Jaql: A Scripting Language for Large Scale Semi-Structured Data Analysis

Kevin Beyer Vuk Ercegovac Rainer Gemulla
Andrey Balmin Mohamed Eltabakh Fatma Ozcan Eugene Shekita

IBM Almaden Research Center
650 Harry Road, San Jose CA, 95120

ABSTRACT

Jaql is a declarative scripting language for enterprise data analysis powered by a scalable runtime that leverages Hadoop's MapReduce parallel programming framework. Jaql is used in IBM's Cognos Consumer Insight [6], the announced InfoSphere BigInsights [3], as well as several research projects. Through these interactions and use-cases, we have focused on the following core design principles: (1) a flexible data model, (2) reusability, (3) varying levels of abstraction, and (4) scalability. The data model is inspired by JSON and can be used to represent data that varies from flat, relational tables to collections of semi-structured documents. To support the various phases of data analysis, Jaql is able to operate without knowledge of the schema of the data; queries can evolve towards partial or rigid schema definitions over time. Reusability is provided through the use of higher-order functions, and by packaging related functions and their required resources into modules. While Jaql is declarative, it is built from layers that vary in their level of abstraction. Higher levels allow concise specification of logical operations (e.g., join), while lower levels blend in physical aspects (e.g., hash join or MapReduce job). We have exposed such functionality so that it is easier to add new operators or to pin down an execution plan for greater control. Finally, Jaql automatically rewrites scripts to use Hadoop's MapReduce for parallel data processing, when possible. Our experimental results illustrate that Jaql scales well for real and synthetic workloads and highlights how access to lower-level operators enabled us to parallelize typically sequential flows for scalably analyzing large datasets.

1. INTRODUCTION

The overwhelming trend towards digital services, combined with cheap storage, has generated massive amounts of data that enterprises need to effectively gather, process, and analyze. Techniques from the data warehousing and high-performance computing communities are invaluable for many enterprises. However, oftentimes their cost or complexity of scale-up discourages the accumulation of data without an immediate need [9]. As valuable knowledge may nevertheless be buried in this data, related and complementary technologies have been developed. Examples include Google's MapReduce [11], its open-source implementation Apache Hadoop [15], and Microsoft's Dryad [18]. These systems are conducive to the "collect first, ask questions later" principle. Web-centric enterprises, which were both developers and early adopters of such scaled-up architectures, quickly recognized the value of higher-level languages within this environment, as evidenced by Google's Sawzall [33], Microsoft's DryadLINQ [42], Yahoo's work on Apache Pig [30], and Facebooks's work on Apache Hive [38].

Traditional enterprises are increasingly experimenting with—and, in some cases, relying on—such scaled-up architectures. In this context, Jaql is used most notably by IBM in several products, including Cognos Consumer Insights [6] and the announced BigInsight's [3] for building data centric applications and for ad hoc data analysis. The workloads from these products, as well as from various research projects are diverse and range from analyzing internal data sources for intranet search [2], cleansing and integrating multiple external data sources for the financial and government sectors [35, 1], developing and using models through Monte Carlo simulation [41], monitoring network data for security purposes [25], analyzing both transaction and system log data, and employing collaborative filtering techniques to predict customer behavior [28] [10]. Such use cases guided the development of the Jaql language and its processing system.

In this paper, we describe the Jaql (JSON Query Language) project [16], which at its core consists of two main components: (1) Jaql, a declarative scripting language for enterprise data analysis, and (2) the Jaql system, which includes the query compiler and processing subsystems. We refer to all three—the project, the language, and the system—as Jaql and disambiguate only when needed. Jaql is used to develop data processing flows that are executed in parallel on Hadoop's MapReduce implementation, when possible. Jaql is influenced by Pig [30], Hive [38], DryadLINQ [42], and others, but has a unique focus on the following combination of core design principles: (1) a flexible data model, (2) reusable and modular scripts, (3) the ability to specify scripts at varying levels of abstraction, henceforth referred to as *physical transparency*, and (4) scalable query processing.

We now summarize how Jaql addresses its design goals.

Flexible Data Model: Jaql's data model is based on JSON, a simple text format and standard (RFC 4627). As a result, Jaql's data model is sufficiently flexible to handle semi-structured documents, which are often found in the early, exploratory stages of data analysis (such as logs), as well as the structured records that are often produced after data cleansing stages. Jaql is able to process data without or with only partial schema information, which is also useful for exploration. For enhanced efficiency, Jaql can en-

VLDB '11, August 29- September 3, 2011, Seattle, WA

force and exploit rigid schema information for both type checking and improved performance. In addition, since 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. Borrowing from functional languages, Jaql supports lazy evaluation and higher-order functions, i.e., functions are treated as first-class data types. 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. Finally, many related functions and their corresponding resources can be bundled together into modules, each with their own namespace.

Physical Transparency: Jaql exposes every internal physical operators as functions 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 execution, 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 the optimizer "hints". Although useful, such hints only control a limited set of query execution plan features (such as join orders and access paths). Jaql's physical transparency goes further in that it allows full control over the execution 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 in 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* (see [23] for an example) and, to the best of our knowledge, Jaql is the first data processing language that exploits this technique and makes it available to 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 an Jaql plan when needed.

Scalability: Jaql is designed to parallelize scripts over collections of relatively small semi-structured 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 such 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 6 show that Jaql scales well and illustrates how physical transparency enabled us to parallelize typically sequential flows for scalably analyzing large datasets.

The remainder of the paper is organized as follows. We review related work in Section 2. Jaql's data model and schema is described in Section 3. We discuss the Jaql language in Section 4, and the system implementation in Section 5. Section 6 contains the experimental results. Finally, we conclude in Section 7.

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 [12, 37], MapReduce [11], 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 scaled-out architectures such as MapReduce and Dryad [18]. In particular, Pig Latin [30], Hive [38], and DryadLINQ [42] have many design goals and features in common. Pig Latin 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 but its syntax is based on SQL, and it is statically typed. Microsoft's Scope [7] 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.

While SQL, Jaql, and Pig Latin are distinct data processing languages, Microsoft's LINQ [24], Google's FlumeJava [8], and the Cascading project [5] 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 functionality 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[33] 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 scripts 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 [19]. Examples of systems that support such functionality include AsterData's SQL/MapReduce [13] and Oracle's pipelined table functions [31]. AT&T's Daytona [14] is a proprietary system that efficiently manages and processes massive flat files using SQL and procedural language features. NESL [4] is a parallel programming language that specializes on nested data. In contrast, Jaql is designed to process semi-structured, self-describing data and its support for higher-order functions offer 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 [40] such as functions and modules/namespaces. However, we are not aware of an XQuery implementation that also supports higher-order functions. In addition, except for DB2's PureXML [17] and MarkLogic Server [27], 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. DATA MODEL AND SCHEMA

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

3.1 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 sequence files, relational databases, key-value stores, or XML documents.

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}`). Note that JDM textual representation is a part of Jaql grammar, thus we use terms JDM value and Jaql value interchangeably. Also note that this representation closely resembles JSON, a popular and standardized text format for data exchange. 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. This resemblance has multiple advantages. While JSON was designed for JavaScript, it has been found useful as a format for data exchange between programs written in many different programming languages, including C, C++. C#, Java, Python, and Ruby, to name just a few. Due to its closeness to JSON, Jaql can readily exchange data with all those languages.

Example 1 *Consider a hypothetical company KnowItAll, Inc. that maintains a repository of documents. 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" } },
  ...
]
```

```
<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.

```
<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

Relationship to other data models. Jaql's data model consciously avoids many complexities that are inherent in other semi-structured data models, such as the XQuery Data Model (XDM) and the Object Exchange Model (OEM). For example, Jaql's data model 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.

3.2 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 [39], RELAX NG [34], JSON schema [21], and JSONR [20]. The schema information does not need to be complete and rigid because Jaql supports partial schema specification.

Jaql uses a simple pattern language to describe schemas, see Figure 2. 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 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:*

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

where ? *marks optional fields. 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 code. 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 code can be reused as is, 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 as `[{fields:map}...]` in Pig Latin [30] and LINQ [24], 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.

4. LANGUAGE

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, reusability, and physical transparency.

4.1 A Simple Example

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] 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.

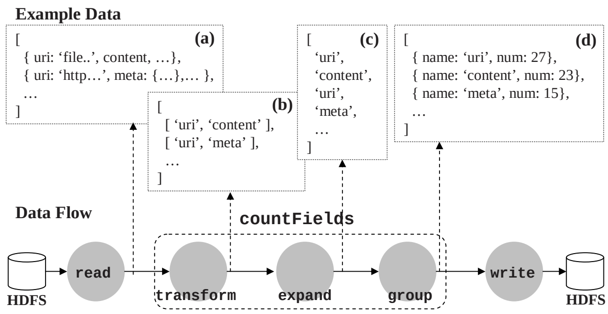

Figure 3: Conceptual data flow for counting fields

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

Lines 11–13 in the example script correspond directly to the conceptual data flow of Figure 3. Jaql uses a pipe syntax (->) which was inspired by Unix pipes. 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 Latin[30]), or WITH clauses (as in SQL), for every computational step. The pipe syntax is also more readable than the functional notation (as in XQuery[40]), when a series of functions are invoked back to back.

In Example 3, `read`, `hdfs`, `countFields`, and `write` are *functions*; their composition and invocation constitutes an expression. The remaining part of the script concerns `countFields`. Line 1 is an import statement that imports several record-related functions from the `myrecord` module. Lines 3–9 constitute the assignment that defines the `countFields` function, which is discussed in detail in the Section 4.3.

4.2 Core Expressions

Jaql has several expressions for manipulating data collections, including `transform`, `expand`, `filter`, `join`, `sort`, `group by`, multi-input `group by` (equivalent to Pig's `co-group` [30]), `merge`, `tee`, and `split`. Note that some of these expressions (such as join, group by, filter) can be found in database management systems, while others (such as transform, merge, tee, split) are typical for ETL engines. A complete discussion of these expressions is beyond the scope of this paper, but see [16].

This section illustrates some of the core expressions using our running example. Consider lines 4–8 of Example 3 as well as the corresponding data shown in Figure 3. The `transform` expression applies a function (or projection) to every element of an array. It has the form e_1`->transform` e_2, where e_1 is an expression that describes the input array, and e_2 describes the transformation. 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 invocation makes use of Jaql's default iteration variable $; most expressions allow renaming this variable using the `each` keyword. For example, `...->transform each r myrecord::names(r)` illustrates how $ can be renamed to `r`. The `names` function itself takes as input a record and produces an array of field names (represented as strings). The output after `transform` is shown Figure 3(b). The `expand` expression in line 6 unnests the array of field names, cf. Figure 3(c).

The subsequent `group by` expression counts the number of occurrences of each distinct field name. In contrast to SQL's `GROUP BY`, Jaql's `group by` expression allows arbitrary Jaql values as grouping key (including arrays and records) and does not necessarily perform aggregation. The grouping key and array of all values in the group are accessible via iteration variables. In the example, these variables are named `key` and `values`. The expression in the `into` clause is used to construct the result of the group by. Here, we construct a record with two fields `name` and `num`, the latter by applying the `count` *aggregate function* to `values`. As with the grouping key, the result of aggregation can be an arbitrary Jaql value.

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 [40] and LINQ [24] in this respect, but differs from Pig Latin [30] and Hive [38] 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.

4.3 Functions

Functions are first-class values in Jaql, i.e., they can be assigned to a variable, 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 4.5).

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.

Jaql makes heavy usage of the *pipe symbol* `->` in its core expressions. 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". 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 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.

4.4 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 [22]) or statistical analysis (e.g., via R 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.
3. read( hdfs("docs.dat") )
4. -> transform { author: $.meta.author,
5.                products: systemt($.content, rules) };
6.
```

When run on the example data, the script may produce

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

Example 4 illustrates how functions and semi-structured data are often used in Jaql. The *javaudf* function shown on line (1) is an example of a function that returns a function that is parameterized by a Java class name *c*, and when invoked, knows how to bind invocation parameters and invoke the appropriate method of *c*. As a result, 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 sample result from Example 4 illustrates a common usage pattern– per string (e.g., `$.content`), SystemT enriches each record with extracted data. In this case, the SystemT *rules* found multiple products produced by multiple companies (e.g., "acme" and "knowitall"). More generally, the extracted information includes additional attributes, such as where in the original input 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.

Functions that are implemented using external programs are similarly used by Jaql. The *externalFn* function wraps the invocation of an external program into a function, which can then be assigned to a variable and invoked just like any other function. The support for external programs include several other notable features. First, data must be serialzed to and from the external process. For this, Jaql's I/O capability is re-used so that data can be converted flexibly between Jaql and the format that the external program expects. Second, the protocol by which Jaql interacts with the external program has been abstracted and made pluggable. Two protocols have been implemented to inter-operate with programs: (1) bulk invocation and (2) value invocation. With *bulk* invocation, Jaql sends all data to the program while using a second thread to asynchronously consume the program's output. While efficient, a complication with bulk invocation is that the external program must consume all data, which makes it cumbersome to send just a projection of the data to the program, then correlate its result back to the original data. For such cases, per-*value* invocation is used to synchronously send

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

a value to the program and receive its output. For both protocols, Jaql re-uses the same process for as long as it can. Bulk invocation is supported in Hadoop (e.g., Hadoop Streaming), Pig and Hive whereas value invocation is supported for UDF's. Neither system unifies *push* and *pull* into a single abstraction, which lets Jaql users easily switch between various implementations.

4.5 Physical Transparency

Physical transparency—i.e., the exposure of lower-level abstractions in the language—enables *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.

Example 5 *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 entries 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.*

Order–sensitive operations like tumbling windows are notoriously more difficult to parallelize than multi–set operations. At this stage, Jaql is not clever enough to automatically parallize this script, so it runs sequentially. For small logs, this is acceptable, but for large logs we clearly need to do better. A user of a traditional database system might make a feature request and wait several years for a solution to be delivered. Physical transparancy allows the power user to implement a solution at a lower level of abstraction.

Example 6 *Imagine somebody found a way to run tumbling windows in parallel. Perhaps not supporting the full generality of the builtin windowing support, but enough to solve the problem at hand. Then we could use this function as a replacement for the original function. The following script is very similar to the previous one, but the read and tumblingWindow have been composed into a single function, ptumblingWindow, that runs in parallel:*

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

The new script remains at a fairly high level, even though it is exploiting low–level operations via *ptumblingWindow*.

The key to *ptumblingWindow*'s implementation is *split* manipulation. Ordinarily, Hadoop is responsible for partitioning an input into *splits* and assigning each split to a single map task. Fortunately, Hadoop's API's are very flexible, making it easy to re-define how a given input is partitioned into splits. For *ptumblingWindow*, we directly access the splits and manipulate them using Jaql to redefine the *splits* so that the semantics of `tumblingWindow` are preserved. Each task processes its split and peeks at the next split to handle the case where a partial, logical record spans a split boundary.

While the implementation of *ptumblingWindow* consists of a handful of simple functions, these functions access very low-level Hadoop API's so 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.

4.6 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 sends and receives data in bulk to the forked process or per value. The

[3]A complete discussion of all the window facilities requires an article in its own right. For the purposes of this paper, a basic understanding is sufficient.

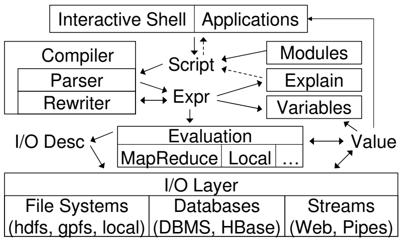

Figure 4: System architecture.

`timeout` function places a limit on how long its input expression can run. If exceeded, 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.

5. SYSTEM IMPLEMENTATION

At a high-level, the Jaql architecture depicted in Figure 4 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.

5.1 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, 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 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.

Input / output *descriptors* are open-ended structures used to describe storage objects that are passed to the `read` and `write` functions, among others. A descriptor is a record with a simple schema: `{adapter:string, *:any}`. The `adapter` field refers to an internal adapter object: a Java class name in our implementation.

The remaining fields are interpreted by the adapter. Most HDFS files use the default Hadoop adapter, which extends the input descriptor to include a `format` and `converter` field. The `format` field refers to a Hadoop `InputFormat` class that provides access to HDFS files or other storage objects; the `converter` translates the Java objects into Jaql's data model. Converters may expect additional parameters in the descriptor, e.g., information needed to decode lines in the file. Similar functionality is provided for `OutputFormats`.

The descriptors exemplify the power of irregular schemas. The `read` function simply requires a record with an `adapter` field. Because the descriptor often provides parameters for several objects, like an adapter and a converter, the descriptor schemas do not form a type hierarchy. Instead they are more like mixin types, using multiple inheritance to mix the fields needed for each of the objects. The default result schema `[any...]` of a `read` is also irregular; it produces an array of values. However, the adapter may refine the result schema. This flexibility in descriptors and results allows users to quickly implement access to new storage objects.

5.2 Evaluation

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. Each map task scans a partition of an input data set—e.g., an HDFS file spread across the cluster—and produces a set of key-value pairs. If appropriate, the map output is partially reduced using a combine task. The map output is redistributed across the cluster by key so that all values with the same key are processed by the same reduce task. Both the combine and reduce tasks are optional.

Unlike traditional databases, MapReduce clusters run on less reliable hardware 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 significantly 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.

The Jaql interpreter begins evaluation of 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 the `mapReduceFn` function of Jaql, but more often a Jaql script is rewritten into one or more MapReduce jobs, as described in Section 5.3.2.

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 obviously must run locally.

5.3 Compiler

The Jaql compiler automatically detects parallelism in a Jaql script and translates it to a set of MapReduce jobs. The rewrite

[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.

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.

5.3.1 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. 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, the `lookup` function in a hash-join is only partitionable over its probe input because we do not want to load the build array for every probe value.

Most expressions are purely functional, meaning that they evaluate each input expression once to produce their value and they do not have any external effects. However, an `Expr` may be *nondeterministic* (e.g., `randomDouble`) or *side-effecting* (e.g., `write`), which restricts the types of rewrites performed by the system. An `Expr` may also declare that it selectively or repeatedly evaluates child expressions.

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.

An `Expr` may be *evaluable at compile-time*, given that its inputs are constants. Evaluating expressions at compile-time obviously eliminates run-time work, potentially to a large degree if the expression is inside of a loop. More importantly, constants can be freely inlined (to be later composed) without creating repeated work and they improve transparency so that important constants can be found. For example, Jaql must determine the adapter in the I/O descriptor to determine the schema of a `read` expression. Note that all expressions have an output schema— in many cases schemas are determined during compilation. Such *schema inference* is used by Jaql to improve the space efficiency of its storage formats. The idea is that if more information is known about the structure of the data, a more compact storage format can be used. In particular, such techniques are used between the map and reduce steps, thereby reducing I/O bandwidth for both disks and network.

5.3.2 Rewrites

At present, the Jaql compiler simply consists of a heuristic rewrite engine that greedily applies approximately 40 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 peel back the abstractions created for readability and modularity, and to compose expressions separated for reusability. The engine simplifies the script, discovers parallelism, and translates declarative aspects into lower-level operators. The most important rules are illustrated in Example 7 and described below.

Example 7 *Steps in rewriting a function call.*

```
1. f = fn(r) r.x + r.y;
2. f({x:1,y:2});
⇒ (fn(r) r.x + r.y)({x:1,y:2}); // variable inline
⇒ (r = {x:1,y:2}, r.x + r.y);   // function inline
⇒ {x:1,y:2}.x + {x:1,y:2}.y;    // variable inline
⇒ 1 + 2;   // constant field access
⇒ 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.

Function inlining: When a function call is applied to a Jaql function, it is replaced by a block[5] 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.

Filter push-down: Filters which do not contain non-deterministic or side-effecting functions are pushdown 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 [12], whereas filter pushdown through `expand` is more similar to predicate pushdown through XPath expressions [32], 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.

$$e_1 \;\rightarrow\; group\ by\ x = \$.x\ into\ \{\ x,\ n:\ count(\$)\ \}$$
$$\rightarrow\ filter\ \$.x\ ==\ 1\ \equiv$$
$$e_1 \;\rightarrow\; filter\ \$.x\ ==\ 1\ \rightarrow\ group\ into\ \{\ \$.x,\ n:\ count(\$)\}$$

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 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 the inlining and composition, Jaql searches for sequences of expressions that can be evaluated in a

[5]A block expression is a parenthesized sequence of expressions separated by commas, with optional local variable assignment. The result of a block is the value of its last expression.

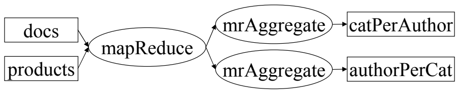

Figure 5: Data-flow diagram for Example 8

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[6]. 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 8 *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:*

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

This definition is similar to the one used in Example 4, but unnests the `products` *array. More specifically, the inner* `transform` *produces an array of author-product pairs, which is then unnested using* `expand`*. The* `product` *dataset 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'));
```

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

Compilation produces a DAG of three MapReduce jobs as shown in Figure 5. 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 9.*

Although our current focus is on generating MapReduce jobs, Jaql should not be categorized simply as a language for MapReduce. Though Jaql should be useful for manipulating small collections (an implementation for Javascript running in a Web browser would be interesting), our focus is on large-scale data analysis. In this space, many paradigms besides MapReduce were proposed recently and in the past, e.g., Pregel for graph processing [26] or ScaLAPACK for matrix computations [36]. Each of these are designed for certain specialized classes of computation. Our long-term goal is to glue many such paradigms together using Jaql, which will increase the types of nodes in the DAG. For example, we created an iterative, parallel model building function called `buildModel` for data-mining tasks that easily expresses, e.g., a parallel k-means computation.

5.4 Decompilation and Explain

Every expression knows how to decompile itself back into a semantically equivalent Jaql script. Immediately after parsing and after every rewrite fires, the `Expr` tree 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* [23].

Example 9 *The following is the result of* `explain` *for the script of Example 8. The list of jobs can be visualized as the DAG in Figure 5.*

```
(// 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] },
  })
)
```

By mandating that every Expr tree supports decompilation, we ensure that every evaluation plan is expressible in Jaql itself, thus providing physical transparency. Since the plan in Example 9 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 9 illustrates how higher-order functions, like mapReduceFn, are represented in Jaql. Note 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 4.6). The MapReduce *map* and *reduce* functions are simply Jaql functions that are serialized into a configuration file and deserialized when the MapReduce job is initialized. For error handling, the fence function simply decompiles its input, forks a child process that expects a Jaql function f, and sends it to the child. Since functions are part of Jaql's data model, serialization and deserialization between child and parent is trivial.

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.

6. 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 three workloads that are based on: (1) a synthetic data set designed for XML-based systems, (2) a real workload that is used analyze intranet data sources, and (3) the log processing example that is described in Example 6.

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. Each worker was configured to run upto 4 map and 4 reduce tasks concurrently. The following configuration parameters were overridden in order to boost performance: HDFS block size was set to 64MB, sort buffer size was set to 512MB, JVM's were re-used, speculative execution was turned off, and 4GB JVM heap space was used per task. All experiments were repeated 3 times and the average of those measurements is reported here.

6.1 Synthetic, Semi-structured Workload

We use the dataset from the TPoX benchmark (Transaction Processing over XML) [29] to illustrate several of Jaql's features over

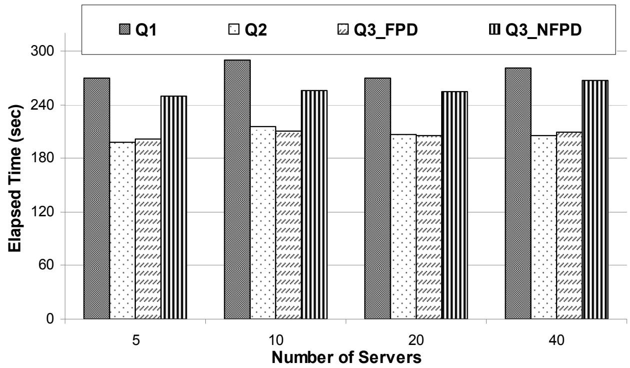

Figure 6: Scale-up experiment using TPox workload.

semi-structured data. The dataset consists of financial transactions with five logical entities: Customers, each customer has a set of Accounts, each account may have Holdings and Orders issued over that account, and each account/order pair has Security types. We used the TPoX data generator [29] to generate *CustAcc* documents with three levels of nesting; *Customer* as the top level entity, one or more *Account* entities, and zero or more *Holding* entities, as the second and third levels of nesting, respectively. We used a Jaql function to convert the XML documents into Jaql's data model while maintaining the same levels of nesting.

We use the following three queries over *CustAcc* documents:
Q1: *For each customer, report the accounts that have holdings along with the holdings sum under each account.*

```
read(hdfs('CustAcct')) -> sumCustAccts();
```

Q2: *As in Q1 while restricting the reported accounts to certain category, e.g., "Business".*

```
read(hdfs('CustAcct')) -> sumCustAccts()
  -> transform each rec {
       Cust: rec.Cust,
         BusinessAccts:
           rec.Accts -> filter $.AcctCategory =="Business")};
```

Q3: *As in Q1 while reporting only a sample from the accounts grouped by customers' countries.*

```
read(hdfs('CustAcct')) -> SampleByCountry(1000)
  -> filter $.Country == "USA";
```

These top-level queries are implemented using the following helper functions:

```
//Report customer accts w/ holdings and their sums
sumCustAccts = fn( cust ) (
  cust
  -> transform each t {
       Cust:  t.Customer.id,
       Accts: holdingSums(t.Customer.Accounts)}
);
// Sample N accounts per country.
SampleByCountry = fn( cust, sampleSize )(
  cust
  -> transform each t {
         Country: ValidateCountry(t.Customer.Country),
         Accts: holdingSums(t.Customer.Accounts)}
  -> group by location = $.Country into {
       Country: location,
       Sample: $[*].Accts -> top sampleSize}
);
// Sum of holdings for accounts that have holdings
```

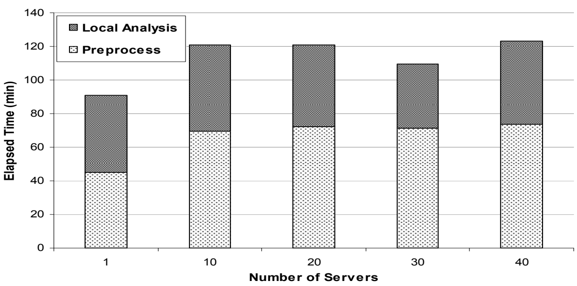

Figure 7: Scale-up expr. using intranet analysis workload.

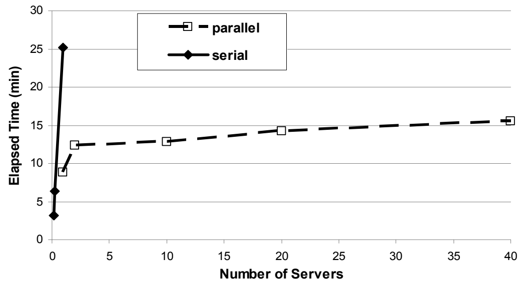

Figure 8: Scale-up expr. using application log workload.

```
holdingSums = fn( accts ) (
  accts
  -> filter exists($.Holdings)
  -> transform each acct {
       Acct: acct.id, AcctCategory: acct.Category,
       SumHoldings:
         acct.Holdings.Position
         -> transform $.Quantity -> sum()}
);
```

These queries illustrate several core Jaql features. First, the queries are expressed by composing multiple functions that abstract away where the data is stored and how its formatted. The body of the functions simply operate on arrays of values, allowing functions to be re-used across queries. Next, Jaql's focus on composable syntax supports data manipulation of values that are nested multiple-levels deep. For example, the `acct.Holdings.Position` is projected and aggregated in the `holdingSums` function. Finally, Jaql is able to peel away the functions and aggressively optimize across function call boundaries.

Notice that in Q3 the filter predicate will be pushed down, at compile time, below the group by statement because the predicate is on the grouping column used inside `SampleByCountry()`. But, it cannot be push-down below the transform statement, because the **ValidateCountry**() function, which validates the country name, is an example of a user-defined function with unknown semantics.

In Figure 6, we report the results obtained from scale-up experiments. The dataset size was increased from 40GB to 320GB while proportionally increasing the cluster size from 5 nodes to 40 nodes. The results show that Jaql scaled well while evaluating queries that manipulate deeply nested data. Jaql required approximately the same amount of time to process larger data sets when using proportionally more hardware. Furthermore, the results illustrate the benefit of the filter push-down rewrite where Q3_FPD (Q3 with filter push-down enabled) performed about 20% better than Q3_NFPD (Q3 with filter push-down disabled). Such rewrites are crucial for Jaql to support functions and achieve better performance.

6.2 Intranet Analysis Workload

Jaql is used at IBM to analyze internal data sources to create specialized, high-quality indexes as described in [2]. 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 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– key functionality includes language identification and information extraction 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 semi-structured 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 filters in the Local Analysis step was more selective for that sample of data. Overall, the results illustrate that Jaql scales well for the given workload.

6.3 Log Processing Workload

We evaluated the scale-up performance of the record cleansing task from Section 4.5. 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.

7. CONCLUSION

We have described Jaql, an extensible declarative scripting language and scalable processing system. 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, functions, 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.

8. REFERENCES

[1] S. Balakrishnan et al. Midas: integrating public financial data. In *SIGMOD '10*, pages 1187–1190, New York, NY, USA, 2010. ACM.

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

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

[4] G. E. Blelloch. Nesl: A nested data-parallel language (3.1). *CMU-CS-95-170*, 1995.

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

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

[7] R. Chaiken et al. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. In *VLDB*, 2008.

[8] C. Chambers et al. Flumejava: easy, efficient data-parallel pipelines. In *PLDI*, pages 363–375, 2010.

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

[10] S. Das et al. Ricardo: Integrating r and hadoop. In *SIGMOD Conference*, pages 987–998, 2010.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[12] D. DeWitt et al. GAMMA – A High Performance Dataflow Database Machine. In *VLDB*, 1986.

[13] 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.

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

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

[16] http://code.google.com/p/jaql/.

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

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

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

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

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

[22] R. Krishnamurthy et al. SystemT: A System for Declarative Information Extraction. *SIGMOD Record*, 37(4):7–13, 2008.

[23] 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.

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

[25] G. Lodi et al. Defending financial infrastructures through early warning systems: the intelligence cloud approach. In *CSIIRW '09*, pages 18:1–18:4, New York, NY, USA, 2009. ACM.

[26] G. Malewicz et al. Pregel: A System for Large-Scale Graph Processing. In *SIGMOD*, 2010.

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

[28] Netflix prize. *http://www.netflixprize.com*.

[29] M. Nicola, I. Kogan, and B. Schiefer. An xml Transaction Processing Benchmark. In *SIGMOD*, 2007. See `http://tpox.sourceforge.net/`.

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

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

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

[33] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. In *Scientific Programming Journal*, page 2005, 2005.

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

[35] A. Sala, C. Lin, and H. Ho. Midas for government: Integration of government spending data on hadoop. In *ICDEW*, pages 163 –166, 2010.

[36] Scalapack project. `http://www.netlib.org/scalapack/`.

[37] M. Stonebraker. The Case for Shared-Nothing. In *IEEE Data Engineering*, March 1986.

[38] A. Thusoo et al. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

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

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

[41] F. Xu, K. Beyer, V. Ercegovac, P. J. Haas, and E. J. Shekita. E = MC3: Managing Uncertain Enterprise Data in a Cluster-computing Environment. In *SIGMOD*, 2009.

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