A Common Compiler Framework for Big Data Languages: Motivation, Opportunities, and Benefits

Vinayak R. Borkar Michael J. Carey
University of California, Irvine
{vborkar,mjcarey}@ics.uci.edu

1 Introduction

We are in the era of Big Data and cluster computing. Data sizes have been growing at an exponential rate. At the same time, growth in computing power has been stagnating due to physical limits in processor technology. The only cost effective way to keep up with the growing data trend has been to harness multiple commodity computers in a shared-nothing configuration. Google, needing to manage extremely large amounts of web data, developed the MapReduce [16] platform. The MapReduce system provides a simple way for developers to express a data-oriented computation, by implementing two single-threaded functions (**map** and **reduce**), that is then automatically parallelized to run on large clusters of commodity machines. Yahoo! soon created the Hadoop [5] platform, based on the MapReduce specification, and made it available as open source software. Hadoop has since become the de facto MapReduce implementation outside of Google.

Initially MapReduce (Hadoop) greatly empowered engineers to run parallel jobs on large clusters to crunch virtually unlimited amounts of data while writing what appeared to be simple functions. However, over time many developers found themselves writing similar, yet different, functions to implement new jobs. Having to write MapReduce functions in an imperative language like Java proved to be time consuming and the proliferation of functions created a maintenance problem, prompting developers to explore the possibility of creating declarative high-level languages to express computation. Sawzall [30] was created inside Google for processing large corpora of text in parallel using MapReduce. Yahoo! developed the Pig [29] system along with its Pig Latin [27] language to express data processing in a declarative language resembling the relational algebra [15]. Facebook created Hive [2], an implementation of a SQL-like language. IBM developed the Jaql [23] language for processing large amounts of JSON data. Pig, Hive, and Jaql all compile queries in their respective languages into MapReduce jobs to run on the Hadoop platform. Microsoft proposed SCOPE [13], a system to compile a sequence of SQL-like statements to run in parallel on their own Dryad [21] data-parallel platform. The Dremel [25] system was created by Google for expressing analytical queries interactively in a subset of SQL using a custom column-based runtime platform. At the University of California, Irvine, we developed the AQL language for processing large amounts of semi-structured data, as part of the ASTERIX platform [9, 12, 7]. VXQuery [6] is a project under incubation at the Apache Software Foundation that aims to run XQuery [4] queries over large corpora of XML documents using a cluster of shared-nothing computers.

Declarative languages targeting various data-parallel platforms have seen dramatic growth in popularity in the past few years. In 2010 Facebook told us that upwards of 95% of their Hadoop jobs were automatically

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

generated by the Hive platform as a result of compiling HiveQL queries [32]. In around the same time frame Yahoo! was seeing more than 65% of their Hadoop jobs being submitted through their Pig platform.

The current Big Data movement has unleashed a wave of efforts where researchers and practitioners are creating new languages that better suit various target audiences without having to conform to established data processing standards like SQL. The need to process non-relational data in the Big Data space has also contributed to the invention of new languages. We conjecture that we will continue to witness the creation of new data processing languages that target specific domains, just as we have seen a growth of domain-specific languages in the Scala [26] world. This presents an important opportunity, as declarative languages for data processing share a common set of requirements and properties which suggest a unified platform upon which future languages can be built. Here we analyze four data processing languages (SQL, XQuery, HiveQL, and Pig Latin) with an aim to compare and contrast language features in Section 2. In Section 3 we take a deeper look at the internals of two open source systems that are popular for Big Data processing, Pig and Hive, to understand the commonalities and differences of the two systems. Based on our observations, Section 4 enumerates requirements for a unified platform and describes how one such platform, one that we are building (called Algebricks), is shaping up. Finally, we conclude in Section 5.

2 Languages and Models for Data Processing

In this section, we look at four known declarative languages that have been used successfully for large-scale data processing tasks, namely SQL, XQuery, HiveQL, and Pig Latin.

2.1 SQL: Structured Query Language

SQL emerged long ago as the de facto winner for processing relational data stored in relational database management systems. The relational model [15] that forms the basis of the logical model for data representation in RDBMSs is based on set-theory, and the SQL language is based on first-order logic. Data stored in relational databases is logically modeled as a collection of tables (relations) containing records (tuples) comprised of fields (attributes). Every stored table has a well-defined schema that describes the names and data types of the fields that are contained in the records contained in that table. Every record in a table must have all the fields mentioned in the schema (although fields are allowed to have unknown or missing values in the form of NULLs). Another characteristic of the relational model is that fields of a record in a table must be of a scalar type; a field value cannot itself be a table. In other words, the relational model allows for a strict two-level hierarchical structure, where the outer level is a table and the inner level contains scalar fields. It is important to note that while the logical model in RDBMSs is based on flat tables, an actual implementation is free to store the data using any physical format as long as the logical information is captured completely. For example, relational databases frequently store tables in the form of B+Trees using the unique value in a record (its primary key) as the indexing key. Columnar storage [31] is another example of a different physical storage format, one where tabular information is stored a column at a time. In a parallel database, where data is stored by partitioning the data, more choices with respect to partitioning strategies are available to the implementation.

The SQL language is based on relational algebra, which in turn is based on first-order logic. A SQL query describes what is expected in the result and does not specify how the result is to be computed, making the language "declarative". In contrast, most programming languages such as C, C++, or Java tend to be "imperative"; the programs expressed in those languages actually describe the exact steps to be performed. Declarative languages leave the language implementation free to explore a range of choices with regards to how to actually compute the final result. An implementation can choose the best way to evaluate a query based on the physical storage strategy used to store the tables participating in the query. This property of SQL (and declarative languages, more generally) has made query optimization a flourishing area of research for the last few decades.

2.2 XQuery

Although SQL and relational databases were widely successful for storing and managing enterprise data, the rigid nature of most implementations with respect to schema management made way for XML as a way of representing data that was more semi-structured. As XML started to become more popular for managing semi-structured data, multiple languages were proposed. XML-QL [17] and Quilt [14] were initially proposed as possible languages for querying XML data. Eventually XQuery [4] was developed, including concepts from XML-QL and Quilt, and it was standardized by the W3C for querying XML data.

An abstract data model called the XQuery Data Model (XDM) [19] is the logical model used to describe the semantics of the XQuery language. XDM is more general and subsumes XML for representing data. XQuery allows data to be typed using the XML Schema standard [10], but does not require it, making XQuery well suited for data whose types are not known upfront. The fundamental concept of representing data for processing with XQuery is the "sequence of items". Every query expression in XQuery operates on a sequence of items to produce a sequence of items. The order of items within a sequence is important and the semantics of XQuery specify the exact order of items that every expression must produce.

The FLWOR expression, along with Path Expressions, forms the core of the XQuery language. The FLWOR construct (For-Let-Where-Orderby-Return) is similar to the Select block (Select-From-Where-Orderby) found in SQL. Path Expressions in XQuery can be translated into FLWOR expressions using the Path Step Expression as a primitive building block, as shown in [18]. Researchers have proposed various algebras based on relational algebra and extensions to express XQuery semantics [22, 28], showing that slight extensions to relational algebra are sufficient to capture the meaning of XQuery queries and to provide a basis for optimizing those queries.

2.3 HiveQL

Facebook created the Hive system to be able to process large amounts of data using the Hadoop platform. Hive was created with the express goal of making large-data processing accessible to data analysts who were already familiar with SQL. HiveQL has thus been developed to be a subset of SQL with some extensions to the supported data model and query syntax.

On the data model front, Hive provides support for tables containing standard primitive data types like in SQL. In addition, Hive adds support for nested data through Array, Map, and Struct data types. Fields in tables can be declared to be of these types and can be used to nest data to arbitrary levels.

HiveQL supports most of the standard SQL syntax. HiveQL limits joins to only equality joins. Non-equality joins or cross products are not directly supported by the syntax. Although non-equality joins can be simulated through the use of a "1 = 1" join predicate and a Where clause to capture the non-equality predicate, such queries are not optimized by the HiveQL optimizer. In order to access nested data in the form of Arrays, Structs, and Map types, HiveQL provides functions to navigate values of these types. In addition to scalar values, HiveQL also includes tuple-functions that can be used to explode nested data into tuples for performing standard relational processing. The current release of HiveQL does not allow the construction of a Map-valued field by grouping multiple records. However, third-party packages such as [3] can be used to add this functionality to Hive.

2.4 Pig Latin

PigLatin, the language exposed by the Pig system from Yahoo!, allows users to express queries in the form of steps. Each step in PigLatin converts its input data (a collection of records) into an output collection of records. At each step, the user can use Load, Filter, Group, Cogroup, ForEach, Join, Order, Distinct, Cross, Union, or Split operators to transform data. Most of the resulting steps (except Cogroup and Split) can be directly mapped to relational algebra operators. The Cogroup operation can be expressed as a Group operation followed by a Join operation and hence can be expressed as a combination of relational algebra operators. The Split operation

in PigLatin can be transformed into multiple Filter (Select) operations when translating PigLatin into relational algebra operators.

Records in Pig can contain scalar valued fields or nested data in the form of tuples, bags and maps.

2.5 Wrap-up

Table 4 summarizes the similarities and differences between these four query processing languages.

Feature	SQL	XQuery	HiveQL	PigLatin
Declarative	Yes	Yes	Yes	Yes
Collection data model	Bag	Sequence	Bag	Bag
Instance data model	Tuples of scalar values	XQuery data model item	Tuples of scalar and complex values	Tuples of scalar and complex values
Support for nested data	No	Yes	Yes (Limited)	Yes (Limited)

Table 4: Query language similarities and differences

3 Data Processing Systems: Pig and Hive

In this section we look at two popular Big Data query processing systems that were built independently, Pig and Hive, with an eye for their system architectures and the steps followed in accepting user queries and translating them into executable artifacts to produce results.

3.1 Apache Pig

The Apache Pig platform first parses the textual representation of a user query into a logical plan representing the query. The logical plan is comprised of a DAG structure where nodes represent operators such as Load, Filter, Group, etc. The resulting logical plan is transformed using rules to create an equivalent, more efficient plan. Currently, the rules implemented in the Pig compiler perform the following rewritings (rules meant to purely aid in the rewriting process by normalizing plans are not listed):

1. Push Filters to eliminate records early
2. Push "flatten" style operators that increase tuple cardinality to later in the plan
3. Merge multiple chained ForEach operators when possible
4. Optimize placement of Limit operators to reduce the number of tuples early

After the optimization process, Pig translates the final logical plan into a sequence of one or more MapReduce jobs to be executed using the Hadoop MapReduce platform. More details regarding Pig's optimizer can be found in [20].

3.2 Apache Hive

The architecture of Apache Hive looks very similar to that of Pig. Hive first parses queries in HiveQL into a logical parse tree representation. A Semantic Analyzer consults metadata information in the system catalog to check the existence of the tables named in the query. The semantic analyzer also makes sure that the query is well-formed with respect to the fields of tables accessed in the query and flags any operations that are incorrect with respect to data types. The parse tree is then translated into a logical plan by the Logical Plan Generator. The logical plan is a tree of operators where some of the operators are relational algebra operators while others are specific to Hive to ease the construction of MapReduce jobs. The logical plan is optimized using a rule-based optimizer (just as in Pig). The Hive rules perform logically similar tasks as those performed by the Pig optimizer,

but they have been implemented to process plans represented using the logical operators implemented in Hive. Hive has additional optimizations as compared to Pig, including:

1. Convert a series of joins into a multi-way join
2. Perform map-side partial aggregation for aggregate queries
3. Preserve and exploit interesting orders in subplans when possible
4. Use sampling to better plan queries

After the optimization process, as in Pig, the logical plan is converted into a series of MapReduce jobs to be evaluated on Hadoop [2, 33].

4 An Approach to Unification: Algebricks

Observing the amount of overlap in functionality in Pig and Hive (and query processors for XQuery, SQL, and AQL that the authors have implemented before) and the similarities in algebras used to express their semantics, we have embarked on creating a unified framework to help future implementors of parallel query processors to quickly build new language implementations to be evaluated over Big Data using a data-parallel platform. AQUA [24] was a similar effort undertaken back in the object database era to create a framework where bulk types (collections) and their operations were separated from OO data model details so as to develop one algebra capable of expressing the query semantics for a variety of OO languages.

Our platform, Algebricks [1], is a model-agnostic, algebraic layer for parallel query processing and optimization.

The Algebricks toolkit consists of the following parts:

1. A set of logical operators: Algebricks includes about fifteen logical operators that language implementors can use to construct query plans. In addition to the standard relational operators (Select, Project, etc), Algebricks contains operators for representing nested query plans.
2. A set of physical operators: While most logical operators have one corresponding physical operator, some operators provide alternate physical choices. In Algebricks, the Select operator has a single physical operator while the Join operator provides multiple choices such as Nested-Loop Join, Hybrid-Hash Join, Grace-Hash Join etc.
3. A rewrite rule framework: Algebricks currently includes a rule-based query transformation framework that provides interfaces for users to implement rewrite rules and a rule engine to execute rules to transform queries.
4. A set of generally applicable rewrite rules: Algebricks includes query transformation rules that usually show promise in most scenarios. For example, pushing Select operators to filter data early in a query is usually considered to help with query efficiency. Algebricks includes a rewrite rule to push Select operators lower in the query plans. Another example of a generally applicable rewrite rule for parallel query processing is one that converts an aggregation query into a two-step aggregation query (when the aggregate function lends itself to partial computation). Since aggregation functions are language dependent and implemented by the language author, their properties (such as the ability to be split into two-level aggregation functions) have to be indicated to the Algebricks framework for the parallel aggregation optimization rule to trigger.
5. A metadata provider API that exposes metadata (catalog) information to Algebricks: The metadata API serves as the window through which Algebricks looks at the host language's ecosystem. It is through the metadata API that Algebricks is able to learn about physical data properties of sources (for example), that is then used by the compiler to reason about interesting orders and interesting partitions during query optimization.
6. A mapping of physical operators to the runtime operators in Hyracks [11] (a data-parallel platform built from the ground up at UCI): Currently, Algebricks provides a translation of queries to a single runtime

layer, viz. Hyracks. While this is an artifact of the current implementation of Algebricks, in principle, it should be possible to extend Algebricks to generate runtime plans for other data-parallel platforms.

4.1 Data Model Agnostic Bulk Operators

An important principle that underlies the design of Algebricks, as mentioned earlier, is a data model agnostic design of bulk operators. To be useful for implementing arbitrary parallel data-intensive languages, Algebricks takes a two-level approach to represent operations on data. "Bulk Operators" are used to manipulate collections (sets, bags, lists, etc.) of tuples. Tuples serve as abstract containers that hold scalar data values whose types are defined by the language being implemented using Algebricks. While Algebricks provides bulk operators, language specific scalar data operations have to be implemented by the author of the language. Bulk operators in Algebricks are parametrized with "accessor" functions that expose to them data model properties necessary to perform their task. For example, the Select operator used to restrict the items in a collection based on a boolean predicate only needs a "function" that, when applied to an item, indicates if the item is to be preserved in the result or is to be discarded. By encapsulating data model specific operations in the function, the Select Operator itself remains agnostic to the specifics of data types needed to support a high-level language using Algebricks. Although this principle has been adopted by several data-parallel execution engines [5, 8, 11] to build extensible runtime systems, we use this principle in Algebricks to implement a data model agnostic compiler framework. Every piece of functionality usually present in a parallel query compiler, such as the query transformation framework used to manipulate query plans, the data property computation logic used to reason about interesting orders and interesting partitioning properties of data, and finally the logic necessary to generate runtime artifacts, is implemented in Algebricks following the same two-level approach described earlier. Query transformation rules that depend entirely on the semantics of bulk operators are made available out-of-the-box with Algebricks and can be re-used by every language implemented using the compiler framework. Similarly, runtime plan generation (a process usually referred to as "code generation") of bulk operations is provided by Algebricks requiring the author of the higher-level language to only provide code generation logic for data model specific functions.

4.2 The Anatomy of an Algebricks-based Compiler

Figure 1 shows the typical sequence of steps followed by a query compiler built using Algebricks. An incoming query string is first lexically analyzed and parsed to construct an abstract syntax tree (AST). The resulting AST is then checked for semantic and type errors before transforming it into an Algebricks logical plan. The Algebricks logical plan is composed of logical operators (described next) and serves as an intermediate language for query compilation. The initial logical plan is handed to the logical optimizer to be transformed into an equivalent, but more efficient, logical plan. The resulting optimized logical plan is then translated to an Algebricks physical plan by selecting physical operators for every logical operation in the plan; this is done by the physical optimizer. Both optimizers in Algebricks are rule-based and are configured by selecting the set of rules to execute during the optimization phases. Finally, the resulting physical plan is processed by the Hyracks job generator to produce a Hyracks job that is then parallelized and evaluated by the Hyracks runtime platform.

The query parser and the translator (the first two stages in Figure 1) are query language specific and have to be implemented by the developer of a new language's compiler. The next three stages (the two optimizers and the Hyracks job generator) are provided by the Algebricks library to be used by developers. Algebricks also includes a library of language independent rewrite rules that can be reused by the compiler developer. Additional language-specific rules are usually required in the optimization process and can be implemented by the developer by extending well-defined interfaces in the Algebricks library.

As shown in Figure 1, the various phases of the query compilation process need access to information about the environment in which the query is being compiled (usually described as catalog metadata). The

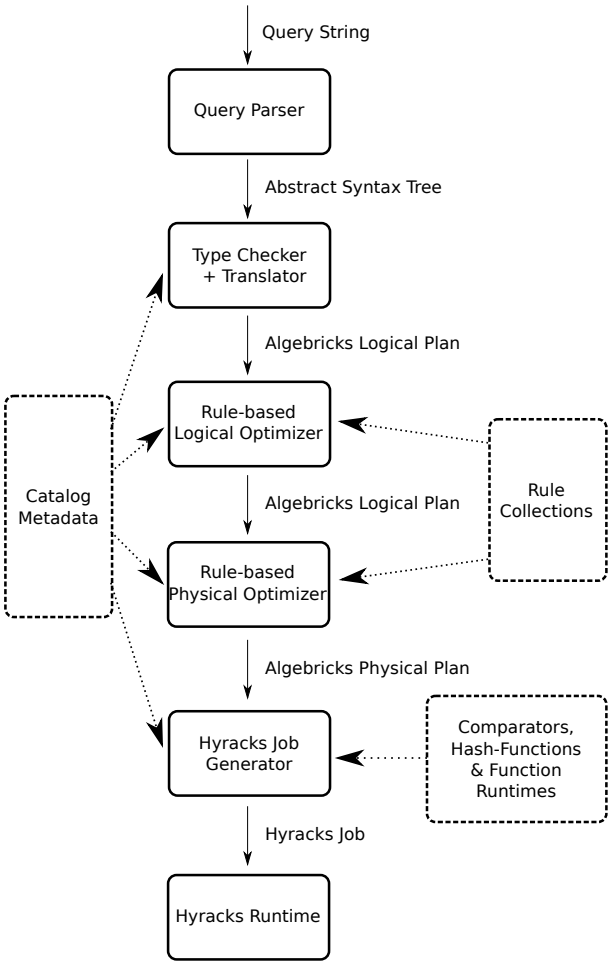

Figure 1: Flowchart of a typical Algebricks-based compiler

catalog contains metadata about the data sources that are accessible in a query. The catalog serves as the authoritative source of information about the logical properties of sources (such as schemas, integrity constraints, key information, etc.) and about their physical properties (access methods, physical locations, etc.). Algebricks provides a metadata interface that must be implemented by the compiler developer so that the various parts of the Algebricks compiler can access the relevant metadata information.

The Hyracks job generator maps the physical operators selected by the physical optimizer into Hyracks runtime operators. In the process of this translation, the Hyracks operators need to be injected with certain data model specific operations. The exact nature of the operations depend on the Hyracks operator being used. For example, the Sort operator in Hyracks must be provided with a comparator to compare two data model instances so that the input records can be sorted. Similarly, the Hash-Join operator needs a hash-function and a comparator to compute its result. When using Algebricks, the compiler developer must provide a family of operations that might be needed for Hyracks job construction.

5 Conclusion and Status

In this paper, we briefly looked at four declarative languages (SQL, XQuery, HiveQL, and PigLatin) used for large-scale query processing. We saw that an abstraction that extends relational algebra, allows the modeling of various types of collections (sets, bags, and lists), and is capable of providing support for nested data and

queries could be used to capture the semantics of queries expressed in the four languages we have seen here. The overlap in functionality in the Pig and Hive compilers shows that a framework that provides commonly required functionality in query compilers would help reduce the amount of code a developer has to write to implement a new language compiler.

To these ends we have developed Algebricks, a model-agnostic query compiler framework that meets the properties listed above. At UCI, we are implementing our own query language to process semi-structured data, called the ASTERIX Query Language (AQL). AQL is being built using Algebricks as the underlying compiler framework. As another proof of concept, we have already ported the HiveQL compiler to use Algebricks to compile HiveQL queries to run on the Hyracks data-parallel platform. We have seen a performance improvement of up to 9x and an average performance improvement of about 4x, at scale, compared to Hive on Hadoop. VXQuery is a project currently being incubated at the Apache Software Foundation that aims to provide a standards compliant XQuery 1.0 query processor that runs on large amounts of XML data using the Hyracks data-parallel platform. VXQuery also uses Algebricks as its compiler framework. Currently, VXQuery reuses Hyracks (90 KLOC) and Algebricks (36 KLOC) and provides a complete parallel XQuery processor for an additional 45 KLOC. More Algebricks implementation and performance details will be available in [1].

References

[1] Algebricks: A Compiler Toolkit for Building Parallel Query Languages for Big Data. In Preparation.

[2] Apache Hive website. `http://hive.apache.org`.

[3] Two Hive UDAF to convert an aggregation to a Map. http://www.adaltas.com/blog/2012/03/06/hive-udaf-map-conversion/.

[4] XQuery 1.0: An XML Query Language (Second Edition), 2010.

[5] Hadoop: Open-source implementation of MapReduce. http://hadoop.apache.org.

[6] The VXQuery Project. http://incubator.apache.org/projects/vxquery.html.

[7] ASTERIX Website. `http://asterix.ics.uci.edu/`.

[8] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: a Programming Model and Execution Framework for Web-Scale Analytical Processing. In *SoCC*, pages 119–130, New York, NY, USA, 2010. ACM.

[9] Alexander Behm, Vinayak R. Borkar, Michael J. Carey, Raman Grover, Chen Li, Nicola Onose, Rares Vernica, Alin Deutsch, Yannis Papakonstantinou, and Vassilis J. Tsotras. ASTERIX: Towards a Scalable, Semistructured Data Platform for Evolving-World Models. *Distrib. Parallel Databases*, 29:185–216, June 2011.

[10] P. Biron, A. Malhotra, World Wide Web Consortium, et al. XML Schema Part 2: Datatypes. *World Wide Web Consortium Recommendation REC-xmlschema-2-20041028*, 2004.

[11] Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing. In *ICDE*, pages 1151–1162, 2011.

[12] Vinayak R. Borkar, Michael J. Carey, and Chen Li. Inside "Big Data Management": Ogres, Onions, or Parfaits? In *EDBT*, 2012.

[13] Ronnie Chaiken, Bob Jenkins, Per A. Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.

[14] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In Gerhard Goos, Juris Hartmanis, Jan Leeuwen, Dan Suciu, and Gottfried Vossen, editors, *The World Wide Web and Databases*, volume 1997 of *Lecture Notes in Computer Science*, pages 1–25. Springer Berlin Heidelberg, 2001.

[15] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[16] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI '04*, pages 137–150, December 2004.

[17] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A Query Language for XML. *Computer Networks*, 31:1155 – 1169, 1999.

[18] P. Fankhauser, M. Fernández, A. Malhotra, M. Rys, J. Siméon, and P. Wadler. Xquery 1.0 formal semantics. *W3C Working Draft*, 2001.

[19] M. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 data model. *W3C working draft*, 15, 2002.

[20] Alan F. Gates, Jianyong Dai, and Thejas Nair. Apache Pig's Optimizer. *IEEE Data Engineering Bulletin*, 35(1), March 2013.

[21] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, pages 59–72, 2007.

[22] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, and Keith Thompson. TAX: A Tree Algebra for XML. In *Revised Papers from the 8th International Workshop on Database Programming Languages*, DBPL '01, pages 149–164, London, UK, UK, 2002. Springer-Verlag.

[23] Jaql Website. `http://jaql.org/`.

[24] Theodore W. Leung, Gail Mitchell, Bharathi Subramanian, Bennet Vance, Scott L. Vandenberg, and Stanley B. Zdonik. The AQUA Data Model and Algebra. In Catriel Beeri, Atsushi Ohori, and Dennis Shasha, editors, *DBPL*, Workshops in Computing, pages 157–175. Springer, 1993.

[25] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[26] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An Overview of the Scala Programming Language. Technical report, IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[27] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-so-Foreign Language for Data Processing. In *SIGMOD Conference*, pages 1099–1110, 2008.

[28] Yannis Papakonstantinou, Vinayak Borkar, Maxim Orgiyan, Kostas Stathatos, Lucian Suta, Vasilis Vassalos, and Pavel Velikhov. XML Queries and Algebra in the Enosys Integration Platform. *Data and Knowledge Engineering*, 44(3):299 – 322, 2003.

[29] Pig Website. `http://hadoop.apache.org/pig`.

[30] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[31] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-Store: A Column-Oriented DBMS. In *VLDB*, pages 553–564. VLDB Endowment, 2005.

[32] Ashish Thusoo. Personal Communication, August 2009.

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