Apache Spark, IBM InfoSphere Streams, Apache Accumulo, MARIO, and so forth. However, each area has the potential to benefit from a converged DSL such as Crucible (particularly those with hybrid analytics, or both batch and stream requirements), or an analytic planning approach like that offered by Mendeleev (those with exploratory analytics, or analytic reconfiguration problems in particular). These can therefore also benefit from the speculative execution approach described in Chapter 6.

This speculative execution approach, in particular, is amenable to generalisation: while the work was demonstrated on IBM's MARIO system, and its applicability to Mendeleev was described, the approach could be valid for any workflow assembly system. All that is required is some library of components, and an interactive workflow design process – whether this is on a vertically integrated platform like BioMOBY/Taverna [81, 113] or any one of the Yahoo! Pipes [87] derivatives [4, 68, 114].

7.3 Further work

The research presented in this thesis is amenable to a number of extensions and further investigation. The work described in Chapter 6 highlighted the value of reusing partial computation across analytic jobs. Pushing this capability down into the Crucible runtime framework could have interesting ramifications for the performance of both hand-written Crucible code, as well as for Mendeleev-generated jobs. In this way, only the subsets of analyses which are different across given topologies must be computed separately, significantly enhancing the overall utilisation of cluster resources. In a similar vein, the capability to subscribe to results published from one job in another will permit a manually specified form of these efficiencies.

In order to improve Crucible's runtime performance further, there may be value in investigating alternative compilation strategies for topologies, in order

to enable their execution of alternative compute architectures. This will enable workload-based optimisation for architecture selection – directly impacting both deployment and system procurement decisions. A component of this work could additionally examine the use of PE fusion (combining multiple PEs into a single operator) and fission (replicating an operator multiple times over subsets of its data inputs) techniques to enhance data parallelism.

One promising area of research is in the automated learning of analytic design patterns. As a MENDELEEV instance is deployed over an extended period of time, analysis of usage patterns may permit the system to recommend to the user analysis for a given data source, or to alter rankings based on those analytics users typically deploy for a given query. These advanced models of analytic design patterns can then feed directly into more advanced modelling of user behaviour for the Speculative Plugin. Taking into account further user attributes, such as business unit or job role, as well as behavioural seasonality (e.g., "Users with role X tend to deploy analytic Y at the start of the month and Z most other mornings") would be an extremely valuable extension of the Speculative Plugin work presented in this thesis.

7.4 Final Remarks

This thesis has presented a number of approaches designed to ease the process of deploying scalable data analytics across a variety of platforms. As requirements and techniques continue to evolve, such approaches will likely increase in prevalence and significance – albeit in increasingly advanced and usable fashions. Many of the themes of this research are expected to pervade future work in this area: empowering domain experts, ensuring scalability, and enabling complex deployments of hybrid analytics.

Bibliography

[1] D. Agrawal, S. Das, and A. El Abbadi. Big data and cloud computing: Current state and future opportunities. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 530–533, Uppsala, Sweden, 2011. ACM.

[2] T. Aihkisalo and T. Paaso. A performance comparison of web service object marshalling and unmarshalling solutions. In *Proceedings of the 2011 IEEE World Congress on Services*, pages 122–129, Washington, DC, USA, 2011. IEEE.

[3] M. Ali. An introduction to Microsoft SQL Server StreamInsight. In *Proceedings of the 1st International Conference and Exhibition on Computing for Geospatial Research & Applications*, page 66, Washington, DC, USA, 2010. ACM.

[4] M. Altinel, P. Brown, S. Cline, R. Kartha, E. Louie, V. Markl, L. Mau, Y.-H. Ng, D. Simmen, and A. Singh. Damia: A data mashup fabric for intranet applications. In *Proceedings of the 33rd international conference on Very large data bases*, pages 1370–1373, San Jose, CA, USA, 2007. VLDB Endowment.

[5] Apache Software Foundation. Apache Accumulo, accessed 2015. URL `http://accumulo.apache.org/`.

[6] Apache Software Foundation. Apache HBase, accessed 2015. URL `http://hbase.apache.org/`.

[7] Apache Software Foundation. Apache Storm, accessed 2015. URL `http://storm.apache.org/`.

[8] Apache Software Foundation. Apache ZooKeeper, accessed 2015. URL `http://zookeeper.apache.org/`.

[9] D. E. Bell and L. J. La Padula. Secure computer system: Unified exposition and multics interpretation. Technical report, MITRE Corporation, ESD-TR-75-306 (MTR-2997), McLean, VA, USA, 1976.

[10] R. Bergmann and Y. Gil. Retrieval of semantic workflows with knowledge intensive similarity measures. In *Case-Based Reasoning Research and Development*, pages 17–31. Springer, New York, NY, USA, 2011.

[11] A. Beygelzimer, A. Riabov, D. Sow, D. S. Turaga, and O. Udrea. Big data exploration via automated orchestration of analytic workflows. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 153–158, San Jose, CA, USA, 2013.

[12] M. Birbeck and S. McCarron. CURIE syntax 1.0: A syntax for expressing compact URIs. *W3C Working Group Note*, 2008.

[13] P. J. Bird. *LEO: The first business computer*. Hasler Publishing Ltd., Wokingham, UK, 1994.

[14] G. Bracha. Generics in the Java programming language. *Sun Microsystems, java.sun.com*, 2004.

[15] D. Brickley, R. V. Guha, and B. McBride. RDF vocabulary description language 1.1: RDF schema. *W3C Recommendation*, 2014.

[16] Cascading Project. Cascading – Platform for big data, accessed 2015. URL `http://www.cascading.org/`.

[17] C. Castillo, M. Mendoza, and B. Poblete. Information credibility on Twitter. In *Proceedings of the 20th International Conference on World wide web*, pages 675–684, Hyderabad, India, 2011. ACM.

[18] F. Chang and G. A. Gibson. Automatic i/o hint generation through speculative execution. In *Proceedings of the Third Symposium on Operating*

Systems Design and Implementation, volume 99, pages 1–14, New Orleans, LA, USA, 1999. USENIX Association.

[19] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[20] T. Cheatham, A. Fahmy, D. Stefanescu, and L. Valiant. Bulk synchronous parallel computing: A paradigm for transportable software. In *Tools and Environments for Parallel and Distributed Systems*, pages 61–76. Springer, New York, NY, USA, 1996.

[21] Q. Chen, C. Liu, and Z. Xiao. Improving MapReduce performance using smart speculative execution strategy. *IEEE Transactions on Computers*, 63(4):954–967, 2014.

[22] Y. Chen, S. Byna, X.-H. Sun, R. Thakur, and W. Gropp. Hiding i/o latency with pre-execution prefetching for parallel applications. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 40, New York, NY, USA, 2008. IEEE Press.

[23] P. Coetzee and S. Jarvis. CRUCIBLE: Towards unified secure on- and off-line analytics at scale. In *Proceedings of the 2013 International Workshop on Data-Intensive Scalable Computing Systems*, pages 43–48, Denver, CO, USA, 2013. ACM.

[24] P. Coetzee and S. Jarvis. Goal-based analytic composition for on- and off-line execution at scale. In *Proceedings of IEEE Trustcom/BigDataSE/ISPA, 2015*, volume 2, pages 56–65, Helsinki, Finland, 2015. IEEE.

[25] P. Coetzee and S. A. Jarvis. Goal-based composition of scalable hybrid analytics for heterogeneous architectures. *Journal of Parallel and Distributed Computing*, 2016. URL `http://doi.org/10.1016/j.jpdc.2016.11.009`.

[26] P. Coetzee, M. Leeke, and S. Jarvis. Towards unified secure on-and off-line analytics at scale. *Parallel Computing*, 40(10):738–753, 2014.

[27] P. L. Coetzee, A. V. Riabov, and O. Udrea. Methods and systems for improving responsiveness of analytical workflow runtimes, November 2016. US Patent 9,495,137.

[28] I. Constantinescu, B. Faltings, and W. Binde. Large scale, type-compatible service composition. In *Proceedings of the IEEE International Conference on Web Services*, pages 506–513, San Diego, CA, USA, 2004. IEEE.

[29] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154, Indianapolis, IN, USA, 2010. ACM.

[30] F. Daniel, C. Rodríguez, S. Roy Chowdhury, H. R. Motahari Nezhad, and F. Casati. Discovery and reuse of composition knowledge for assisted mashup development. In *Proceedings of the 21st International Conference on World Wide Web*, pages 493–494, Lyon, France, 2012. ACM.

[31] T. H. Davenport and J. G. Harris. *Competing on analytics: The new science of winning*. Harvard Business Press, Boston, MA, USA, 2007.

[32] T. H. Davenport and L. Prusak. *Working knowledge: How Publishers manage what they know*. Harvard Business Press, Boston, MA, USA, 1998.

[33] G. De Francisci Morales. SAMOA: A platform for mining big data streams. In *Proceedings of the 22nd International Conference on the World Wide Web companion*, pages 777–778, Rio de Janeiro, Brazil, 2013. International World Wide Web Conferences Steering Committee.

[34] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[35] H. Demirkan and D. Delen. Leveraging the capabilities of service-oriented decision support systems: Putting analytics and big data in cloud. *Decision Support Systems*, 55(1):412–421, 2013.

[36] S. Efftinge and M. Völter. oAW XText: A framework for textual DSLs. In *Proceedings of Eclipse Modeling Symposium at Eclipse Summit*, volume 32, page 118, Esslingen, Germany, 2006.

[37] EsperTech Inc. Esper – complex event processing, accessed 2015. URL `http://www.espertech.com/esper/`.

[38] M. Franklin et al. Mllib: A distributed machine learning library. In *Proceedings of NIPS Workshop on Machine Learning Open Source Software*, Lake Tahoe, NV, USA, 2013.

[39] A. Fuchs. Accumulo – Extensions to Google's Bigtable design. Technical report, National Security Agency, March 2012.

[40] F. Gabbay and A. Mendelson. *Speculative execution based on value prediction*. PhD thesis, Technion – Israel Institute of Technology, 1996.

[41] A. E. Gattiker, F. H. Gebara, A. Gheith, H. P. Hofstee, D. A. Jamsek, J. Li, E. Speight, J. W. Shi, G. C. Chen, and P. W. Wong. Understanding system and architecture for big data. *Journal of IBM research*, 2012.

[42] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen. BigBench: Towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 1197–1208, New York, NY, USA, 2013. ACM.

[43] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *ACM SIGOPS operating systems review*, 37(5):29–43, 2003.

[44] I. Goiri, F. Julia, J. Ejarque, M. De Palol, R. M. Badia, J. Guitart, and J. Torres. Introducing virtual execution environments for application life-

cycle management and SLA-driven resource distribution within service providers. In *Proceedings of the 8th IEEE International Symposium on Network Computing and Applications. (NCA 09)*, pages 211–218, Cambridge, MA, USA, 2009. IEEE.

[45] L. Golab, T. Johnson, J. S. Seidel, and V. Shkapenyuk. Stream warehousing with DataDepot. In *Proceedings of the 35th SIGMOD Conference on Management of Data*, pages 847–854, Providence, RI, USA, 2009. ACM.

[46] Google, Inc. Google Mashup Editor, accessed 2015. URL `https://developers.google.com/mashup-editor/`.

[47] Google Inc. Protocol Buffers, accessed 2015. URL `https://developers.google.com/protocol-buffers/`.

[48] T. J. Hacker, F. Romero, and C. D. Carothers. An analysis of clustered failures on large supercomputing systems. *Journal of Parallel and Distributed Computing*, 69(7):652–665, 2009.

[49] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan. The rise of "big data" on cloud computing: Review and open research issues. *Information Systems*, 47:98–115, 2015.

[50] M. Hausenblas and J. Nadeau. Apache Drill: Interactive ad-hoc analysis at scale. *Big Data*, 1(2):100–104, 2013.

[51] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[52] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The HiBench benchmark suite: Characterization of the mapreduce-based data analysis. In *IEEE 26th International Conference on Data Engineering Workshops*, pages 41–51, Long Beach, CA, USA, 2010. IEEE.

[53] W. S. Humphrey. Why big software projects fail: The 12 key questions.

In D. J. Reifer, editor, *Software Management*. IEEE, New York, NY, USA, seventh edition, 2005.

[54] M. Islam, A. K. Huang, M. Battisha, M. Chiang, S. Srinivasan, C. Peters, A. Neumann, and A. Abdelnur. Oozie: Towards a scalable workflow management system for Hadoop. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, page 4, Scottsdale, AZ, USA, 2012. ACM.

[55] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. Zdonik. Towards a streaming SQL standard. *Proceedings of the VLDB Endowment*, 1(2): 1379–1390, 2008.

[56] C. Ji, Y. Li, W. Qiu, U. Awada, and K. Li. Big data processing in cloud computing environments. In *12th International Symposium on Pervasive Systems, Algorithms and Networks*, pages 17–23, San Marcos, TX, USA, 2012. IEEE.

[57] N. P. Jouppi and D. W. Wall. *Available instruction-level parallelism for superscalar and superpipelined machines*, volume 17. ACM, New York, NY, USA, 1989.

[58] C. R. Kalmanek, I. Ge, S. Lee, C. Lund, D. Pei, J. Seidel, J. Van der Merwe, and J. Ates. Darkstar: Using exploratory data mining to raise the bar on network reliability and performance. In *Proceedings of the 7th International Workshop on Design of Reliable Communication Networks*, pages 1–10, Alexandria, VA, USA, 2009. IEEE.

[59] S.-H. Kang, D.-H. Koo, W.-H. Kang, and S.-W. Lee. A case for flash memory SSD in Hadoop applications. *International Journal of Control and Automation*, 6(1):201–210, 2013.

[60] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An analysis of traces from a production MapReduce cluster. In *10th IEEE/ACM International*

Conference on Cluster, Cloud and Grid Computing, pages 94–103, Melbourne, Australia, 2010. IEEE.

[61] J. Kepner, W. Arcand, D. Bestor, B. Bergeron, C. Byun, V. Gadepally, M. Hubbell, P. Michaleas, J. Mullen, A. Prout, et al. Achieving 100,000,000 database inserts per second using Accumulo and D4M. In *IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Waltham, MA, USA, 2014. IEEE.

[62] M. Kornacker and J. Erickson. Cloudera Impala: Real-time queries in Apache Hadoop, 2012. URL `http://blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real/`.

[63] T. M. Kroeger, D. D. Long, and J. C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *USENIX Symposium on Internet Technologies and Systems*, pages 13–22, Monterey, CA, USA, 1997.

[64] V. Kumar, H. Andrade, B. Gedik, and K.-L. Wu. DEDUCE: At the intersection of MapReduce and stream processing. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 657–662, Lausanne, Switzerland, 2010. ACM.

[65] O. Lassila, R. Swick, et al. Resource Description Framework (RDF) model and syntax specification. *W3C Recommendation*, 1998.

[66] Y.-N. Law, H. Wang, and C. Zaniolo. Query languages and data models for database sequences and data streams. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pages 492–503, San Jose, CA, USA, 2004. VLDB Endowment.

[67] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–15, Seattle, WA, USA, 2014. ACM.

[68] T. Loton. *Introduction to Microsoft Popfly, No Programming Required*. Lotontech Limited, 2008.

[69] M. Loukides. What is data science, 2010. URL `http://radar.oreilly.com/2010/06/what-is-data-science.html`.

[70] H. P. Luhn. A business intelligence system. *IBM Journal of Research and Development*, 2(4):314–319, 1958.

[71] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, Indianapolis, IN, USA, 2010. ACM.

[72] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, et al. OWL-S: Semantic markup for web services. *W3C member submission*, 22:2007–04, 2004.

[73] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., Greenwich, CT, USA, 2015.

[74] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[75] J. W. Mickens, J. Elson, J. Howell, and J. Lorch. Crom: Faster web browsing using speculative execution. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, volume 10, page 9, Boston, MA, USA, 2010.

[76] Z. Nabi, E. Bouillet, A. Bainbridge, and C. Thomas. Of Streams and Storms. *Journal of IBM research*, 2014.

[77] D. S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and

F. Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20(1):379–404, 2003.

[78] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 170–177, Sydney, Australia, 2010.

[79] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical report, EPFL Lausanne, IC/2004/64, 2004.

[80] C. Ogbuji et al. FuXi 1.4: A python-based, bi-directional logical reasoning system for the semantic web, accessed 2015. URL `https://code.google.com/p/fuxi/`.

[81] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, et al. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[82] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. 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, Vancouver, BC, Canada, 2008. ACM.

[83] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve world wide web latency. *ACM SIGCOMM Computer Communication Review*, 26(3):22–36, 1996.

[84] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 35th SIGMOD Conference on Management of Data*, pages 165–178, Providence, RI, USA, 2009. ACM.

[85] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated synthesis of composite BPEL4WS web services. In *Proceedings of the IEEE International Conference on Web Services*, pages 293–301, Orlando, FL, USA, 2005. IEEE.

[86] R. Procter, F. Vis, A. Voss, M. Cantijoch, Y. Manykhina, M. Thelwall, R. Gibson, A. Hudson-Smith, and S. Gray. Riot rumours: How misinformation spread on Twitter during a time of crisis, 2011. URL `http://www.guardian.co.uk/uk/interactive/2011/dec/07/london-riots-twitter`.

[87] M. Pruett. *Yahoo! Pipes*. O'Reilly, Sebastopol, CA, USA, first edition, 2007.

[88] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *The Journal of Supercomputing*, 7(1-2):9–50, 1993.

[89] R. Rea and K. Mamidipaka. IBM InfoSphere Streams: Enabling complex analytics with ultra-low latencies on data in motion. *IBM White Paper*, 2009.

[90] A. Riabov and Z. Liu. Planning for stream processing systems. *Proceedings of the AAAI National Conference on Artificial Intelligence*, 20(3):1205, 2005.

[91] A. Riabov and Z. Liu. Scalable planning for distributed stream processing systems. In *Proceedings of The International Conference on Automated Planning and Scheduling*, pages 31–41, Pittsburgh, PA, USA, 2006. AAAI Press.

[92] A. V. Riabov, E. Boillet, M. D. Feblowitz, Z. Liu, and A. Ranganathan. Wishful search: Interactive composition of data mashups. In *Proceedings of the 17th International Conference on World Wide Web*, pages 775–784, Beijing, China, 2008. ACM.

[93] A. V. Riabov, S. Sohrabi, D. Sow, D. Turaga, O. Udrea, and L. Vu. Planning-based reasoning for automated large-scale data analysis. In *Proceedings of the 25th International Conference on Automated Planning and Scheduling*, Jerusalem, Israel, 2015.

[94] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache Tez: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1357–1369, Melbourne, Victoria, Australia, 2015. ACM.

[95] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications? *ACM SIGARCH Computer Architecture News*, 40(3):440–451, 2012.

[96] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. In *Journal of Physics: Conference Series*, volume 78, page 012022, Boston, MA, USA, 2007. IOP Publishing.

[97] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, Lake Tahoe, NV, USA, 2010. IEEE.

[98] E. Sirin and B. Parsia. Planning for semantic web services. In *Semantic Web Services Workshop at 3rd International Semantic Web Conference*, pages 33–40, Hiroshima, Japan, 2004. Springer.

[99] E. Smith. JVM serializers project, accessed 2015. URL `https://github.com/eishay/jvm-serializers/wiki`.

[100] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[101] S. Sohrabi, O. Udrea, and A. Riabov. Knowledge engineering for planning-based hypothesis generation. In *Proceedings of the Automated Planning and Scheduling (ICAPS) Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, pages 46–53, Portsmouth, NH, USA, 2014.

[102] K. T. Stolee, S. Elbaum, et al. Solving the search for source code. *ACM Transactions on Software Engineering and Methodology*, 23(3):26, 2014.

[103] L. Tassiulas and A. Ephremides. Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. *IEEE Transactions on Automatic Control*, 37(12):1936–1948, 1992.

[104] F. W. Taylor. *Shop management*. McGraw-Hill, New York, NY, USA, 1911.

[105] E. Tejedor and R. M. Badia. COMP Superscalar: Bringing grid superscalar and GCM together. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid. (CCGRID 08)*, pages 185–193, Lyon, France, 2008. IEEE.

[106] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart. Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads, 2011. URL `http://disruptor.googlecode.com/files/Disruptor-1.0.pdf`.

[107] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a MapReduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[108] R. Vanbrabant. *Google Guice: Agile Lightweight Dependency Injection Framework*. Apress Media LLC, New York, NY, USA, 2008.

[109] H. Varian. How the web challenges managers. *McKinsey Quarterly*, 2009.

[110] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.

[111] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, et al. Bigdatabench: A big data benchmark suite from internet services. In *IEEE 20th International Symposium on High Performance Computer Architecture*, pages 488–499, Orlando, FL, USA, 2014. IEEE.

[112] K. Whitehouse, F. Zhao, and J. Liu. Semantic streams: A framework for composable semantic interpretation of sensor data. In *Wireless Sensor Networks*, pages 5–20. Springer, Banff, Alberta, Canada, 2006.

[113] M. D. Wilkinson and M. Links. BioMOBY: An open source biological web services proposal. *Briefings in Bioinformatics*, 3(4):331–341, 2002.

[114] J. Wong. Marmite: Towards end-user programming for the web. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 270–271, Coeur d'Alene, ID, USA, 2007. IEEE.

[115] R. Xin, J. Rosen, et al. Shark: SQL and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 13–24, New York, NY, USA, 2013. ACM.

[116] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on Spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2, New York, NY, USA, 2013. ACM.

[117] Q. Yang and J. Ren. I-CASH: Intelligently coupled array of SSD and HDD. In *IEEE 17th International Symposium on High Performance Computer Architecture*, pages 278–289, San Antonio, TX, USA, 2011. IEEE.

[118] YourKit LLC. YourKit Java Profiler, accessed 2016. URL `https://www.yourkit.com/java/profiler/features/`.

[119] J. Yu, B. Benatallah, F. Casati, and F. Daniel. Understanding mashup development. *Internet Computing, IEEE*, 12(5):44–52, 2008.

[120] Y. Yuan, Y. Wu, Q. Wang, G. Yang, and W. Zheng. Job failures in high performance computing systems: A large-scale empirical study. *Computers & Mathematics with Applications*, 63(2):365–377, 2012.

[121] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of 8th USENIX Symposium on Operating Systems Design and Implementation*, volume 8, page 7, San Diego, CA, USA, 2008.

[122] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user MapReduce clusters. Technical report, EECS Department, University of California, Berkeley, 2009.

[123] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, Boston, MA, USA, 2010. USENIX Association.

[124] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, page 2, Berkeley, CA, USA, 2012. USENIX Association.

[125] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing*, page 10, Boston, MA, USA, 2012. USENIX Association.

[126] Y. Zhou, R. M. Weiss, E. McArthur, D. Sanchez, X. Yao, D. Yuen, M. R. Knox, and W. W. Czech. WebViz: A web-based collaborative interactive visualization system for large-scale data sets. In *GPU Solutions to Multi-scale Problems in Science and Engineering*, pages 587–606. Springer, New York, NY, USA, 2013.

Appendices

APPENDIX A

CRUCIBLE DSL Grammar

```
grammar uk.ac.warwick.dcs.crucible.lang.Crucible with org.eclipse.xtext.xbase.Xbase

generate crucible "http://dcs.warwick.ac.uk/crucible/lang/Crucible"

import "http://www.eclipse.org/xtext/common/JavaVMTypes"

Document:
    package = Package?
    imports += Import*
    pes += PE*
;

// Package declaration
Package:
    'package' name=QualifiedName ';'?
;

// Imports
Import:
    'import' importedNamespace=QualifiedNameOptionalWildcard ';'?
;

QualifiedNameOptionalWildcard :
    QualifiedName '.*'?
;


// PE Definition
PE:
    'process' name=ValidID ('extends' extended=JvmTypeReference)? '{'
        ((('conf'   | 'config')  ':' conf+=ConfigBlock) |
        (       'state'         ':' state+=StateBlock) |
        (('output' ':' outputs+=Output ';'?) | ('outputs' ':' '[' outputs+=Output? (',' outputs+=Output)* ']' ';'?)) |
        (('input'  | 'inputs')  ':' inputs+=InputBlock)) *
    '}'
;

// Configuration
ConfigBlock:
    {ConfigBlock}
    '{' lines+=ConfigLine* '}' | lines+=ConfigLine
;
```

```
43
44 ConfigLine:
45     type=JvmTypeReference? name=ValidID '=' value=XExpression ';'?
46 ;
47
48 // Mutable state
49 StateBlock:
50     {StateBlock}
51     '{' lines+=StateLine* '}' | lines+=StateLine
52 ;
53
54 StateLine:
55     local?='local'? type=JvmTypeReference? name=ValidID '=' value=
           XExpression ';'?
56 ;
57
58 // Stream inputs
59 InputBlock:
60     {InputBlock}
61     '{' lines+=Input* '}' | lines+=Input
62 ;
63
64 Input:
65     source=OutputReference '->' (body=XBlockExpression|super?='super')
66 ;
67
68 // Outputs
69 Output:
70     name=ValidID
71 ;
72
73 OutputReference:
74     pe=[PE] '.' output=[Output]
75 ;
```

Listing A.1: Crucible DSL grammar, expressed using XText's language specification syntax.

APPENDIX B

MENDELEEV Inference Results

Listing B.1: Modelling an SPL (IBM's Streams Processing Language) HTTP Fetch PE in RDF: Inference Results.

```
@prefix mlv: <http://go.warwick.ac.uk/crucible/mendeleev/ns#> .
@prefix pe: <urn:pe://> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix type: <http://go.warwick.ac.uk/crucible/mendeleev/types#> .

rdfs:Class a rdfs:Class ;
    rdfs:subClassOf rdfs:Class .

mlv:type a rdfs:Class ;
    rdfs:subClassOf mlv:type .

mlv:genericType a rdfs:Class ;
    rdfs:subClassOf mlv:genericType ,
        mlv:type .

mlv:pe a rdfs:Class ;
    rdfs:subClassOf mlv:pe .

mlv:spl_pe a rdfs:Class ;
    mlv:runtime mlv:streams ;
    rdfs:subClassOf mlv:pe ,
        mlv:spl_pe .

type:URL a mlv:genericType ,
        mlv:type ,
        rdfs:Class ;
    mlv:nativeCode "java.net.URL" ;
    mlv:parent type:string ;
    rdfs:subClassOf type:URL ,
        type:string .

type:byteStream a mlv:type ,
        rdfs:Class ;
    mlv:nativeCode "java.nio.ByteBuffer" ;
    rdfs:subClassOf type:byteStream .

type:image a mlv:type ,
        rdfs:Class ;
    mlv:nativeCode "java.nio.ByteBuffer" ;
    mlv:parent type:byteStream ;
    rdfs:subClassOf type:byteStream ,
        type:image .
```

```
type:header_list a mlv:type,
        rdfs:Class ;
    mlv:nativeCode "uk.ac.warwick.dcs.mendeleev.pe.http.HeaderList"
         ;
    rdfs:subClassOf type:header_list .

type:http_response a mlv:type,
        rdfs:Class ;
    mlv:nativeCode "java.lang.String" ;
    rdfs:subClassOf type:http_response .

type:mime_type a mlv:type,
        rdfs:Class ;
    mlv:nativeCode "java.lang.String" ;
    rdfs:subClassOf type:mime_type .

_:mlv_type_1 a mlv:type,
        rdfs:Class ;
    rdfs:label "fetch_url generic type parameter" ;
    mlv:nativeCode "java.nio.ByteBuffer" ;
    mlv:parent type:byteStream ;
    rdfs:subClassOf _:mlv_type_1,
        type:byteStream .

_:mlv_param_1 a type:URL ;
    rdfs:label "url" ;
    mlv:genericParameter _:mlv_type_1 .

pe:fetch_url_spl a mlv:pe,
        mlv:spl_pe ;
    rdfs:label "Fetch the contents of a URL" ;
    mlv:input [ rdfs:label "data" ;
            mlv:parameter _:mlv_param_1 ] ;
    mlv:nativeCode "uk.ac.warwick.dcs.mendeleev.pe::FetchURL" ;
    mlv:output [ rdfs:label "Output" ;
            mlv:parameter [ a _:mlv_type_1 ;
                    rdfs:label "body" ],
                [ a type:http_response ;
                    rdfs:label "response" ],
                [ a type:header_list ;
                    rdfs:label "headers" ],
                [ a type:mime_type ;
                    rdfs:label "type" ] ] ;
    mlv:postCondition [ mlv:hasField type:string ],
        [ mlv:hasField type:URL ;
            mlv:provenance mlv:preCondition ],
        [ mlv:provenance mlv:preCondition ;
            mlv:runtime mlv:streams ],
        [ mlv:peUsed pe:fetch_url_spl ;
            mlv:provenance mlv:baseRule ],
        [ mlv:hasField type:header_list ;
            mlv:provenance mlv:output ],
        [ mlv:hasField type:mime_type ;
            mlv:provenance mlv:output ],
        [ mlv:hasField _:mlv_type_1 ;
            mlv:provenance mlv:output ],
        [ mlv:hasField type:byteStream ],
        [ mlv:hasField type:http_response ;
            mlv:provenance mlv:output ],
```

```
    [ mlv:paramType type:URL ;
        mlv:unboundGenericParameter _:mlv_param_1 ] ;
mlv:preCondition [ mlv:hasField type:URL ;
        mlv:provenance mlv:input ],
    [ mlv:provenance rdf:type ;
        mlv:runtime mlv:streams ] .
```

APPENDIX C

MENDELEEV Case Study Library

Listing C.1: MENDELEEV PE library used in case studies.

```
@prefix mlv: <http://go.warwick.ac.uk/crucible/mendeleev/ns#> .
@prefix pe: <urn:pe://> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix type: <http://go.warwick.ac.uk/crucible/mendeleev/types#> .

# Basic datatype declarations
type:ANY a mlv:type .
type:string a mlv:type .
type:int a mlv:type .
type:double a mlv:type .
type:location a mlv:type;
  mlv:parent type:double .
type:expression a mlv:type .

mlv:genericType rdfs:subClassOf mlv:type .
type:URL a mlv:genericType;
  mlv:parent type:string .
type:byteStream a mlv:type .
type:image a mlv:type;
  mlv:parent type:byteStream .
type:text a mlv:type;
  mlv:parent type:byteStream, type:string .

# PE Super-types
mlv:crucible_pe rdfs:subClassOf mlv:pe; mlv:runtime mlv:crucible .
mlv:spark_pe rdfs:subClassOf mlv:pe; mlv:runtime mlv:spark .
mlv:spl_pe rdfs:subClassOf mlv:pe; mlv:runtime mlv:streams .
mlv:accumulo_pe rdfs:subClassOf mlv:pe; mlv:runtime mlv:accumulo .
mlv:accumulo_table rdfs:subClassOf mlv:pe;
  mlv:runtime mlv:accumulo .
mlv:meteor_pe rdfs:subClassOf mlv:pe; mlv:runtime mlv:meteor .

# Import/Export PEs to transfer between runtimes
pe:cru_to_streams_export a mlv:crucible_pe;
  rdfs:label "Export for CRUCIBLE -> Streams";
  mlv:nativeCode
    "uk.ac.warwick.dcs.mendeleev.pe.StreamsExportNode";
  mlv:input [rdfs:label "Data"];
  mlv:postCondition [mlv:clearRuntime pe:cru_to_streams_export];
  mlv:postCondition [mlv:runtime mlv:cru_to_streams]
.
pe:cru_to_streams_import a mlv:spl_import_pe;
  rdfs:label "Import for CRUCIBLE -> Streams";
  mlv:nativeCode "ImportNode";
  mlv:output [rdfs:label "Data"];
```

```
  mlv:preCondition [mlv:runtime mlv:cru_to_streams];
  mlv:postCondition [mlv:clearRuntime pe:cru_to_streams_import];
  mlv:postCondition [mlv:runtime mlv:streams]
.

pe:cru_to_accumulo_export a mlv:crucible_pe;
  rdfs:label "Export for CRUCIBLE -> Accumulo";
  mlv:nativeCode
    "uk.ac.warwick.dcs.mendeleev.pe.AccumuloExportNode";
  mlv:input [rdfs:label "Data"];
  mlv:postCondition [mlv:clearRuntime pe:cru_to_accumulo_export];
  mlv:postCondition [mlv:runtime mlv:cru_to_accumulo]
.
pe:cru_to_accumulo_import a mlv:accumulo_import_pe;
  rdfs:label "Import for CRUCIBLE -> Accumulo";
  mlv:nativeCode "ImportNode";
  mlv:output [rdfs:label "Data"];
  mlv:preCondition [mlv:runtime mlv:cru_to_accumulo];
  mlv:postCondition [mlv:clearRuntime pe:cru_to_accumulo_import];
  mlv:postCondition [mlv:runtime mlv:accumulo]
.

pe:cru_to_meteor_export a mlv:crucible_pe;
  rdfs:label "Export for CRUCIBLE -> Meteor";
  mlv:nativeCode "uk.ac.warwick.dcs.mendeleev.pe.MongoExportNode";
  mlv:input [rdfs:label "Data"];
  mlv:postCondition [mlv:clearRuntime pe:cru_to_meteor_export];
  mlv:postCondition [mlv:runtime mlv:cru_to_meteor]
.
pe:cru_to_meteor_import a mlv:meteor_import_pe;
  rdfs:label "Import for CRUCIBLE -> Meteor";
  mlv:nativeCode "MongoCrucibleDataSource";
  mlv:output [rdfs:label "Data"];
  mlv:preCondition [mlv:runtime mlv:cru_to_meteor];
  mlv:postCondition [mlv:clearRuntime pe:cru_to_meteor_import];
  mlv:postCondition [mlv:runtime mlv:meteor]
.

pe:accumulo_to_cru_export a mlv:accumulo_pe;
  rdfs:label "Export for Accumulo -> CRUCIBLE";
  mlv:nativeCode
    "uk.ac.warwick.dcs.mendeleev.pe.CrucibleExportIterator";
  mlv:input [rdfs:label "Data"];
  mlv:postCondition [mlv:clearRuntime pe:accumulo_to_cru_export];
  mlv:postCondition [mlv:runtime mlv:accumulo_to_cru]
.
pe:accumulo_to_cru_import a mlv:accumulo_import_pe;
  rdfs:label "Import for Accumulo -> CRUCIBLE";
  mlv:nativeCode
    "uk.ac.warwick.dcs.mendeleev.pe.AccumuloImportNode";
  mlv:output [rdfs:label "Data"];
  mlv:preCondition [mlv:runtime mlv:accumulo_to_cru];
  mlv:postCondition [mlv:clearRuntime pe:accumulo_to_cru];
  mlv:postCondition [mlv:runtime mlv:crucible]
.

pe:accumulo_to_streams_export a mlv:accumulo_pe;
  rdfs:label "Export for Accumulo -> Streams";
  mlv:nativeCode "uk.ac.warwick.dcs.mendeleev.pe.StreamsIterator";
```

```
  mlv:input [rdfs:label "Data"];
  mlv:postCondition [mlv:clearRuntime pe:accumulo_to_streams_export
      ];
  mlv:postCondition [mlv:runtime mlv:accumulo_to_streams]
.
pe:accumulo_to_streams_import a mlv:spl_import_pe;
  rdfs:label "Import for Accumulo -> Streams";
  mlv:nativeCode "uk.ac.warwick.dcs.mendeleev.pe::AccumuloImport";
  mlv:output [rdfs:label "Data"];
  mlv:preCondition [mlv:runtime mlv:accumulo_to_streams];
  mlv:postCondition [mlv:clearRuntime pe:accumulo_to_streams_import
      ];
  mlv:postCondition [mlv:runtime mlv:streams]
.

pe:streams_to_meteor_export a mlv:spl_pe;
  rdfs:label "Export for Streams -> Meteor";
  mlv:nativeCode
    "uk.ac.warwick.dcs.mendeleev.pe::MeteorExportNode";
  mlv:input [rdfs:label "Data"];
  mlv:postCondition [mlv:clearRuntime
    pe:streams_to_meteor_export];
  mlv:postCondition [mlv:runtime mlv:streams_to_meteor]
.
pe:streams_to_meteor_import a mlv:meteor_import_pe;
  rdfs:label "Import for Streams -> Meteor";
  mlv:nativeCode "ImportNode";
  mlv:output [rdfs:label "Data"];
  mlv:preCondition [mlv:runtime mlv:streams_to_meteor];
  mlv:postCondition [mlv:clearRuntime
    pe:streams_to_meteor_import];
  mlv:postCondition [mlv:runtime mlv:meteor]
.

pe:streams_to_accumulo_export a mlv:spl_pe;
  rdfs:label "Export for Streams -> Accumulo";
  mlv:nativeCode
    "uk.ac.warwick.dcs.mendeleev.pe::AccumuloExportNode";
  mlv:input [rdfs:label "Data"];
  mlv:postCondition [mlv:clearRuntime
    pe:streams_to_accumulo_export];
  mlv:postCondition [mlv:runtime mlv:streams_to_accumulo]
.
pe:streams_to_accumulo_import a mlv:accumulo_import_pe;
  rdfs:label "Import for Streams -> Accumulo";
  mlv:nativeCode "ImportNode";
  mlv:output [rdfs:label "Data"];
  mlv:preCondition [mlv:runtime mlv:streams_to_accumulo];
  mlv:postCondition [mlv:clearRuntime
    pe:streams_to_accumulo_import];
  mlv:postCondition [mlv:runtime mlv:accumulo]
.

pe:streams_to_cru_export a mlv:spl_pe;
  rdfs:label "Export for Streams -> CRUCIBLE";
  mlv:nativeCode "uk.ac.warwick.dcs.mendeleev.pe::CruExportNode";
  mlv:input [rdfs:label "Data"];
  mlv:postCondition [mlv:clearRuntime pe:streams_to_cru_export];
  mlv:postCondition [mlv:runtime mlv:streams_to_cru]
```

```
.
pe:streams_to_cru_import a mlv:cru_import_pe;
  rdfs:label "Import for Streams -> CRUCIBLE";
  mlv:nativeCode "ImportNode";
  mlv:output [rdfs:label "Data"];
  mlv:preCondition [mlv:runtime mlv:streams_to_cru];
  mlv:postCondition [mlv:clearRuntime pe:streams_to_cru_import];
  mlv:postCondition [mlv:runtime mlv:crucible]
.

# Suite of join PEs for each runtime
pe:cru_join a mlv:crucible_pe;
  rdfs:label "Adaptive join";
  mlv:nativeCode "uk.ac.warwick.dcs.crucible.lib.join.AdaptiveJoin"
      ;
  mlv:config [
    rdfs:label "Join type hint";
    mlv:configElement "JoinHint";
    rdf:type type:string
  ];
  mlv:input [
    rdfs:label "Left";
    mlv:parameter [
      rdfs:label "join_id";
      rdf:type type:identifier
    ]
  ];
  mlv:input [
    rdfs:label "Right";
    mlv:parameter [
      rdfs:label "join_id";
      rdf:type type:identifier
    ]
  ];
  mlv:output [
    rdfs:label "JoinedData";
  ]
.

pe:spl_join a mlv:spl_pe;
  rdfs:label "Adaptive join";
  mlv:nativeCode "uk.ac.warwick.dcs.mendeleev.pe.join::
      StreamsAdaptiveJoin";
  mlv:config [
    rdfs:label "Join type hint";
    mlv:configElement "JoinHint";
    rdf:type type:string
  ];
  mlv:input [
    rdfs:label "Left";
    mlv:parameter [
      rdfs:label "join_id";
      rdf:type type:identifier
    ]
  ];
  mlv:input [
    rdfs:label "Right";
    mlv:parameter [
      rdfs:label "join_id";
```

```
      rdf:type type:identifier
    ]
  ];
  mlv:output [
    rdfs:label "JoinedData";
  ]
.

pe:cru_geo_annotate_join a mlv:crucible_pe;
  rdfs:label "Geospatial join";
  mlv:nativeCode "uk.ac.warwick.dcs.crucible.lib.join.
      GeoAnnotateJoin";
  mlv:config [
    rdfs:label "Join Tolerance (metres)";
    mlv:configElement "Tolerance";
    rdf:type type:double
  ];
  mlv:input [
    rdfs:label "Database";
    mlv:parameter [
      rdfs:label "db_lat";
      rdf:type type:latitude
    ];
    mlv:parameter [
      rdfs:label "db_lon";
      rdf:type type:longitude
    ]
  ];
  mlv:input [
    rdfs:label "Stream";
    mlv:parameter [
      rdfs:label "data_lat";
      rdf:type type:latitude
    ];
    mlv:parameter [
      rdfs:label "data_lon";
      rdf:type type:longitude
    ]
  ];
  mlv:output [
    rdfs:label "AnnotatedData";
  ]
.

# General library PEs
pe:fetch_url_spl a mlv:spl_pe;
  rdfs:label "Fetch the contents of a URL";
  mlv:nativeCode "uk.ac.warwick.dcs.mendeleev.pe::FetchURL";
  mlv:input [
    rdfs:label "data";
    mlv:parameter [
      rdfs:label "url";
      rdf:type type:URL;
      mlv:genericParameter _:fetch_type_spl
    ]
  ];
  mlv:output [
    rdfs:label "Output";
    mlv:parameter [
```

```
      rdfs:label "response";
      rdf:type type:http_response
    ];
    mlv:parameter [
      rdfs:label "headers";
      rdf:type type:header_list
    ];
    mlv:parameter [
      rdfs:label "type";
      rdf:type type:mime_type
    ];
    mlv:parameter [
      rdfs:label "body";
      rdf:type _:fetch_type_spl
    ]
  ]
.

_:fetch_type_spl
  rdfs:label "fetch_url generic type parameter";
  mlv:parent type:byteStream # Default type, but restricts
      subclasses also
.

pe:fetch_url a mlv:crucible_pe;
  rdfs:label "Fetch the contents of a URL";
  mlv:nativeCode "uk.ac.warwick.dcs.crucible.lib.web.FetchURL";
  mlv:input [
    rdfs:label "data";
    mlv:parameter [
      rdfs:label "url";
      rdf:type type:URL;
      mlv:genericParameter _:fetch_type
    ]
  ];
  mlv:output [
    rdfs:label "Output";
    mlv:parameter [
      rdfs:label "response";
      rdf:type type:http_response
    ];
    mlv:parameter [
      rdfs:label "headers";
      rdf:type type:header_list
    ];
    mlv:parameter [
      rdfs:label "type";
      rdf:type type:mime_type
    ];
    mlv:parameter [
      rdfs:label "body";
      rdf:type _:fetch_type
    ]
  ]
.

_:fetch_type
  rdfs:label "fetch_url generic type parameter";
  mlv:parent type:byteStream # Default type, but restricts
```

```
      subclasses also
.

pe:location_clustering a mlv:crucible_pe;
  rdfs:label "Location Clustering";
  mlv:nativeCode "uk.ac.warwick.dcs.mendeleev.pe.LocationClustering
      ";
  mlv:config [
    rdfs:label "Number of clusters";
    mlv:configElement "ClusterCount";
    rdf:type type:int
  ];
  mlv:config [
    rdfs:label "Window Size (tuples)";
    mlv:configElement "WindowSize";
    rdf:type type:int
  ];
  mlv:input [
    mlv:parameter [
      rdfs:label "key";
      rdf:type type:identifier
    ];
    mlv:parameter [
      rdfs:label "latitude";
      rdf:type type:latitude
    ];
    mlv:parameter [
      rdfs:label "longitude";
      rdf:type type:longitude
    ]
  ];
  mlv:output [
    rdfs:label "Clusters";
    mlv:parameter [
      rdfs:label "clusterLat";
      rdf:type type:cluster_latitude
    ];
    mlv:parameter [
      rdfs:label "clusterLon";
      rdf:type type:cluster_longitude
    ];
    mlv:parameter [
      rdfs:label "members";
      rdf:type type:cluster_density
    ]
  ];
  mlv:postCondition [mlv:clearPreConditions pe:location_clustering]
.
type:cluster_latitude mlv:parent type:latitude .
type:cluster_longitude mlv:parent type:longitude .
type:cluster_density mlv:parent type:int .


pe:spl_location_clustering a mlv:spl_pe;
  rdfs:label "SPL Native Location Clustering";
  mlv:nativeCode "uk.ac.warwick.dcs.mendeleev.pe::
      LocationClustering";
  mlv:config [
    rdfs:label "Number of clusters";
```

```
    mlv:configElement "ClusterCount";
    rdf:type type:int
  ];
  mlv:config [
    rdfs:label "Window Size (tuples)";
    mlv:configElement "WindowSize";
    rdf:type type:int
  ];
  mlv:input [
    mlv:parameter [
      rdfs:label "key";
      rdf:type type:identifier
    ];
    mlv:parameter [
      rdfs:label "latitude";
      rdf:type type:latitude
    ];
    mlv:parameter [
      rdfs:label "longitude";
      rdf:type type:longitude
    ]
  ];
  mlv:output [
    rdfs:label "Clusters";
    mlv:parameter [
      rdfs:label "clusterLat";
      rdf:type type:cluster_latitude
    ];
    mlv:parameter [
      rdfs:label "clusterLon";
      rdf:type type:cluster_longitude
    ];
    mlv:parameter [
      rdfs:label "members";
      rdf:type type:cluster_density
    ]
  ];
  mlv:postCondition [mlv:clearPreConditions pe:location_clustering]
.

pe:FFT a mlv:crucible_pe;
  rdfs:label "2-Dimensional FFT";
  mlv:nativeCode "uk.ac.warwick.dcs.crucible.image.2dFFT";
  mlv:input [
    mlv:parameter [
      rdfs:label "image";
      rdf:type type:image
    ]
  ];
  mlv:output [
    rdfs:label "fft";
    mlv:parameter [
      rdfs:label "fft";
      rdf:type type:fft2d
    ];
  ]
.
type:fft2d mlv:parent type:byteStream .
```

```
pe:bounding_box_iterator a mlv:accumulo_pe;
  rdfs:label "Lat/Lon bounding box filtering Accumulo Iterator";
  mlv:nativeCode "uk.ac.warwick.dcs.mendeleev.pe.BBoxFilterIterator
      ";
  mlv:input [
    mlv:parameter [
      rdfs:label "latitude";
      rdf:type type:latitude
    ];
    mlv:parameter [
      rdfs:label "longitude";
      rdf:type type:longitude
    ]
  ];
  mlv:output [
    rdfs:label "filtered"
  ]
.

pe:bounding_box_pe a mlv:spl_pe;
  rdfs:label "Lat/Lon bounding box filtering SPL PE";
  mlv:nativeCode "uk.ac.warwick.dcs.mendeleev.pe.GeoIPIterator";
  mlv:input [
    mlv:parameter [
      rdfs:label "latitude";
      rdf:type type:latitude
    ];
    mlv:parameter [
      rdfs:label "longitude";
      rdf:type type:longitude
    ]
  ];
  mlv:output [
    rdfs:label "filtered"
  ]
.

# PEs specific to case studies
pe:flickr a mlv:crucible_pe;
  rdfs:label "Flickr crawl data";
  mlv:nativeCode "uk.ac.warwick.dcs.crucible.lib.web.
      FlickrInterestingnessSource";
  mlv:output [
    rdfs:label "Output";
    mlv:parameter [
      rdfs:label "accuracy";
        rdf:type type:hdop
      ];
    mlv:parameter [
        rdfs:label "dateTaken";
        rdf:type type:timestamp
      ];
    mlv:parameter [
        rdfs:label "description";
        rdf:type type:description
      ];
    mlv:parameter [
        rdfs:label "id";
        rdf:type type:flickr_photo_id
```

```
      ];
    mlv:parameter [
        rdfs:label "largeUrl";
        rdf:type [
          rdfs:label "Generic instance of URL<Image> (flickr:
              largeUrl)";
          mlv:parent type:URL;
          mlv:genericParameter type:image
        ]
      ];
    mlv:parameter [
        rdfs:label "owner";
        rdf:type type:flickr_user
      ];
    mlv:parameter [
        rdfs:label "placeId";
        rdf:type type:flickr_placeId
      ];
    mlv:parameter [
        rdfs:label "tags";
        rdf:type type:taglist
      ];
    mlv:parameter [
        rdfs:label "title";
        rdf:type type:title
      ];
    mlv:parameter [
        rdfs:label "url";
        rdf:type [
          rdfs:label "Generic instance of URL<Image> (flickr:
              largeUrl)";
          mlv:parent type:URL;
          mlv:genericParameter type:image
        ]
      ]
  ]
.

type:username mlv:parent type:identifier .

pe:flickr_user_details a mlv:crucible_pe;
  rdfs:label "Fetch Flickr user details";
  mlv:nativeCode "uk.ac.warwick.dcs.crucible.lib.web.
      FlickrUserDetails";
  mlv:input [
    rdfs:label "User ID";
    mlv:parameter [
      rdfs:label "user";
      rdf:type type:flickr_user
    ]
  ];
  mlv:output [
    rdfs:label "UserDetails";
    mlv:parameter [
      rdfs:label "realName";
      rdf:type type:fullname
    ];
    mlv:parameter [
      rdfs:label "location";
```

```
      rdf:type type:flickr_location
    ];
    mlv:parameter [
      rdfs:label "mbox_sha1sum";
      rdf:type type:mbox_sha1sum
    ];
    mlv:parameter [
      rdfs:label "photosCount";
      rdf:type type:count_of_photos
    ];
    mlv:parameter [
      rdfs:label "photosurl";
      rdf:type [
        rdfs:label "Generic instance of URL<flickr_photo_page> (
            flickr:photosurl)";
        mlv:parent type:URL;
        mlv:genericParameter type:flickr_photo_page
      ]
    ];
    mlv:parameter [
      rdfs:label "profileurl";
      rdf:type [
        rdfs:label "Generic instance of URL<flickr_profile> (flickr
            :profileurl)";
        mlv:parent type:URL;
        mlv:genericParameter type:flickr_profile
      ]
    ];
    mlv:parameter [
      rdfs:label "buddyIconUrl";
      rdf:type [
        rdfs:label "Generic instance of URL<Image> (flickr:
            buddyIconUrl)";
        mlv:parent type:URL;
        mlv:genericParameter type:image
      ]
    ]
  ]
.

pe:exif_extract a mlv:crucible_pe;
  rdfs:label "Extract EXIF data from image";
  mlv:nativeCode "uk.ac.warwick.dcs.crucible.image.EXIF";
  mlv:input [
    rdfs:label "Image Data";
    mlv:parameter [
      rdfs:label "image";
      rdf:type type:image
    ]
  ];
  mlv:output [
    rdfs:label "extracted_exif";
    mlv:parameter [
      rdfs:label "creator";
      rdf:type type:person_name
    ];
    mlv:parameter [
      rdfs:label "camera";
      rdf:type type:camera
```

```
    ];
    mlv:parameter [
      rdfs:label "manufacturer";
      rdf:type type:camera_manufacturer
    ];
    mlv:parameter [
      rdfs:label "latitude";
      rdf:type type:latitude
    ];
    mlv:parameter [
      rdfs:label "longitude";
      rdf:type type:longitude
    ];
    mlv:parameter [
      rdfs:label "exif_timestamp";
      rdf:type type:exif_timestamp
    ];
    mlv:parameter [
      rdfs:label "fnumber";
      rdf:type type:fnum
    ];
    mlv:parameter [
      rdfs:label "colour_space";
      rdf:type type:colourspace
    ]
  ]
 .

 type:imsi mlv:parent type:identifier .

 pe:imsi_ip a mlv:crucible_pe;
  rdfs:label "Stream of IMSI-IP Address observations";
  mlv:nativeCode "uk.ac.warwick.dcs.crucible.telephony.
      IMSI_IP_Observations";
  mlv:output [
    rdfs:label "imsi_ip";
    mlv:parameter [
      rdfs:label "imsi";
      rdf:type type:imsi
    ];
    mlv:parameter [
      rdfs:label "ip";
      rdf:type type:ipaddress
    ]
  ]
 .

pe:telephony a mlv:crucible_pe;
  rdfs:label "Telephony data";
  mlv:nativeCode "uk.ac.warwick.dcs.crucible.telephony.
      TelephonySource";
  mlv:output [
    rdfs:label "calls";
    mlv:parameter [
      rdfs:label "source";
      rdf:type type:telnum
    ];
    mlv:parameter [
      rdfs:label "dest";
```

```
      rdf:type type:telnum
    ];
    mlv:parameter [
      rdfs:label "mcc";
      rdf:type type:mcc
    ];
    mlv:parameter [
      rdfs:label "mnc";
      rdf:type type:mnc
    ];
    mlv:parameter [
      rdfs:label "lac";
      rdf:type type:lac
    ];
    mlv:parameter [
      rdfs:label "cid";
      rdf:type type:cid
    ];
    mlv:parameter [
      rdfs:label "length";
      rdf:type type:call_length
    ];
    mlv:parameter [
      rdfs:label "imsi";
      rdf:type type:imsi
    ];
    mlv:parameter [
      rdfs:label "imei";
      rdf:type type:imei
    ]
  ]
.

type:cell_id a mlv:type;
  mlv:parent type:identifier .

pe:cell_id a mlv:crucible_pe;
  rdfs:label "Mint Cell Identity";
  mlv:nativeCode "uk.ac.warwick.dcs.crucible.telephony.MintCellID";
  mlv:input [
    rdfs:label "cli";
    mlv:parameter [
      rdfs:label "mcc";
      rdf:type type:mcc
    ];
    mlv:parameter [
      rdfs:label "mnc";
      rdf:type type:mnc
    ];
    mlv:parameter [
      rdfs:label "lac";
      rdf:type type:lac
    ];
    mlv:parameter [
      rdfs:label "cid";
      rdf:type type:cid
    ]
  ];
  mlv:output [
```

```
    rdfs:label "cellid";
    mlv:parameter [
      rdfs:label "cellid";
      rdf:type type:cell_id
    ]
  ]
.

pe:cell_id_spl a mlv:spl_pe;
  rdfs:label "Mint Cell Identity";
  mlv:nativeCode "uk.ac.warwick.dcs.mendeleev.pe::MintCellID";
  mlv:input [
    rdfs:label "cli";
    mlv:parameter [
      rdfs:label "mcc";
      rdf:type type:mcc
    ];
    mlv:parameter [
      rdfs:label "mnc";
      rdf:type type:mnc
    ];
    mlv:parameter [
      rdfs:label "lac";
      rdf:type type:lac
    ];
    mlv:parameter [
      rdfs:label "cid";
      rdf:type type:cid
    ]
  ];
  mlv:output [
    rdfs:label "cellid";
    mlv:parameter [
      rdfs:label "cellid";
      rdf:type type:cell_id
    ]
  ]
.

pe:cell_location_beacon a mlv:spl_pe;
  rdfs:label "Periodically trigger release of cell locations";
  mlv:nativeCode "uk.ac.warwick.dcs.mendeleev.pe::Beacon";
  mlv:output [
    rdfs:label "trigger";
    mlv:parameter [
      rdfs:label "trigger";
      rdf:type type:cell_location_trigger
    ]
  ]
.

pe:cell_location_trigger a mlv:crucible_pe;
  rdfs:label "Periodically trigger release of cell locations";
  mlv:nativeCode "uk.ac.warwick.dcs.crucible.telephony.Trigger";
  mlv:output [
    rdfs:label "trigger";
    mlv:parameter [
      rdfs:label "trigger";
      rdf:type type:cell_location_trigger
```

```
    ]
  ]
.

type:tower_latitude mlv:parent type:latitude .
type:tower_longitude mlv:parent type:longitude .

pe:cell_locations a mlv:crucible_pe;
  rdfs:label "Stream of cell tower location updates";
  mlv:nativeCode "uk.ac.warwick.dcs.crucible.telephony.
      CellLocations";
  mlv:input [
    rdfs:label "trigger";
    mlv:parameter [
      rdfs:label "trigger";
      rdf:type type:cell_location_trigger
    ]
  ];
  mlv:output [
    rdfs:label "cell_locations";
    mlv:parameter [
      rdfs:label "cellid";
      rdf:type type:cell_id
    ];
    mlv:parameter [
      rdfs:label "lat";
      rdf:type type:tower_latitude
    ];
    mlv:parameter [
      rdfs:label "lon";
      rdf:type type:tower_longitude
    ]
  ]
.

pe:spl_cell_locations a mlv:spl_pe;
  rdfs:label "Stream of cell tower location updates";
  mlv:nativeCode "uk.ac.warwick.dcs.crucible.telephony::
      CellLocations";
  mlv:input [
    rdfs:label "trigger";
    mlv:parameter [
      rdfs:label "trigger";
      rdf:type type:cell_location_trigger
    ]
  ];
  mlv:output [
    rdfs:label "cell_locations";
    mlv:parameter [
      rdfs:label "cellid";
      rdf:type type:cell_id
    ];
    mlv:parameter [
      rdfs:label "lat";
      rdf:type type:tower_latitude
    ];
    mlv:parameter [
      rdfs:label "lon";
      rdf:type type:tower_longitude
```

```
    ]
  ]
.

type:ipaddress mlv:parent type:identifier .

pe:network_data a mlv:crucible_pe;
  rdfs:label "Network trace data";
  mlv:nativeCode "uk.ac.warwick.dcs.crucible.model.impl.DataSource"
      ;
  mlv:config [
    rdfs:label "Data Source URI (netflow://...)";
    mlv:configElement "Source";
    rdf:type type:string
  ];
  mlv:output [
    rdfs:label "Data";
    mlv:parameter [
      rdfs:label "srcIP";
      rdf:type type:ipaddress
    ];
    mlv:parameter [
      rdfs:label "destIP";
      rdf:type type:ipaddress
    ];
    mlv:parameter [
      rdfs:label "port";
      rdf:type type:tcp_port
    ];
    mlv:parameter [
      rdfs:label "ttl";
      rdf:type type:ttl
    ];
    mlv:parameter [
      rdfs:label "ts";
      rdf:type type:ns_timestamp
    ]
  ];
.

pe:network_data_table a mlv:accumulo_table;
  rdfs:label "Network trace data table";
  mlv:nativeCode "uk.ac.warwick.dcs.mendeleev.pe.AccumuloTable";
  mlv:config [
    rdfs:label "Table name (try: 'netflow')";
    mlv:configElement "Source";
    rdf:type type:string
  ];
  mlv:output [
    rdfs:label "Data";
    mlv:parameter [
      rdfs:label "srcIP";
      rdf:type type:ipaddress
    ];
    mlv:parameter [
      rdfs:label "destIP";
      rdf:type type:ipaddress
    ];
    mlv:parameter [
```

```
      rdfs:label "port";
      rdf:type type:tcp_port
    ];
    mlv:parameter [
      rdfs:label "ttl";
      rdf:type type:ttl
    ];
    mlv:parameter [
      rdfs:label "ts";
      rdf:type type:ns_timestamp
    ]
  ];
.

pe:geo_ip a mlv:spl_pe;
  rdfs:label "Geo-IP Service";
  mlv:nativeCode "uk.ac.warwick.dcs.mendeleev.pe::GeoIP";
  mlv:input [
    mlv:parameter [
      rdfs:label "ip";
      rdf:type type:ipaddress
    ]
  ];
  mlv:output [
    rdfs:label "Geo";
    mlv:parameter [
      rdfs:label "ip";
      rdf:type type:ipaddress
    ];
    mlv:parameter [
      rdfs:label "lat";
      rdf:type type:latitude
    ];
    mlv:parameter [
      rdfs:label "lon";
      rdf:type type:longitude
    ]
  ]
.

pe:geo_ip_iterator a mlv:accumulo_pe;
  rdfs:label "Geo-IP Service as an Accumulo Iterator";
  mlv:nativeCode "uk.ac.warwick.dcs.mendeleev.pe.GeoIPIterator";
  mlv:input [
    mlv:parameter [
      rdfs:label "ip";
      rdf:type type:ipaddress
    ]
  ];
  mlv:output [
    rdfs:label "Geo";
    mlv:parameter [
      rdfs:label "lat";
      rdf:type type:latitude
    ];
    mlv:parameter [
      rdfs:label "lon";
      rdf:type type:longitude
    ]
```

```
  ]
.

pe:detect_face a mlv:crucible_pe;
  rdfs:label "Face Detection";
  mlv:nativeCode "uk.ac.warwick.dcs.crucible.faces.FaceDetection";
  mlv:input [
    mlv:parameter [
      rdfs:label "image";
      rdf:type type:image
    ]
  ];
  mlv:output [
    rdfs:label "person";
    mlv:parameter [
      rdfs:label "id";
      rdf:type type:faceID
    ];
  ]
.
type:faceID mlv:parent type:identifier, type:int .

pe:face_details a mlv:crucible_pe;
  rdfs:label "Data about Faces";
  mlv:nativeCode "uk.ac.warwick.dcs.crucible.faces.FaceData";
  mlv:output [
    rdfs:label "face";
    mlv:parameter [
      rdfs:label "id";
      rdf:type type:faceID
    ];
    mlv:parameter [
      rdfs:label "email";
      rdf:type type:email
    ];
    mlv:parameter [
      rdfs:label "name";
      rdf:type type:person_name
    ]
  ]
.

# Generic sinks
pe:table a mlv:meteor_pe;
  rdfs:label "Data Table Visualisation";
  mlv:nativeCode "uk.ac.warwick.dcs.crucible.vis.DataTable";
  mlv:input [
    rdfs:label "data"
  ]
.

pe:chart a mlv:meteor_pe;
  rdfs:label "Chart Visualisation";
  mlv:nativeCode "uk.ac.warwick.dcs.crucible.vis.Chart";
  mlv:config [
    rdfs:label "Chart Type";
    mlv:configElement "chart";
    rdf:type type:string
  ];
```

```
  mlv:input [
    rdfs:label "data";
    mlv:parameter [
      rdfs:label "x_label";
      rdf:type type:string
    ];
    mlv:parameter [
      rdfs:label "y_value";
      rdf:type type:double
    ]
  ]
.

pe:temporal_chart a mlv:meteor_pe;
  rdfs:label "Chart over time";
  mlv:nativeCode "uk.ac.warwick.dcs.crucible.vis.TemporalChart";
  mlv:input [
      rdfs:label "1d_data";
      mlv:parameter [
        rdfs:label "timestamp";
        rdf:type type:timestamp
      ];
      mlv:parameter [
        rdfs:label "value";
        rdf:type type:double
      ]
  ]
.

 pe:labelled_temporal_chart a mlv:meteor_pe;
  rdfs:label "Chart over time (labelled)";
  mlv:nativeCode "uk.ac.warwick.dcs.crucible.vis.
      LabelledTemporalChart";
  mlv:input [
        rdfs:label "labelled_data";
        mlv:parameter [
          rdfs:label "timestamp";
          rdf:type type:timestamp
        ];
        mlv:parameter [
          rdfs:label "value";
          rdf:type type:double
        ];
        mlv:parameter [
          rdfs:label "group";
          rdf:type type:string
        ]
  ]
 .


pe:file_sink a mlv:crucible_pe;
  rdfs:label "File Sink";
  mlv:nativeCode "uk.ac.warwick.dcs.crucible.lib.pe.FileSink";
  mlv:config [
    rdfs:label "File name";
    mlv:configElement "Filename";
    rdf:type type:string
  ];
```

```
  mlv:config [
    rdfs:label "Write data as CSV?";
    mlv:configElement "WriteCSV";
    rdf:type type:bool
  ];
  mlv:input [
    rdfs:label "data";
  ]
.

pe:google_maps a mlv:meteor_pe;
  rdfs:label "Pins on a map";
  mlv:nativeCode "uk.ac.warwick.dcs.crucible.vis.GoogleMaps";
  mlv:input [
    rdfs:label "locations";
    mlv:parameter [
      rdfs:label "id";
      rdf:type type:identifier
    ];
    mlv:parameter [
      rdfs:label "latitude";
      rdf:type type:latitude
    ];
    mlv:parameter [
      rdfs:label "longitude";
      rdf:type type:longitude
    ]
  ]
.

 pe:open_street_map a mlv:crucible_pe;
  rdfs:label "Open Street Map pins on a map";
  mlv:nativeCode "uk.ac.warwick.dcs.crucible.vis.OSMaps";
  mlv:input [
    rdfs:label "locations";
    mlv:parameter [
      rdfs:label "id";
      rdf:type type:identifier
    ];
    mlv:parameter [
      rdfs:label "latitude";
      rdf:type type:latitude
    ];
    mlv:parameter [
      rdfs:label "longitude";
      rdf:type type:longitude
    ]
  ]
 .
```

Listing C.2: Forward-chaining inference rules applied to the above.

```
@prefix mlv: <http://go.warwick.ac.uk/crucible/mendeleev/ns#> .
@prefix pe: <urn:pe://> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
#
#   TYPE INFERENCE / HIERARCHY RULES
#

#  Types are types, types are classes
{ ?x a ?t . } => { ?t a rdfs:Class } .
{ ?x a mlv:type . } => { ?x a rdfs:Class } .
{
  ?x mlv:parameter ?p .
  ?p a ?t .
} => {
  ?t a mlv:type.
} .

#  Parents are super-classes
{ ?t mlv:parent ?t2 } => {
  ?t rdfs:subClassOf ?t2 .
  ?t2 a rdfs:Class .
  ?t a mlv:type .
  ?t2 a mlv:type .
} .

#  Sub-classes are classes. Also, are sub-classes of themselves
{ ?t a rdfs:Class . } => { ?t rdfs:subClassOf ?t } .
{ ?x rdfs:subClassOf ?y } => {
  ?x a rdfs:Class .
  ?y a rdfs:Class .
} .

#  Propagate PE type as far as necessary
{
  ?x rdfs:subClassOf mlv:pe .
  ?n a ?x .
} => {
  ?n a mlv:pe .
} .

#  Generic parameter values are still types
{
  ?y mlv:genericParameter ?p .
} => {
  ?p a mlv:type .
} .

#  X<T>, Y<U>, X SUB Y, T SUB U -> X<T> SUB Y<U>
{
  ?x a mlv:type ;
  mlv:parent ?x_parent ;
  mlv:genericParameter ?x_param .

  ?y a mlv:type ;
  mlv:parent ?y_parent ;
  mlv:genericParameter ?y_param .
```

```
  ?x_parent rdfs:subClassOf ?y_parent .
  ?x_param rdfs:subClassOf ?y_param .
} => {
  ?x rdfs:subClassOf ?y .
} .

#  Parent relationship copies all statements
{
  ?x mlv:parent ?y .
  ?y ?p ?o .
} => {
  ?x ?p ?o .
} .

#
#   PRE-CONDITION EXPANSION RULES
#
{
  ?x a mlv:pe .
} => {
  ?x mlv:postCondition [ mlv:peUsed ?x ; mlv:provenance mlv:
      baseRule ] .
} .

# Pre-condition propagates by default, unless magic postCondition
#  is present. Also, don't copy the provenance information.
# Runtimes are handled explicitly by the second rule.
{
  ?x mlv:preCondition [ ?condP ?condO ] .
  ?condP log:notEqualTo mlv:provenance .
  ?condP log:notEqualTo mlv:runtime .
  ?x mlv:has_no_post_condition mlv:clearPreConditions .
} => {
  ?x mlv:postCondition [ ?condP ?condO ; mlv:provenance mlv:
      preCondition ] .
} .
{
  ?x mlv:preCondition [ mlv:runtime ?runtime ] .
  ?x mlv:has_no_post_condition mlv:clearRuntime .
} => {
  ?x mlv:postCondition [ mlv:runtime ?runtime ; mlv:provenance mlv:
      preCondition ] .
} .

# PE type implies preCondition, if it receives inputs (i.e. isn't a
    datasource)
{
  ?x a mlv:pe .
  ?x a [ mlv:runtime ?runtime ].
  ?x mlv:has_predicate mlv:input .
} => {
  ?x mlv:preCondition [
    mlv:runtime ?runtime ; mlv:provenance rdf:type
  ] .
} .

# Datasource PE implies its own runtime as postCondition
{
```

```
  ?x a mlv:pe .
  ?x a [ mlv:runtime ?runtime ].
  ?x mlv:has_no_predicate mlv:input .
} => {
  ?x mlv:postCondition [
    mlv:runtime ?runtime ; mlv:provenance mlv:isSource
  ] .
} .

# PE input implies preCondition
{
  ?x mlv:input [
    mlv:parameter ?param
  ] .
  ?param a ?type .
} => {
  ?x mlv:preCondition [
    mlv:hasField ?type ; mlv:provenance mlv:input
  ] .
} .

# PE output implies postCondition
{
  ?x mlv:output [
    mlv:parameter ?param
  ] .
  ?param a ?type .
} => {
  ?x mlv:postCondition [
    mlv:hasField ?type ; mlv:provenance mlv:output
  ] .
} .

# Annotate unbounded generic postconditions
{
  ?x mlv:postCondition ?cond .
  ?cond mlv:hasField ?type .
  ?param mlv:genericParameter ?type .
  ?param a ?paramType
} => {
  ?x mlv:postCondition [
    mlv:unboundGenericParameter ?param ;
    mlv:paramType ?paramType
  ].
} .

# Expand bounded generic postconditions
{
  ?x mlv:postCondition ?cond .
  ?cond mlv:hasField ?type .
  ?type a mlv:genericType ;
    mlv:genericParameter ?param ;
    mlv:parent ?parent .
} => {
  ?x mlv:postCondition [
    mlv:boundedGenericField ?parent ;
    mlv:genericParameter ?param
  ].
} .
```

```
# Post condition expansion through RDFS subclass inference
{
  ?x mlv:postCondition [
    mlv:hasField ?o
  ] .
  ?o mlv:parent ?super0 .
  ?o log:notEqualTo ?super0 .
} => {
  ?x mlv:postCondition [
    mlv:hasField ?super0
  ] .
} .
```