Jaql: A Scripting Language for Large Scale Semistructured Data Analysis

Kevin S. Beyer Vuk Ercegovac Rainer Gemulla[*] Andrey Balmin

Mohamed Eltabakh[†] Carl-Christian Kanne Fatma Ozcan Eugene J. Shekita

IBM Research - Almaden
San Jose, CA, USA

{kbeyer,vercego,abalmin,ckanne,fozcan,shekita}@us.ibm.com
rgemulla@mpi-inf.mpg.de, meltabakh@cs.wpi.edu

ABSTRACT

This paper describes Jaql, a declarative scripting language for analyzing large semistructured datasets in parallel using Hadoop's MapReduce framework. Jaql is currently used in IBM's InfoSphere BigInsights [5] and Cognos Consumer Insight [9] products. Jaql's design features are: (1) a flexible data model, (2) reusability, (3) varying levels of abstraction, and (4) scalability. Jaql's data model is inspired by JSON and can be used to represent datasets that vary from flat, relational tables to collections of semistructured documents. A Jaql script can start without any schema and evolve over time from a partial to a rigid schema. Reusability is provided through the use of higher-order functions and by packaging related functions into modules. Most Jaql scripts work at a high level of abstraction for concise specification of logical operations (e.g., join), but Jaql's notion of *physical transparency* also provides a lower level of abstraction if necessary. This allows users to pin down the evaluation plan of a script for greater control or even add new operators. The Jaql compiler automatically rewrites Jaql scripts so they can run in parallel on Hadoop. In addition to describing Jaql's design, we present the results of scale-up experiments on Hadoop running Jaql scripts for intranet data analysis and log processing.

1. INTRODUCTION

The combination of inexpensive storage and automated data generation is leading to an explosion in data that companies would like to analyze. Traditional SQL warehouses can certainly be used in this analysis, but there are many applications where the relational data model is too rigid and where the cost of a warehouse would be prohibitive for accumulating data without an immediate need for it [12]. As a result, alternative platforms have emerged for large scale data analysis, such as Google's MapReduce [14], Hadoop [18], the open-source implementation of MapReduce, and Microsoft's Dryad [21]. MapReduce was initially used by Internet companies to analyze Web data, but there is a growing interest in using Hadoop [18], to analyze enterprise data.

[*]Max-Planck-Institut für Informatik, Saarbrücken, Germany. This work was done while the author was visiting IBM Almaden.

[†]Worcester Polytechnic Institute (WPI), Worcester, USA. This work was done while the author was visiting IBM Almaden.

To address the needs of enterprise customers, IBM recently released InfoSphere BigInsights [5] and Cognos Consumer Insights [9] (CCI). Both BigInsights and CCI are based on Hadoop and include analysis flows that are much more diverse than those typically found in a SQL warehouse. Among other things, there are analysis flows to annotate and index intranet crawls [4], clean and integrate financial data [1, 36], and predict consumer behavior [13] thru collaborative filtering.

This paper describes Jaql, which is a declarative scripting language used in both BigInsights and CCI to analyze large semistructured datasets in parallel on Hadoop. The main goal of Jaql is to simplify the task of writing analysis flows by allowing developers to work at a higher level of abstraction than low-level MapReduce programs. Jaql consists of a scripting language and compiler, as well as a runtime component for Hadoop, but we will refer to all three as simply Jaql. Jaql's design has been influenced by Pig [31], Hive [39], DryadLINQ [42], among others, but has a unique focus on the following combination of core features: (1) a flexible data model, (2) reusable and modular scripts, (3) the ability to specify scripts at varying levels of abstraction, referred to as *physical transparency*, and (4) scalability. Each of these is briefly described below.

Flexible Data Model: Jaql's data model is based on JSON, which is a simple format and standard (RFC 4627) for semistructured data. Jaql's data model is flexible to handle semistructured documents, which are often found in the early, exploratory stages of data analysis, as well as structured records, which are often produced after data cleansing stages. For exploration, Jaql is able to process data with no schema or only a partial schema. However, Jaql can also exploit rigid schema information when it is available, for both type checking and improved performance. Because JSON was designed for data interchange, there is a low impedance mismatch between Jaql and user-defined functions written in a variety of languages.

Reusability and Modularity: Jaql blends ideas from programming languages along with flexible data typing to enable encapsulation, composition, and ultimately, reusability and modularity. All of these features are not only available for end users, but also used to implement Jaql's core operators and library functions. Borrowing from functional languages, Jaql supports lazy evaluation and higher-order functions, i.e., functions are treated as first-class data types. Functions provide a uniform representation for not only Jaql-bodied functions but also those written in external languages. Jaql is able to work with expressions for which the schema is unknown or only partially known. Consequently, users only need to be concerned with the portion of data that is relevant to their task. Partial schema also allows a script to make minimal restrictions on the schema and thereby remain valid on evolving input. Finally, many related functions and their corresponding resources can be bundled together into modules, each with their own namespace.

Physical Transparency: Building on higher order functions, Jaql's evaluation plan is entirely expressible in Jaql's syntax. Consequently, Jaql exposes every internal physical operator as a function in the language, and allows users to combine various levels of abstraction within a single Jaql script. Thus the convenience of a declarative language is judiciously combined with precise control over query evaluation, when needed. Such low-level control is somewhat controversial but provides Jaql with two important benefits. First, low-level operators allow users to pin down a query evaluation plan, which is particularly important for well-defined tasks that are run regularly. After all, query optimization remains a challenging task even for a mature technology such as an RDBMS. This is evidenced by the widespread use of optimizer "hints". Although useful, such hints only control a limited set of query evaluation plan features (such as join orders and access paths). Jaql's physical transparency goes further in that it allows full control over the evaluation plan.

The second advantage of physical transparency is that it enables *bottom-up extensibility*. By exploiting Jaql's powerful function support, users can add functionality or performance enhancements (such as a new join operator) and use them in queries right away. No modification of the query language or the compiler is necessary. The Jaql rewriter exploits this design principle to compile high-level declarative expressions to lower-level function calls, which also have a valid representation in Jaql's syntax. This well-known compiler design pattern is called *source-to-source compilation* [27] and, to the best of our knowledge, Jaql is the first data processing language that exploits this technique and makes it available to end users.

In summary, physical transparency offers users a complete spectrum of control: declarative expressions are preferable when they work, hints cover the common lapses in optimization, and physical transparency offers direct access to a Jaql plan when needed.

Scalability: Jaql is designed to parallelize scripts over large collections of semistructured objects distributed among a cluster of commodity servers. The achieved scalability is essential for both large datasets and expensive per-object computations. By focusing on large, partitionable collections, many of the innovations developed for shared nothing databases are applicable. Some of these techniques—such as parallel scan, repartitioning, and parallel aggregation—are also present in MapReduce, along with fault-tolerance and dynamic scheduling to circumvent hardware and software failures. Given a script, Jaql translates it into an evaluation plan consisting of MapReduce jobs and, when necessary, intermediate sequential steps. Results in Section 8 show that Jaql scales well and illustrates how physical transparency enabled us to parallelize typically sequential flows for analyzing large datasets.

In the remainder of the paper, we first review related work in Section 2 and background in Section 3. Jaql's data model and schema is described in Section 4. Core language features, an example of physical transparency, and the system implementation are described in Sections 5, 6, and 7 respectively. Section 8 contains the experimental results and we conclude in Section 9.

2. RELATED WORK

Many systems, languages, and data models have been developed to process massive data sets, giving Jaql a wealth of technologies and ideas to build on. In particular, Jaql's design and implementation draw from shared nothing databases [15, 38], MapReduce [14], declarative query languages, functional and parallel programming languages, the nested relational data model, and XML. While Jaql has features in common with many systems, we believe that the combination of features in Jaql is unique. In particular, Jaql's use of higher-order functions is a novel approach to physical transparency, providing precise control over query evaluation.

Jaql is most similar to the data processing languages and systems that were designed for scale-out architectures such as MapReduce and Dryad [21]. In particular, Pig [31], Hive [39], and DryadLINQ [42] have many design goals and features in common. Pig is a dynamically typed query language with a flexible, nested relational data model and a convenient, script-like syntax for developing data flows that are evaluated using Hadoop's MapReduce. Hive uses a flexible data model and MapReduce with syntax that is based on SQL (so it is statically typed). Microsoft's Scope [10] has similar features as Hive, except that it uses Dryad as its parallel runtime. In addition to physical transparency, Jaql differs from these systems in three main ways. First, Jaql scripts are reusable due to higher-order functions. Second, Jaql is more composable: all language features can be equally applied to any level of nested data. Finally, Jaql's data model supports partial schema, which assists in transitioning scripts from exploratory to production phases of analysis. In particular, users can contain such changes to schema definitions without a need to modify existing queries or reorganize data (see Section 4.2 for an example).

ASTERIX Query Language (AQL) [3] is also composable, supports "open" records for data evolution, and is designed for scalable, shared-nothing data processing. Jaql differs in its support for higher-order functions and physical transparency.

While SQL, Jaql, and Pig are data processing languages, Microsoft's LINQ [28], Google's FlumeJava [11], the PACTs programming model [2], and the Cascading project [8] offer a programmatic approach. We focus on LINQ, due to its tighter integration with the host language (e.g., C#). LINQ embeds a statically typed query language in a host programming language, providing users with richer encapsulation and tooling that one expects from modern programming environments. DryadLINQ is an example of such an embedding that uses the Dryad system for its parallel runtime. Jaql differs in three main ways. First, Jaql is a scripting language and lightweight so that users can quickly begin to explore their data. Second, Jaql exploits partial schema instead of a programming language type system. Finally, it is not clear whether DryadLINQ allows its users to precisely control evaluation plans to the same degree that is supported by Jaql's physical transparency.

Sawzall [34] is a statically-typed programming language for Google's MapReduce, providing domain specific libraries to easily express the logic of a single MapReduce job. In comparison, Jaql can produce data flows composed of multiple MapReduce jobs.

A key ingredient for reusability is for functions to be polymorphic, often through table-valued parameters [22]. Examples of systems that support such functionality include AsterData's SQL/MapReduce [16] and Oracle's pipelined table functions [32]. AT&T's Daytona [17] is a proprietary system that efficiently manages and processes massive flat files using SQL and procedural language features. NESL [6] is a parallel programming language specialized for nested data. In contrast to these systems, Jaql is designed to process semistructured data and its support for higher-order functions offers more options for reusability.

RDBMS' and native XML data management systems offer a wide range of flexibility for processing semistructured data and have had a significant influence on Jaql's design. For reusability, Jaql includes many of the features found in XQuery [41] such as functions, modules/namespaces, and higher-order functions [37]. In addition, except for DB2's PureXML [20] and MarkLogic Server [30], most XQuery systems have not been implemented for shared-nothing architectures. Finally, Jaql's physical transparency is a significant departure from such declarative technology.

3. HADOOP BACKGROUND

Jaql relies on Hadoop's MapReduce infrastructure to provide parallelism, elasticity, and fault tolerance for long-running jobs on commodity hardware. Briefly, MapReduce is a parallel programming framework that breaks a *job* into *map* and *reduce* tasks. In Hadoop, each *task* is managed by a separate process, all tasks on a single node are managed by a *TaskTracker*, and all TaskTrackers in a cluster are managed by a single *JobTracker*. The input data set– e.g., an HDFS file spread across the cluster– is partitioned into *splits*. Each map task scans a single split to produce an intermediate collection of (*key*, *value*) pairs. If appropriate, the map output is partially reduced using a *combine* function. The map output is redistributed across the cluster by *key* in the *shuffle* phase so that all *values* with the same *key* are processed by the same reduce task. Both the combine function and reduce phase are optional. For the remainder of the paper, the term MapReduce is used interchangeably with the Hadoop MapReduce implementation and we distinguish between the general MapReduce programming framework where needed.

Unlike traditional databases, MapReduce clusters run on less reliable hardware and are significantly less controlled; for example MapReduce jobs by definition include significant amounts of user code with their own resource consumption, including processes and temporary files. Hence, MapReduce nodes are less stable than a DBA managed database system, which means that nodes frequently require a reboot to clean out remnants from previous tasks. As a result, the system replicates input data, materializes intermediate results, and restarts failed tasks as required.

4. JAQL'S DATA MODEL AND SCHEMA

Jaql was designed to process large collections of semistructured and structured data. In this section, we describe Jaql's data model (JDM) and schema language.

4.1 The Data Model

Jaql uses a very simple data model: a JDM *value* is either an atom, an array, or a record. Most common atomic types are supported, including strings, numbers, nulls, and dates. Arrays and records are compound types that can be arbitrarily nested. In more detail, an *array* is an ordered collection of values and can be used to model data structures such as vectors, lists, sets, or bags. A *record* is an unordered collection of name-value pairs– called *fields*– and can model structs, dictionaries, and maps.

Despite its simplicity, JDM is very flexible. It allows Jaql to operate with a variety of different data representations for both input and output, including delimited text files, JSON files, binary files, Hadoop's SequenceFiles, relational databases, key-value stores, or XML documents. Jaql often uses binary representations of JDM, but the textual representation expresses Jaql literals and is useful for debugging and interoperating with other languages.

Textual Representation. Figure 1 shows the grammar for the textual representation of JDM values. The grammar unambiguously identifies each data type from the textual representation. For example, strings are wrapped into quotation marks (`"text"`), numbers are represented in decimal or scientific notation (`10.5`), and booleans and null values are represented as literals (`true`). As for compound types, arrays are enclosed in brackets (`[1,2]`) and records are enclosed in curly braces (`{a:1, b:false}`). Example 1 models a *collection* of product reviews as an array of records; we interchangeably refer to such top-level arrays as collections. Since JDM textual representation is a part of Jaql grammar, we use the terms JDM value and Jaql value interchangeably. Also note that this representation closely resembles JSON. In fact, Jaql's grammar and JDM subsumes JSON: Any valid JSON instance can be read by Jaql. The converse it not true, however, as JDM has more atomic types. Since JSON is a useful format for data exchange between programs written in many programming languages (e.g., Java, Python, Ruby, ...), JDM's closeness to JSON simplifies data exchange with all those languages.

```
<value>  ::= <atom> | <array> | <record>
<atom>   ::= <string> | <binary> | <double> |
             <date> | <boolean> | 'null' | ...
<array>  ::= '[' ( <value> (',' <value>)* )? ']'
<record> ::= '{' ( <field> (',' <field>)* )? '}'
<field>  ::= <name> ':' <value>
```

Figure 1: Grammar for textual representation of Jaql values.

Example 1 *Consider a hypothetical company KnowItAll, Inc. that maintains a collection of product reviews. The following is an excerpt in JDM's textual representation.*

```
[
  { uri: "http://www.acme.com/prod/1.1reviews",
    content: "Widget 1.1 review by Bob ...",
    meta: { author  : "Bob",
            contentType: "text",
            language: "EN" } },


  { uri: "file:///mnt/data/docs/memo.txt",
    content: "The first memo of the year ...",
    meta: { author: "Alice",
            language: "EN" } },
  ...
]
```

Relationship to other data models. JDM consciously avoids many complexities that are inherent in other semistructured data models, such as the XQuery Data Model (XDM) and the Object Exchange Model (OEM). For example, JDM does not have node identity or references. As a consequence, Jaql does not have to deal with multiple equality semantics (object and value) and Jaql values are always trees (and not graphs). These properties not only simplify the Jaql language, but also facilitate parallelization.

4.2 The Schema

Jaql's ability to operate without a schema, particularly in conjunction with self-describing data, facilitates exploratory data analysis because users can start working with the data right away, without knowing its complete type. Nevertheless, there are many well known advantages to schema specification, including static type checking and optimization, data validation, improved debugging, and storage and runtime optimization. For these reasons, Jaql allows and exploits schema specifications. Jaql's schema and schema language are inspired by XML Schema [40], RELAX NG [35], JSON schema [25], and JSONR [24]. The schema information

```
<schema> ::= <basic> '?'? ('|' <schema>)*
<basic>  ::= <atom> | <array> | <record>
              | 'nonnull' | 'any'
<atom>   ::= 'string' | 'double' | 'null' | ...
<array>  ::= '[' ( <schema> (',' <schema>)*
                  '...'? )? ']'
<record> ::= '{' ( <field> (',' <field>)* )? '}'
<field>  ::= (<name> '?'? | '*') (':' <schema>)?
```

Figure 2: Grammar for schema language.

does not need to be complete and rigid because Jaql supports partial schema specification.

Jaql uses a simple pattern language, shown in Figure 2, to describe schemas. The schema of an atomic type is represented by its name, e.g., `boolean` or `string`. The schema of an array is represented by using a list of schemas in brackets, e.g., `[boolean, string]`. Optionally, usage of `...` indicates that the last array element is repeatable, e.g., `[string, double ...]`. The schema of records are defined similarly, e.g., `{uri: string}` describes a record with a single field `uri` of type string. Question marks indicate optionality (for fields) or nullability (otherwise). We refer to a schema as *regular* if it can be represented with the part of the language just described. Regular schemas give a fairly concise picture of the actual types in the data. In contrast, *irregular* schemas make use of wildcards such as `nonnull` or `any` for values of arbitrary type, * for fields of arbitrary name, or omission of a field's type, or to specify different alternative schemas (using `|`). In general, irregular schemas are more vague about the data. The simplest irregular schema is `any`; it matches any value and is used in the absence of schema information.

Example 2 *The excerpt of the KnowItAll data shown in Example 1 conforms to the following regular schema (the* `?` *marks optional fields):*

```
[ { uri: string, content: string,
    meta: { author: string, contentType?: string,
            language: string }
  } ... ],
```

This schema is unlikely to generalize to the entire dataset, but the following irregular schema may generalize:

```
[ { uri: string, content: any, meta: {*: string} }
  ... ].
```

The ability to work with regular and irregular schemas allows Jaql to exploit schema information in various degrees of detail. In contrast to many other languages, Jaql treats schema as merely a *constraint* on the data: A data value (and its type) remains the same whether or not its schema is specified.[1] This makes it possible to add schema information– whether partial or complete– as it becomes available without changing the data type or any of the existing Jaql scripts. For example, initial screening of the KnowItAll dataset might be performed using schema `[{*}...]`, which indicates that the data is a collection of arbitrary records. When in later phases, as more and more information becomes available, the schema is refined to, say, `[{uri:string,*}...]`, all existing scripts can be reused, but will benefit from static type checking and increased efficiency. In contrast, refinement of schema often requires a change of data type, and consequently query, in many other languages. For example, a dataset of arbitrary records is modeled

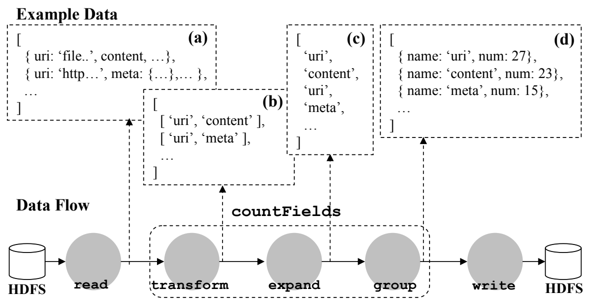

Figure 3: Conceptual data flow for counting fields.

as `[{fields:map}...]` in Pig [31] and LINQ [28], which both support flexible map containers that can store heterogeneous data. When information about field `uri` becomes available, it is propagated by pulling `uri` out of `fields`. The schema *and data type* becomes `[{uri:string, fields:map}...]` and all references to `uri` in the query have to be modified.

5. JAQL LANGUAGE OVERVIEW

This section describes the core features of the Jaql language. A series of examples is used to emphasize how the language meets its design goals of flexibility and reusability.

5.1 Core Expressions

Jaql is a scripting language. A *Jaql script* is simply a sequence of statements, and each *statement* is either an import, an assignment, or an *expression*. The following example describes a simple task and its Jaql implementation.

Example 3 *Consider a user who wants to gain some familiarity with the KnowItAll data by learning which fields are present and with what frequency. Figure 3 shows a conceptual data flow that describes this task. The data flow consists of a sequence of "operators"; example data is shown at various intermediate points. The* `read` *operator loads raw data, in this case from Hadoop's Distributed File System (HDFS), and converts it into Jaql values. These values are processed by the* `countFields` *subflow, which extracts field names and computes their frequencies. Finally, the* `write` *operator stores the result back into HDFS.*

This task is accomplished by the following Jaql script:

```
 1. import myrecord;
 2.
 3. countFields = fn(records) (
 4.   records
 5.   -> transform myrecord::names($)
 6.   -> expand
 7.   -> group by fName = $ as occurrences
 8.      into { name: fName, num: count(occurrences) }
 9. );
10.
11. read(hdfs("docs.dat"))
12. -> countFields()
13. -> write(hdfs("fields.dat"));
```

Working our way from the bottom of the script to the top, the conceptual data flow of Figure 3 is specified on lines 11–13. `read`, `hdfs`, `countFields`, and `write` are *functions*; their composition and invocation constitutes an expression. Lines 3–9 constitute the assignment that defines the `countFields` function. The `countFields` function depends on an externally defined function, `names`, which is imported on line 1 from the `myrecord` module.

[1]In contrast, parsing the same XML document with or without an XML Schema may result in different XQuery data model instances with different data types.

Unix pipes inspired Jaql's syntax "->" for flowing the output of one expression to the next. Although this symbol "->" has multiple interpretations in Jaql, the expression to the left of "->" always provides the context for what is on the right-hand side. Thus, e_1->e_2 can be read as "e_1 flows into e_2". The pipe syntax explicitly shows the data flow in a Jaql script, making it easier to read and debug. We chose this syntax to avoid defining variables (as in Pig[31]), or WITH clauses (as in SQL), for every computational step. The pipe syntax is also more readable than the functional notation (as in XQuery[41]), when a series of functions are invoked back-to-back.

Jaql has several expressions for manipulating data collections (also referred to as arrays), including `transform`, `expand`, `filter`, `join`, `sort`, `group by`, multi-input `group by`, `merge`, and `tee`. Note that some of these expressions (such as `join`, `group by`, `filter`,`transform`, `merge`) are found in database management systems, while others (such as `tee`) are typical for ETL engines. This section illustrates some of the core expressions using our running example and we refer the reader to [23] for further details.

transform: The `transform` expression applies a function (or projection) to every element of an array to produce a new array. It has the form e_1`->transform` e_2, where e_1 is an expression that describes the input array, and e_2 is applied to each element of e_1.

Consider lines 4–8 of Example 3 as well as the corresponding data shown in Figure 3. In lines 4 and 5 of the example, e_1 refers to the `records` variable, and e_2 invokes the `names` function from the `myrecord` module. The `names` function takes as input a record and produces an array of field names (represented as strings). The output after `transform` is shown Figure 3(b). By default, `transform` binds each element of e_1 to a variable named $, which is placed in e_2's scope. As with most Jaql expressions, the `each` keyword is used to override the default iteration variable. For example, `...->transform each r myrecord::names(r) ...` renames $ to `r`. Example 5 illustrates how such renaming is useful when iterating over nested data.

expand: The `expand` expression is most often used to unnest its input array. It differs from `transform` in two primary ways: (1) e_2 must produce a value v that is an array type, and (2) each of the elements of v is returned to the output array, thereby removing one level of nesting.

The `expand` expression in line 6 unnests the array of field names, cf. Figure 3(c). In this case, e_2 is assumed to be the identity function. That is, each element in the output from the previous `transform` operator is assumed to be an array (of strings, in this case). If needed, the iteration variable $ (or override) is in scope for e_2.

group by: Similar to SQL's `GROUP BY`, Jaql's `group by` expression partitions its input on a grouping expression and applies an aggregation expression to each group. In contrast to SQL, Jaql's `group by` operates on multiple input collections (e.g., *co-group* [31]), supports complex valued grouping keys, and provides the entire group to the aggregation expression.

In the example, the `group by` expression on lines 7–8 counts the number of occurrences of each distinct field name and returns an array of records. The grouping key, `fName`, is assigned to $, which is bound to each value from the input (e.g., a string). The group, `occurrences`, is assigned to the array of values (e.g., string array) associated with each distinct key. Both `fName` and `occurrences` are in scope for the `into` clause on line 8, where they are used to construct a record (denoted by {`name, num`}) per group. The `name` field is set to `fName` and a `count` is computed over `occurrences`.

filter: The `filter` expression, e`->filter` p, retains input values from e for which predicate p evaluates to `true`. The following example modifies Example 3 by retaining only those field names that belong to an *in-list*:

```
...
5.   -> transform myrecord::names($)
6.   -> expand
7.   -> filter $ in [ "uri", "meta" ]
...
```

join: The `join` expression supports equijoin of 2 or more inputs. All of the options for inner and outer joins are also supported. See Example 9 for an example of a join between two collections.

union: The `union` expression is a Jaql function that merges multiple input arrays into a single output array. It has the form: `union(`$e_1, \ldots$`)` where each e_i is an array.

tee: The `tee` expression is a Jaql function that takes an array as input that it outputs "as-is", along with broadcasting it to one or more expressions. It is a generalization of Unix `tee` where a stream can be redirected to a file and consumed by another process. Jaql's `tee` has the form: e_0`->tee(`$e_1, \ldots e_n$`)`. Each of the expressions, $e_1, \ldots e_n$ consume e_0 and `tee` outputs e_0. Typically, $e_1, \ldots, e_n$ terminate in a `write` so are side-effecting. The following modifies Example 3 by transforming the result of the `group by` on lines 7–8 and writing out two additional files, "low" and "high". Note that the `countFields` function still returns the same result since `tee` passes through its input.

```
 ...
9.  -> transform $.num
10. -> tee( -> filter $ <= 10 -> write(hdfs("low")),
11.         -> filter $ > 10 -> write(hdfs("high")))
 ...
```

Control-flow: The two most commonly used control-flow expressions in Jaql are `if-then-else` and `block` expressions. The `if-then-else` expression is similar to conditional expressions found in most scripting and programming languages. A `block` establishes a local scope where zero or more local variables can be declared and the last statement provides the return value of the block. The `block` expression is similar to XQuery'y `LET` clause. The following modifies the `group by` expression in Example 3. Now, the record that is returned per group computes a `count` and a classification that is based on the `count`.

```
...
7.  -> group by fName = $ as occurrences
8.     into ( n = count(occurrences),
9.            b = if( n <= 10 ) "low" else "high",
10.           { name: fName, num: n, bin: b }
11.         )
...
```

5.2 Functions

Functions are first-class values in Jaql, i.e., they can be assigned to a variable and are high-order in that they can be passed as parameters or used as a return value. Functions are the key ingredient for *reusability*: Any Jaql expression can be encapsulated in a function, and a function can be parameterized in powerful ways. Also, functions provide a principled and consistent mechanism for *physical transparency* (see Section 6).

In Example 3, the `countFields` function is defined on lines 3–9 and invoked on line 12. In Jaql, named functions are created by constructing a *lambda function* and assigning it to a variable. Lambda functions are created via the `fn` expression; in the example, the resulting function value is assigned to the `countFields` variable. The function has one parameter named `records`. Although not shown, parameters can be constrained by a schema when desired. Note that the definition of `countFields` is abstracted from the details of where its input is stored and how its formatted. Such details are only specified during invocation (lines 11–12), which makes `countFields` reusable.

Jaql makes heavy usage of the *pipe symbol* -> in its core expressions. Lines 12 and 13 in the example script show a case where the right-hand side is not a core expression but a function invocation. In this case, the left-hand side is bound to the first argument of the function, i.e., e->$f(\ldots) \equiv f(e, \ldots)$. This interpretation *unifies core expressions and function invocations* in that input expressions can occur up front. User-defined functions, whether they are defined in Jaql or an external language, thus integrate seamlessly into the language syntax.

To see this, compare the Jaql expressions

read(e_1) -> **transform** e_2 -> *myudf*() -> **group by** e_3

to the equivalent but arguably harder-to-read expression

myudf(*read*(e_1) -> **transform** e_2) -> **group by** e_3.

5.3 Extensibility

Jaql's set of built-in functions can be extended with *user-defined functions* (UDF) and *user-defined aggregates* (UDA), both of which can be written in either Jaql or an external language. Such functions have been implemented for a variety of tasks, ranging from simple string manipulation (e.g., `split`) to complex tasks such as information extraction (e.g., via System T [26]) or statistical analysis (e.g., via R [13] and SPSS[2]). As mentioned before, the exchange of data between Jaql and user code is facilitated by Jaql's use of a JSON-based data model.

Example 4 *Continuing from Example 3, suppose that the user wants to extract names of products mentioned in the* `content` *field. We make use of a UDF that, given a document and a set of extraction rules, uses System T for information extraction. The following Jaql script illustrates UDF declaration and invocation:*

```
1. systemt = javaudf("com.ibm.ext.SystemTWrapper");
2. rules   = read(hdfs("rules.aql"));
3
4. read(hdfs("docs.dat"))
5. -> transform { author: $.meta.author,
6.                products: systemt($.content, rules) };
```

When run on the example data, the script may produce

```
[ { author: "Bob", products: ["Widget 1.1",
                              "Service xyz",...] },
  { author: "Alice", products: [ ... ] }, ... ].
```

Example 4 illustrates how functions and semistructured data are often used in Jaql. The *javaudf* function shown on line 1 is a function that returns a function which is parameterized by a Java class name *c*, and when invoked, knows how to bind invocation parameters and invoke the appropriate method of *c*. By leveraging Jaql functions, Jaql does *not* distinguish between native Jaql functions and external UDFs and UDAs—both can be assigned to variables, passed as parameters, and returned from functions. The *externalFn* function provides a similar wrapping of external programs (run in separate a process) with which Jaql can exchange data.

5.4 Processing Semistructured Data

While Jaql's nested data model and partial schemas let users flexibly represent their data, Jaql's language features enable powerful processing of semistructured data. In this section, we highlight the Jaql features that are most commonly exploited. First, Jaql's expressions are composable— expressions that can be applied to top-level, large collections can also be applied to data that is deeply nested. For example, `transform` can be used on a deeply nested array as easily as it can be used on a large, top-level array. Next, Jaql includes a path language that lets users "dig into" nested data, along with complex constructors to generate complex data. Finally, higher-order functions, schema inference, control-flow, and various syntactic conveniences are often combined for rich, reusable functionality.

5.4.1 Language Composability

The sample result from Example 4 illustrates a common usage pattern: per string (e.g., `$.content`), SystemT enriches each record with extracted data (also called annotations). In this case, the SystemT *rules* found multiple products. More generally, extracted data is more deeply nested. For example, multiple types of annotations can be found (e.g., "products" and "services") where each type includes zero or more annotations, and each annotation includes where in the source document it was found. As a result, the relatively "flat" input data is transformed into more nested data. Often, the script shown in Example 4 is followed by additional steps that filter, transform or further classify the extracted data. Jaql's composability is crucial to support such manipulation of nested data.

Example 5 *Consider a* `wroteAbout` *dataset that contains pairs of authors and product names (similar to Example 4), and a* `products` *dataset that contains information about individual products. The* `wroteAbout` *dataset is defined as follows:*

```
1. wroteAbout = read(hdfs("docs.dat"))
2. -> transform { author: $.meta.author,
3.                products: systemt($.content, rules) }
4. -> transform each d (
5.      d.products -> transform { d.author, product: $ }
6.    )
7. -> expand;
```

The definition in Example 5 is similar to the one used in Example 4, but unnests the `products` array. The pairing of author's with products makes use of Jaql's composability by nesting `transforms`. The `transform` on Line 4 iterates over annotated documents (outer) and associates the `author` with each nested `product` (inner). The products per document are iterated over using a nested `transform` on Line 5, and the `each` from Line 4 lets the user access the outer context.

Jaql treats *all* its expressions uniformly. In particular, there is no distinction between "small" expressions (such as additions) and "large" expressions (such as a group by). As a consequence, all expressions can be used at both the top-level and within nested structures. Jaql is similar to XQuery [41] and LINQ [28] in this respect, but differs from Pig [31] and Hive [39] which provide little support for manipulating nested structures without prior unnesting. Limiting the language to operate on mostly the top level or two may simplify the implementation and early learning of the language but becomes tedious when manipulating richer objects.

5.4.2 Path Expressions and Constructors

Example 5 also illustrates basic accessors and constructors for nested data. The `$.meta.author` is an example of a field access. There are numerous syntactic short-hands to "dig into" nested data. For example, `wroteAbout[*].product` projects `product` from each record, producing an array of products (with duplicates). These can easily be deduplicated and counted

[2]See http://www.r-project.org and http://www.spss.com.

as follows: `wroteAbout[*].product -> distinct() -> count()`.

Records and arrays are constructed by combining the JSON-like representation with expressions. In Line 5, a record is constructed, `{ ...}` and populated with two fields. The first field inherits its name from `d.author` and a new field, `product`, is declared. Record construction also includes syntax for optionally dropping a field name in the event that its value expression evaluates to null. Another common task that is supported is to retain all fields except a few which are either dropped or replaced.

5.5 Error Handling

Errors are common place when analyzing large, complex data sets. A non-exhaustive list of errors includes corrupt file formats, dynamic type errors, and a myriad of issues in user-defined code that range from simple exceptions, to more complicated issues such as functions that run too long or consume too much memory. The user must be able to specify how such errors effect script evaluation and what feedback the system must supply to improve analysis.

Jaql handles errors by providing coarse-grained control at the script level and fine-grained control over individual expressions. For coarse-grained control, core expressions (e.g., `transform`, `filter`) have been instrumented to adhere to an *error policy*. The policies thus far implemented control if a script is aborted when there is: (1) any error, or (2) more than *k* errors. When an error occurs, the input to the expression is logged, and in the case where errors are permitted, the expression's output is skipped.

For fine-grained control, the user can wrap an arbitrary expression with `catch`, `fence`, or `timeout` functions. The `catch` function allows an error policy to be specified on a specific expression instead of at the script level. The `fence` function evaluates its input expression in a forked process. Similar to `externalFn`, Jaql supports the option to send and receive data in bulk or per value to the forked process. The `timeout` function places a limit on how long its input expression can run. If exceeded, the expression is terminated and an exception is thrown.

Since fine-grained error handling is implemented using Jaql functions, composing them and having them work in parallel using MapReduce comes for free. Consider the following expression:

```
read(hdfs("docs.dat"))
-> transform catch( timeout( fence(
                         fn(r) myudf(r.content)
                    ), 5000), $.uri);
```

This expression is evaluated as a parallel scan (e.g., a Map-only job). Each *map task* (e.g., parent process) processes a partition of the input and evaluates `myudf` in a child process that it forks (once per map task). Each invocation of `myudf` is passed an input record, r, and limited to 5 seconds. If an exception occurs or the operation times out, the script-level error policy is used and `$.uri` is logged.

6. JAQL'S PHYSICAL TRANSPARENCY

Physical transparency, i.e., the ability for all Jaql plans to be expressed in Jaql's syntax, enables users to hard-wire plans (e.g., force a join order or strategy) as well as *bottom-up extensibility* to get functionality first and abstraction later. The sophisticated Jaql user can add a new run-time operator by means of a new (perhaps higher-order) function. The new operator can be used immediately, without requiring any changes to Jaql internals. If the operator turns out to be important enough to the Jaql community, a Jaql developer can add new syntax, rewrites, statistics, or access methods to Jaql itself. In a traditional database system design, all of these tasks must be accomplished before new run-time functionality is exposed, which makes adding new operators a daunting task.

This section illustrates the extensibility aspect of physical transparency through a scenario where Jaql was used for log processing (Section 7.4 discusses the plan language aspect). While the full details are out of scope for this paper, we explain the overall approach and highlight the key functions required to provide a scalable solution that needed to be aware of the time-ordered log records.

Example 6 *Consider a* `log` *dataset that resembles many Apache HTTP Server error logs or Log4J Java application logs. This dataset contains a sequence of log records with the following schema:*

```
{ date: date, id: long, host: string, logger: string,
  status: string, exception?: string, msg?: string,
  stack?: string }
```

The data in our example log is stored in a text file and originally intended for human consumption. The log records are generated in increasing date order, so the files are sorted by the timestamp. There are two types of log records in the file based on the status field: a single line 'success' record or a multi–line 'exception' record. All records have the first five fields separated by a comma. When the status is 'exception', the next line contains the type of exception and a descriptive message separated by a colon. The next several lines are the stack trace.

To form a single logical record, multiple consecutive lines need to be merged into single record. The following script uses Jaql's built–in tumbling window facility to glue the exception lines with the standard fields to create a single line per record for easy processing in later steps:

```
1. read(lines('log'))
2.  -> tumblingWindow( stop = fn(next) isHeader(next) )
3.  -> transform cleanRec($)
4.  -> write(lines('clean'));
```

The `read` *in line (1) reads the file as a collection of* `lines`*. Next in line (2), the function tumblingWindow is a higher–order function that takes an ordered input and a predicate to define the points where the window breaks*[3]*. The isHeader function returns true when the next line starts with a timestamp and has at least 5 fields. The cleanRec function combines the header and all the exception lines into a single line by escaping the newlines in the stack trace.*

At this stage, Jaql is not clever enough to automatically parallelize this script, so it runs sequentially. For small logs, this is acceptable, but for large logs we clearly need to do better. Physical transparency allows the power user to implement a solution at a lower level of abstraction immediately. In contrast, a user of a traditional database system might make a feature request and wait several years for a solution to be delivered.

Order–sensitive operations like tumbling windows are notoriously more difficult to parallelize than multi–set operations. The high-level description of MapReduce in Section 3 uncovers the main challenge that we deal with here. The input to a Hadoop MapReduce job is partitioned into *splits* that are simply determined by byte offsets into the original file. The log file in Figure 4 is partitioned into four splits, $S_0 - S_3$. Since each split is processed by a separate map task, the challenge is in handling those records that span split boundaries, for example, the "exception" record that straddles splits S_1 and S_2.

The key idea needed to parallelize `tumblingWindows` is the ability to manipulate splits. If all mappers can peek into the next

[3]A complete discussion of all the window facilities is not in this paper's scope.

mapper's split, then all records would be processed. Fortunately, Hadoop's API's are very flexible, making it easy to re-define how a given input is partitioned into splits. `tumblingWindows` was parallelized by a new function called *ptumblingWindow* which directly accesses the splits to pair them into consecutive splits as shown by $S'_0 - S'_2$. This allows mappers to peek into the "next" split.

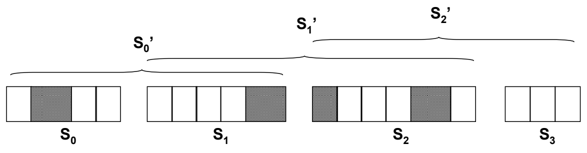

Figure 4: A partitioned log file with "success" (white) and "exception" (shaded) records.

Example 7 *The following script is very similar to the previous one that is shown n Example 6, but the read and tumblingWindow have been composed into a single function, ptumblingWindow, that has been designed to run in parallel:*

```
ptumblingWindow(lines('log'), isHeader)
 -> transform cleanRec($)
 -> write(lines('clean'));
```

The new script remains at a fairly high level, but uses a new parallel version of `tumblingWindow` implemented in Jaql using low–level primitives for split manipulation. Next, we describe the low level operations needed for *ptumblingWindow*, but omit the full Jaql source for conciseness.

Building Blocks: Split manipulation in Jaql is accomplished through three low-level functions: 1) `inputSplits`, 2) `readSplit`, and 3) `worklist`. The `inputSplits` function takes an I/O descriptor as input (see Section 7.1) and returns an array of split descriptors. The `readSplit` function takes a split descriptor as input and returns an iterator over the split's content. So far, these two functions emulate how a map task iterates over its partition of the input. The missing piece is to assign splits to map tasks. The `worklist` gives us such control by providing a *virtual* input to MapReduce where the user controls the number of map tasks to run and what data to pass to each map task. A `worklist` takes an array as input—given n values, n map tasks will be started where the ith map task, $0 \le i < n$, is assigned the array's ith input value. For example, `worklist` is used for synthetic data generation where all that is needed for a given map task is a seed value and distribution parameters.

Putting it Together: `inputSplits` is used to obtain the physical splits (e.g., $S_0 - S_3$). These splits are paired into logical splits (e.g., $S'_0 = (S0, S1)$, $S'_1 = (S_1, S_2), \ldots$)) that are given as input to `worklist`. For each such split, the map task uses `readSplit` to read the first physical split while running `tumblingWindow` sequentially. If needed, the mapper will peek into the first part of the next physical split to find the end of its last record. While we assume that a log record does not span more than two physical splits, it would be straightforward to generalize to larger log records.

While the implementation of *ptumblingWindow* consists of a handful of simple functions, these functions access very low-level Hadoop API's so it is unlikely to be understood by the casual user. The level of abstraction that is needed is comparable to directly programming a MapReduce job. However, physical transparency enabled a solution to the problem and functions allowed these details to be hidden in the implementation of the top-level *ptumblingWindow* function. In addition, *ptumblingwindow* is sufficiently abstract so that it can be applied to any collection. Using features like the ones described here, we have built parallel enumeration, sliding windows, sampling, and various join algorithms, to name a few.

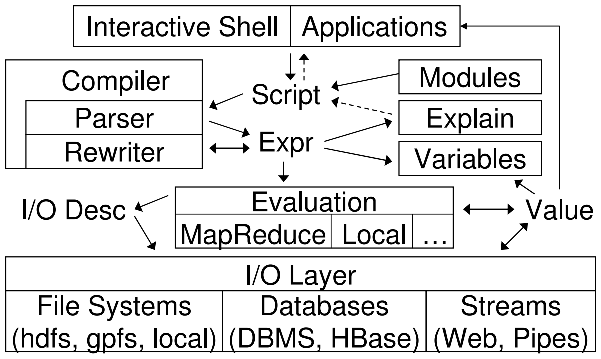

Figure 5: System architecture.

7. JAQL'S SYSTEM IMPLEMENTATION

At a high-level, the Jaql architecture depicted in Figure 5 is similar to most database systems. Scripts are passed into the system from the interpreter or an application, compiled by the parser and rewrite engine, and either explained or evaluated over data from the I/O layer. Jaql modules provide organization and abstraction over reusable components, which are introspected during compilation. Scripts may bind variables to values, or more often to expressions that serve as temporary views. This section describes the major components of the architecture, starting from the lowest layer.

7.1 I/O Layer

The storage layer is similar to a federated database. Rather than requiring data to be loaded into a system-specific storage format based on a pre-defined schema, the storage layer provides an API to access data *in-situ* in other systems, including local or distributed file systems (e.g., Hadoop's HDFS, IBM's GPFS), database systems (e.g., DB2, Netezza, HBase), or from streamed sources like the Web. Unlike federated databases, however, most of the accessed data is stored within the same cluster and the I/O API describes data partitioning, which enables parallelism with data affinity during evaluation. Jaql derives much of this flexibility from Hadoop's I/O API.

Jaql reads and writes many common file formats (e.g., delimited files, JSON text, Hadoop Sequence files). Custom adapters are easily written to map a data set to or from Jaql's data model. The input can even simply be values constructed in the script itself. Adapaters are parameterized through descriptor records which exploit Jaql's flexible data model to represent a wide variety of configurations.

7.2 Evaluation

The Jaql interpreter evaluates the script locally on the computer that compiled the script, but spawns interpreters on remote nodes using MapReduce. A Jaql script may directly invoke MapReduce jobs using Jaql's `mapReduceFn`, but more often, developers use high-level Jaql and depend on the compiler to rewrite the Jaql script into one or more MapReduce jobs, as described in Section 7.3.2.

The `mapReduceFn` function is higher-order; it expects input/output descriptors, a map function, and an optional reduce function. Jaql includes a similar function, `mrAggregate`, that is specialized for running algebraic aggregate functions[4] in parallel using MapReduce. `mrAggregate` requires an `aggregate` parameter that provides a list of aggregates to compute. During evaluation of `mapReduceFn` or `mrAggregate`, Jaql instructs Hadoop to start a MapReduce job, and each map (reduce) task starts a new Jaql interpreter to execute its map (reduce) function.

Of course, not everything can be parallelized, either inherently or because of limitations of the current Jaql compiler. Therefore, some parts of a script are run on the local computer. For example, access to files in the local file system run locally.

7.3 Compiler

The Jaql compiler automatically detects parallelization opportunities in a Jaql script and translates it to a set of MapReduce jobs. The rewrite engine generates calls to `mapReduceFn` or `mrAggregate`, moving the appropriate parts of the script into the map, reduce, and aggregate function parameters. The challenge is to peel through the abstractions created by variables, higher-order functions, and the I/O layer. This section describes the salient features used during the translation.

7.3.1 Internal Representation

Like the internal representation of many programming languages, the Jaql parser produces an abstract syntax tree (AST) where each node, called an *`Expr`*, represents an expression of the language (i.e., an operator). The children of each node represent its input expressions. Other AST nodes include variable definitions and references, which conceptually create cross-links in the AST between each variable reference and its definition. Properties associated with every `Expr` guide the compilation. The most important properties are described below.

Each `Expr` defines its result *schema*, be it regular or irregular, based on its input schemas. The more complete the schema, the more efficient Jaql can be. For example, when the schema is fully regular, storage objects can record structural information once and avoid repeating it with each value. Such information is used to write out smaller temporary files between the map and reduce stage, thereby using disk and network resources more efficiently. However, even limited schema information is helpful; e.g., simply knowing that an expression returns an array enables streaming evaluation in our system.

An `Expr` may be *partitionable* over any of its array inputs, which means that the expression can be applied independently over partitions of its input: $e(I, ...) \equiv \uplus_{P \in parts(I)} e(P, ...)$. In the extreme, an `Expr` may be *mappable* over an input, which means that the expression can be applied equally well to individual elements: $e(I, ...) \equiv \uplus_{i \in I} e([i], ...)$. These properties are used to determine whether MapReduce can be used for evaluation. The `transform` and `expand` expressions are mappable over their input. Logically, any expression that is partitionable should also be mappable, but there are performance reasons to distinguish these cases. For example, Jaql includes a broadcast join strategy that loads the inner table to an in-memory hashtable. Loading and repeated probing are implemented by a function called `lookup`. In this case, `lookup` is only partitionable over its probe input (outer) because we do not want to load the build input (inner) for each key probe.

An `Expr` may deny *remote evaluation*. For example, the `mapReduceFn` function itself is not allowed to be invoked from within another MapReduce job because it would blow up the number of jobs submitted and could potentially cause deadlock if there are not enough resources to complete the second job while the first is still holding resources.

[4]Algebraic aggregation functions are those that can be incrementally evaluated on partial data sets, such as sum or count. As a result, we use combiners to evaluate them.

7.3.2 Rewrites

At present, the Jaql compiler simply consists of a heuristic rewrite engine that greedily applies approximately 100 transformation rules to the `Expr` tree. The rewrite engine fires rules to transform the `Expr` tree, guided by properties, to another semantically equivalent tree. In the future, we plan to add dynamic cost-based optimization to improve the performance of the declarative language features, but our first priority is providing physical transparency to a powerful run-time engine.

The goal of the rewrites is to simplify the script, discover parallelism, and translate declarative expressions into lower-level operators. This task is complicated by higher order functions, variables, and modules. In essence, the rewriter enables database–style optimization while supporting common programming language features. We first describe the most important rules used for simplification through an example, then show sample database-style rewrites (e.g., filter pushdown), and finally describe the rewrite to MapReduce.

Example 8 *Steps in rewriting a function call.*

```
f = fn(r) r.x + r.y;  // declare function f
f({x:1,y:2});         // invoke f
// Rewriter transformation:
1. (fn(r) r.x + r.y)({x:1,y:2}); // variable inline
2. (r = {x:1,y:2}, r.x + r.y);    // function inline
3. {x:1,y:2}.x + {x:1,y:2}.y;    // variable inline
4. 1 + 2;   // constant field access
5. 3;       // compile-time computable
```

Variable inlining: Variables are defined by expressions or values. If a variable is referenced only once in a expression that is evaluated at most once, or the expression is cheap to evaluate, then the variable reference is replaced by its definition. Variable inlining opens up the possibility to compose the variable's definition with the expressions using the variable. In Example 8, variables f and r are inlined.

Function inlining: When a function call is applied to a Jaql function, it is replaced by a block in which parameters become local variables: `(fn(x)` e_1`)(`e_2`)` $\Rightarrow$ `(x =` e_2`,` e_1`)`. Variable inlining may further simplify the function call. In Example 8, the body of function f is inlined.

Filter push-down: Filters that do not contain non-deterministic or side-effecting functions are pushed down as low as possible in the expression tree to limit the amount of data processed. Filter pushdown through `transform`, `join`, and `group by` is similar to relational databases [15], whereas filter pushdown through `expand` is more similar to predicate pushdown through XPath expressions [33], as `expand` unnests its input data. For example, the following rule states that we can pushdown the predicate before a `group by` operator if the filter is on the grouping key.

```
e -> group by x = $.x into { x, n: count($)}
  -> filter $.x == 1
  ≡
e -> filter $.x == 1
  -> group into { $.x, n: count($) }
```

Field access: When a known field of a record is accessed, the record construction and field access are composed: `{x:` e`, ...}.x` $\Rightarrow$ e . A similar rule applies to arrays. This rule forms the basis for selection and projection push-down as well. The importance of this property was a major reason to move away from

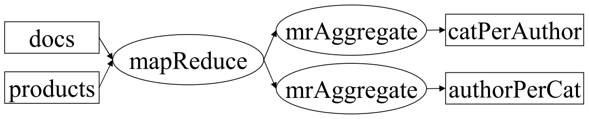

Figure 6: Data-flow diagram for Example 9.

XML. Node construction in XQuery includes several effects that prevent a simple rewrite rule like this: node identity, node order, parent axis, sibling axis, and changing of primitive data types when an item is inserted into a node.

To MapReduce: After simplifying the script, Jaql searches for sequences of expressions that can be evaluated in a single MapReduce job. It looks for a `read` followed by a second sequence of partitionable expressions, followed by a `write` to a distributed output. If the `group` is not present, a map-only job is produced. If the group is only used inside of algebraic aggregates, an `mrAggregate` call is produced[5]. Otherwise, a `mapReduceFn` call is produced. The partitionable expressions before the `group` and the grouping key expression are placed in the map function. The map function is called once to process an entire partition, not per element. Any expressions after the aggregates and the second sequence of partitionable expressions are placed in the reduce function. The `group` may have multiple inputs (i.e., co-group), in which case each input gets its own map function, but still a single reduce function is created. The rewrite must consider expressions that are nondeterministic, are side-effecting, or disallow remote evaluation.

Via these and the remaining rules, scripts are conceptually translated into a directed-acyclic graph (DAG), where each node is a MapReduce job or a sequential step.

Example 9 *Consider* `wroteAbout` *as defined in Example 5 and the* `product` *dataset that is defined as:*

```
products = read(hdfs("products.dat"));
```

The following Jaql script computes two summary files for categories and authors. Lines 1–3 join the `wroteAbout` *and* `products` *collections. Lines 5–8 count the distinct product categories mentioned by each author. Lines 10–13 count the distinct authors for each product category. The notation* `R[*].f` *is short-hand to project a field from an array of records. The* `cntDist` *function is a user-defined aggregate that computes a count of distinct values in parallel.*

```
 1. joinedRefs = join w in wroteAbout, p in products
 2.   where w.product == p.name
 3.   into { w.author, p.* };
 4.
 5. joinedRefs
 6.   -> group by author = $.author as R
 7.      into {author, n: cntDist(R[*].prodCat)}
 8.   -> write(hdfs('catPerAuthor'));
 9.
10. joinedRefs
11.   -> group by prodCat = $.prodCat as R
12.      into {prodCat, n: cntDist(R[*].author)}
13.   -> write(hdfs('authorPerCat'));
```

Compilation produces a DAG of three MapReduce jobs as shown in Figure 6. The DAG is actually represented internally as a block of `mapReduceFn` *and* `mrAggregate` *calls, with edges created by data-flow dependencies, variables, and read/write conflicts. The complete compilation result is given in Example 10.*

[5]Note that in this case a combiner with the same aggregation function is also produced.

Although our current focus is on generating MapReduce jobs, Jaql should not be categorized simply as a language only for MapReduce. New platforms are being developed that offer different (e.g., Pregel [29]) and more general operators (e.g., Nephele [7], Hyracks [3]). Our long-term goal is to glue many such paradigms together using Jaql.

7.4 Decompilation and Explain

Every expression knows how to decompile itself back into a semantically equivalent Jaql script, thus providing the aspect of physical transparency needed for pinning down a plan. Immediately after parsing and after every rewrite rule fires, the `Expr` tree (e.g., evaluation plan) can be decompiled. The `explain` statement uses this facility to return the lower-level Jaql script after compilation. This process is referred to as *source-to-source translation* [27].

Example 10 *The following is the result of* `explain` *for the script of Example 9. The list of jobs can be visualized as the DAG in Figure 6.*

```
(// Extract products from docs, join with access log
  tmp1 = mapReduce({
    input: [ { location: 'docs.dat', type: 'hdfs' },
             { location: 'products', type: 'hdfs' } ],
    output: HadoopTemp(),
    map: [fn(docs) (
        docs
          -> transform
              { w: { author: $.meta.author,
                     products: systemt(
                       $.content, 'rules...' ) }}
          -> transform [$.w.product, $] ),
      fn(prods) (prods
          -> transform { p: $ }
          -> transform [$.p.name, $] )
      ],
    reduce: fn(pname, docs, prods) (
      if( not isnull(pname) ) (
        docs -> expand each d (
          prods -> transform each p { d.*, p.* } ))
      -> transform { $.w.author, $.p.* } )
    }),

  // Count distinct product categories per author
  mrAggregate({
    input: tmp1,
    output: { location: 'catPerAuthor', type: 'hdfs' },
    map: fn(vals) vals -> transform [$.author, $],
    aggregate: fn(author, vals)
       [ vals -> transform $.prodCat -> cntDist() ],
    final: fn(author, aggs) { author, n: aggs[0] },
   }),

  // Count distinct authors per product category
  mrAggregate({
    input: tmp1,
    output: { location: 'authorPerCat', type: 'hdfs' },
    map: fn(vals) vals -> transform [$.prodCat, $],
    aggregate: fn(author, vals)
       [ vals -> transform $.prodCat -> cntDist() ],
    final: fn(prodCat, aggs) { prodCat, n: aggs[0] },
   })
)
```

Since the plan in Example 10 is a valid Jaql query, it can be modified with a text editor and submitted "as-is" for evaluation. In certain situations where a particular plan was required, the capability to edit the plan directly, as opposed to modifying source code, was invaluable. In addition, Example 10 illustrates how higher-order functions, like `mapReduceFn`, are represented in Jaql. Note

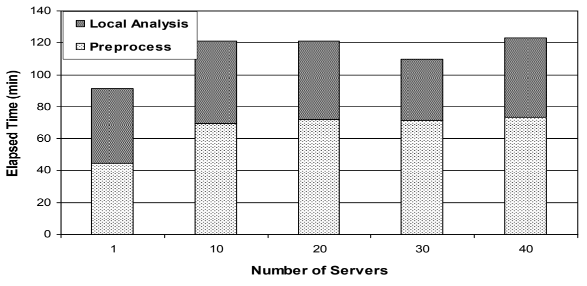

Figure 7: Scale-up using intranet analysis workload.

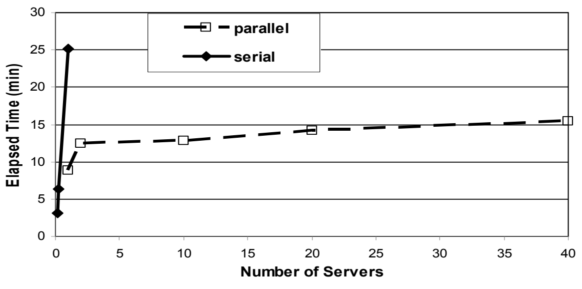

Figure 8: Scale-up using application log workload.

that run-time operators of many database systems can be viewed as higher-order functions. For example, hash-join takes two tables as input, a key generation function for each table, and a function to construct the output. Jaql simply exposes such functionality to the sophisticated user.

Support for decompilation was instrumental in bridging Jaql to MapReduce and in implementing error handling features (see Section 5.5). The MapReduce *map* and *reduce* functions are Jaql functions that are written into a configuration file and parsed when the MapReduce job is initialized. For error handling, the `fence` function decompiles its input expression, forks a child process and sends the decompiled expression to the child.

Jaql's strategy is for a user to start with a declarative query, add hints if needed, and move to low-level operators as a last resort. Even when a declarative query is producing the right plan, the user can use `explain` to get a low-level script for production use that ensures a particular plan over changing input data.

8. EXPERIMENTAL EVALUATION

In this section, we describe our experiments and summarize the results. We focused on Jaql's scalability while exercising its features to manage nested data, compose data flows using Jaql functions, call user-defined functions, and exploit physical transparency. We considered two workloads that are based on: a real workload that is used to analyze intranet data sources, and the log processing example that is described in Example 7.

Hardware: The experiments were evaluated on a 42-node IBM SystemX iDataPlex dx340. Each server consisted of two quad-core Intel Xeon E5540 64-bit 2.8GHz processors, 32GB RAM, 4 SATA disks, and interconnected using 1GB Ethernet.

Software: Each server had Ubuntu Linux (kernel version 2.6.32-24), IBM Java 1.6, Hadoop 0.20.2, and Jaql 0.5.2. Hadoop's "master" processes (MapReduce JobTracker and HDFS NameNode) were installed on one server and another 40 servers were used as workers. All experiments were repeated 3 times and the average of those measurements is reported here.

8.1 Intranet Data Analysis

Jaql is used at IBM to analyze internal data sources to create specialized, high-quality indexes as described in [4]. The steps needed for this process are: (1) crawl the sources (e.g., Web servers, databases, and Lotus Notes), (2) pre-process all inputs, (3) analyze each document (Local Analysis), (4) analyze groups of documents (Global Analysis), and (5) index construction. Nutch [19] is used for step (1) and Jaql is used for the remaining steps.

For the evaluation, we took a sample of the source data and evaluated how Jaql scales as both the hardware resources and data are proportionally scaled up. Per server, we processed 36 GB of data, scaling up to 1.4 TB for 40 servers. We focused on steps (2) and (3) since these steps manage the most data. The preprocess step (2) transforms the input into a common schema, and for Web data, resolves redirections, which requires an aggregation. The local analysis step (3) analyzes each document, for example, identifying language and extracting structured information using SystemT. These steps exercise many of Jaql's features, which range from standard data processing operators (e.g., group-by, selection, projection), to extensibility (e.g., Jaql and Java functions), and semistructured data manipulation.

The results are shown in Figure 7. The pre-process phase (step 2) reads directly from Nutch crawler output and resolves redirections in the reducer tasks. The time needed to shuffle all data across the network dominated overall run-time, which explains why the result for scale-factor 1 was much faster— the shuffle looped back to the same machine. Most of the other steps used Map-only jobs so scaling was more predictable. The one exception was at scale factor 30 where the Local Analysis step was more selective for that sample of data. Overall, the results illustrate that Jaql scales well for the given workload.

8.2 Log Processing

We evaluated the scale-up performance of the log cleansing task from Section 6 that relied on physical transparency for parallelization. We generated 30M records per CPU core of synthetic log data with 10% of the records representing exceptions with an average of 11 additional lines per exception record, which resulted in approximately 3.3 GB / core. We varied the number of servers from 1 to 40, which varied the number of cores from 8 to 320 and data from 26GB to 1TB. The result in Figure 8 shows that the original sequential algorithm works well for small data, but quickly gets overwhelmed. Interestingly, the parallel algorithm also runs significantly faster at small scale than at the high end (from 1 machine to 2). However, the parallel algorithm scales well from 2 to 40 machines, drastically outperforming the sequential algorithm even at a single machine because of its use of all 8 cores.

9. CONCLUSION

We have described Jaql, an extensible declarative scripting language that uses Hadoop's MapReduce for scalable, parallel data processing. Jaql was designed so that users have access to the system internals—highlighting our approach to physical transparency. As a result, users can add features and solve performance problems when needed. For example, we showed how tumbling windows and physical transparency can be exploited to scalably process large logs. A key enabler of physical transparency is Jaql's use of (higher-order) functions, which addresses both composition and encapsulation so that new features can be cleanly reused.

Jaql's design was also molded by the need to handle a wide variety of data. The flexibility requirement guided our choice of data model and is evident in many parts of the language design. First, all expressions can be uniformly applied to any Jaql value, whether it represents the entire collection or a deeply nested value. Second, the schema information at every expression can range from none, through partial schema, to full schema. Thus, Jaql balances the need for flexibility with optimization opportunities. The performance results illustrate that Jaql scales on a variety of workloads that exercise basic data processing operations, extensibility features, and nested data manipulation.

Jaql is still evolving, and there are many challenges that we plan to pursue as future work. A non-exhaustive list includes: further investigation of errors handling and physical transparency, adaptive and robust optimization, exploitation of materialized views, discovery-based techniques for storage formats and partition elimination, and novel aspects for tools that assist with design as well as runtime management.

10. REFERENCES

[1] S. Balakrishnan, V. Chu, M. A. Hernández, H. Ho, et al. Midas: Integrating Public Financial Data. In *SIGMOD*, pages 1187–1190, 2010.

[2] D. Battré, S. Ewen, F. Hueske, O. Kao, et al. Nephele/PACTs: A Programming Model and Execution Framework for Web-scale Analytical Processing. In *SoCC*, pages 119–130, 2010.

[3] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, et al. ASTERIX: Towards a Scalable, Semistructured Data Platform for Evolving-world Models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[4] K. S. Beyer, V. Ercegovac, R. Krishnamurthy, S. Raghavan, et al. Towards a Scalable Enterprise Content Analytics Platform. *IEEE Data Eng. Bull.*, 32(1):28–35, 2009.

[5] Biginsights. `http://www-01.ibm.com/software/data/infosphere/biginsights/`.

[6] G. E. Blelloch. NESL: A Nested Data-Parallel Language (3.1). *CMU-CS-95-170*, 1995.

[7] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, et al. Hyracks: A Flexible and Extensible Foundation for Data-intensive Computing. In *ICDE*, pages 1151–1162, 2011.

[8] Cascading. *http://www.cascading.org/*.

[9] Cognos Consumer Insight. `http://www-01.ibm.com/software/analytics/cognos/analytic-applications/consumer-insight/`.

[10] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsen, et al. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *PVLDB*, 1(2):1265–1276, 2008.

[11] C. Chambers, A. Raniwala, F. Perry, S. Adams, et al. FlumeJava: Easy, Efficient Data-parallel Pipelines. In *PLDI*, pages 363–375, 2010.

[12] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, et al. MAD Skills: New Analysis Practices for Big Data. *Proc. VLDB Endow.*, 2(2):1481–1492, 2009.

[13] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, et al. Ricardo: Integrating R and Hadoop. In *SIGMOD*, pages 987–998, 2010.

[14] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[15] D. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, et al. GAMMA – A High Performance Dataflow Database Machine. In *VLDB*, pages 228–237, 1986.

[16] E. Friedman, P. Pawlowski, and J. Cieslewicz. SQL/MapReduce: A Practical Approach to Self-describing, Polymorphic, and Parallelizable User-defined Functions. *Proc. VLDB Endow.*, 2(2):1402–1413, 2009.

[17] R. Greer. Daytona and the Fourth-generation Language Cymbal. *SIGMOD Rec.*, 28(2):525–526, 1999.

[18] Hadoop. *http://hadoop.apache.org*.

[19] http://nutch.apache.org/.

[20] IBM DB2 PureXML. *http://www-01.ibm.com/software/data/db2/xml/*.

[21] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, pages 59–72, 2007.

[22] M. Jaedicke and B. Mitschang. User-Defined Table Operators: Enhancing Extensibility for ORDBMS. In *VLDB*, pages 494–505, 1999.

[23] Biginsights jaql. `http://publib.boulder.ibm.com/infocenter/bigins/v1r1/index.jsp/`.

[24] JSONR. *http://laurentszyster.be/jsonr/*.

[25] JSON Schema . *http://json-schema.org/*.

[26] R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, et al. SystemT: A System for Declarative Information Extraction. *SIGMOD Record*, 37(4):7–13, 2008.

[27] S.-I. Lee, T. A. Johnson, and R. Eigenmann. Cetus - An Extensible Compiler Infrastructure for Source-to-Source Transformation. In *LCPC*, pages 539–553, 2003.

[28] LINQ. *http://msdn.microsoft.com/en-us/netframework/aa904594.aspx*.

[29] G. Malewicz, M. Austern, A. Bik, J. Dehnert, et al. Pregel: A System for Large-Scale Graph Processing. In *SIGMOD*, pages 135–146, 2010.

[30] MarkLogic Server . *http://developer.marklogic.com/pubs/4.1/books/cluster.pdf*.

[31] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, pages 1099–1110, 2008.

[32] Oracle9i Pipelined Table Functions. *http://www.oracle.com/technology/sample_code/tech/pl_sql*.

[33] F. Ozcan, N. Seemann, and L. Wang. XQuery Rewrite Optimization in IBM DB2 pureXML. *IEEE Data Eng. Bull.*, 31(4):25–32, 2008.

[34] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. In *Scientific Programming Journal*, pages 277–298, 2005.

[35] Relax NG . *http://relaxng.org/*.

[36] A. Sala, C. Lin, and H. Ho. Midas for Government: Integration of Government Spending Data on Hadoop. In *ICDEW*, pages 163 –166, 2010.

[37] Saxon-ee 9.3. `http://saxon.sourceforge.net/`.

[38] M. Stonebraker. The Case for Shared-Nothing. In *IEEE Data Engineering*, pages 4–9, March 1986.

[39] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, et al. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[40] XML Schema . *http://www.w3.org/XML/Schema*.

[41] *XQuery 1.0: An XML Query Language*, January 2007. W3C Recommendation, See `http://www.w3.org/TR/xquery`.

[42] Y. Yu, M. Isard, D. Fetterly, M. Budiu, et al. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI*, pages 1–14, 2008.