Algebricks: A Data Model-Agnostic Compiler Backend for Big Data Languages

Vinayak Borkar[2*] Yingyi Bu[1] E. Preston Carman, Jr.[3] Nicola Onose[2*] Till Westmann[4]
Pouria Pirzadeh[1] Michael J. Carey[1] Vassilis J. Tsotras[3]

[1]University of California, Irvine [2]X15 Software, Inc. [3]University of California, Riverside [4]Oracle Labs
ecarm002@ucr.edu

Abstract

A number of high-level query languages, such as Hive, Pig, Flume, and Jaql, have been developed in recent years to increase analyst productivity when processing and analyzing very large datasets. The implementation of each of these languages includes a complete, data model-dependent query compiler, yet each involves a number of similar optimizations. In this work, we describe a new query compiler architecture that separates language-specific and data model-dependent aspects from a more general query compiler backend that can generate executable data-parallel programs for shared-nothing clusters and can be used to develop multiple languages with different data models. We have built such a data model-agnostic query compiler substrate, called *Algebricks*, and have used it to implement three different query languages — HiveQL, AQL, and XQuery — to validate the efficacy of this approach. Experiments show that all three query languages benefit from the parallelization and optimization that Algebricks provides and thus have good parallel speedup and scaleup characteristics for large datasets.

Categories and Subject Descriptors H.2 Database Management [*Languages*]: Query languages; H.2 Database Management [*Systems*]: Query processing

General Terms Languages, Design, Experimentation

Keywords Big Data, Query Languages, Parallel Query Processing

1. Introduction

Data has been growing at an astounding rate in recent years, primarily due to the growth of the Internet and social media. Stagnating single-processor speeds and the low cost of hardware have led companies to invest in clusters of commodity machines to cope with the storage and processing needs of this growing data.

Traditionally, businesses have used relational databases to store and query business data. However, most web companies have found that traditional databases are restrictive (owing to the flat relational model and transactional semantics) or inefficient for their use-cases. These shortcomings of relational databases have motivated companies to explore custom systems tailored to their use-cases instead of using a "one-size fits-all" paradigm. The MapReduce paradigm developed by Google [18] ignited this phenomenon. MapReduce was developed to support web-scale data processing on a cluster of commodity machines by providing a simple yet powerful user-model. Apache Hadoop [6] soon became the de-facto open-source implementation of the MapReduce model.

While the MapReduce model is simple, it is extremely low-level for most data processing tasks; it can require sophisticated puzzle-solving and Java skills to develop jobs that reflect the requirements of high-level end-user problems. As a consequence, higher-level declarative languages have since been developed by the same web companies to make their developers more productive in performing data-intensive jobs. For example, Google proposed Sawzall (an awk-like language for text processing) [37], Facebook created Hive (based on the SQL language) [41], and Yahoo developed Pig [34]. The success of declarative languages is evident from the job statistics released by the major web companies. A few years ago Facebook stated that upwards of 95% of the MapReduce jobs on their cluster are produced by the Hive compiler and Yahoo put the number of MapReduce jobs created by Pig at 65% (and growing). Recently, several new and more efficient SQL-on-Hadoop systems, e.g., Impala [33] and SparkSQL [7], have been built to improve the performance of Hive and the MapReduce runtime.

The proposed declarative languages differ significantly in terms of data models and semantics. For example, in the two queries listed below, the first query is a SQL query over the relational data model, while the second is an XQuery over the XML data model. Despite their differences, however, they share a common theme in that they provide a set of bulk operations to manipulate collections of values.

```
select l_returnflag, l_linestatus, sum(l_quantity),
  sum(l_extendedprice),
  sum(l_extendedprice*(1-l_discount)),
  sum(l_extendedprice*(1-l_discount)*(1+l_tax)),
  avg(l_quantity), avg(l_extendedprice), avg(l_discount),
  count(1)
from lineitem
where l_shipdate <= '1998-09-02'
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus;
```

```
fn:sum(
  for $r in collection("sensors")/dataCollection/data
  where $r/station eq "GHCND:USW00014771"
    and $r/dataType eq "PRCP"
    and year-from-dateTime(xs:dateTime(data($r/date)))
        eq 1999
  return $r/value
) div 10
```

[*] work done while at the University of California, Irvine.

SoCC '15, August 27 - 29, 2015, Kohala Coast, HI, USA.

http://dx.doi.org/10.1145/2806777.2806941

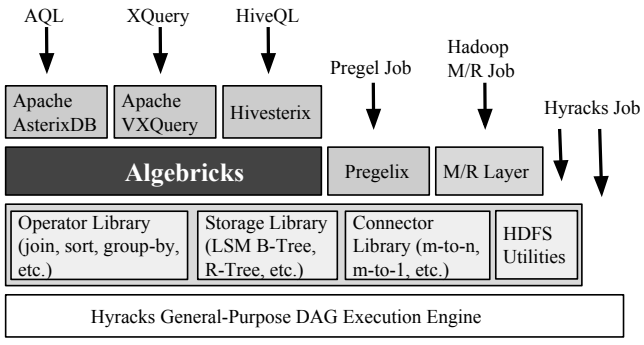

Figure 1. Algebricks Software Stack.

Based on the observed commonality between compilers of high-level declarative languages and the authors' prior experience building query compilers, this paper motivates the need for a framework that encapsulates common functionality that can be reused to build different languages for processing data at large scale. As part of the framework, we describe a set of data model-neutral operators and describe their semantics via examples. The framework provides a set of interfaces that must be implemented by the developer of a high-level language to stitch in the data model specifics of the target high-level language. The framework includes a rule-based rewriter* with a set of pre-implemented rules that are relevant for any language implemented using our operators (e.g., pushing filters closer to sources is a rule that is independent of the data model).

The reusable algebraic framework we have built for the compilation of higher level languages into data parallel jobs is called Algebricks. Algebricks is a by-product of the ASTERIX stack[3] that has been under development at UC Irvine and UC Riverside. Figure 1 shows the present set of projects that make up the ASTERIX Stack. Algebricks is shown within the highlighted box in Figure 1 and serves as the common compilation layer for three separate high-level languages: HiveQL [41], XQuery [17] and AQL [3]. All of these languages are declarative; they specify what is required to be in the result, but do not specify how to compute it. The implementation is free to use any access paths (indexes) and partitioning strategy in order to evaluate the query. The output of Algebricks are jobs that can be scheduled and executed in parallel on the Hyracks runtime platform, which is described in the following section.

The rest of the paper is organized as follows. Section 2 introduces the target runtime platform for Algebricks. Section 3 walks through the data model-neutral Algebricks interfaces and internals. Section 4 presents three query languages built on-top-of Algebricks: HiveQL, AQL, and XQuery. Experimental evaluations are discussed in Section 5. Related work appears in Section 6, and we conclude the paper in Section 7.

2. The Hyracks Runtime

Hyracks [13] is the compilation target platform for the Algebricks framework. This section covers the two bottom layers in Figure 1: the general-purpose DAG (directed acyclic graph) execution engine and its associated runtime libraries.

Execution Engine. Hyracks is a push-based data-parallel runtime in the same space as Hadoop [6] and Dryad [30]. Jobs are submitted to Hyracks in the form of DAGs made up of operators and connectors. Operators are responsible for consuming input partitions and producing output partitions. Connectors redistribute data between operators. For a job submitted in a Hyracks cluster, a master machine instructs a set of worker machines to execute clones of the operator DAG in parallel and orchestrates data exchanges. The Hyracks scheduler analyzes a job DAG to identify groups of operators (stages) that can be executed together at any time while adhering to the blocking requirements of the operators. The stages are then parallelized and executed in the order of their data dependencies. The Hyracks execution engine is extensible — users can define their own operators and connectors to construct a job DAG.

Runtime Libraries. The following Hyracks libraries serve as key runtime building blocks for Algebricks query plans (i.e., intermediate representations for queries in high-level languages).

- *Operator Library*. This library includes operators that support bulk operations on large datasets, such as hybrid-hash join [21], external sort, and both sort-based and hash-based group-bys. All those operators gracefully spill intermediate data to disks when memory is under pressure.
- *Connector Library*. Data exchange patterns in the connector library include hash partitioning, range partitioning, random partitioning, and broadcasting. A connector is also associated with a materialization policy that dictates whether or not to materialize data at the sender and/or receiver side and whether the materialization should block data sending/receiving. The Hyracks connector library covers a spectrum of data redistribution strategies beyond the Hadoop shuffle.
- *Storage Library*. A number of native storage and indexing mechanisms, such as B-Tree, R-Tree, inverted index, and their LSM-based counterparts [4], are included in this library.
- *HDFS Utilities*. HDFS [6] read and write operators (for multiple HDFS versions) are provided in this library. In addition, there is an associated client-side scheduler in this library to set Hyracks location constraints [13] properly to make a best effort for achieving data-local HDFS reads.

Algebricks targets Hyracks to create executable data parallel programs. However, Algebricks concepts could also be applied to other recent runtime platforms such as Spark [44] or Tez [8].

3. The Algebricks Framework

Algebricks is an algebraic layer for parallel query processing and optimization. To be useful to implement various data-intensive query languages, Algebricks has been carefully designed to be *agnostic* of the data model of the data that it processes. Logically, operators operate on collections of tuples containing data values. The types and formats of data values carried inside a tuple are not specified by the Algebricks toolkit; language implementors are free to define any value types as abstract data types. For example, a language developer implementing a SQL compiler on top of Algebricks would define SQL's scalar data types to be the data model, provide a type computer for SQL expressions, and implement runtime functions such as scalar functions and aggregate functions as well as runtime operations such as comparison and hashing. AQL [3] has a richer set of data types, including various collection types and nested types, and these have been implemented on top of the Algebricks API as well.

The Algebricks framework consists of the following parts:

- A set of logical operators,
- A set of physical operators,
- A rewrite rule framework,
- A set of generally applicable rewrite rules,
- A metadata provider API that exposes metadata (catalog) information to Algebricks, and,
- A mapping of physical operators to the runtime operators and connectors in Hyracks.

* A prototype of a cost-based optimizer based on Algebricks has been implemented at the Indian Institute of Technology, Bombay. However, this is not yet a part of the Algebricks code distribution.

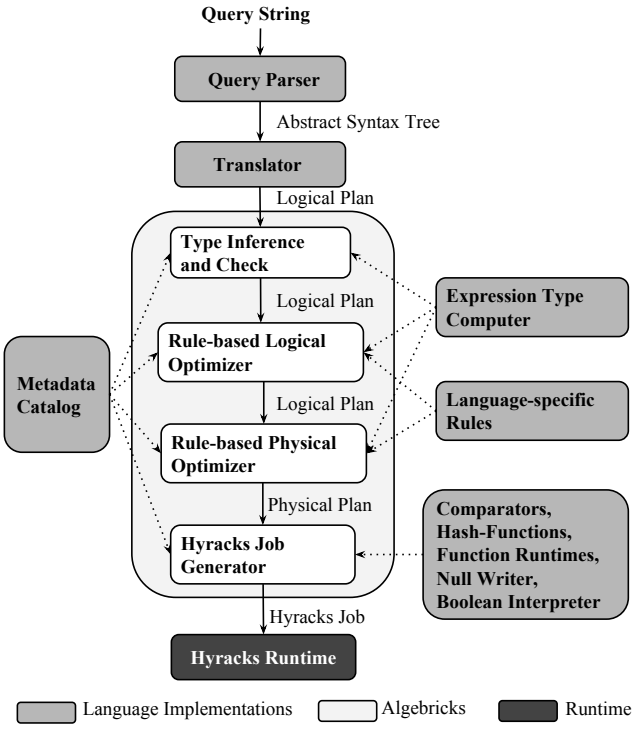

Figure 2. Flowchart of a typical Algebricks-based compiler.

Compilation Flow. Figure 2 shows the typical sequence of compilation steps followed by a query processor built using Algebricks. An incoming query string is first lexically analyzed and parsed to construct an abstract syntax tree (AST). This AST is then translated into the Algebricks logical plan, which is composed of logical operators (described in Section 3.2) and serves as an intermediate representation to perform the next steps involved in query compilation. Type inference and checking is done over the initial logical plan, using a language-provided expression type computer which infers the type and checks type errors for each individual expression. The logical plan is then handed to the logical optimizer which rewrites it heuristically using logical rewrite rules. The physical optimizer translates the optimized logical plan to an Algebricks physical plan by selecting physical operators (described in Section 3.3) for every logical operation in the plan. Both optimizers are rule-based and configured by selecting the set of rules to execute during the optimization phases. The resulting physical plan is processed by the Hyracks Job Generator to produce a Hyracks job that is parallelized and evaluated by the Hyracks execution engine.

Rewriting Rules. The query parser and translator (the first two stages in Figure 2 are query language specific and must be implemented by a query language developer. The next three stages (the optimizers and Hyracks job generator) are provided by the Algebricks library to be used by the developer. Algebricks also includes a library of language-agnostic rewrite rules that can be reused by the compiler developer. Additional language-specific rules are usually required in the optimization process, too. These need to be implemented by the developer by extending well-defined interfaces in the Algebricks library. Note that a language-provided expression type computer should be called during certain rule applications to propagate updated types for the updated intermediate logical plans. The rewrite rule framework is described in Section 3.4.

Metadata Catalog. As shown in Figure 2, 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 catalog is the authoritative source of logical properties of sources (such as schema, integrity constraints, key information, etc.) and their physical properties (access methods, physical location, etc.) Algebricks provides a Metadata Interface (Section 3.1) that must be implemented by the compiler developer so that the various parts of the Algebricks compiler can access relevant metadata information.

Runtime Operations/Functions. The Hyracks Job Generator maps physical operators selected by the Physical Optimizer to Hyracks runtime operators and connectors. In the process of this translation, the runtime operators need to be injected with data model-specific operations. The exact nature of the operation depends on the runtime operator being used. For example, the Sort operator must be provided a comparator to compare two data model instances so that the input records can be sorted; the Hash-Join operator needs a hash-function and a comparator to compute its result; the Select operator requires a boolean interpreter to interpret the result of its filtering condition expression; the Outer-Join operators need a null writer to generate null fields according to the language-defined data format. When using Algebricks, the language developer must provide families of operations (usually one family for each data type, for example, comparators and hash functions) that are needed for correct runtime Job construction. Finally, the language developer must implement a mapping from function expressions to its own function runtime implementations (e.g., arithmetic functions, string functions, aggregate function, etc.) so that the data model agnostic runtime operators (e.g., select, assign, aggregate, etc.) can evaluate the data model-dependent functions.

3.1 Metadata Interface

Every data-management system that accesses stored data or external data sources to process queries requires some form of metadata that describes locations of the stored files or external data sources and the various access paths available to get to the data bits. For example, relational databases use a catalog to store information (schema) about the tables and indexes present in the database; such information is used by the query compiler when compiling query plans. Algebricks provides a Metadata Interface which must be implemented by the author of the host language so that the Algebricks compiler can access metadata information required during the compilation process. The goal of this interface is to allow Algebricks access to all relevant details about datasources while not constraining the host language to a specific representation of the metadata internally. The Metadata Interface provides the following pieces of information to the compiler:

- **Data Source Metadata**. Stored collections of data or external data sources are both modeled as a Data Source in Algebricks. For example, in HiveQL, a table would be considered a Data Source while a dataset or an external data feed would be modeled as a Data Source in AsterixDB [3, 28]. The Data Source interface enables Algebricks to get metadata about the data that it represents. Source metadata can be broadly classified as Logical Metadata and Physical Metadata. Logical Metadata about a Data Source includes the type information of the data it represents and other logical properties such as functional dependencies and integrity constraints (keys, etc.). Physical Metadata provides details of how the data is physically stored or obtained, specifically providing partitioning information and any local ordering present in the stored data.
- **Access Path Binding**. The Metadata Interface in Algebricks also serves as a factory to create the runtime binding to connect the compiled Hyracks job to the "last mile" (operators that are capable of reading the data from the stored location).
- **Function Metadata**. Algebricks further uses the Metadata Interface to find Function Metadata to optimize function applications that appear in the plan being compiled.

3.2 Logical Operators

The logical operators in Algebricks draw inspiration from a variety of algebraic systems proposed for nested-relational models description [29, 31] and for the processing of semi-structured data [19, 32, 36]. Tuples in Algebricks are used as an abstraction that hold values. Algebricks operators logically consume and produce collections of tuples. Tuples contain fields that are created and manipulated by the operators; field names are represented by names prepended with a $ sign in the description that follows. Note that tuples as used in this section are not to be confused with tuples in the relational model, as Algebricks tuple field values are allowed to be of arbitrary data types (not just scalar types like in the relational model). For example, in AsterixDB, an entire record (a composite data structure that contains name-value pairs) can be the value of a field in an Algebricks tuple and a concrete field type (i.e., open type in AQL [3]) can even be determined at runtime.

Operators may also contain references to scalar or aggregation functions wrapped inside a generic logical expression interface which exposes the used fields and allows for field name substitution. This allows Algebricks to rewrite the plan, while at runtime native functions are run against the data model of the implemented language. A logical expression also has methods for computing logical constraints that can be inferred from that expression. The constraints currently implemented are functional dependencies and equivalence classes. They are used during rewriting, e.g. to reason about interesting orders and partitioning properties. There are three implementations of the logical expression interface:

- *Constant* holds constant values.
- *VariableReference* points to a logical variable (with a logical id) which is a column in a tuple.
- *FunctionCall* holds a function identifier and references to argument expressions. It models all expressions which are not constants nor variables and it is the only kind of expression that is not a leaf in an expression tree.

FunctionCall is further refined by four implementations:

- *ScalarFunctionCall* is used for functions that compute their result by looking at only one tuple.
- *AggregateFunctionCall* functions produce a result after iterating over a collection of tuples. SQL aggregate functions belong here.
- *StatefulFunctionCall* is similar to AggregateFunctionCall, but produces a result after each tuple, not only at the end. Position variables in XQuery fall in this category: for every binding in a for-clause, they output the binding's index in the sequence.
- *UnnestingFunctionCall* is given an input tuple and then can be called multiple times, each time returning a possibly different result, until a special value is returned. Examples are the *range* function in Asterix which returns integer values from a specified range or the *collection* function in XQuery which returns nodes according to the arguments' available collection.

From a given function expression, a query language implementation can create the runtime artifact, i.e., an evaluator, which is capable of implementing its language-specific runtime semantics. Evaluators run code specific to every language: in the case of Hivesterix (a HiveQL implementation on top of Algebricks; see Section 4.1), it runs the native Hive evaluators, which were not designed specifically for Algebricks. Translation between logical function calls and evaluators is done at job generation time (Section 3.5). This design has the advantage that it allows multiples ways of implementing the translation from logical to physical. In Hivesterix, this is done by having the function call expressions reference their evaluators. In AsterixDB, evaluators register themselves in a global map which is keyed on function identifiers.

3.3 Physical Operators

While the logical operators described in Section 3.2 capture the semantics of an Algebricks logical plan, physical operators are used to specify the exact algorithm to use for evaluating each operator. During the query rewrite process (described in Section 3.4) rules assign physical operators to each logical operator in the query plan, thus deciding the concrete algorithms to use in evaluating the query. Most logical operators in Algebricks map to a unique physical operator. However, Join and Group-By operators have multiple concrete implementations. Physically, joins can be performed using the Hybrid-Hash Join [21] or the Nested-Loop Join algorithms. The Group-By operation can be performed using a pre-clustered implementation that assumes that its input is already clustered with respect to the grouping keys or by using a hash-based or sort-based Group-By algorithm that is capable of spilling to disk when the in-memory state exceeds available memory [42].

In addition to physical operators corresponding to logical operators, Algebricks also provides physical operators to enforce order properties as well as partitioning properties of data distributed to individual operations during parallel evaluation of the query. Data distribution between operator partitions is expressed using an Exchange operator, motivated by [24]. Currently, the data distribution strategies offered by Exchange operators in Algebricks are:

- *One-to-One Exchange*: This strategy is used for the local movement of data without redistribution of data between partitions.
- *Hash Exchange*: The Hash Exchange strategy hashes specified fields to determine how tuples are to be routed to the destination partitions. This operator helps in partitioning data based on field values so that two tuples with the same partitioning field values are routed to the same destination.
- *Range Exchange*: The Range Exchange strategy uses a provided range vector to determine the destination partition to send a record to, based on the values of specified fields. The range vector maps disjoint ranges of values to target partitions. The value of the partitioning fields of each record are used to probe the range vector to decide the target partition. In addition to sending equal values to the same partition, range-based partitioning also sends close-by values to the same site. Such a distribution strategy is useful for performing a global sort in a parallel manner.
- *Random Exchange*: Sometimes it is desirable to redistribute data to a set of partitions in a load-balanced manner, but without necessarily maintaining a value-based partitioning property. The Random Exchange Operator does exactly that by sending each record to a randomly determined partition.
- *Broadcast Exchange*: The Broadcast Exchange strategy is used to make sure that the same data is delivered to every partition of the next operator. An example situation for the use of the Broadcast Exchange operator is while joining a small relation with a large partitioned relation. The small relation is broadcast to each machine where a partition of the large relation resides prior to performing a local join at each site.

Moreover, the Hash Exchange, Range Exchange, and Random Exchange strategies have a second variant each, namely, Hash-Merge Exchange, Range-Merge Exchange, and Random-Merge Exchange. A merge variant applies the same partitioning strategy as the non-merge variant, but uses a priority queue on the receiving side that merges all incoming streams so as to maintain a specified sort order that the data originally possessed before partitioning.

In Algebricks, the physical operator layer is extensible. Compilers on top of Algebricks can register new physical operators by implementing the following interface:

```
1 public interface IPhysicalOperator {
2   public PhysicalRequirements requiredPropertiesForChildren(
3     IPhysicalPropertiesVector requiredByParent);
4
```

```
  public void deliveredProperties(ILogicalOperator op,
    IOptimizationContext context);

  public void contributeRuntimeOperator(
    IHyracksJobBuilder builder,
    IOpSchema propagatedSchema, IOpSchema[] inputSchemas,
    IOpSchema outerPlanSchema);
}
```

The methods `requiredPropertiesForChildren` and `deliveredProperties` compute and propagate ordering, grouping and partitioning properties, close in spirit to SCOPE [45]. These properties guarantee that the correct semantics of the operators are implemented, e.g., a Pre-Clustered Group-By will have its inputs partitioned by (a subset of) the group-by key and locally clustered on each partition, while allowing for optimizations, e.g., if the data partitioning already satisfies the requirements, no re-partitioning of the input is needed. The `contributeRuntimeOperator` method is essential in the job generation phase, discussed in Section 3.5.

In AsterixDB, we need physical operators for B-tree indexes, to enable efficient access to the native storage layer [4]. Since they are AsterixDB specific, they cannot be part of the Algebricks library, so they are added as language specific operators for accessing its native storage layer [4]. The BTreeSearch physical operator's `contributeRuntimeOperator` constructs a Hyracks B-Tree search runtime operator which is added to the job DAG together with an incoming edge from the Hyracks operator that produces the search intervals. BTreeSearch generally delivers a local ordering property on the key fields. For a primary index, the operator also delivers a hash-partitioning property of the key fields. The operator generally requires its input (a stream of key intervals) be broadcast over the locations where the index is partitioned, but for primary indexes, the required property is hash-partitioning.

3.4 Rewriter

The Algebricks optimizer uses Logical-to-Logical rewrite rules to create alternate logical formulations of the initial DAG. Logical-to-Physical rewrite rules then generate a DAG of physical operators that specify the algorithms to use to evaluate the query. For example, a Join operator might be rewritten to a Hash-Join physical operator. We expect that most language implementors using Algebricks will need a rewriting framework to perform additional useful data model-specific optimizations. The Algebricks toolkit contains a rewriting framework that allows users to write their own rewrite rules, but it also comes with a number of "out of the box" rules that the user can choose to reuse for compiling their high-level language. Examples of rules that apply for most languages include:

- **Push Selects**: Rule for pushing filters lower in the plan to eliminate data that is not useful to the query.
- **Introducing Projects**: Rule to limit the width of intermediate tuples by eliminating values that are no longer needed in the plan.
- **Query Decorrelation**: Rule to decorrelate nested queries to use joins when possible.

An example rule implemented in AsterixDB that is not of general applicability is one that uses ASTERIX metadata [3] to determine if a field is present in the declared schema (i.e., if its presence is known a priori) in order to determine which field access method to use: field-access-by-index or field-access-by-name.

The rewriter uses the properties below, which, together with the graph of operators, dictate whether a rule is fired or not.

- **Used/Produced Variables**. Operators schemas determine how projections and selections are pushed and are needed by most rules that change the order of operators.
- **Functional Dependencies and Data Properties**. The process of rewriting into a physical DAG uses partitioning properties and local physical properties (grouping, ordering) of input data sources. Similar to SCOPE [45], these are computed based on functional dependencies and variable equivalence classes. Based on partitioning properties, Data Exchange operators [22, 25] are introduced to perform data redistribution between partitions.
- **Equivalence Classes**. The optimizer analyzes equivalence classes of variables. For example, after an equal inner join, a variable in the left-hand side join key and its corresponding variable in the right-hand side join key are equivalent after the join operator. With the information of equivalent classes, functional dependencies and data properties are further propagated.

3.5 Job Generation

The job generation component outputs a Hyracks job implementing the Algebricks plan (Figure 2). The job is constructed by walking the plan and calling the `contributeRuntimeOperator` method on the physical operators. A physical operator can contribute either (a) a full Hyracks operator, (b) a micro-operator, which becomes part of a pipeline running inside one Hyracks operator, (c) a connector, typically when the Algebricks operator is an exchange. The Hyracks runtime for `DataSourceScan` and `WriteResult` operators are obtained through the Metadata Interface which also returns partitioning information needed to instantiate those operators. While the job generator currently only generates Hyracks jobs, its architecture conceptually supports other runtimes (e.g. Spark, Tez). E.g., instead of generating Hyracks connectors, in Tez we would create "edges" and in Spark "partitioners", which are all similar concepts representing data movement from producers to consumers.

4. Using Algebricks

In this section we delve into the details of how three query processing systems — Hivesterix (Hive-on-Hyracks), AsterixDB, and VXQuery — use Algebricks to compile queries.

4.1 Hivesterix

Hivesterix compiles HiveQL queries to run on the Hyracks platform. For each HiveQL query, Hivesterix obtains the physical query plan from the Hive compiler and turns that into an Algebricks logical plan for Algebricks to produce a Hyracks job that calls back to Hive's function evaluators at runtime. Let us walk through a HiveQL example. A simple HiveQL query shown below filters records from a TPC-H "lineitem" table to retain only those records which satisfy the filter predicate. Furthermore, the aggregated overall "revenue" from selected records are returned. The "lineitem" table used in the query has the schema shown in Table 1.

```
select sum(l_extendedprice*l_discount) as revenue
from lineitem
where l_shipdate >= '1994-01-01'
  and l_shipdate < '1995-01-01'
  and l_discount >= 0.05 and l_discount <= 0.07
  and l_quantity < 24;
```

Hivesterix translates this query into the Algebricks plan below. In the plan, "algebricks-*" (e.g., alegebricks-gte, algebricks-lte) is a function that Algebricks can recognize during rewriting but will use a language-provided runtime implementation at evaluation time.

```
WRITE_RESULT( $$revenue )
AGGREGATE( $$revenue:sum($$l_extendedprice*$$l_discount) )
SELECT( algebricks-and(
    algebricks-gte($$l_shipdate, '1994-01-01'),
    algebricks-lt($$l_shipdate, '1995-01-01'),
    algebricks-gte($$l_discount, 0.05),
    algebricks-lte($$l_discount, 0.07),
    algebricks-lt($$l_quantity, 24)) )
```

Column Name	l_orderkey	l_partkey	l_suppkey	l_linenumber	l_quantity	l_extendedprice	l_discount	l_tax
Data Type	bigint	bigint	bigint	bigint	float	float	float	float
Column Name	l_returnflag	l_linestatus	l_shipdate	l_commitdate	l_receiptdate	l_shipinstruct	l_shipmode	l_comment
Data Type	string	string	string	string	float	float	string	string

Table 1. Schema of the TPC-H lineitem table.

```
ASSIGN( $$l_shipdate, $$l_discount, $$l_extendedprice,
    $$l_quantity:
    column_expr($l, "l_shipdate"),
    column_expr($l,"l_discount"),
    column_expr($l, "l_extendedprice"),
    column_expr($l,"l_quantity") )
UNNEST( $$l:dataset(lineitem) )
EMPTY_TUPLE_SOURCE
```

While reading Algebricks plans, it is important to note that the dataflow is assumed to flow from the bottom of the plan to the top. The "unnest" operator produces the tuple streams from rows in the "lineitem" table. The Metadata Interface (Section 3.1) implemented in Hivesterix probes the Hive meta-store (the catalog database used by Hive-on-Hadoop) to get schema information about the table accessed by the query. As seen in Table 1, the "lineitem" table has sixteen columns. Note that the translator does not prune the source fields not used in the query. Algebricks includes the logic necessary to perform this pruning. Continuing on with the initial plan for the query, the translator creates an "assign" operator to extract fields from input tuples and a "select" operator to represent the "where" clause in the query. Finally, the "aggregate" operator is used to perform the aggregate sum and a "write" operator is used to write the results to HDFS.

After the translated plan is constructed, Hivesterix proceeds to optimize the query. The Hivesterix optimizer is populated with a total of 61 optimization rules out of which 4 rules are specific to Hivesterix and the rest are part of the common Algebricks framework. Figure 3 visualizes the finally generated Hyracks job from the optimizer. The Algebricks rewrite rules have all the logic necessary to parallelize the query without the need for Hivesterix to provide any additional code.

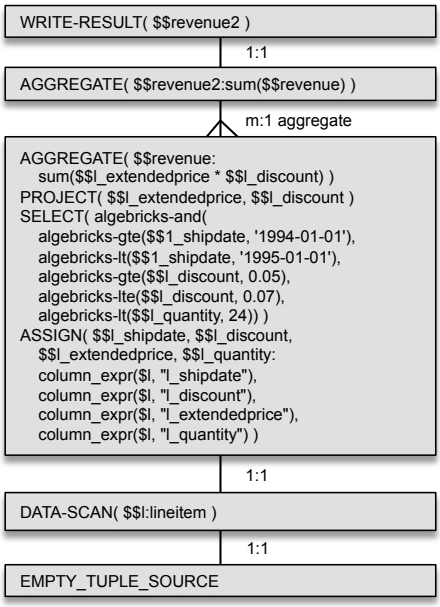

Figure 3. Hivesterix Hyracks Job.

Note that most of the final Hyracks job looks similar to the original translated plan with the following differences:

- A project operator is injected into the plan to prune unnecessary columns.
- An additional aggregate operator is injected to perform local aggregate in order to save the network bandwidth consumption, since the aggregation function *sum* is distributive.
- An exchange operator with the associated data redistribution strategy has been introduced between every pair of operators. Operators below the lower aggregate operator use only one-to-one exchange operators. A m-to-one exchange is placed between the lower (local) aggregate and the higher (global) aggregate.

4.2 Apache AsterixDB

Apache AsterixDB is a full-featured Big Data Management System for managing large quantities of semi-structured data. It is currently undergoing incubation at the Apache Software Foundation (ASF). AsterixDB's data model supports JSON-like data and adds more primitive and structured data types to JSON's data types.

As a slightly more complex example, consider the query below in the Asterix Query Language (AQL) [3]. This query performs a join between two natively stored data collections, "GleambookMessages" and "GleambookUsers", that matches records from the two datasets on the "author_id" and "id" fields respectively. Note that as allowed by the flexible AsterixDB data model, "author_id" is not in the declared schema of "GleambookMessages" but "id" is part of the "GleambookUsers" schema. Furthermore, we limit the results to contain users whose "user_since" is within a time range and messages whose "send_time" is within a specific time interval. Results are in the form of ADM (Asterix Data Model) [3] records with two attributes, "uname" and "message".

```
for $message in dataset GleambookMessages
for $user in dataset GleambookUsers
where $message.author_id = $user.id
  and $user.user_since >= datetime('2008-10-24T14:21:21')
  and $user.user_since < datetime('2008-10-25T14:21:21')
  and $message.send_time >=datetime('2011-02-24T20:01:48')
  and $message.send_time < datetime('2011-02-25T05:01:48')
return {"uname": $user.name, "message": $message.message}
```

The AQL translator translates the query to the following equivalent logical plan representation:

```
DISTRIBUTE-RESULT( $$20 )
PROJECT( $$20 )
ASSIGN( $$20:open-record-constructor(
    "uname", field-access-by-name($$1, "name"),
    "message", field-access-by-name($$0, "message")) )
SELECT( algebricks-and(
    algebricks-eq(field-access-by-name($$0, "author_id"),
        field-access-by-name($$1, "id")),
    algebricks-ge(field-access-by-name($$1, "user_since"),
        datetime("2008-10-24T14:21:21")),
    algebricks-lt(field-access-by-name($$1, "user_since"),
        datetime("2008-10-25T14:21:21")),
    algebricks-ge(field-access-by-name($$0, "send_time"),
        datetime("2011-02-24T20:01:48")),
    algebricks-lt(field-access-by-name($$0, "send_time"),
        datetime("2011-02-25T05:01:48"))) )
UNNEST( $$1:dataset("GleambookUsers") )
UNNEST( $$0:dataset("GleambookMessages") )
EMPTY_TUPLE_SOURCE
```

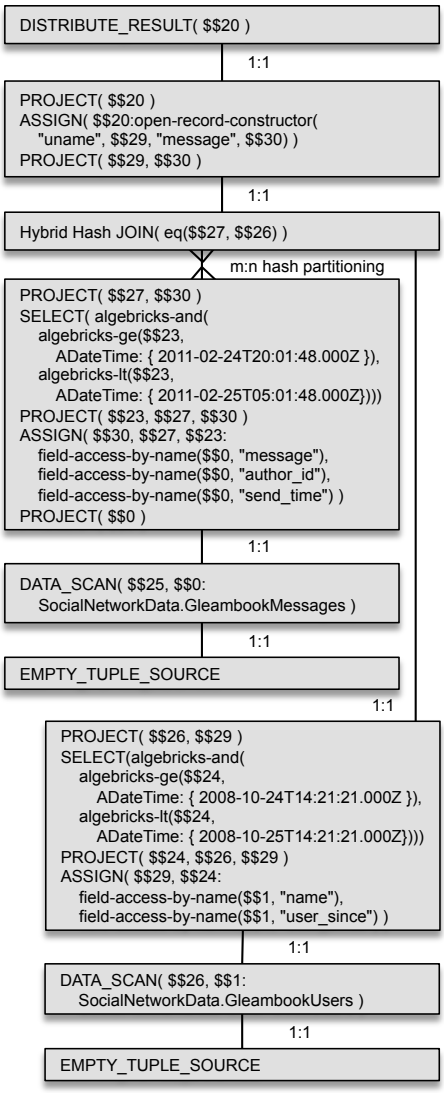

Figure 4. AsterixDB Hyracks Job.

Then, the Algebricks optimizer transforms the initial logical plan into an optimized physical plan by applying logical and physical rewriting rules. The optimizer utilizes 48 AsterixDB specific rules and 41 generic Algebricks rules. Figure 4 visualizes the generated Hyracks job for this example query.

Note that the final optimized Hyracks job has the following main differences from the original plan:

- A hybrid hash join operator is introduced to evaluate the equality join condition.
- The original select conditions are pushed into the two join branches.
- Project operators are injected into the plan wherever necessary.
- A hash partition exchange operator is enforced for the "GleambookMessages" branch to make sure data from this branch is partitioned by "author_id". Note that there is no re-partition exchange for the "GleambookUsers" branch because data from this branch has already been hash partitioned across the cluster according to the primary key "id".

4.3 Apache VXQuery

Apache VXQuery is an XQuery processor built for handling large collections of XML data. It uses Algebricks and Hyracks to process XML by adding a binary representation of the XQuery Data Model (XDM), an XQuery parser, an XQuery optimizer, and the data model dependent expressions. VXQuery is intended to implement the complete XQuery specification.

The example XQuery statement is based on weather data provided by the NOAA [1]. The query finds all precipitation ("PRCP") records for the "GHCND:USW00014771" station that occurred during 1999. The precipitation readings are recorded in tenths of an inch. After summing all readings, the result is divided by 10 to find the total precipitation for 1999 in inches. The query (also included in our experiments as VQ2) is shown below followed by a sample XML weather record:

```
fn:sum(
  for $r in collection("sensors")/dataCollection/data
  where $r/station eq "GHCND:USW00014771"
    and $r/dataType eq "PRCP"
    and year-from-dateTime(xs:dateTime(data($r/date)))
        eq 1999
  return $r/value
) div 10
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<dataCollection pageCount="1" totalCount="3">
  <data>
    <date>1999-12-02T00:00:00.000</date>
    <dataType>PRCP</dataType>
    <station>GHCND:USW00014771</station>
    <value>0</value>
    <attributes>
      <attribute></attribute>
      <attribute></attribute>
      <attribute>a</attribute>
      <attribute></attribute>
    </attributes>
  </data>
  <data>
    <date>1999-12-02T00:00:00.000</date>
    <dataType>TMIN</dataType>
    <station>GHCND:USW00014771</station>
    <value>-6</value>
    <attributes>
      <attribute></attribute>
      <attribute></attribute>
      <attribute>0</attribute>
      <attribute></attribute>
    </attributes>
  </data>
</dataCollection>
```

Apache VXQuery begins by translating the XQuery statement into a logical query plan using the Algebricks logical operators. The translator follows the XQuery specification and creates a correct Algebricks query plan with all the XQuery type checking expressions translated into the query plan. The optimizer will use these expressions to type check the query plan.

XQuery is a complex language with well-defined but complicated semantics for various parts of the language [2]. Below we show the initial Algebricks logical plan generated by the VXQuery translator for the weather query. Note that what appears as a simple path-step in XQuery (the / operator), is equivalent to a more complex expression that involves unnesting, sorting, duplicate-elimination, etc. For example, lines 68-74 in the listing below represent "/dataCollection". Please note that while we do not expect the reader to glean all the details of the plan shown, we wish to communicate the fact that Algebricks is capable of implementing the semantics of a complex query language like XQuery.

```
DISTRIBUTE-RESULT( $$59 )
UNNEST( $$59:iterate($$58) )
ASSIGN( $$58:divide(promote(<anyAtomicType?>, data($$56)),
    promote(<anyAtomicType?>, data($$57))) )
ASSIGN( $$57:10 )
ASSIGN( $$56:sum(promote(<anyAtomicType*>, data($$55))) )
SUBPLAN {
  AGGREGATE( $$55:sequence($$54) )
  ASSIGN( $$54:sort-distinct-nodes-asc-or-atomics($$53) )
  SUBPLAN {
    AGGREGATE( $$53:sequence(child("value",
        treat(<node*>, $$51))) )
    UNNEST( $$51:iterate($$49)
    NESTED-TUPLE-SOURCE
  }
  ASSIGN( $$49:treat(<item>, $$19) )
  SELECT( boolean($$48) )
  ASSIGN( $$48:and(boolean(data($$36)),
      boolean(data($$47))) )
  ASSIGN( $$47:value-eq(promote(<anyAtomicType?>,
      data($$45)), promote(<anyAtomicType?>,data($$46))) )
  ASSIGN( $$46:1999 )
  ASSIGN( $$45:year-from-dateTime(promote(<dateTime?>,
      data($$44))) )
  ASSIGN( $$44:cast(<dateTime?>, $$43) )
  ASSIGN( $$43:data(treat(<item*>, $$42)) )
  ASSIGN( $$42:sort-distinct-nodes-asc-or-atomics($$41) )
  SUBPLAN {
    AGGREGATE( $$41:sequence(child("date",
        treat(<node*>, $$39))) )
    UNNEST( $$39:iterate($$37) )
    NESTED-TUPLE-SOURCE
  }
  ASSIGN( $$37:treat(<item>, $$19) )
  ASSIGN( $$36:and(boolean(data($$27)),
      boolean(data($$35))) )
  ASSIGN( $$35:value-eq(promote(<anyAtomicType?>,
      data($$33)), promote(<anyAtomicType?>,data($$34))) )
  ASSIGN( $$34:"PRCP" )
  ASSIGN( $$33:sort-distinct-nodes-asc-or-atomics($$32))
  SUBPLAN {
    AGGREGATE( $$32:sequence(child("dataType",
        treat(<node*>, $$30))) )
    UNNEST( $$30:iterate($$28) )
    NESTED-TUPLE-SOURCE
  }
  ASSIGN( $$28:treat(<item>, $$19) )
  ASSIGN( $$27:value-eq(promote(<anyAtomicType?>,
      data($$25)),promote(<anyAtomicType?>,data($$26)) ) )
  ASSIGN( $$26:"GHCND:USW00014771" )
  ASSIGN( $$25:sort-distinct-nodes-asc-or-atomics($$24) )
  SUBPLAN {
    AGGREGATE( $$24:sequence(child("station",
        treat(<node*>, $$22))) )
    UNNEST( $$22:iterate($$20) )
    NESTED-TUPLE-SOURCE
  }
  ASSIGN( $$20:treat(<item>, $$19) )
  UNNEST( $$19:iterate($$18) )
  ASSIGN( $$18:sort-distinct-nodes-asc-or-atomics($$17) )
  SUBPLAN {
    AGGREGATE( $$17:sequence(child("data",
        treat(<node*>, $$15))) )
    UNNEST( $$15:iterate($$13) )
    ASSIGN( $$13:sort-distinct-nodes-asc-or-atomics($$12))
    NESTED-TUPLE-SOURCE
  }
  SUBPLAN {
    AGGREGATE( $$12:sequence(child("dataCollection",
        treat(<node*>, $$10))) )
    UNNEST( $$10:iterate($$8) )
    ASSIGN( $$8:sort-distinct-nodes-asc-or-atomics($$7) )
    NESTED-TUPLE-SOURCE
  }
  SUBPLAN {
    AGGREGATE( $$7:sequence(child("root",
        treat(<node*>, $$5))) )
    UNNEST( $$5:iterate($$3) )
    NESTED-TUPLE-SOURCE
  }
  ASSIGN( $$3:collection(promote(<string>, data($$2))) )
  ASSIGN( $$2:treat(<item>, $$1) )
  ASSIGN( $$1:"/sensors" )
  NESTED-TUPLE-SOURCE
}
EMPTY-TUPLE-SOURCE
```

After translating the plan, the Algebricks query optimization phase uses 22 Apache VXQuery rules and 30 generic Algebricks rules to create an optimized logical plan. Apache VXQuery rules apply basic XQuery data type rules and path step expression optimizations to streamline the plan. One of Algebricks' benefits to this plan is a more efficient way to calculate the aggregation. Using Algebricks logical operators, the built-in rules enable efficient two-step aggregation across the cluster. Further, the VXQuery query optimizer provides data specific details - enabling parallelization of the query plan. Apache VXQuery achieves parallel execution simply by providing XQuery function properties to Algebricks.

In the job generation phase, the VXQuery Metadata Interface implementation finds how the non-fragmented XML documents are partitioned throughout a cluster. Each machine has a unique set of XML documents residing in a directory specified by the query's fn:doc or fn:collection function. While AsterixDB has a native storage option, Apache VXQuery scans external XML documents for each query. The weather query uses the collection function to identify the "sensors" directory on all nodes. Using the plan and the metadata interface, Algrebricks creates the Hyracks job shown in Figure 5.

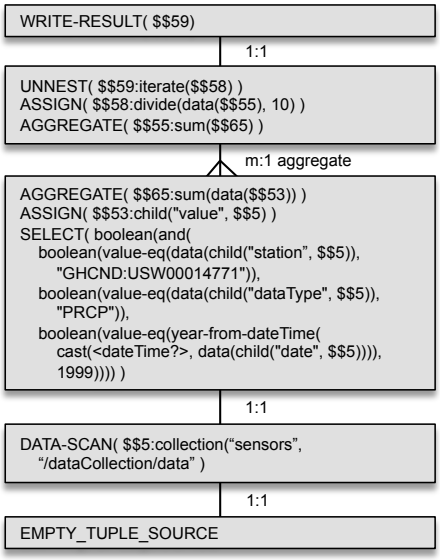

Figure 5. VXQuery Hyracks Job.

In the current implementation of VXQuery, we assume a collection of many (typically small) XML files. Nevertheless, Algebricks can also handle parallelization of queries over a single large document stored on a distributed storage system such as HDFS. We are currently implementing a parallel XML parser similar to [35] to enable these use cases.

5. Experimental Evaluation

In this section, we demonstrate experimentally the parallel efficiency of Hivesterix, AsterixDB, and VXQuery. In addition to showing proof of their existence, these experiments aim to expose the direct benefits of using Algebricks in achieving good parallelization characteristics. We report performance for a representative set of queries for each system for different sizes of data and

different numbers of nodes, with the goal of showing the scale-up and speed-up characteristics of each system. (It is not a goal of the paper to compare query performance across the different systems.)

5.1 Hivesterix

Cluster: We ran Hivesterix experiments on a 40-node cluster with a Gigabit Ethernet switch. Each node had a Quadcore Intel Xeon CPU E3-1230 V2 3.30GHz, 16GB of RAM, and 3TB RAID0 (3x1TB disks, linux software RAID). We compared Hivesterix and Hive-on-Hadoop (Hive-0.12.0). In the experiments, the RAID0 disk on each node is used for the HDFS data directory and the spilling workspace for query processing. **Data**: For speedup experiments, we used TPC-H 250× (250GB). For scaleup experiments, we used TPC-H 250×, 500×, 750×, 1000× (1TB) for 10, 20, 30, and 40 machines respectively. **Configuration**: We ran eight partitions per machine. **Queries**: We report three representative queries from the TPC-H benchmark, a filter and aggregate query, a group-by query, and a join with group-by query, shown below.

HQ1: Filter + Aggregate Query (TPC-H Q14)

```
select sum(l_extendedprice*l_discount) as revenue
from lineitem
where l_shipdate >= '1994-01-01'
  and l_shipdate < '1995-01-01'
  and l_discount >= 0.05 and l_discount <= 0.07
  and l_quantity < 24;
```

HQ2: Group-by Query (TPC-H Q1)

```
select l_returnflag, l_linestatus, sum(l_quantity),
  sum(l_extendedprice),
  sum(l_extendedprice*(1-l_discount)),
  sum(l_extendedprice*(1-l_discount)*(1+l_tax)),
  avg(l_quantity), avg(l_extendedprice), avg(l_discount),
  count(1)
from lineitem
where l_shipdate <= '1998-09-02'
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus;
```

HQ3: Join + Group-By Query (TPC-H Q9)

```
select nation, o_year, sum(amount) as sum_profit
from (
  select n_name as nation, year(o_orderdate) as o_year,
    l_extendedprice * (1 - l_discount) -
    ps_supplycost * l_quantity as amount
  from orders o join (
    select l_extendedprice, l_discount, l_quantity,
      l_orderkey, n_name, ps_supplycost
    from part p join (
      select l_extendedprice, l_discount, l_quantity,
        l_partkey, l_orderkey, n_name, ps_supplycost
      from partsupp ps join (
        select l_suppkey, l_extendedprice, l_discount,
          l_quantity, l_partkey, l_orderkey, n_name
        from (
          select s_suppkey, n_name
          from nation n join supplier s
            on n.n_nationkey = s.s_nationkey
        ) s1 join lineitem l on s1.s_suppkey = l.l_suppkey
      ) l1 on ps.ps_suppkey = l1.l_suppkey
        and ps.ps_partkey = l1.l_partkey
    ) l2 on p.p_name like '%green%'
      and p.p_partkey = l2.l_partkey
  ) l3 on o.o_orderkey = l3.l_orderkey
) profit
group by nation, o_year
order by nation, o_year desc;
```

As indicated by Figures 6 and 7, all queries show good speed-up and scale-up characteristics. All three benefit from scanning blocks of HDFS data in parallel. In HQ1 and HQ2, filtering is parallelized and aggregation is done in two phases, reducing the amount of data transferred across machines. HQ3 benefits from parallelizing joins across the cluster. We have also executed TPC-H using Hive-on-Hadoop (Hive-0.12.0) and a comparison with Hivesterix is shown in Figure 8. (An interesting future exercise might include the new generation of "SQL on Hadoop" systems.)

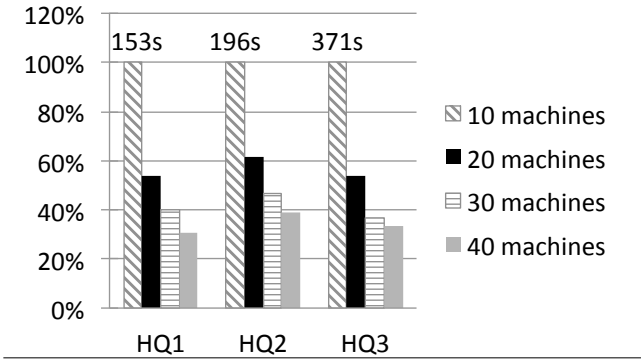

Figure 6. Hivesterix cluster speed up (percentage of 10 machines).

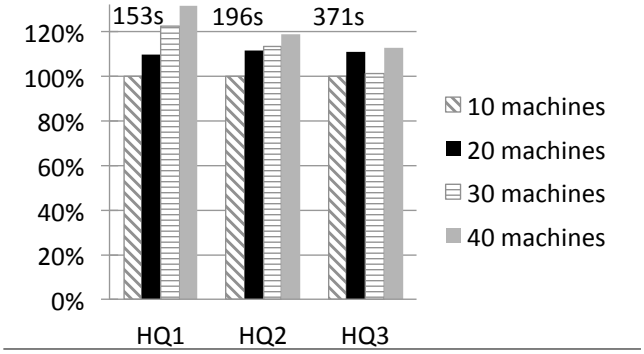

Figure 7. Hivesterix cluster scale up (percentage of 10 machines).

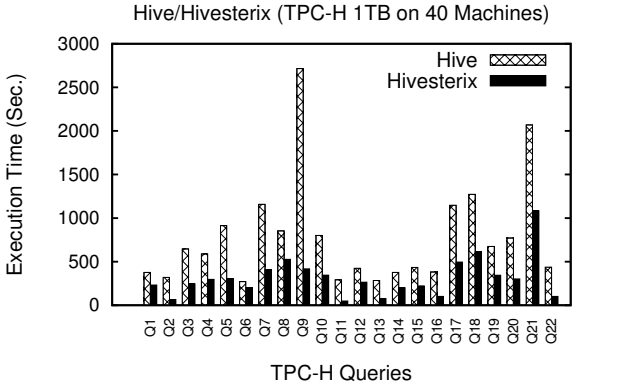

Figure 8. Hivesterix and Hive-on-Hadoop Comparison on TPC-H.

5.2 Apache AsterixDB

Cluster: We ran the reported experiments on a 10-node IBM x3650 cluster with a Gigabit Ethernet switch. Each node had one Intel Xeon processor E5520 2.26GHz with four cores, 12GB of RAM, and four 300GB, 10K RPM hard disks. On each machine 3 disks

were used for data. The other disk was used to store "system's data" (transaction logs and system logs). **Data**: We used a synthetic data generator to create records for the collections related to these tests ("GleambookUsers" etc.) For the speedup experiments, we generated 338GB of data, loaded on 9, 18 and 27 partitions. For the scaleup experiments, we generated 169GB, 338GB and 507GB of data for 9, 18 and 27 partitions respectively. A secondary index was constructed on the user_since field of the GleambookUsers dataset. **Configuration**: We ran three partitions on each machine. We assigned a maximum of 6GB of memory to each node controller. The buffercache size for each node controller was set to be 1GB. **Queries**: We executed four representative AQL queries, as follows.

AQ1: Filter Query

```
for $t in dataset GleambookMessages
where $t.author_id < 0
return $t
```

AQ2: Filter Query Using Indexes

```
for $user in dataset GleambookUsers
where $user.user_since >= datetime('2005-08-16T15:52:14')
and $user.user_since < datetime('2005-08-16T15:57:14')
return $user
```

AQ3: Aggregate Query

```
avg(
  for $t in dataset ChirpMessages
  where $t.send_time >= datetime('2008-11-15T19:42:51')
    and $t.send_time < datetime('2008-11-15T23:27:51')
  return string-length($t.message_text)
)
```

AQ4: Join Query

```
for $message in dataset GleambookMessages
for $user in dataset GleambookUsers
where $message.author_id = $user.id
  and $user.user_since >= datetime('2011-02-24T20:01:48')
  and $user.user_since < datetime('2011-02-25T05:01:48')
  and $message.send_time >=datetime('2008-10-24T14:21:21')
  and $message.send_time < datetime('2008-10-25T14:21:21')
return {
  "uname": $user.name,
  "message": $message.message
}
```

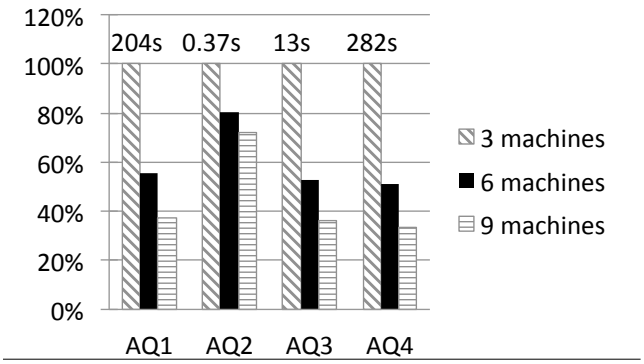

Figure 9. AsterixDB cluster speed up (percentage of 3 machines).

Figure 9 and Figure 10 depict the parallel speedup and scaleup of the four AQL queries. AQ1 and AQ3 benefit from the same rules

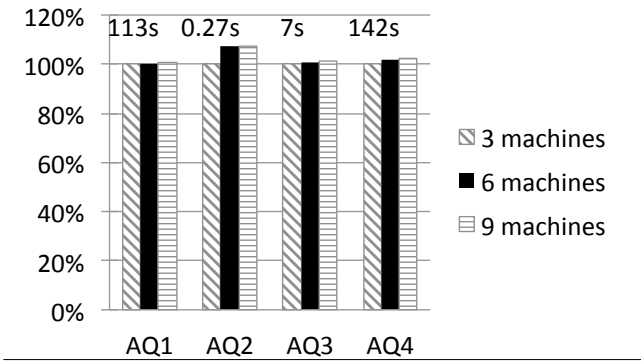

Figure 10. AsterixDB cluster scale up (percentage of 3 machines).

in Algebricks that helped HQ1 and HQ2 in Hivesterix. Scanning partitions and filtering is done in parallel on the different nodes of the cluster. In AQ3, the aggregation is performed in two phases to reduce network transfer. Join parallelism allows AQ4 to use the cluster effectively. AQ2 uses an index in AsterixDB to evaluate filters instead of scanning all the data. The index range scan is performed in parallel on the different partitions. Note that Algebricks includes facilities for utilizing indexes for query processing, but AsterixDB is currently the only system built on top of Algebricks that implements indexing at the storage level. Detailed AsterixDB performance characteristics can be found in [3].

5.3 Apache VXQuery

Cluster: Experiments were run on a cluster whose nodes have two Dual-Core AMD Opteron(tm) processor 2212 HE CPUs, 8GB of RAM, and two 1TB hard drives. **Data**: We used NOAA's Global Historical Climatology Network (GHCN)-Daily dataset that includes daily summaries of climate recordings (e.g., high and low temperatures, wind speed, rainfall). The complete XML data definition can be found on NOAA's site [1]. For the speed-up experiments, we used 57GB of weather XML data partitioned over the number of machines (varied from 1 to 8) for each run. The scale-up experiments were performed while keeping the amount of data per machine constant at 7.2GB and varying the number of machines from 1 to 8. **Configuration**: We ran four partitions per machine. **Queries**: We used the following three XQuery queries:

VQ1: Filter Query

Query VQ1 filters the data to show all readings that report an extreme wind warning. Such warnings occur when the wind speed exceeds 110 mph. (The wind measurement unit, tenths of a meter per second, has been converted to miles per hour.)

```
for $r in collection("sensors")/dataCollection/data
where $r/dataType eq "AWND"
  and xs:decimal(data($r/value)) gt 491.744
return $r
```

VQ2: Aggregate Query

Query VQ2 finds the annual precipitation for Syracuse, NY using the airport weather station (USW00014771) for 1999. The precipitation is reported in tenths of an inch.

```
sum(
  for $r in collection("sensors")/dataCollection/data
  where $r/station eq "GHCND:USW00014771"
    and $r/dataType eq "PRCP"
    and year-from-dateTime(xs:dateTime(data($r/date)))
        eq 1999
  return $r/value
) div 10
```

VQ3: Join + Aggregate Query

Query VQ3 finds the lowest recorded temperature (TMIN) in the United States for 2001. This query includes nested loops which are commonly used in XQuery. While XQuery does not have a join expression, these nested loops can be converted into a join operation using the Algebricks join operator. Converting a nested loop into a more efficient join algorithm provides the expected performance improvement for Apache VXQuery.

```
min(
  for $s in
      collection("stations")/stationCollection/station
  for $r in collection("sensors")/dataCollection/data
  where $s/id eq $r/station
    and (some $x in $s/locationLabels satisfies
        ($x/type eq "CNTRY" and $x/id eq "FIPS:US"))
    and $r/dataType eq "TMIN"
    and year-from-dateTime(xs:dateTime(data($r/date)))
        eq 2001
  return $r/value
) div 10
```

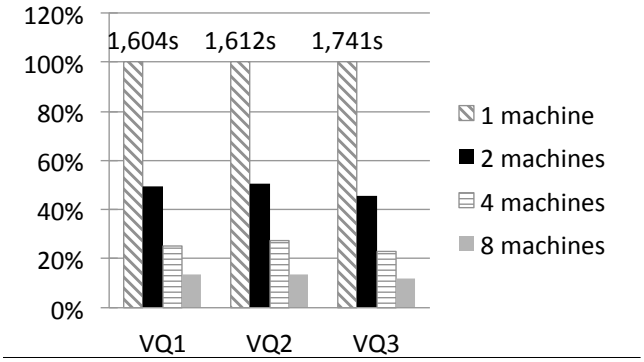

Figure 11. VXQuery cluster speed up (percentage of 1 machine).

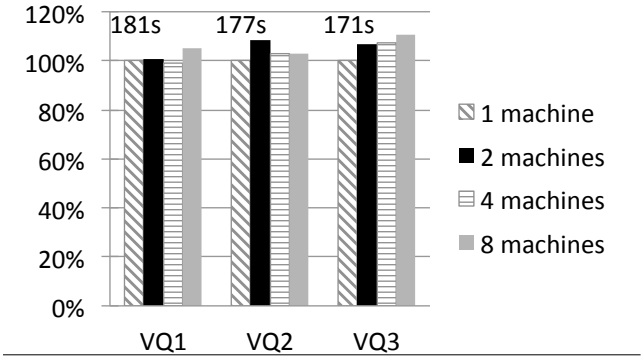

Figure 12. VXQuery cluster scale up (percentage of 1 machine).

Figure 11 and Figure 12 show the parallel speedup and scaleup of Apache VXQuery. The optimizations implemented in Algebricks help VQ1, VQ2, and VQ3 to achieve good parallel performance by parallelizing scans, filters, aggregations, and joins. More performance results for the current Apache VXQuery implementation on top of Algebricks can be found in [16].

6. Related Work

The early 1980s to the mid 1990s saw a flurry of parallel database research that resulted in research systems like Gamma [22], the Grace Database machine [23], and Bubba [12]. The Teradata [38] commercial parallel database system was built during that same time period. All of these systems were designed to evaluate SQL on the flat relational data model without the aim of extensibility on either the query language or data model fronts. Since these systems targeted relatively small clusters, fault-tolerance during query processing was also not a goal. However, in terms of parallel query evaluation, they exploited partitioned, pipelined, and independent parallel query evaluation techniques [20].

Genesis [10] proposed decomposing database systems into smaller reusable components so that a new custom database system could be assembled from the pieces. The design goals of Hyracks [13] and Algebricks are strongly motivated by reusability similar in spirit to Genesis but allow more flexibility in terms of the data model (as Genesis largely targeted relational style queries).

Integration of abstract data types into database systems [40] was a first step into achieving data model extensibility. EXODUS [15] was a system built in an era of object-oriented databases when there was heavy experimentation at the language and data model levels. The high-level goal of EXODUS was to provide a general configurable tool-set to build data management systems efficiently. In this regard, the goals of this work are similar to those of EXODUS. The EXODUS optimizer generator [27] was built as part of the EXODUS project to provide an extensible optimizer generator. System designers could use its extensibility to implement their own equivalence rules that the rewriter applied to enumerate query plans, much like the rewrite rule system in Algebricks. Volcano [25] and Cascades [26] took the work from EXODUS further by adding support for parallelism. Algebricks continues this line of research and addresses the challenges of the post-MapReduce world, where query plans are executed by highly scalable runtime engines.

Google's unveiling of MapReduce [18] led to a renewed interest in creating systems for evaluating programs in parallel on shared-nothing clusters. Microsoft's Dryad platform [30] allows developers to think of parallel programs as a DAG of vertices and edges. DryadLINQ [43] and the SCOPE runtime system [45], both implemented on top of Dryad, are functionally close to Algebricks. However, their extensibility is limited to .NET framework classes and the LINQ/SCOPE language, while Algebricks has been used to implement languages ranging from the SQL-like HiveQL to XQuery.

Nephele/PACTs [11] extends the MapReduce model with additional second-order operators (e.g. Cross, Match, CoGroup) while maintaining the rigidity of having a fixed set of operators known to the system. SparkSQL [7] and Spark DataFrames [5] share the common Catalyst optimizer[9] and its relational algebra underneath, but they are bound to a predefined data model. Apache Calcite[14], formerly Optiq, is a dynamic framework for parsing and planning queries now incubating at Apache; Calcite did not explore the query language dimension, focusing only on SQL, while Algebricks has been used for other languages like AQL and XQuery. Orca [39] is a new implementation of a top-down optimizer that exploits SMP parallelism to optimize SQL queries.

Another technique that was popular a few years ago for implementing new data models and query languages was to build a mapping from those models to the relational model and use an RDBMS to store the data in the form of tables. Queries in the new language are then translated into SQL to produce the correct output.

7. Conclusions and Future Work

We presented the design, implementation, use cases, and evaluation of Algebricks, a data model-agnostic query compiler backend built on the Hyracks parallel dataflow engine. We described how three query processing systems — Hivesterix, Apache AsterixDB, and Apache VXQuery — have been built using Algebricks with good scaling properties. While Algebricks is built on Hyracks, a similar architecture and methodology could be adopted by other Big Data

stacks, e.g., Spark [44], Stratosphere [11], or Tez [8]. We have made several open source releases of the Algebricks framework under the Hyracks repository (http://code.google.com/p/hyracks), and we invite other Big Data researchers to download and try the system. (It would be particularly interesting to see it explored for array- or graph-based languages.) In the future, we plan to add cost-based optimizations to Algebricks and to enhance the interactions between Algebricks and the Hyracks scheduler to support dynamic query re-optimization at execution time.

Acknowledgements

Algebricks work has been supported by a UC Discovery grant, NSF IIS awards 0910989 and 0910859, and NSF CNS awards 1305430 and 1305253. Industrial supporters include Amazon, eBay, Facebook, Google, HTC, Infosys, Microsoft, Oracle Labs, and Yahoo!.

References

[1] National Climate Data Center: Data Access. `http://www.ncdc.noaa.gov/data-access/`.

[2] XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition), 2010.

[3] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu *et al.* AsterixDB: A Scalable, Open Source BDMS. *PVLDB*, 7(14): 1905–1916, 2014.

[4] S. Alsubaiee, A. Behm, V. R. Borkar, Z. Heilbron, Y. Kim, M. Carey *et al.* Storage Management in AsterixDB. *PVLDB*, 7(10):841–852, 2014.

[5] Apache. The Distributed DataFrame Project. `http://ddf.io/`,.

[6] Apache. Hadoop: Open-Source Implementation of MapReduce. `http://hadoop.apache.org`,.

[7] Apache. The SparkSQL Project. `https://spark.apache.org/sql/`,.

[8] Apache. The Tez Project. `http://tez.apache.org/`,.

[9] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley *et al.* Spark SQL: Relational Data Processing in Spark. In *SIGMOD*, 2015.

[10] D. S. Batory. Genesis: a Project to Develop an Extensible Database Management System. In *OODS*, pages 207–208.

[11] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/PACTs: a Programming Model and Execution Framework for Web-Scale Analytical Processing. In *SoCC*, pages 119–130, 2010.

[12] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin *et al.* Prototyping Bubba, A Highly Parallel Database System. *IEEE Trans. Knowl. Data Eng.*, 2(1):4–24, Mar. 1990.

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

[14] calcite. Apache Calcite. `http://calcite.incubator.apache.org/`.

[15] M. J. Carey, D. J. DeWitt, D. Frank, G. Graefe, J. E. Richardson, E. J. Shekita *et al.* The Architecture of the EXODUS Extensible DBMS. In *OODS*, pages 231–256. 1991.

[16] E. P. Carman, Jr., T. Westmann, V. R. Borkar, M. J. Carey, and V. J. Tsotras. Apache VXQuery: A Scalable XQuery Implementation. *CoRR*, abs/1504.00331, 2015.

[17] D. D. Chamberlin. XQuery: A Query Language for XML. In *SIGMOD*, page 682, 2003.

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

[19] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The NEXT Framework for Logical XQuery Optimization. In *VLDB*, pages 168–179, 2004.

[20] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Commun. ACM*, 35:85–98, 1992.

[21] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Database Systems. In *SIGMOD*, pages 1–8, 1984.

[22] D. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *IEEE Trans. Knowl. Data Eng.*, 2(1):44–62, 1990.

[23] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An Overview of The System Software of a Parallel Relational Database Machine GRACE. In *VLDB*, pages 209–219, 1986.

[24] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *SIGMOD*, pages 102–111, 1990.

[25] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[26] G. Graefe. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[27] G. Graefe and D. DeWitt. The EXODUS Optimizer Generator. *SIGMOD Rec.*, 16(3):160–172, Dec. 1987.

[28] R. Grover and M. J. Carey. Data Ingestion in AsterixDB. In *EDBT*, pages 605–616, 2015.

[29] R. H. Güting, R. Zicari, and D. M. Choy. An Algebra for Structured Office Documents. *ACM Trans. Inf. Syst.*, 7(2):123–157, 1989.

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

[31] G. Jaeschke and H. Schek. Remarks on the Algebra of Non First Normal Form Relations. In *PODS*, pages 124–138, 1982.

[32] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A Tree Algebra for XML. In *DBPL*, pages 149–164, 2001.

[33] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi *et al.* Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*, 2015.

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

[35] Oracle. Using Oracle XQuery for Hadoop. `http://docs.oracle.com/cd/E49465_01/doc.23/e49333/oxh.htm#BDCUG526`.

[36] Y. Papakonstantinou, V. Borkar, M. Orgiyan, K. Stathatos, L. Suta, V. Vassalos *et al.* XML Queries and Algebra in the Enosys Integration Platform. *Data and Knowl. Eng.*, 44(3):299 – 322, 2003.

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

[38] J. Shemer and P. Neches. The Genesis of a Database Computer. *Computer*, 17(11):42 –56, Nov. 1984. ISSN 0018-9162.

[39] M. A. Soliman, L. Antova, V. Raghavan, A. El-Helw, Z. Gu, E. Shen *et al.* Orca: A Modular Query Optimizer Architecture for Big Data. In *SIGMOD*, pages 337–348, 2014.

[40] M. Stonebraker. Inclusion of New Types in Relational Data Base Systems. In *ICDE*, pages 262–269, 1986.

[41] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony *et al.* Hive: a Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009. ISSN 2150-8097.

[42] J. Wen, V. R. Borkar, M. J. Carey, and V. J. Tsotras. Revisiting Aggregation for Data Intensive Applications: A Performance Study. *CoRR*, abs/1311.0059, 2013.

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

[44] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley *et al.* Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, pages 15–28, 2012.

[45] J. Zhou, N. Bruno, M.-C. Wu, P.-Å. Larson, R. Chaiken, and D. Shakib. SCOPE: Parallel Databases Meet MapReduce. *VLDB J.*, 21(5):611–636, 2012.