THE UNIVERSITY OF
WARWICK

Original citation:
Coetzee, Peter and Jarvis, Stephen A., 1970- (2015) Goal-based analytic composition for on- and off-line execution at scale. In: The 9th IEEE International Conference on Big Data Science and Engineering, Helsinki, Finland, 20-22 Aug 2015

Permanent WRAP url:
http://wrap.warwick.ac.uk/72354

Copyright and reuse:
The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions. Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

Publisher's statement:
"© 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting /republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works."

A note on versions:
The version presented here may differ from the published version or, version of record, if you wish to cite this item you are advised to consult the publisher's version. Please see the 'permanent WRAP url' above for details on accessing the published version and note that access may require a subscription.

For more information, please contact the WRAP Team at: publications@warwick.ac.uk

http://wrap.warwick.ac.uk

Goal-Based Analytic Composition for On- and Off-line Execution at Scale

Peter Coetzee
Department of Computer Science
University of Warwick
Coventry, CV4 7AL
United Kingdom
Email: p.l.coetzee@warwick.ac.uk

Stephen Jarvis
Department of Computer Science
University of Warwick
Coventry, CV4 7AL
United Kingdom
Email: s.a.jarvis@warwick.ac.uk

***Abstract*—Crafting scalable analytics in order to extract actionable business intelligence is a challenging endeavour, requiring multiple layers of expertise and experience. Often, this expertise is irreconcilably split between an organisation's engineers and subject matter or domain experts. Previous approaches to this problem have relied on technically adept users with tool-specific training. These approaches have generally not targeted the levels of performance and scalability required to harness the sheer volume and velocity of large-scale data analytics.**

In this paper, we present a novel approach to the automated planning of scalable analytics using a semantically rich type system, the use of 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 execution both on- and off-line, with no lower-level engineering support.

We describe in detail (i) an abstract model of analytic assembly and execution; (ii) goal-based planning and (iii) code generation using this model for both on- and off-line analytics. Our implementation of this model, MENDELEEV, is used to (iv) demonstrate the applicability of our approach through a series of case studies, in which a single interface is used to create analytics that can be run in real-time (on-line) and batch (off-line) environments. We (v) analyse the performance of the planner, and (vi) show that the performance of MENDELEEV's generated code is comparable with that of hand-written analytics.

I. INTRODUCTION

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

Transferring skills from one group into the other can be challenging, and so typically organisations are left with two options. They either make use of a traditional development model, in which engineers attempt to elucidate requirements from stakeholders, develop a solution to meet those requirements, and then seek the approval of their customers; or they aim to empower the domain experts by offering increasingly high-level abstractions onto their execution environment, concealing the difficulty (and often the power) of hand-tuning an analytic.

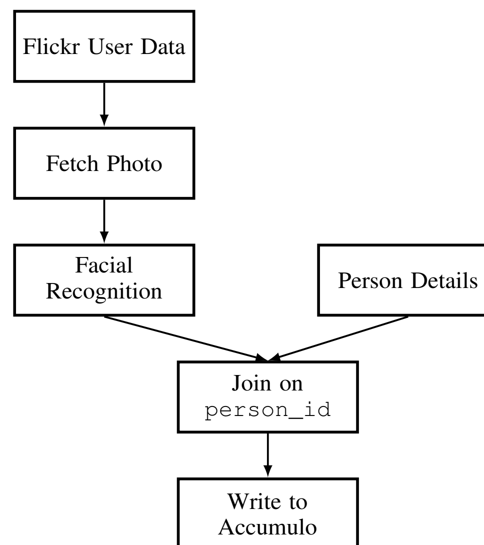


Fig. 1: A sample analytic, reading profile pictures from Flickr and using facial recognition to populate an Accumulo table.

Consider, for example, the Flickr[1] analytic described in Figure 1. Each component of the analysis is described here by a box, with arrows indicating the flow of data from one component to another. There are many runtime environments in which this analytic could be run, depending on the wider system context (*e.g.*, if user data is being crawled, a streaming (on-line) analytic engine such as Apache Storm [1] or IBM InfoSphere Streams [2] might be used; alternatively, if the data is already present in an HDFS (Hadoop Distributed File System) [3] store, an off-line MapReduce [4] or Apache Spark [5] runtime might fit better). Each of these runtime environments specify their own programming model, and have their own optimisation constraints and engineering best practices, which a domain expert may not be aware of.

There are additionally challenges in even a relatively simple analytic such as this through which an engineer could misunderstand the requirements of their users (using the wrong input data, persisting the wrong fields to the Accumulo [6] datastore, bad parameters for the facial recognition system, *etc.*).

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

This divide between engineering expertise and domain knowledge has led researchers to consider approaches which will make the best use of the available skills, without the risks inherent in traditional models of cooperation. This paper presents a new approach, which for the first time will enable domain experts to compose scalable analytics without any engineering knowledge. This approach composes an analytic from primitives crafted by engineers, which can be immediately deployed in a scalable and performant streaming and batch context, without any additional engineering required.

The specific contributions of this work are as follows:

- A new abstract model of analytic assembly and execution, centred around a semantically rich type system (Section IV);
- Goal-based planning of on- and off-line analytic applications using the above model, requiring little programming ability or prior knowledge of available analytic components from the user (Section V);
- Code generation for execution of planned analytics at scale, with equivalent semantics, in both on- and off-line scenarios (Section VI);
- Validation of the applicability and performance of the types-based planning approach using four case studies from the domains of telecommunications and image analysis (Section VII);
- An analysis of the performance of the planning engine over each of these case studies (Section VIII-A);
- Performance analysis of analytics at scale in both on- and off-line runtime environments, demonstrating comparable performance with equivalent hand-written alternatives (Section VIII-B).

The remainder of this paper is structured as follows: Section II describes related work; Section III outlines the methodology adopted in this research; Sections IV and V detail the approach to modelling analytics, and planning their execution respectively; Section VI discusses code generation; while Section VII discusses the application of this approach through four case studies. Finally, Sections VIII and IX provide a performance evaluation and conclude the paper.

II. Related Work

A variety of approaches to the problem of analytic planning exist in the literature. Historical research in this area tends to be in the context of web-based mashups, but some of the requirements behind such systems are relevant. Yu *et al.* [7] provide a rich overview of a number of approaches, including Yahoo! Pipes [8] – one of the first in a number of recent dataflow-based visual programming paradigms for mashups and analytics. These require enough technical knowledge from their users to be able to navigate and select the components of a processing pipeline, as well as connecting those components together, without requiring the use of a programming language. This removes the challenges of learning programming syntax, but 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 [9], PopFly [10], and Marmite [11]. The work of Daniel *et al.* [12] aims to simplify the use of tools like Pipes by providing recommendations to a non-expert on how to compose their flows. Others, such as Google's (discontinued) Mashup Editor [13] take a more technical approach, requiring an in-depth knowledge of XML, JavaScript, and related technologies, but permit a greater degree of flexibility as a result. Finding domain experts with sufficient expertise in these areas to express their queries can be challenging.

Some vendors offer more technical solutions to the problem of creating analytics by non-engineers, without entering the realm of full programming languages. SQL is a common vehicle for this; Apache Spark SQL [5], [14] and Cloudera Impala [15] both offer an SQL-style interface onto NoSQL data stores. The work of Jain *et al.* [16] 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, there exist entire classes of analytics that cannot be represented in SQL [17]. Approaches which assemble general-purpose code into complex analytics do not suffer these limitations.

Whitehouse *et al.* [19] propose a semantic approach to composing queries over streams of sensor data using a declarative mechanism to drive a backward chaining reasoner, solving for possible plans at execution time. Sirin *et al.* [20] introduce the use of OWL-S for query component descriptions in the SHOP2 [21] planner (a hierarchical task network planner). Another common approach, taken by Pistore *et al.* in BPEL4WS [22], uses transition systems as a modelling 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 [23] 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 [24] offers a traditional "search" interface (making use of full-text and tag-based search) to locate web services which a user may manually compose in the Taverna interface. This form of manual search and assembly requires considerable user experience, and an understanding of the art of the possible, which a general-purpose analytic planner does not.

Research in Software Engineering has examined a related problem to this: searching for an existing code sample to perform some desired function. Stolee *et al.* [25] examined the use of semantic models of source code as an indexing strategy to help find a block of code that will pass a set of test cases – one form of goal-based search. Such semantic searches have additionally been used in web service composition [26], [27]. However, the complexity of the semantic model and the uncertainty in its retrieval accuracy make assembly of multiple blocks of code on this basis risky – there is little guarantee that the retrieved code samples are compatible.

These web services based systems typically involve considerable user training (whether in the composition interface, or the

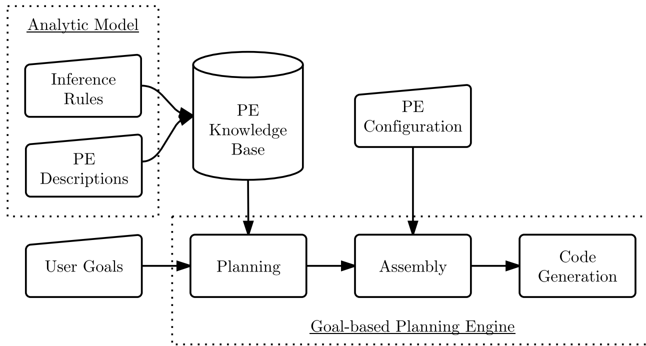


Fig. 2: Steps in composing an analytic.

formal specification of their query language), and at their core aim to answer a single question at a time through service-oriented protocols like WSDL and SOAP. Often, large-scale data analytics workflows aim instead to analyse a huge amount of data in parallel – an execution model which is closer to High Performance Computing simulations than web mashups.

One noteworthy approach to the problem is that taken by IBM's research prototype, MARIO [28], which builds on SPPL, the Streaming Processing Planning Language [29], [30]. IBM characterises MARIO as offering *wishful search*, in which a user enters a set of goal tags. The MARIO planning engine then aims to construct a sequence of analytical components for deployment on InfoSphere Streams that will satisfy those goals. These tags correspond to those applied to the individual analytical components by the engineers responsible for their creation. In practice, due to the tight coupling between the engineer-created tagsonomy of patterns and the flows 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.

This research builds on the wishful search concept behind MARIO, permitting the composition of complex analytics while using a more granular model of analytic behaviour, by making use of existing techniques in AI planning. It additionally targets execution of these analytics equivalently in both on- and off-line runtime environments. As such, it is the only automated planning engine of its type to offer this degree of applicability for its analytics, regardless of runtime, without also requiring significant technical ability and input from the end-user.

III. High-Level Overview

To compose an analytic from a user's goals, the components outlined in Figure 2 are used. An abstract Analytic Model (detailed in Section IV) generates a knowledge-base of processing elements. With this knowledge-base in place, the system collects goals from the user as input to the planning process. There are three types of goals that can be used to constrain the planning process (see Section V):

- Types that the analytic must produce in its output;
- The datasource with which the analytic must begin;
- The type of data sink to terminate the analytic.

For example, to create the sample analytic described in Figure 1, the user might specify:

- Types: `person_id`, `person_name`, `postal_address`, `email_address`
- Source: `FlickrUserData`
- Sink: `Accumulo`

The planning process generates a set of possible analytics, which can be presented to the user. 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. Finally, code generation is invoked on the concrete plan to create an executable analytic.

The approach described in this paper has been implemented and tested using real analytics in a system we call Mendeleev, named after the scientist responsible for composing and organising the periodic table as we know it today.

IV. Modelling Analytics

The first contribution of this paper is a model of analytic behaviour, used to separate the planning process from the concrete implementation of an analytic. This research models an analytic as a set of communicating sequential processes, 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. Unlike existing modelling languages, such as S-NET [18], this abstract analytic model makes no assumptions about the statefulness or atomicity of each PE. While this may limit the scope of potential optimisations during code generation, it significantly enhances the expressivity and applicability of the model.

The model is encoded in an RDF [31] graph describing the available types and PEs[2]. Types may exhibit polymorphic inheritance, as in a typical second-order type system. This inheritance is indicated using the `mlv:parent` relationship, and may form an inheritance graph provided each type cycle declares only one `mlv:nativeCode`; that is, the name of the type in the target language that is represented by this concept (in the case of our target language, this is a Java class). 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 1.

Listing 1: RDF graph for a simple type hierarchy

```
# The "raw" ByteBuffer parent type
type:byteBuffer rdf:type mlv:type ;
  mlv:nativeCode  "java.nio.ByteBuffer" .
```

[2]RDF types are given in this paper using W3C CURIE [32] 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#
```

```
# 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 type constraint (akin to a generic type in Java [33], or a template in C++ [34]). This is used in PEs which transform an input type to an output without precise knowledge of the information encoded in the data. For example (see Listing 2), a PE for fetching data over HTTP might take an input of `type:URL` parameterised with ⟨`?T mlv:parent type:byteBuffer`⟩, and output data with the same type as the variable `?T`, an as-yet unbound subtype of `type:byteBuffer`. A priori, `?T` is known to be a `type:byteBuffer`; during planning it may be bound to a more specific type (*e.g.*, `type:image` in the Flickr analytic described above).

Listing 2: Modelling unbound type variables in RDF

```
# Declaration of a generic type
type:URL rdf:type mlv:genericType ;
  mlv:nativeCode "java.net.URL" .
# PE input declaration for url<?T>
#    (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 (along with a subset of the PE model described in Listing 3) can be seen in Figure 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`.

A. PE Formalism

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. 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:

$$\tau_n = \nu_0 \cup \nu_1 \cup ... \cup \nu_{n-1} \cup \nu_n \quad (1)$$

Or, inductively:

$$\tau_n = \tau_{n-1} \cup \nu_n \quad (2)$$

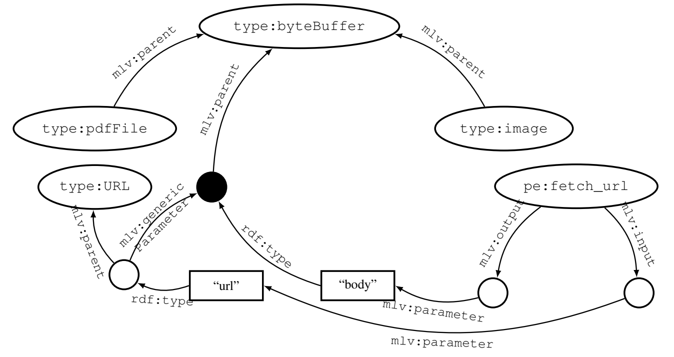


Fig. 3: Graph visualisation of the RDF description of a portion of the example model. `_:urlType` bnode represented by ●.

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 will not be considered here, in order to simplify the description of the planning process.

One important extension to this model is in support of join operators (discussed in further detail in Section V). These must receive two sets of input types, and emit the union of their accumulated inputs. Thus, for join operator χ_n with inputs χ_i and χ_j, τ_n is given as follows:

$$\tau_n = \tau_i \cup \tau_j \quad (3)$$

PE connectivity utilises a form of subsumption compatible with the type model described above. A type u can be said to be subsumed by a type v ($u \lhd v$) if one of the following cases hold true:

$$u \lhd v \Leftarrow \begin{cases} u \texttt{ mlv:parent } v \\ u \texttt{ mlv:parent } t, t \lhd v \end{cases} \quad (4)$$

$$u\langle t\rangle \lhd v\langle s\rangle \Leftarrow u \lhd v \wedge t \lhd s \quad (5)$$

A PE χ_x is considered *fully compatible* with χ_y, and is thus able to emit tuples to PE χ_y, if the following holds true:

$$\forall t \in \mu_y, \exists u \in \tau_x \mid u \lhd t \quad (6)$$

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 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 3.

Listing 3: Modelling the HTTP Fetch PE in RDF

```
pe:fetch_url rdf:type mlv:pe ;
  mlv:nativeCode "lib.web.FetchURL" ;
  mlv:input [
    mlv:parameter [  # url is a URL<?T>
```

Algorithm 1 Bidirectional Planning, searching for a given set of target types (ϕ), source PE (σ), accumulated types (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
      ▷ Update τ with outputs of σ, and check for completion
12:     τ ← τ ∪ ν_σ
13:     if τ satisfies φ then
14:         return [[σ]]
15:     end if
      ▷ Depth-first search of PEs in β
16:     forward ← consumers_of(τ)
17:     candidates ← dfs_search(forward ∩ β, φ, σ, τ)
18:     results ← ∅
19:     for all candidate in candidates do
20:         results ← results ∪ [σ, candidate]
21:     end for
      ▷ Depth-first search of remaining candidates
22:     if results == ∅ then
23:         results ← dfs_search(forward − β, φ, σ, τ)
24:     end if
      ▷ If nothing found still, join with other paths
      ▷ EXPLORE_JOINS is mutually recursive with SOLVE
25:     if results == ∅ then
26:         results ← explore_joins(σ, φ, τ)
27:     end if
28:     return results
29: end procedure
```

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

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
```

V. Goal-Based Planning

The second contribution of this paper is a goal-based planner based around the semantically rich type system described above. The goal 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.

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 FuXi [35] RETE-UL algorithm), and compute three key types of closure. First, a subset of RDFS-style 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* rules described in Section IV, Equation 6. A PE χ_x is considered *partially compatible* with χ_y, and is thus a candidate for sending tuples to PE χ_y, if one of the following holds true:

$$\exists t \in \mu_y, u \in \nu_x \mid u \lhd t \quad (7)$$

$$\exists t \in \mu_y, u\langle v\rangle \in \mu_x, v \in \nu_x \mid v \lhd t \quad (8)$$

$$\exists s\langle t\rangle \in \mu_y, u\langle v\rangle \in \nu_x \mid u \lhd s, v \lhd t \quad (9)$$

For example, consider `pe:fetch_url` described above; it requires a URL parameterised with any `type:byteBuffer`. Consider also `pe:exif`, which requires a `type:image` on its input (where `type:image` $\lhd$ `type:byteBuffer`), and outputs a number of Exif[3]-related fields:

$$\mu_{\text{fetch_url}} = \{\texttt{type:url}\langle ?T \lhd \texttt{type:byteBuffer}\rangle\} \quad (10)$$

$$\nu_{\text{fetch_url}} = \{?T\} \quad (11)$$

$$\mu_{\text{exif}} = \{\texttt{type:image}\} \quad (12)$$

$$\nu_{\text{exif}} = \{\texttt{type:camera}, \texttt{type:lat}, \texttt{type:lon}, \texttt{type:fstop}, ...\} \quad (13)$$

Through Equation 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`.

In practice, this closure is calculated offline and stored for future use. The search through the graph of partially compatible PEs is

[3] Exchangeable image file format; image file metadata

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 types, while fulfilling the input requirements of each PE. In order to minimise the search-space explosion, the search is performed bi-directionally, with an empirically selected heuristic expanding the search space backwards for every three levels of forward search. Similarly, if a source or a sink constraint is specified, it is used to optimise the search process. The algorithm has 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;
- L12-15: Update the set of accumulated type data (τ), and test to see if all required types are satisfied; if so, this branch of the search terminates;
- L16-21: Attempt to search the next level (recursively), using only the set of backwards candidates;
- L22-24: If the above step did not yield any new paths, repeat the search with PEs not in the set of backwards candidates;
- L25-27: If the above steps still have not yielded any results, attempt to explore joins (see below).

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 τ, 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.

The EXPLORE_JOINS procedure, referenced in Algorithm 1 on line 26, explores non-linear analytic assembly, by joining the current accumulated type τ with other sources in the knowledge-base. This automated join process is calculated on the basis of identity-joins; if two PEs output data with a common sub-type of the built-in `type:identifier`, then they can be considered for a join operation. The implementation of this join PE is runtime-dependent; MENDELEEV uses an adaptive windowed join which is sensitive to data source types.

Once an execution plan is selected from the generated options, it must be assembled into a concrete plan. This process involves binding keys from 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 the nature of the fields in their data.

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. The goal node will have a τ of the union of all PE outputs up to that point in the analytic – however, many of these types may not be needed in order to correctly complete the computation. During assembly, a second pass is therefore taken backwards across the topology (in a breadth-first traversal from the goal node) using the type pruning algorithm outlined in Algorithm 2 to prevent the topology from passing

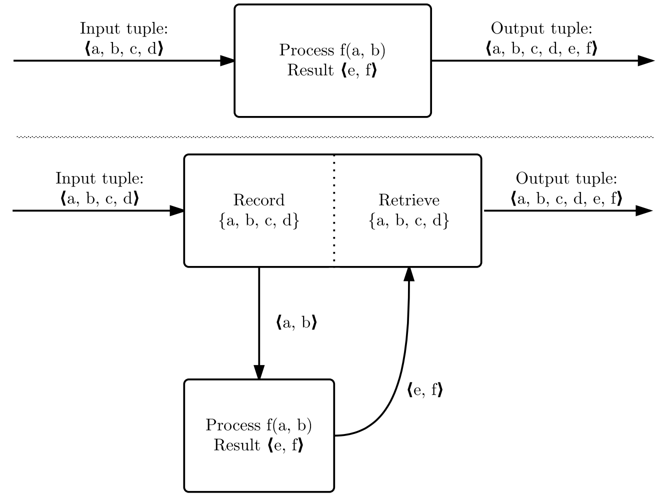


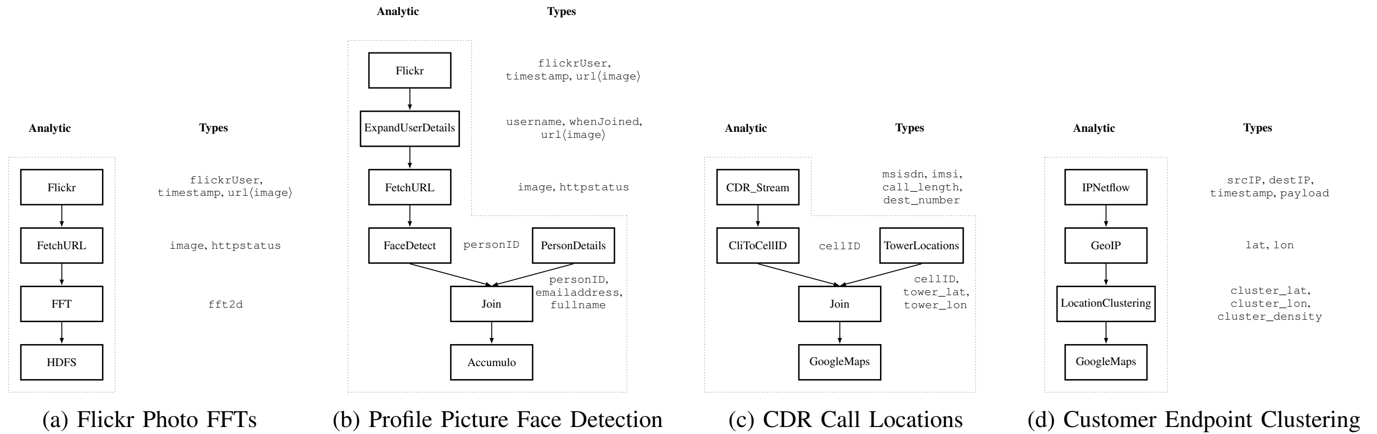
Fig. 4: Top: MENDELEEV message passing model for a process `f`. Bottom: CRUCIBLE model of tuple field copying semantics.

unnecessary data forwards. This helps to control the otherwise unlimited expansion of tuple width, improving the space, time, and message passing complexity of the resultant analytic.

VI. CODE GENERATION

Once the concrete execution plan is assembled, it is passed to a pluggable code generator; this is the third contribution of this paper. MENDELEEV's planner produces a concrete plan, which the code generator must turn into native code for execution on both on- and off-line runtimes. To achieve this, it may either generate native code for each runtime directly, or use an intermediate representation to manage the differences in runtime models. MENDELEEV currently generates code using the CRUCIBLE [36] domain specific language (DSL) as such an intermediate representation. CRUCIBLE offers a DSL and suite of runtime environments, adhering to a common runtime model, that provide consistent execution semantics for an analytic across on- and off-line runtimes. A performance evaluation of CRUCIBLE [36] showed consistent scalability of CRUCIBLE topologies in a standalone environment, in batch mode on Apache Spark, and in streaming mode on IBM's InfoSphere Streams. CRUCIBLE's DSL is source-to-source compiled through CRUCIBLE to optimised native code for each runtime environment on which it can be executed.

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 4 illustrates how this might be achieved in the basic CRUCIBLE execution model. MENDELEEV's conceptual model (top) 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. On the bottom of Figure 4, 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.



(a) Flickr Photo FFTs (b) Profile Picture Face Detection (c) CDR Call Locations (d) Customer Endpoint Clustering

Fig. 5: Planned analytics for Flickr Image Analysis and Telecommunications analytics.

While this theoretical approach produces correct results, the extra message passing it involves would slow topologies down considerably. 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. At this time it will also prune out unused tuple fields from the output per Algorithm 2.

VII. CASE STUDIES

To better understand the composition process in MENDELEEV, we present here our fourth contribution; a series of case studies and an evaluation of this technique. Figure 5 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 set of types added to the result, given beside each PE. These analytics have been generated with MENDELEEV, using a small library of general-purpose PEs.

A. Case Study: Flickr Image Analysis

The user wishes to compute and store the Fourier transform of a series of images from Flickr, and store those results as HDFS files for use elsewhere in their workflow. Engineers have exposed a datasource consisting of a Flickr photo metadata stream, and described it to the MENDELEEV system. The user selects the following bounds from the user interface:

1) Sink: HDFS
2) Requested 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 system 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(a).

Another analyst has an interest in annotating people from their Flickr profile images with their email addresses from another source using a facial recognition system, sending their results to an Accumulo table (as described in the original example in Figure 1). They configure MENDELEEV to search as follows:

1) Sink: Accumulo
2) Requested Types: `person`, `emailaddress`

The user is presented with a number of analytics, but on closer inspection none of these use Flickr as a datasource. They refine their query interactively to bind the source to "Flickr". This returns a small number of candidate analytics, and the user selects that shown in Figure 5(b). 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.

B. Case Study: Telecommunications Visualisations

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:

1) Sink: GoogleMaps
2) Requested Types: `msisdn`[5], `tower_latitude`, `tower_longitude`

Figure 5(c) shows their selected analytic. This result particularly highlights the importance of a strong adaptive join capability; as this is a live streaming analytic, the rate at which CDRs arrive is likely to be far slower than that of the (offline) TowerLocations datasource. While MENDELEEV represents this as a join operation, in practice the join PE operates akin to an in-memory lookup against the TowerLocations dataset.

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

[5]A unique telecoms subscriber identifier

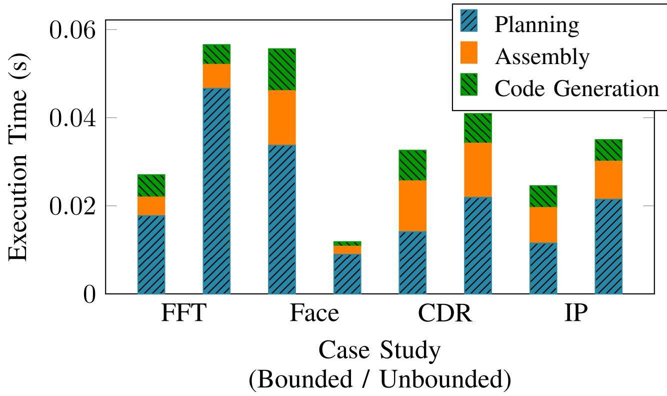


Fig. 6: Benchmark results for the MENDELEEV planner when applied to the case studies.

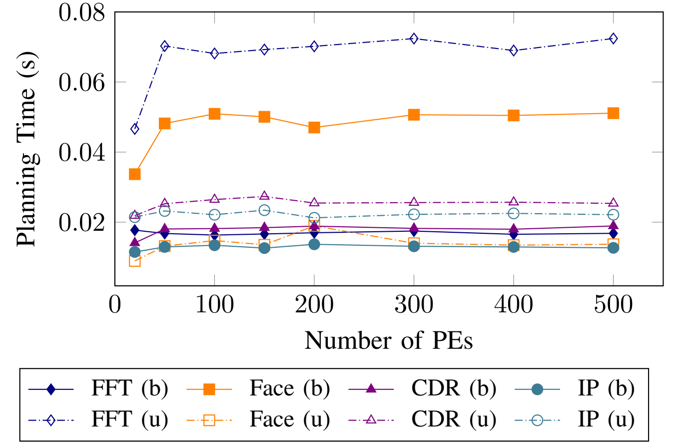


Fig. 7: Scaling of the MENDELEEV planner with knowledge-base size for both (b)ounded and (u)bounded case studies.

Another 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 telco has the right peering agreements in place. They configure a query:

1) Sink: GoogleMaps
2) Requested Types: `ipaddress`, `cluster_latitude`, `cluster_longitude`

Their resulting analytic is shown in Figure 5(d). 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 this same analytic can be employed against streaming IP Netflow data, or against an historical database of events without any further user interaction. The generated code is simply deployed to either their streaming or their offline platform, and the CRUCIBLE framework selects the relevant instance of the datasource.

VIII. 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; the time taken for the planning and assembly process, and the runtime performance of its resulting output.

A. Planner Performance

As the fifth contribution of this work, the performance of the planning process has been examined using the previously discussed four case studies. Each case study has been benchmarked as a bounded query (with a fixed source), 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 6. These experiments were performed on a system with a 4-core Intel Core i7 CPU and 8GB 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 fails to generate a correct solution altogether in its unbounded form (but does so very quickly). The bounded Face Detection query is, in fact, the longest-running assembly and generation process. This is 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 order to better understand how the bidirectional search in the planning phase scales as the knowledge-base expands, further benchmarks were run against knowledge bases of varying size. The PEs in this expanded knowledge base were all “reachable” in the graph search, and as such could be expected to have an impact on planning time. The results of these benchmarks for both the bounded (b) and unbounded (u) query variants can be seen in Figure 7. They show that in scaling the size of the knowledge-base from 20 to 50 PEs there is a noticeable performance impact. However, as the search is bidirectional, beyond this scale there is little negative impact on the search time. At no point does the planning take longer than 0.08 seconds in the tested case studies. Further detail about the number of plans considered in the search, and the number found and returned, can be seen in Table I below.

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 I: Number of plans considered and returned in the 500 PE stress test knowledge-base.

		5		10		20		30		40		50	
		Records Processed (millions)											
	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

TABLE II: Benchmarking results (makespan time and per-tuple latency) for each runtime mode and code type.

B. Runtime Performance

It is valuable to compare the performance of MENDELEEV's generated code to analytics hand-written in both the CRUCIBLE DSL and in native code. For this, hand-written native and CRUCIBLE code for each runtime is compared to MENDELEEV, using a shared library of basic Java operations to implement the "IP Communications Endpoints" case study described above (Figure 5d). These analytics were executed against 194 offline packet capture files, corresponding to 100GB of raw capture data (5.8GB of packet headers).

Five equivalent variants of this 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. Results were collected on a small test cluster, consisting of three Data Nodes, one NameNode, and two Streams nodes. Each node hosts 2x3.0GHz CPUs, 8GB RAM, and 2x1GbE interfaces. Table II shows the performance and scalability (makespan time, and latency per tuple) of the analytic on each runtime type in turn; Standalone, Apache Spark (HDFS mode), and on IBM InfoSphere Streams.

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 engineering expertise from the user. Due to the additional compile-time knowledge that MENDELEEV infers about the input and output tuples, it is able to make stronger assumptions about the input data; and thus generate more efficient tuple processing methods, with less data copying and validation required than in the base CRUCIBLE runtime's implementation.

Equivalent analytics, hand-written and tuned for each runtime, outperforms MENDELEEV by a maximum of 1.3× in these experiments. Furthermore, the per-tuple latency 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 III below.

Environment	MENDELEEV vs CRUCIBLE	MENDELEEV vs Manual
Standalone	1.12×	0.77×
Spark	1.39×	0.87×
Streams	1.10×	0.82×

TABLE III: Relative speedup of MENDELEEV to CRUCIBLE and manually implemented runtimes.

IX. CONCLUSIONS & FURTHER WORK

This work has demonstrated (i) a new abstract model for the assembly and execution of analytics, based on a semantically rich type system, along with (ii) a novel approach to goal-based planning using this model, requiring little engineering expertise from the user. It has shown (iii) how code generation for these analytics can be leveraged for execution at scale in both on- and off-line scenarios. It has (iv) validated the applicability of the approach, as well as (v) its performance for interactive use. Finally, (vi) performance results have shown MENDELEEV-generated analytics offer runtime performance comparable with hand-written code. The performance penalty over a completely hand-written and tuned analytic has been shown to be a maximum of 1.3× in these experiments – a potentially acceptable cost for enabling domain experts to compose scalable analytics without the need for any prior programming or engineering experience.

There are a number of avenues to be explored in future work. 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 end up deploying for a given type of query.

It would additionally be valuable to investigate an analytic design approach with a shorter gap between generating and validating an analytic, by demonstrating an example set of results a user can expect to receive from candidate analytics before the assembly is completed. This would necessitate some engineering around the automated compilation and deployment of analytics in an interactive timeframe, but could significantly aid a user's understanding of available plans.

Finally, there are currently two primitives used in the planning algorithm; basic PEs and the join operator. We propose that there may be value in modelling more advanced primitives (*e.g.*, reductions or filters) and exploring their impact on both usability and the range of analytics MENDELEEV can express.

ACKNOWLEDGMENTS

With thanks to Graham Cormode and Steven Wright for their insightful comments and feedback on early drafts of this paper. This work was funded under an Industrial EPSRC CASE Studentship, entitled "Platforms for Deploying Scalable Parallel Analytic Jobs over High Frequency Data Streams".

REFERENCES

[1] N. Marz, "Apache Storm." [Online]. Available: http://storm.apache.org/

[2] R. Rea and K. Mamidipaka, "IBM InfoSphere Streams: Enabling Complex Analytics with Ultra-Low Latencies on Data in Motion," 2009.

[3] K. Shvachko, H. Kuang *et al.*, "The Hadoop Distributed File System," in *Proceedings of the 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2010.

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, 2008.

[5] M. Zaharia, M. Chowdhury *et al.*, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010.

[6] A. Fuchs, "Accumulo - Extensions to Google's Bigtable Design," National Security Agency, Tech. Rep., March 2012.

[7] J. Yu, B. Benatallah, F. Casati *et al.*, "Understanding mashup development," *Internet Computing, IEEE*, vol. 12, no. 5, 2008.

[8] M. Pruett, *Yahoo! Pipes*, 1st ed. O'Reilly, 2007.

[9] M. Altinel, P. Brown *et al.*, "Damia: a data mashup fabric for intranet applications," in *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007.

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

[11] J. Wong, "Marmite: Towards end-user programming for the web," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2007.

[12] F. Daniel, C. Rodriguez *et al.*, "Discovery and reuse of composition knowledge for assisted mashup development," in *Proceedings of the 21st International Conference on World Wide Web*. New York, NY, USA: ACM, 2012.

[13] Google, Inc., "Google Mashup Editor," http://editor.googlemashups.com.

[14] 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*. New York, NY, USA: ACM, 2013.

[15] M. Kornacker and J. Erickson, "Cloudera Impala: real-time queries in Apache Hadoop," 2012. [Online]. Available: http://blog.cloudera.com/

[16] N. Jain, S. Mishra *et al.*, "Towards a streaming SQL standard," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, 2008.

[17] Y. Law, H. Wang *et al.*, "Query languages and data models for database sequences and data streams," in *Proceedings of the 30th International Conference on Very Large Data Bases*. VLDB Endowment, 2004.

[18] C. Grelck, S.-B. Scholz, and A. Shafarenko, "S-Net: A typed stream processing language," in *Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages (IFL'06), Budapest, Hungary*, ser. Technical Report 2006-S01, Z. Horváth and V. Zsók, Eds. Eötvös Loránd University, Faculty of Informatics, Budapest, Hungary, 2006, pp. 81–97.

[19] K. Whitehouse, F. Zhao *et al.*, "Semantic streams: A framework for composable semantic interpretation of sensor data," in *Wireless Sensor Networks*. Springer, 2006.

[20] E. Sirin and B. Parsia, "Planning for semantic web services," in *Semantic Web Services, 3rd International Semantic Web Conference*, 2004.

[21] D. Nau, T. Au *et al.*, "SHOP2: An HTN planning system," *Journal of Artificial Intelligence Research*, vol. 20, 2003.

[22] M. Pistore, P. Traverso *et al.*, "Automated synthesis of composite bpel4ws web services," in *Proceedings of the IEEE International Conference on Web Services*. IEEE, 2005.

[23] M. D. Wilkinson and M. Links, "Biomoby: an open source biological web services proposal," *Briefings in bioinformatics*, vol. 3, no. 4, 2002.

[24] T. Oinn, M. Addis *et al.*, "Taverna: a tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, 2004.

[25] K. T. Stolee, S. Elbaum *et al.*, "Solving the search for source code," *ACM Transactions on Software Engineering and Methodology*, 2014.

[26] R. Bergmann and Y. Gil, "Retrieval of semantic workflows with knowledge intensive similarity measures," in *Case-Based Reasoning Research and Development*. Springer, 2011.

[27] I. Constantinescu, B. Faltings *et al.*, "Large scale, type-compatible service composition," in *Proceedings of the IEEE International Conference on Web Services*. IEEE, 2004.

[28] A. Riabov, E. Boillet *et al.*, "Wishful search: Interactive composition of data mashups," in *Proceedings of the 17th International Conference on World Wide Web*. New York, NY, USA: ACM, 2008.

[29] A. Riabov and Z. Liu, "Planning for stream processing systems," in *Proceedings of the National Conference on Artificial Intelligence*, vol. 20, no. 3, 2005.

[30] A. Riabov and Z. Liu, "Scalable planning for distributed stream processing systems," in *Proceedings of The International Conference on Automated Planning and Scheduling*. AAAI Press, 2006.

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

[32] M. Birbeck and S. McCarron, "CURIE Syntax 1.0: A syntax for expressing Compact URIs," 2008.

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

[34] T. Veldhuizen, "Expression templates," *C++ Report*, vol. 7, no. 5, 1995.

[35] C. Ogbuji *et al.*, "FuXi 1.4: A Python-based, bi-directional logical reasoning system for the semantic web," https://code.google.com/p/fuxi/.

[36] P. Coetzee, M. Leeke, and S. Jarvis, "Towards unified secure on- and off-line analytics at scale," *Parallel Computing*, vol. 40, no. 10, 2014.