Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing

Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, Rares Vernica

Computer Science Department, University of California, Irvine
Irvine, CA 92697
vborkar@ics.uci.edu

(Long Version of paper published in ICDE 2011)

Abstract—**Hyracks is a new partitioned-parallel software platform designed to run data-intensive computations on large shared-nothing clusters of computers. Hyracks allows users to express a computation as a DAG of data operators and connectors. Operators operate on partitions of input data and produce partitions of output data, while connectors repartition operators' outputs to make the newly produced partitions available at the consuming operators. We describe the Hyracks end user model, for authors of dataflow jobs, and the extension model for users who wish to augment Hyracks' built-in library with new operator and/or connector types. We also describe our initial Hyracks implementation. Since Hyracks is in roughly the same space as the open source Hadoop platform, we compare Hyracks with Hadoop experimentally for several different kinds of use cases. The initial results demonstrate that Hyracks has significant promise as a next-generation platform for data-intensive applications.**

I. Introduction

In recent years, the world has seen an explosion in the amount of data owing to the growth of the Internet. In the same time frame, the declining cost of hardware has made it possible for companies (even modest-sized companies) to set up sizeable clusters of independent computers to store and process this growing sea of data. Based on their experiences with web-scale data processing, Google proposed MapReduce [1], a programming model and an implementation that provides a simple interface for programmers to parallelize common data-intensive tasks. Shortly thereafter, Hadoop [2], an open-source implementation of MapReduce, was developed and began to gain followers. Similarly, Microsoft soon developed Dryad [3] as a generalized execution engine to support their coarse-grained data-parallel applications.

It has since been noted that, while MapReduce and Dryad are powerful programming models capable of expressing arbitrary data-intensive computations, it requires fairly sophisticated skills to translate end-user problems into jobs with Map and Reduce primitives for MapReduce or into networks of channels and vertices for Dryad. As a result, higher-level declarative languages such as Sawzall [4] from Google, Pig [5] from Yahoo!, Jaql [6] from IBM, Hive [7] from Facebook, and DryadLINQ [8] and Scope [9] from Microsoft have been developed to make data-intensive computing accessible to more programmers. The implementations of these languages translate declarative programs on partitioned data into DAGs of MapReduce jobs or into DAGs of Dryad vertices and channels.

Statistics reported by companies such as Yahoo! and Facebook regarding the sources of jobs on their clusters indicate that these declarative languages are gaining popularity as the interface of choice for large scale data processing. For example, Yahoo! recently reported that over 60 percent of their production Hadoop jobs originate from Pig programs today, while Facebook reports that a remarkable 95 percent of their production Hadoop jobs are now written in Hive rather than in the lower-level MapReduce model. In light of the rapid movement "up the food chain" to these higher-level languages, an obvious question emerges for the data-intensive computing community: If we had set out from the start to build a parallel platform to serve as a target for compiling higher-level declarative data-processing languages, what should that platform have looked like? It is our belief that the MapReduce model adds significant accidental complexity[1] as well as inefficiencies to the translation from higher-level languages.

In this paper, we present the design and implementation of Hyracks, which is our response to the aforementioned question. Hyracks is a flexible, extensible, partitioned-parallel framework designed to support efficient data-intensive computing on clusters of commodity computers. Key contributions of the work reported here include:

1) The provision of a new platform that draws on time-tested contributions in parallel databases regarding efficient parallel query processing, such as the use of operators with multiple inputs or the employment of pipelining as a means to move data between operators. Hyracks includes a built-in collection of operators that can be used to assemble data processing jobs without needing to write processing logic akin to Map and Reduce code.
2) The provision of a rich API that enables Hyracks oper-

[1] Accidental complexity is complexity that arises in computer systems which is non-essential to the problem being solved [10].

ator implementors to describe operators' behavioral and resource usage characteristics to the framework in order to enable better planning and runtime scheduling for jobs that utilize their operators.

3) The inclusion of a Hadoop compatibility layer that enables users to run existing Hadoop MapReduce jobs unchanged on Hyracks as an initial ”get acquainted” strategy as well as a migration strategy for ”legacy” data-intensive applications.
4) Performance experiments comparing Hadoop against the Hyracks Hadoop compatibility layer and against the native Hyracks model for several different types of jobs, thereby exploring the benefits of Hyracks over Hadoop owing to implementation choices and the benefits of relaxing the MapReduce model as a means of job specification.
5) An initial method for scheduling Hyracks tasks on a cluster that includes basic fault recovery (to guarantee job completion through restarts), and a brief performance study of one class of job on Hyracks and Hadoop under various failure rates to demonstrate the potential gains offered by a less pessimistic approach to fault handling.
6) The provision of Hyracks as an available open source platform that can be utilized by others in the data-intensive computing community as well.

The remainder of this paper is organized as follows. Section II provides a quick overview of Hyracks using a simple example query. Section III provides a more detailed look at the Hyracks programming model as seen by different classes of users, including end users, compilers for higher-level languages, and implementors of new operators for the Hyracks platform. Section IV discusses the implementation of Hyracks, including its approaches to job control, scheduling, fault handling, and efficient data handling. Section V presents a set of performance results comparing the initial implementation of Hyracks to Hadoop for several disparate types of jobs under both fault-free operation and in the presence of failures. Section VI reviews key features of Hyracks and their relationship to work in parallel databases and data-intensive computing. Finally, Section VII summarizes the paper and discusses our plans for future work.

II. Hyracks Overview

Hyracks is a partitioned-parallel dataflow execution platform that runs on shared-nothing clusters of computers. Large collections of data items are stored as local partitions distributed across the nodes of the cluster. A Hyracks job (the unit of work in Hyracks), submitted by a client, processes one or more collections of data to produce one or more output collections (also in the form of partitions). Hyracks provides a programming model and an accompanying infrastructure to efficiently divide computations on large data collections (spanning multiple machines) into computations that work on each partition of the data separately. In this section we utilize a small example to describe the steps involved in Hyracks job execution. We also provide an introductory architectural overview of the Hyracks software platform.

A. Example

As will be explained more completely in section III, a Hyracks job is a dataflow DAG composed of operators (nodes) and connectors (edges). Operators represent the job's partitioned-parallel computation steps, and connectors represent the (re-) distribution of data from step to step. Internally, an individual operator consists of one or several activities (internal sub-steps or phases). At runtime, each activity of an operator is realized as a set of (identical) tasks that are clones of the activity and that operate on individual partitions of the data flowing through the activity.

Let us examine a simple example based on a computation over files containing CUSTOMER and ORDERS data drawn from the TPC-H [11] dataset. In particular, let us examine a Hyracks job to compute the total number of orders placed by customers in various market segments. The formats of the two input files for this Hyracks job are of the form:

CUSTOMER (C_CUSTKEY, C_MKTSEGMENT, ...)
ORDERS (O_ORDERKEY, O_CUSTKEY, ...)

where the dots stand for remaining attributes that are not directly relevant to our computation.

To be more precise about the intended computation, the goal for the example job is to compute the equivalent of the following SQL query:

```
select C_MKTSEGMENT, count(O_ORDERKEY)
from CUSTOMER join ORDERS on C_CUSTKEY = O_CUSTKEY
group by C_MKTSEGMENT
```

A simple Hyracks job specification to perform this computation can be constructed as shown in Figure 1. For each data source (CUSTOMER and ORDERS), a file scanner operator is used to read the source data. A hash-based join operator receives the resulting streams of data (one with CUSTOMER instances and another with ORDERS instances) and produces a stream of CUSTOMER-ORDERS pairs that match on the specified condition (C_CUSTKEY = O_CUSTKEY). The result of the join is then aggregated using a hash-based group operator on the value of the C_MKTSEGMENT field. This group operator is provided with a COUNT aggregation function to compute the count of O_ORDERKEY occurrences within a group. Finally, the output of the aggregation is written out to a file using a file writer operator.

As noted earlier, data collections in Hyracks are stored as local partitions on different nodes in the cluster. To allow a file scanner operator to access the partitions, its runtime tasks must be scheduled on the machines containing the partitions. As indicated in the figure, metadata provided to the two file scanner operators specifies the machines where the partitions of the CUSTOMER and ORDERS files are available. For the sake of this toy example, the CUSTOMER file has two partitions that reside on nodes NC1 and NC2, respectively. The ORDERS file also has two partitions, but each partition is two-way replicated; one can be accessed on either of nodes NC3 or NC2, and the other on NC1 or NC5. (This information about replication gives Hyracks multiple scheduling options for the ORDERS file scanner's tasks.)

Moving downstream, our example Hyracks job specification computes the join in a partitioned manner as well. To enable this, of course, the data arriving at the join tasks (join partitions) must satisfy the property that CUSTOMER and ORDERS instances that match will be routed to the same join task. In the job specification, every edge between two operators carries an annotation indicating the data distribution logic to be used for routing data from the source (producer) partitions to the target (consumer) partitions. The example job in Figure 1 uses hash-partitioning of the CUSTOMER and ORDERS instances on C_CUSTKEY and O_CUSTKEY, respectively, to enforce the required data property for the join operator's partitions. Hash-partitioning is then used again to repartition the output partitions from the join operator to the grouping operator on the C_MKTSEGMENT field to ensure that all CUSTOMER-ORDERS pairs that agree on that field will be routed to the same partition for aggregation. The final result of the job will be created as a set of partitions (with as many partitions as grouping operators being used) by using a 1:1 connector between the group operator and the file writer operator; the 1:1 connector will lead Hyracks to create as many consumer tasks as there are producer tasks and route the results pairwise without repartitioning.

When Hyracks begins to execute a job, it takes the job specification and internally expands each operator into its constituent activities. This results in a more detailed DAG, as shown in Figure 2. This expansion of operators reveals to Hyracks the phases of each operator along with any sequencing dependencies among them. The hash-based join operator in the example expands into two activities; its first activity builds a hashtable on one input and the second activity then probes the resulting hashtable with the other input in order to produce the result of the join. Note that the build phase of the join operation has to finish before the probe phase can begin; this sequencing constraint is captured as a dotted arrow (a blocking edge) in the figure from the join-build activity to the join-probe activity. Similarly, the hash-based grouping operator in our example expands into an aggregation activity that blocks an output generation activity to model the fact that no output can be produced by the aggregator until it has seen all of its input data. The reason for having operators express their internal phases (activities) in this manner is to provide Hyracks with enough insight into the sequencing dependencies of the various parts of a job to facilitate execution planning and coordination. Activities that are transitively connected to other activities in a job only through dataflow edges are said to together form a stage. Intuitively, a stage is a set of activities that can be co-scheduled (to run in a pipelined manner, for example). Section IV describes more details about this process.

The planning of the details of a job's parallel execution is done in the order in which stages become ready to execute. (A given stage in a Hyracks job is ready to execute when all of its dependencies, if any, have successfully completed execution.) Figure 3 shows the runtime task graph that results from planning the activities in a stage. While the figure depicts the tasks for all three stages of the job, Hyracks actually expands each stage into tasks just prior to the stage's execution. At runtime, the three stages in our example will be executed in order of their readiness to run. Hyracks will start by running the scanning of CUSTOMER data along with the hash-build part of the join tasks, pipelining (and routing) the CUSTOMER data between the parallel tasks of these initial activities. Once the first stage is complete, the next stage, which probes the hashtable using the ORDERS data and performs aggregation, will be planned and executed. After completion of the second stage, the output generation tasks of the group-by operator along with the file writer tasks will be activated and executed, thus producing the final results of the job. The job's execution is then complete.

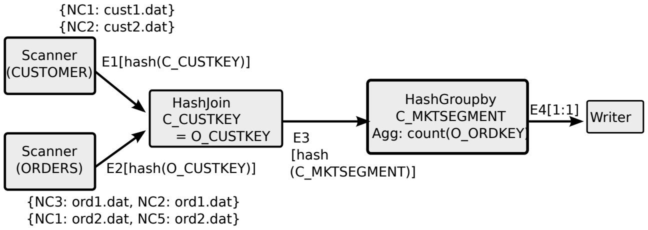


Fig. 1: Example Hyracks job specification

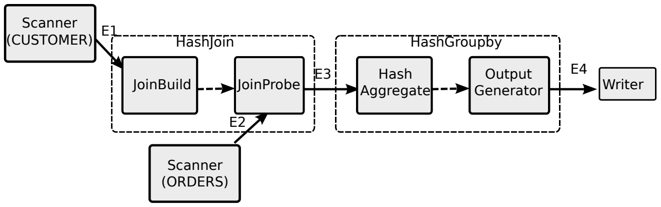


Fig. 2: Example Hyracks Activity Node graph

B. High-level Architecture

Figure 4 provides an overview of the basic architecture of a Hyracks installation. Every Hyracks cluster is managed by a Cluster Controller process. The Cluster Controller accepts job execution requests from clients, plans their evaluation strategies (e.g., computing stages), and then schedules the jobs' tasks (stage by stage) to run on selected machines in the cluster. In addition, it is responsible for monitoring the state of the cluster to keep track of the resource loads at the various worker machines. The Cluster Controller is also responsible for re-planning and re-executing some or all of the tasks of a job in the event of a failure. Turning to the task execution side, each worker machine that participates in a Hyracks cluster runs a Node Controller process. The Node Controller accepts task execution requests from the Cluster Controller and also reports on its health (e.g., resource usage levels) via a heartbeat mechanism. More details regarding the controller architecture and its implementation are provided in section IV.

III. Computational Model

In this section, we describe the Hyracks approach to data representation and the Hyracks programming model as they

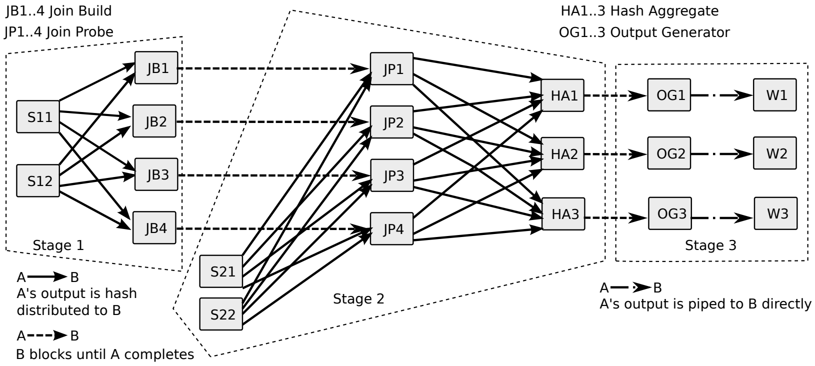


Fig. 3: Parallel instantiation of the example

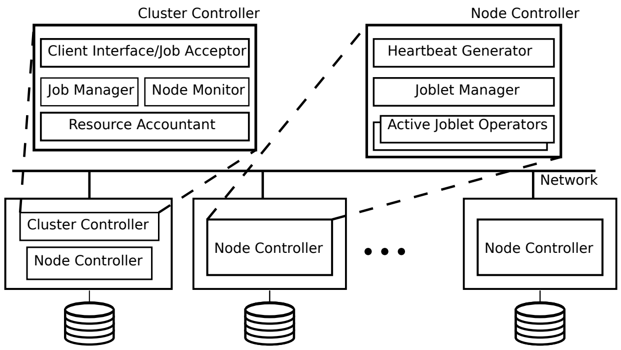


Fig. 4: Hyracks system architecture

pertain to two broad classes of users: end users who use Hyracks as a Job assembly layer to solve problems, and operator implementors who wish to implement new operators for use by end users. (Compilers for higher-level languages essentially fall into the first class of users as well.) We also describe the built-in operators and connectors that are available as part of the initial software distribution of Hyracks.

A. *Data Treatment in Hyracks*

In Hyracks, data flows between operators over connectors in the form of records that can have an arbitrary number of fields. Hyracks provides support for expressing data-type-specific operations such as comparisons and hash functions. The type of each field is described by providing an implementation of a descriptor interface that allows Hyracks to perform serialization and deserialization. For most basic types, e.g., numbers and text, the Hyracks library contains pre-existing type descriptors. The Hyracks use of a record as the carrier of data is a generalization of the (key, value) concept found in MapReduce and Hadoop. The advantage of the generalization is that operators do not have to artificially package (and repackage) multiple data objects into a single "key" or "value" object. The use of multiple fields for sorting or hashing also becomes natural in this model. For example, the Hyracks operator library includes an external sort operator descriptor that can be parameterized with the fields to use for sorting along with the comparison functions to use. Type descriptors in Hyracks are similar to the Writable and WritableComparator interfaces in Hadoop. However, a subtle but important difference is that Hyracks does not require the object instances that flow between operators to themselves implement a specific interface; this allows Hyracks to directly process data that is produced and/or consumed by systems with no knowledge of the Hyracks platform or its interfaces.

B. *End User Model*

Hyracks has been designed with the goal of being a runtime platform where users can hand-create jobs, like in Hadoop and MapReduce, yet at the same time to be an efficient target for the compilers of higher-level programming languages such as Pig, Hive, or Jaql. In fact, the Hyracks effort was born out of the need to create an appropriate runtime system for the ASTERIX project at UCI [12], a project in which we are building a scalable information management system with support for the storage, querying, and analysis of very large collections of semi-structured nested data objects using a new declarative query language (AQL). We also have an effort currently underway to support Hive on top of the Hyracks native end user model. Figure 5 gives an overview of the current user models.

1) Hyracks Native User Model: A Hyracks job is formally expressed as a DAG of operator descriptors connected to one another by connector descriptors, as was indicated in Figure 1. In addition to its input and output connectors, an operator descriptor may take other parameters specific to its operation. For example, the ExternalSortOperatorDescriptor in the Hyracks built-in operator library needs to know which fields in its input record type are to be used for sorting, the comparators to use to perform the sorting operation, and the amount of main memory budgeted for its sorting work.

Hyracks currently allows users to choose between two ways of describing the scheduling choices for each of the operators in a job. One option is to list the exact number of partitions of an operator to be created at runtime along with a list of choices of worker machines for each partition. Another option is just to specify the number of partitions to use for an operator, leaving Hyracks to decide the location assignments for each of its partitions. We are currently working on a third, longer-term option involving automatic partitioning and placement based on the estimated resource requirements for operators and the current availability of resources at the worker machines.

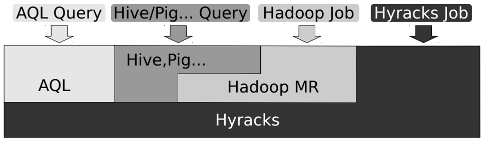


Fig. 5: End User Models of Hyracks

2) Hyracks Map-Reduce User Model: MapReduce has become a popular paradigm for data-intensive parallel computation using shared-nothing clusters. Classic example applications for the MapReduce paradigm have included processing and indexing crawled Web documents, analyzing Web request logs, and so on. In the open source community, Hadoop is a popular implementation of the MapReduce paradigm that is quickly gaining traction even in more traditional enterprise IT settings. To facilitate easy migration for Hadoop users who wish to use Hyracks, we have built a Hadoop compatibility layer on top of Hyracks. The aim of the Hyracks Hadoop compatibility layer is to accept Hadoop jobs unmodified and run them on a Hyracks cluster.

In MapReduce and Hadoop, data is initially partitioned across the nodes of a cluster and stored in a distributed file system (DFS). Data is represented as `(key, value)` pairs. The desired computation is expressed using two functions:

```
map    (k1,v1)       → list(k2,v2);
reduce (k2,list(v2)) → list(k3,v3).
```

A MapReduce computation starts with a map phase in which the `map` functions are applied in parallel on different partitions of the input data. The `(key, value)` pairs output by each `map` function are then hash-partitioned on their `key`. For each partition, the pairs are sorted by their `key` and sent across the cluster in a shuffle phase. At each receiving node, all of the received partitions are merged in sorted order by their `key`. All of the pair values sharing a given key are then passed to a single reduce call. The output of each `reduce` function is finally written to a distributed file in the DFS.

To allow users to run MapReduce jobs developed for Hadoop on Hyracks, we developed two extra operator descriptors that can wrap the `map` and `reduce` functionality.[2] The data tuples provided as input to the *hadoop_mapper* operator descriptor are treated as `(key, value)` pairs and, one at a time, passed to the `map` function. The *hadoop_reducer* operator descriptor also treats the input as `(key, value)` pairs, and groups the pairs by `key`. The sets of `(key, value)` pairs that share the same `key` are passed, one set at a time, to the `reduce` function. The outputs of the `map` and `reduce` functions are directly output by the operator descriptors. To emulate a MapReduce framework we employ a sort operator descriptor and a hash-based distributing connector descriptor.

Figure 6 shows the Hyracks plan for running a MapReduce job. After data is read by the scan operator descriptor, it is fed into the *hadoop_mapper* operator descriptor using a 1:1 edge. Next, using an M:N hash-based distribution edge, data is partitioned based on `key`. (The "M" value represents the number of maps, while the "N" value represents the number of reducers.) After distribution, data is sorted using the sort operator descriptor and passed to the *hadoop_reducer* operator descriptor using a 1:1 edge. Finally, the *hadoop_reducer* operator descriptor is connected using a 1:1 edge to a file-writer operator descriptor.

C. Operator Implementor Model

One of the initial design goals of Hyracks has been for it to be an extensible runtime platform. To this end, Hyracks includes a rich API for operator implementors to use when building new operators. The implementations of operators made available "out of the box" via the Hyracks operator library are also based on the use of this same API.

Operators in Hyracks have a three-part specification:

1) *Operator Descriptors*: Every operator is constructed as an implementation of the Operator Descriptor interface. Using an operator in a job involves creating an instance of the descriptor of that operator. The descriptor is responsible for accepting and verifying parameters required for correct operation from the user during job assembly. During the job planning phase, the descriptor is responsible for expanding itself into its constituent activities (as illustrated back in Figure 2).
2) *Operator Activities*: Hyracks allows an operator to describe, at a high level, the various phases involved in its evaluation. Phases in an operator usually create sequencing dependencies that describe the order in which the inputs of the operator are required to be made available and the order in which the outputs will become available. Again, Figure 2 showed the expansions for the hash-join and the hash-based grouping operators into their component activities. As a part of the expansion, activities indicate any blocking dependencies and the mapping of overall operator inputs and outputs to the activities' inputs and outputs. For example, when the hash-join operator is expanded, it indicates to Hyracks that its join-build activity consumes one input and its join-probe activity consumes the other. It also indicates that its join-probe activity produces the output. At a high level, this describes to Hyracks that the second input is not required to be available until the stage containing the join-build activity is complete. Conversely, it also indicates that output from the join will not be available until the stage containing the join-build activity is complete. Hyracks then uses this information to plan the stages of a job in order of necessity. Deferring planning to the moment just prior to execution allows Hyracks to use current cluster

[2] The `combine` functionality of MapReduce is fully supported as well, but those details are omitted here for ease of exposition.

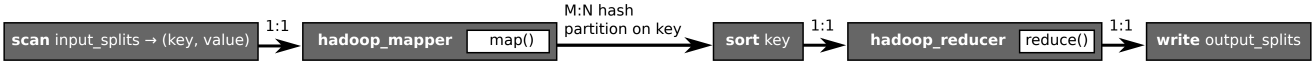


Fig. 6: Hyracks plan for Hadoop jobs

conditions (availability of machines, resource availability at the various machines, etc.) to help determine the schedule for a stage's tasks.

3) *Operator Tasks*: Each activity of an operator actually represents a *set* of parallel tasks to be scheduled on the machines in the cluster. Tasks are responsible for the runtime aspect of the activity – each one consumes a partition of the activity's inputs and produces a partition of its output. The tasks corresponding to activities of the same partition of the same operator may need to share state. For example, the join-probe task needs to be given a handle to the hashtable constructed by the join-build task. To facilitate such state sharing, Hyracks co-locates those tasks that belong to a given partition of an activity of the same operator. For example, in Figure 3, the join-probe tasks are co-located with their corresponding join-build tasks. Note that the join-build task will no longer be active when the join-probe task is scheduled, but Hyracks provides a shared context for the two tasks to use to exchange the required information. In the future we plan to explore alternative state transfer mechanisms (albeit at a higher cost), e.g., to use when co-location is impossible due to resource constraint violations at a machine. In terms of operator implementation, the implementor of a new operator can implement the runtime behavior of its activities by providing single-threaded implementations of an iterator interface (a la [13]), much as in Gamma [14].

D. Hyracks Library

Hyracks strongly promotes the construction of reusable operators and connectors that end users can then just use to assemble their jobs. Hyracks, as part of its software distribution, comes with a library of pre-existing operators and connectors. Here we briefly describe some of the more important operators and connectors.

1) Operators:

- *File Readers/Writers*:

– *Line File Reader/Writer*: Reads/writes files from/to the local file system, treating a file as a sequence of lines.

– *Delimited File Reader/Writer*: Reads/writes files from/to the local file system; the files contain records separated by record delimiters, and the records contain fields separated by field delimiters.

– *HDFS File Reader/Writer*: Reads/writes files from/to the Hadoop Distributed File system.

- *Mappers*:

– *Native mapper*: Used to evaluate a function once for each item in the input. Mappers can be used to implement traditional operators such as projection and selection.

– *Hadoop Mapper*: Used to call the Mapper interface in the Hadoop MapReduce library. It is used by the compatibility layer to invoke Hadoop mappers.

- *Sorters*:

– *In-memory Sorter*: Uses main memory to sort its input.

– *External Sorter*: Tries to use a main memory allocation to sort its input data, but when the input does not fit into memory, it creates and merges runs on disk.

- *Joiners*:

– *In-memory Hash Joiner*: Performs an equijoin by using memory to build a hash-table on one input and then probing the table with the other input.

– *Hybrid Hash Joiner*: Performs an equijoin using the hybrid hash-join algorithm [13].

– *Grace Hash Joiner*: Performs an equijoin using the Grace hash-join algorithm [13].

- *Aggregators*:

– *Hash-based Aggregator*: Groups input data using an in-memory hash table. A custom aggregator function is invoked for each group of the input data.

– *Preclustered Aggregator*: Groups input data using very little state assuming it is already clustered on the grouping fields.

2) Connectors: Connectors distribute data produced by a set of *sender* operators to a set of *receiver* operators.

- *M:N Hash-Partitioner:* Hashes every tuple produced by senders (on a specified set of fields) to generate the receiver number to which the tuple is sent. Tuples produced by the same sender keep their initial order on the receiver side.
- *M:N Hash-Partitioning Merger:* Takes as input sorted streams of data and hashes each tuple to find the receiver. On the receiver side, it merges streams coming from different senders based on a given comparator, thus producing ordered partitions.
- *M:N Range-Partitioner:* Takes the position k of the partitioning field and a set of disjoint n ranges, each range associated with one of the n receiver operators. (The field at position k must always contain integer values.) Every tuple produced by a sender is distributed to the receiver associated with the range in which the value of the k-th field lies.
- *M:N Replicator:* Copies the data produced by every sender to every receiver operator.
- *1:1 Connector:* Connects exactly one sender to one receiver operator.

IV. Hyracks Implementation details

In this section we describe some of the details of the Hyracks implementation. We first describe the control portion of Hyracks, which maintains cluster membership and performs job scheduling, tracking, and recovery. We then discuss some of the salient implementation details of modules that help Hyracks to run its operators and connectors efficiently.

A. Control Layer

As was shown in Figure 4, a Cluster Controller (CC) is used to manage a Hyracks cluster. Worker machines are added to a Hyracks cluster by starting a Node Controller (NC) process on that machine and providing it with the network address of the Cluster Controller process whose cluster it is to join. The NC then initiates contact with the CC and requests registration. As part of its registration request, the NC sends details about the resource capabilities (number of cores, etc.) of its worker machine. On successful registration, the CC replies to the NC with the rate at which the NC should provide heartbeats back to the CC to maintain continued membership in the cluster.

On receiving a job submission from a client, the CC infers the stages of the job (as was illustrated in Figures 2 and 3). Scheduling of a stage that is ready to execute is performed using the latest information available about the cluster. (This may happen much later than when the job was submitted, e.g., in a long multi-stage job.) Once it has determined the worker nodes that will participate in the execution of the stage, the CC initiates task activation on those worker nodes in parallel by pushing a message to start the tasks. This is different from the way the Hadoop Job Tracker disseminates tasks to the Task Trackers – Hadoop uses a pull model where the Task Trackers pull tasks from the Job Tracker by means of a periodic heartbeat. Although eagerly pushing job start messages increases the number of messages on the cluster, we have adopted this approach to minimize the wait time for tasks to start execution. This becomes increasingly important as a platform tries to target short jobs. In the future, we will explore ways of minimizing the network traffic for control messages by exploiting opportunities to piggyback multiple messages targeting the same NC.

The control logic of the Hyracks Cluster Controller is implemented in Overlog [15] using the JOL [16] library. Overlog is an extension of Datalog that has support for events. Integration with JOL was convenient since both JOL and Hyracks are implemented in Java. The JOL implementation of Overlog models data as tables of records with a declared schema. The fields in the tables can be arbitrary Java objects. JOL allows Java code to register callbacks that are invoked by the Overlog interpreter when modifications are made to the tables. JOL also allows insertion and deletion of records in a table from Java. At a very high level, the Java code in the CC is used to translate events from the outside world (clients submitting jobs and NC registrations/heartbeats) into inserts and deletes on JOL tables. This action triggers the JOL interpreter to execute the JOL program which results in modifications to one or more tables that have Java callbacks attached. When invoked with data that is inserted into or deleted from the tables by JOL, these callbacks apply the desired effects in the outside world (e.g., by sending start/abort task messages to the NCs). Figure 7 shows the role of JOL and Java code as a controller framework.

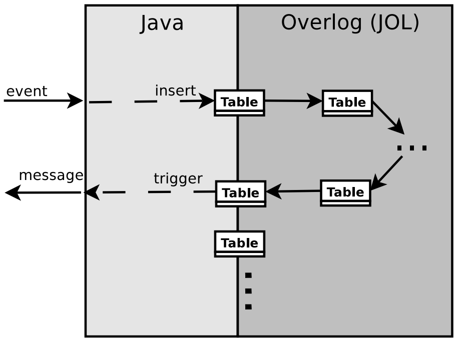


Fig. 7: Mixed Java/JOL controller framework

Table I shows a few of the JOL tables that are relevant to identifying the stages of a Hyracks job. When a job is submitted, the CC translates the job specification into insertions of records into the *operatordescriptor* (one record per Operator Descriptor) and *connectordescriptor* (one record per Connector Descriptor) tables. It then visits every Operator Descriptor in the job specification and inserts a record into the *activitynode* table for every activity of each Operator Descriptor. As part of the expansion of the Operator Descriptor into activities, a mapping from the Operator Descriptor inputs (and outputs) to the Operator Task inputs (and outputs) is provided along with blocking dependencies between activities. The input/output mapping is captured in the *activityconnection* table while the blocking information is captured in the *activityblocked* table. Once the CC has inserted records into these tables, it invokes the JOL runtime to evaluate the Overlog program. Figure 8 shows the part of the Overlog program used by the Cluster Controller to infer stages of activities and populate the *activitystage_rec* and *activitystage* tables. The *activitystage_rec* table is populated using rules 1 thru 4 recursively and the final result of stage assignments to activities is recorded in the *activitystage* table. The rules specified in Overlog can be summarized as follows:

1) Rule 1: Initially assume all activities can be evaluated in the earliest stage by inserting an assignment of stage 0 to every activity in the *activitynode* table.
2) Rule 2: If an activity A1 must block on activity A2, and A1 is assigned a stage earlier than or the same as A2, insert a stage assignment for A1 that is one greater than the stage assigned to A2.
3) Rules 3 and 4: If activity A1 is connected to activity A2 through a Connector Descriptor, and A1 and A2 belong to different stages, insert a stage assignment for A1 that is the later of A1's and A2's stages. This rule occurs twice, once for A1 and once for A2, because the *connectordescriptor* table captures a one-way connection whereas for the stage assignment we concern ourselves with the existence of a connection. Intuitively, if we wish to co-schedule (for pipelining) the activities connected by a data flow connector, they will both need to be assigned to the latter of the stages to which they belong.

4) Rule 5: Finally, the actual stage number associated with an activity is the maximum over all of the stage number assignments and is captured in the *activitystage* table.

The scheduling and fault-recovery logic is also implemented in Overlog using the general principles described above, but not explained here due to space constraints.

B. Task Execution Layer

Once a stage of a Hyracks job becomes ready to run, the CC activates the stage on the set of NCs that have been chosen to run the stage and then waits until the stage completes or until an NC failure is detected. In the remainder of this section we explore the task activation process and then discuss some details of the implementation of the data-processing and data-movement aspects of Tasks and Connectors, respectively.

1) Task Activation: When a stage becomes ready to run, the Cluster Controller sends messages to the Node Controllers participating in that stage's execution to perform task activation in three steps.

1) *Receiver Activation*: The CC initiates a ReceiverActivation request to every NC that participates in a stage's execution. Each NC, on receipt of this request, creates the Task objects that have been designated to run at that NC (as determined by the scheduler). For each Task that accepts inputs from other Tasks, it creates an endpoint that has a unique network address. The mapping from a Task object's input to the endpoint address is relayed back to the CC as the response to this request.
2) *Sender-Receiver Pairing*: Once the CC receives the ReceiverActivation responses containing local endpoint addresses from the NCs, it merges all of them together to create a global endpoint address map. This global map is now broadcast to each NC so that the send side of each Task object at that NC knows the network addresses of its consumer Tasks.
3) *Connection Initiation*: Once pairing has completed on all NCs, the CC informs all of the NCs that the Tasks can be started.

2) Task Execution and Data Movement: The unit of data that is consumed and produced by Hyracks tasks is called a Frame. A frame is a fixed-size (configurable) chunk of contiguous bytes. A producer of data packs a frame with a sequence of records in a serialized format and sends it to the consumer who then interprets the records. Frames never contain partial records; in other words, a record is never split across frames. A naive way to implement tasks that process records is to deserialize them into Java objects and then process them. However, this has the potential to create many objects, resulting in more data copies (to create the object representation) and a burden on the garbage collector (later) to reclaim the space for those objects. To avoid this problem, Hyracks provides interfaces for comparing and hashing fields that can be implemented to work off of the binary data in a frame. Hyracks includes implementations of these interfaces for the basic Java types (String, integer, float, etc.). Hyracks also provides Task implementors with helper functions to perform basic tasks on the binary data such as to copy complete records, copy specific fields, and concatenate records from source frames to target frames.

A task's runtime logic is implemented as a push-based iterator that receives a frame at a time from its inputs and pushes result frames to its consumers. Note that the task-runtime only implements the logic to process streams of frames belonging to one partition without regards to any repartitioning of the inputs or outputs that might be required. Any repartitioning is achieved by using a Connector. The connector runtime has two sides, the send-side and the receive-side. There are as many send-side instances of a connector as tasks in the producing activity, while there are as many receive-side instances as tasks in the consuming activity. The send-side instances are co-located with the corresponding producer tasks, while the receive-side instances are co-located with the consuming tasks. When a send-side instance of a connector receives a frame from its producing task, it applies its redistribution logic to move records to the relevant receive-side instances of the connector. For example, the M:N hash-partitioning connector in Hyracks redistributes records based on their hash-value. The send-side instances of this connector inspect each record in the received frame, compute the hash-value, and copy it to the target frame meant for the appropriate receive-side instance. When a target frame is full, the send-side instance sends the frame to the receive-side instance. If the send-side and receive-side instances of a connector are on different worker machines (which is most often the case), sending a frame involves writing it out to the network layer. The network layer in Hyracks is responsible for making sure that the frame gets to its destination.

In order to use network buffers efficiently on the receive-side of a connector, Hyracks allows the connector implementation to specify the buffering strategy for received frames. Figure 9 shows two buffering strategies that could be chosen by a connector. If the receive-side logic of a connector does not differentiate between the frames sent by the various send-side instances, it can choose to use one network buffer for all senders. On the other hand, if the connector's logic requires treating frames from different senders separately, it can use one network buffer per send-side instance. For example, the M:N hash-partitioning connector does not differentiate frames based on their senders. However, an M:N sort-merge connector, which assumes that frames sent by a sender contain records that are pre-sorted, needs to decide which sender's frame to read next in order to perform an effective merge.

V. Experimental Results

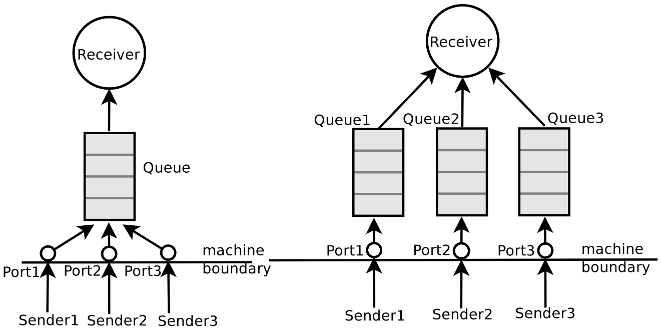
In this section we report the results of performance comparisons of Hadoop and Hyracks as data-intensive computing platforms. First, we compare the performance of jobs written using the MapReduce paradigm by running them on Hadoop and running the same jobs unchanged on Hyracks using its Hadoop compatibility layer. Next, we show the difference in performance that results from not being limited to the MapRe-

TABLE I: Tables identifying stages of a Hyracks job.

Table Name	Table Attributes
operatordescriptor	JobId, OperatorDescriptorId, OperatorDescriptor
connectordescriptor	JobId, ConnectorDescriptorId, SourceOperatorId, SourcePort, TargetOperatorId, TargetPort, ConnectorDescriptor
activitynode	JobId, OperatorDescriptorId, ActivityId, Activity
activityconnection	JobId, OperatorDescriptorId, OperatorPort, Direction, ActivityId, ActivityPort
activitystage	JobId, OperatorDescriptorId, ActivityId, StageNumber
activityblocked	JobId, BlockingOperatorDescriptorId, BlockingActivityId, BlockedOperatorDescriptorId, BlockedActivityId

$$\begin{aligned}
&\mathit{activitystage\_rec}(\mathit{JobId}, \mathit{OpId}, \mathit{ActivityId}, 0) :- \mathit{activitynode}(\mathit{JobId}, \mathit{OpId}, \mathit{ActivityId}, \_); && (1)\\
&\mathit{activitystage\_rec}(\mathit{JobId}, \mathit{OpId2}, \mathit{ActivityId2}, \mathit{Stage}) :- \mathit{activitystage\_rec}(\mathit{JobId}, \mathit{OpId1}, \mathit{ActivityId1}, \mathit{Stage1}), && (2)\\
&\qquad \mathit{activitystage\_rec}(\mathit{JobId}, \mathit{OpId2}, \mathit{ActivityId2}, \mathit{Stage2}),\\
&\qquad \mathit{activityblocked}(\mathit{JobId}, \mathit{OpId1}, \mathit{ActivityId1}, \mathit{OpId2}, \mathit{ActivityId2}),\\
&\qquad \mathit{Stage2} \leq \mathit{Stage1}\ \ \{\ \ \mathit{Stage} := \mathit{Stage1} + 1;\ \ \};\\
&\mathit{activitystage\_rec}(\mathit{JobId}, \mathit{OpId2}, \mathit{ActivityId2}, \mathit{Stage}) :- \mathit{activitystage\_rec}(\mathit{JobId}, \mathit{OpId1}, \mathit{ActivityId1}, \mathit{Stage1}), && (3)\\
&\qquad \mathit{activitystage\_rec}(\mathit{JobId}, \mathit{OpId2}, \mathit{ActivityId2}, \mathit{Stage2}),\\
&\qquad \mathit{activityconnection}(\mathit{JobId}, \mathit{OpId1}, \mathit{Op1Port}, \mathit{Dir.OUT}, \mathit{ActivityId1}, \_),\\
&\qquad \mathit{activityconnection}(\mathit{JobId}, \mathit{OpId2}, \mathit{Op2Port}, \mathit{Dir.IN}, \mathit{ActivityId2}, \_),\\
&\qquad \mathit{connectordescriptor}(\mathit{JobId}, \_, , \mathit{OpId1}, \mathit{Op1Port}, \mathit{OpId2}, \mathit{Op2Port}, \_),\\
&\qquad \mathit{Stage1} \neq \mathit{Stage2}\ \ \{\ \ \mathit{Stage} := \mathit{java.lang.Math.max}(\mathit{Stage1}, \mathit{Stage2});\ \ \};\\
&\mathit{activitystage\_rec}(\mathit{JobId}, \mathit{OpId1}, \mathit{ActivityId1}, \mathit{Stage}) :- \mathit{activitystage\_rec}(\mathit{JobId}, \mathit{OpId1}, \mathit{ActivityId1}, \mathit{Stage1}), && (4)\\
&\qquad \mathit{activitystage\_rec}(\mathit{JobId}, \mathit{OpId2}, \mathit{ActivityId2}, \mathit{Stage2}),\\
&\qquad \mathit{activityconnection}(\mathit{JobId}, \mathit{OpId1}, \mathit{Op1Port}, \mathit{Dir.OUT}, \mathit{ActivityId1}, \_),\\
&\qquad \mathit{activityconnection}(\mathit{JobId}, \mathit{OpId2}, \mathit{Op2Port}, \mathit{Dir.IN}, \mathit{ActivityId2}, \_),\\
&\qquad \mathit{connectordescriptor}(\mathit{JobId}, \_, \mathit{OpId1}, \mathit{Op1Port}, \mathit{OpId2}, \mathit{Op2Port}, \_),\\
&\qquad \mathit{Stage1} \neq \mathit{Stage2}\ \ \{\ \ \mathit{Stage} := \mathit{java.lang.Math.max}(\mathit{Stage1}, \mathit{Stage2});\ \ \};\\
&\mathit{activitystage}(\mathit{JobId}, \mathit{OpId}, \mathit{ActivityId}, \mathit{max} < \mathit{Stage} >) :- \mathit{activitystage\_rec}(\mathit{JobId}, \mathit{OpId}, \mathit{ActivityId}, \mathit{Stage}); && (5)
\end{aligned}$$

Fig. 8: Fragment of Overlog program for inferring stages of activities.



Shared buffer.

One buffer per sender.

Fig. 9: Buffering strategies for connectors.

duce model in Hyracks. Finally, we explore the performance of Hadoop and Hyracks when node failures occur by using a simple model for generating failures.

For all of the experiments shown in this section, we used a cluster of ten IBM machines each with a 4-core Xeon 2.27 GHz CPU, 12GB of main memory, and four locally attached 10,000 rpm SATA drives. The same cluster was used for Hyracks as well as Hadoop. The Hadoop version used was 0.20.1.

A. MapReduce on Hyracks

In this section, we compare the running times of three very different kinds of jobs (K-Means clustering, a TPC-H style query, and Set-Similarity Joins) on Hadoop versus on Hyracks using the compatibility layer.

1) K-Means: K-Means is a well known clustering algorithm that tries to find clusters given a set of points. Mahout [17] is a popular open source project that includes a Hadoop implementation of the K-Means algorithm as MapReduce jobs. The clustering problem consists of grouping a set of n points into k clusters $\{C_i\}_{i=1,k}$ with centers $\{\mu_i\}_{i=1,k}$, such that each point is assigned to the cluster with the closest center. The K-Means MapReduce implementation [18] starts with a randomized algorithm (called canopy computation) to heuristically determine the number of clusters that the K-Means algorithm must try to find. The canopy MapReduce job finds k points to serve as initial centroids of the k clusters that will then be refined by subsequent iterations of the K-Means algorithm. Each iteration of the K-Means algorithm is also implemented as a MapReduce job. The Map phase scans over the set of data points in the input and finds their closest centroid μ_j among the k centroids computed in the previous step. Once the closest centroid is found, that input point is deemed to belong to cluster j. The output of the Map phase is regrouped by the cluster assignment of each point and routed to the Reducers. The Reduce phase computes a new cluster centroid for each of the k clusters by averaging the coordinates of all points assigned to that cluster.

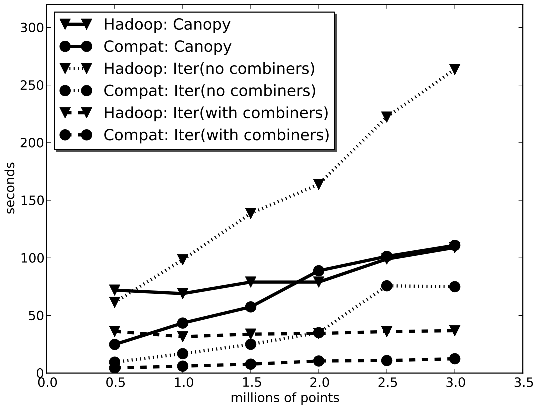


Fig. 10: K-Means on Hadoop and Hyracks

In Figure 10 we show separately the time taken by the canopy computation phase and by the iterations for K-Means on both systems. We further explore the efficiency tradeoffs of Hyracks vs. Hadoop as a data-platform by presenting the average running-time of the iterations both with and without combiners in the two systems. Turning off combiners for the iterative part of the K-Means algorithm increases dramatically the amount of data that is moved from the Map phase to the Reduce phase (although it does not change the final result).

From Figure 10 we see that the canopy computation phase benefits the least from running on Hyracks since it is mostly CPU intensive. This phase initially (for 0.5 and 1 million points) shows faster time to completion on Hyracks. However, as the size of the data grows from 1.5 to 3 million points, the two lines essentially converge. This is because Hyracks runs the same CPU-intensive user functions that Hadoop runs, and hence there is not much room for improvement there. On the other hand, when much of a job's time is spent in re-organizing data by partitioning and sorting, running the MapReduce job on Hyracks performs better. The K-Means iteration results clearly show this behavior. This improvement owes itself to the more efficient nature of the data-movement layer that forms the foundation for the Hyracks operators and connectors.

In Hadoop, moving data from the mappers to the reducers is performed rather inefficiently. All data from the mappers is written to disk before it is moved to the reducer side. When a reducer is notified of a mapper's completion, it initiates a pull of its partition using HTTP. The resultant near-simultaneous fetch requests from all reducers to the mappers leads to disk contention on the Map side as the reducers try to read all of the requested partitions from disk; there is also a surge in network traffic created by such simultaneous requests [19].

In Hyracks, data is instead pushed from the producers to the consumers in the form of fixed sized chunks (on the order of 32KB) incrementally, as they are produced. This leads to a much smoother utilization of the network throughout the processing time of the producer. Also, since the producer decides when a block is sent out, the aforementioned disk contention issue does not arise in Hyracks.

2) Set-Similarity Joins: Set-Similarity Joins match data items from two collections that are deemed to be similar to each other according to a similarity function. Work reported in [20] performed an extensive evaluation of algorithms for computing such joins using the MapReduce framework. From that work we took the best MapReduce algorithm to compare its performance on Hadoop vs. on Hyracks using the Hadoop compatibility layer. This "fuzzy join" computation requires three stages of MapReduce jobs, as shown in Figures 11, 12, and 13. The first stage tokenizes the input records and produces a list of tokens ordered by frequency using two MapReduce jobs. The second stage (one MapReduce job) produces record-id pairs of similar records by reading the original data and using the tokens produced in the first stage as auxiliary data. The final stage of SSJ computes similar record pairs from the record-id pairs from Stage 2 and the original input data using two MapReduce jobs. We refer the reader to [20] for more details about the algorithm.

For comparing the performance of Set-Similarity Joins (SSJ), we used the algorithm shown in the figures to find publications with similar titles in the DBLP dataset [21] by performing a join of the DBLP data with itself. In order to test performance for different sizes of data, we increased the DBLP data sizes to 5x, 10x, and 25x over the original dataset (which contains approximately 1.2M entries) using the replication scheme of [20]. Table II shows the running times for the three stages of the SSJ algorithm on the three different sizes of data. The row labeled "Hadoop" represents the time taken to run each stage as a Hadoop job; "Compat" shows the time taken to run the same MapReduce jobs using the compatibility layer on Hyracks (Please ignore the row labeled "Native" for now; it will be explained shortly).

TABLE II: Size-up for Parallel Set-Similarity Joins (10 nodes)

		Stage 1	Stage 2	Stage 3	Total time
5x	Hadoop	75.52	103.23	76.59	255.34
	Compat	24.50	38.92	23.21	86.63
	Native	11.41	32.69	9.33	53.43
10x	Hadoop	90.52	191.16	107.14	388.82
	Compat	42.15	128.86	38.47	209.49
	Native	16.14	81.70	14.87	112.71
25x	Hadoop	139.58	606.39	266.26	1012.23
	Compat	105.98	538.74	60.62	705.34
	Native	34.50	329.30	27.88	391.68

In the column labeled "Stage 1" in Table II, we see that the MapReduce jobs running on Hadoop scale smoothly with increasing data sizes. The Hadoop compatibility layer on Hyracks also shows this trend. However, the job running on Hyracks is faster because it benefits from two improvements: low job startup overhead and more efficient data movement by the infrastructure. Hyracks uses push-based job activation, which introduces very little wait time between the submission of a job by a client and when it begins running on the node

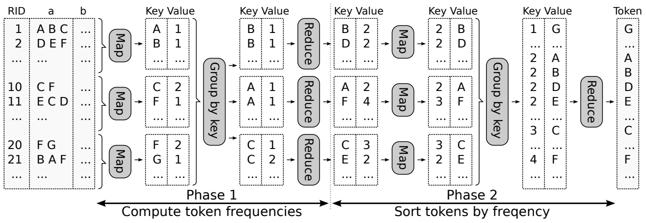


Fig. 11: Example data flow of Stage 1 of Set-Similarity Joins. (Basic Token ordering for a self-join on attribute “a”.)

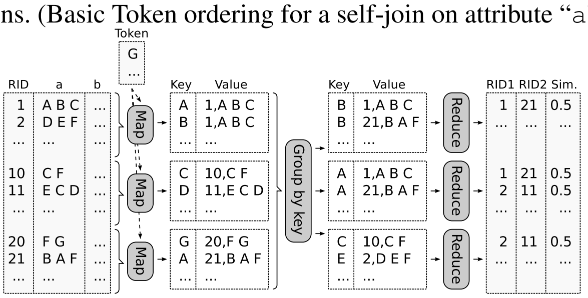


Fig. 12: Example data flow of Stage 2 of Set-Similarity Joins. (Basic Kernel for a self-join on attribute “a”.)

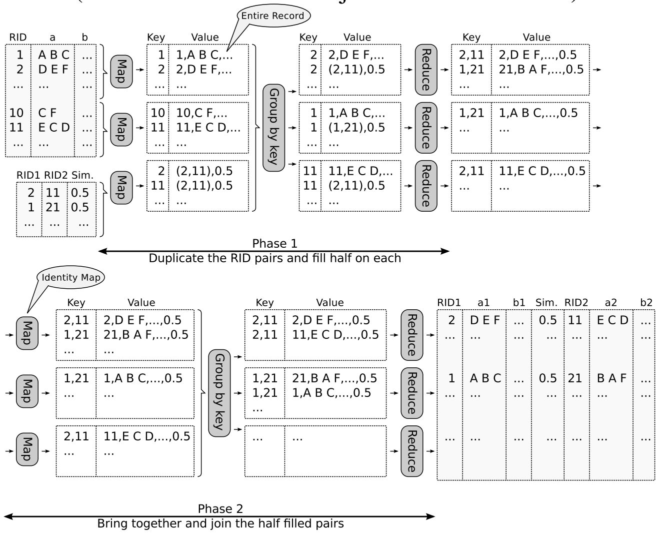

Entire Record
RID a b
RID1 RID2 Sim.
Key Value
Map
Group by key
Reduce
Phase 1
Duplicate the RID pairs and fill half on each
Identity Map
RID1 a1 b1 Sim. RID2 a2 b2
Phase 2
Bring together and join the half filled pairs

Fig. 13: Example data flow of Stage 3 of Set-Similarity Joins using Basic Record Join for a self-join case. “a1” and “a2” correspond to the original attribute “a”, while “b1” and “b2” correspond to attribute “b”.

controllers.

In the column labeled “Stage 2” in Table II, we observe a near constant difference between the running times of Hadoop and Hyracks. This is because most of the time in this stage is spent in the Reducer code that uses a quadratic algorithm to find matching records. We do not describe the details here due to space limitations; we refer the reader to the original paper on this algorithm in [20]. The key thing to note is that most of the work in this stage is done in the Reduce phase in user-defined code, so the only improvement we can expect is due to lower overhead of the infrastructure.

The column labeled “Stage 3” in Table II, shows that at 5x data size, the ratio of the Hadoop completion time to the Hyracks compatibility layer completion time is 3.3; the ratio drops to about 2.8 at the 10x data size. As the amount of data grows to 25x, we see the compatibility layer completing about 4 times faster than Hadoop. This is because Stage 3 performs two “joins” of the record-ids with the original data to produce similar-record-pairs from the similar-record-id-pairs that were produced by the previous stage. Every join step reshuffles all of the original input data from the Map to the Reduce phase, but the output of the join is fairly small. Most of the work is thus performed in moving the data (as the Map and Reduce code is fairly simple). At smaller data sizes, Hadoop times are dominated by startup overhead, while for larger data sizes the dominating factor is data redistribution.

3) TPC-H: The TPC-H query is the query from the Hyracks running example that was introduced in Section II-A. Figure 14 shows the structure of the two MapReduce jobs corresponding to the running example from Figure 1.

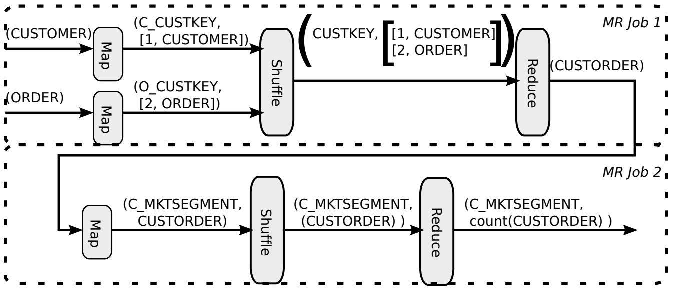

MR Job 1
(CUSTOMER)
Map
(C_CUSTKEY, [1, CUSTOMER])
(ORDER)
Map
(O_CUSTKEY, [2, ORDER])
Shuffle
(CUSTKEY, [[1, CUSTOMER] [2, ORDER]])
Reduce
(CUSTORDER)
MR Job 2
Map
(C_MKTSEGMENT, CUSTORDER)
Shuffle
(C_MKTSEGMENT, (CUSTORDER))
Reduce
(C_MKTSEGMENT, count(CUSTORDER))

Fig. 14: Hadoop implementation of TPC-H query from Fig. 1.

The first MapReduce job in Figure 14 performs the join using the tagged-join method described in [22], while the second job computes the aggregation. The Mapper of the first job generates the value field as a tagged output that indicates whether the record is from the CUSTOMER source or the ORDER source. The associated key is generated by extracting the C_CUSTKEY field if it is a customer record or the O_CUSTKEY if it is an ORDER record. The Reducer of the join job then receives all records (both CUSTOMER and ORDER) that match on the join key. The Reducer function then generates the CUSTOMER-ORDER pairs to be sent to the output. The data generated from this first MapReduce job is used as the input for the second MapReduce job, which performs the aggregation step by grouping the data on the value of the C_MKTSEGMENT field and then computing the desired count. The Mapper of this job produces the value of C_MKTSEGMENT as the key and the CUSTOMER-ORDER pair as the value. The Reducer thus receives a list of CUSTOMER-ORDER pairs for the same value of C_MKTSEGMENT that it then counts. The final query output is emitted as a pair of C_MKTSEGMENT and count values.

As with the other tasks, we measured the end-to-end response time of the jobs by running them on Hadoop and comparing that with the time it took to run the same MapReduce jobs on Hyracks using the compatibility layer. In addition to the standard configuration of running the two MapReduce jobs sequentially, with HDFS used to store the intermediate join result, we also executed the compatibility layer with "pipelining" between jobs turned on. In this case the join job pipelines data to the aggregation MapReduce job without the use of HDFS.

As we see from Figure 15, TPC-H on Hadoop takes about 5.5 times the time taken by the compatibility layer on Hyracks for smaller data sizes (scale 5). Job start-up overhead in Hadoop is responsible for this slowness. As data sizes grow, we observe that the running times for Hadoop and the compatibility layer grow super-linearly. This is not surprising since there are two sort steps involved, one in the first MapReduce job to perform the join of CUSTOMER and ORDER data, and the second to perform aggregation in the second MapReduce job. The result of the join is quite large, and hence a substantial amount of time is spent in partitioning and sorting. To measure the time spent in writing the join results to HDFS and reading it back in, we also ran in compatibility mode with pipelining turned on. We observe from Figure 15 that I/O to and from HDFS adds significant overhead. At scale 40, the compatibility layer takes about 133 seconds (about 25% longer than the pipelined execution). Note however that all of the MapReduce running times exhibit a super-linear curve owing to the sort operations (We will ignore the "Hyracks Native" curve for now).

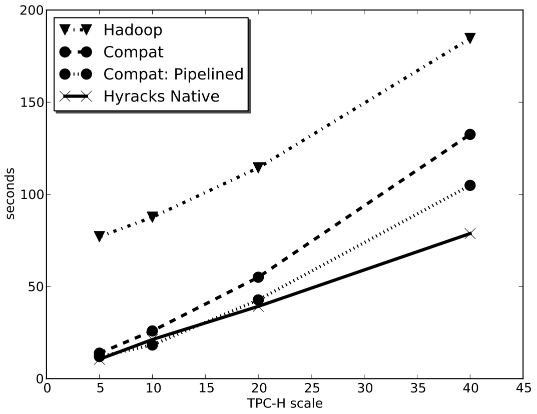


Fig. 15: TPC-H query from Fig. 1 with Hadoop and Hyracks.

B. Beyond MapReduce

We now show the results of running two experiments to study the performance improvements obtainable in Hyracks by not being restricted to the MapReduce programming model. We present results for Set-Similarity Joins and the TPC-H join/aggregate example query.

1) Set-Similarity Joins (Take 2): Figures 16, 17, and 18 show the native Hyracks job specifications to implement the three stages of the Hadoop based Set-Similarity Join examined in the previous subsection.

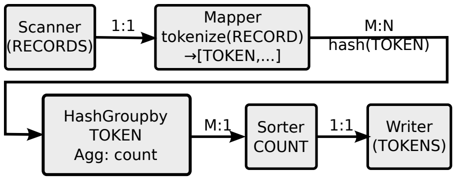


Fig. 16: Hyracks-native plan for Stage 1 of Set-Similarity Self-Joins.

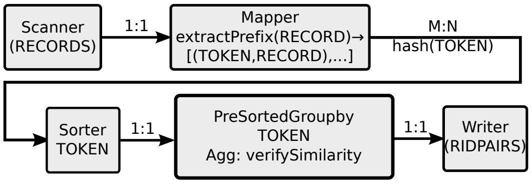

Scanner (RECORDS)
1:1
Mapper extractPrefix(RECORD)→ [(TOKEN,RECORD),...]
M:N hash(TOKEN)
Sorter TOKEN
1:1
PreSortedGroupby TOKEN Agg: verifySimilarity
1:1
Writer (RIDPAIRS)

Fig. 17: Hyracks-native plan for Stage 2 of Set-Similarity Self-Joins.

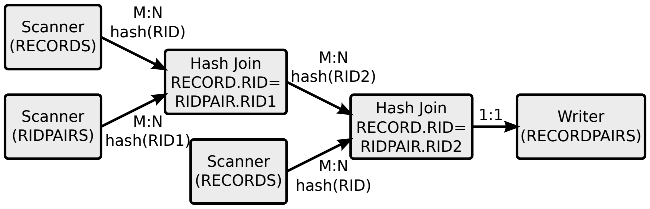


Fig. 18: Hyracks-native plan for Stage 3 of Set-Similarity Self-Joins.

In Table II, the row labeled "Native" shows the time required by the Hyracks jobs to perform Set-Similarity Joins. We can now compare these times to those that we examined previously. These faster Native times are due in part to the savings already discussed, but here we also see additional algorithmic benefits. Stage 1 benefits the most from the Native formulation due to the use of hash based grouping, which does not need to sort the input data like in Hadoop. Stage 3, in the Native case, benefits from using hash join operators that perform the joins by building a hash-table on one of their inputs and then probing the table using the other input. This strategy is cheaper than first sorting both inputs on the join key to bring the matching values from both the sides together (which is what happens in Hadoop). In Stage 2, we used sort-based grouping since it performed slightly better than its hash-based counterpart. Usually, hash-based aggregation works well for traditional aggregates when grouping is performed on relatively low cardinality attributes, as compression is achieved by the aggregation. In the case of Stage 2, however, the grouping operation is used to separate the input into groups without applying an incremental aggregation function. The sort operator in Hyracks implements the "Poor man's normalized key" optimization mentioned in [23] which can be exploited when building Hyracks Native jobs. This optimization helps in terms of providing cheaper comparisons and better cache locality. After sorting, the "aggregation" function performs a quadratic scan of the items within each group. This is seen in Table II as a super-linear growth in the completion time for

Stage 2.

In summary, we see that the Native formulation of Set-Similarity Joins run about 2.6 to 4.7 times faster than their MapReduce formulation running on Hadoop and about 1.6 to 1.8 times faster than running the same MapReduce jobs on Hyracks using its compatibility layer.

2) TPC-H (Take 2): We now compare the two MapReduce TPC-H implementations with the native Hyracks implementation. The "Hyracks Native" line in Figure 15 shows the results of natively running the TPC-H query in Figure 1. Notice that for large data the TPC-H query executes on Hyracks much faster than the MapReduce counterparts on Hadoop or on Hyracks via the compatibility layer. As in the native expression of Set-Similarity Joins, this is due in part to the fact that the TPC-H query uses hash-joins and hash-based aggregation. In addition, Hyracks does not need to materialize intermediate results between the joiner and the aggregator. In contrast, since Hadoop accepts one MapReduce job at a time, it needs to persist the output of the Join job for this example in HDFS. To minimize the "damage" caused by this materialization on the Hadoop completion time, we set the replication factor in HDFS to 1 (meaning no replication). We observe that the Hyracks job scales linearly with growing data size since all operations in the pipeline have linear computational complexity for the memory and data sizes considered here.

To conclude, Hyracks is a more efficient alternative to Hadoop for MapReduce jobs, and even greater performance benefits can be achieved by not being restricted to the MapReduce programming model. Here we have compared the performance characteristics of Hadoop and Hyracks in detail for K-Means clustering, Set-Similarity joins, and an example TPC-H query. We have measured even greater benefits for simpler jobs. For example, the standard Hadoop Word-Count MapReduce job on 12 GB of data ran approximately twice as fast on Hyracks using the compatibility layer than on Hadoop, and a Native Word-Count implementation on Hyracks ran approximately 16 times as fast as the Hadoop MapReduce version.

C. Fault Tolerance Study

A system designed to run on a large number of commodity computers must be capable of detecting and reacting to possible faults that might occur during its regular operation. In this part of the paper, we perform a very simple experiment to illustrate the fact that, while being able to run jobs through to the end despite failures is valuable, doing so by being pessimistic and ever-prepared for incremental forward-recovery is not a general "right" answer. Hadoop uses a naive strategy based on storing all intermediate results to durable storage before making progress. While such a naive strategy is a safe first approach, we wish to begin exploring the path of applying more selective fault-tolerance. Hyracks is in a better position to exploit operator and connector properties to achieve the same degree of fault tolerance while doing less work along the way. As our first step in this direction, the Hyracks fault-tolerance strategy that we use for the experiment in this section is to simply restart all jobs impacted by failures. Although such a strategy cannot be the only one in a system like Hyracks, we believe that this is actually an effective strategy for smaller to medium-sized jobs.

We used the TPC-H example query shown in Figure 1 on data scale 40 for this experiment. In our experimental set-up, we had the client submit the same job request over and over again in an infinite loop without any think time and we measured the observed completion time for every request. Separately, we killed a randomly chosen cluster node at regular intervals. In different runs of the experiment, we changed the mean time between failures from 100 seconds to 1600 seconds. The same experiment was performed with the jobs running on Hyracks and then with the corresponding jobs running on Hadoop. Figure 19 shows the observed request completion time (retry times included) for the two systems for different MTBF (mean time between failure) values. The data inputs to the jobs in both systems were replicated so that a node failure could still allow progress of the job.

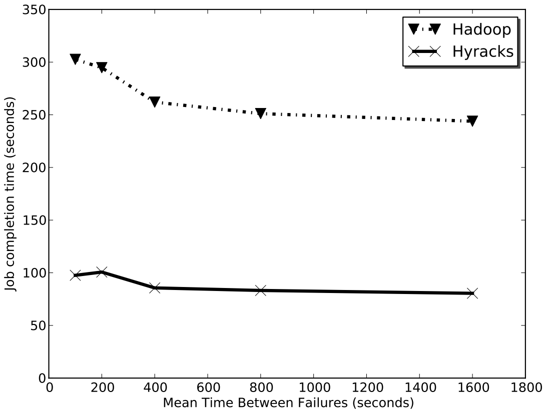


Fig. 19: TPC-H query from Fig. 1 in the presence of failures.

In Hadoop in order to provide as much isolation from failures as possible, every map task writes its output to the local disk before its data is made available to the reducer side. On the reducer side, every partition fetched from every map task is written to local disk before a merge is performed. The goal of the extra I/O in Hadoop is to make sure that at most one task needs to be restarted when a failure occurs. In the current implementation of Hyracks, however, data is pipelined from producers to consumers, so a single failure of any one node requires a restart of all of the operators that are currently part of the same pipelined stage.

We observe from Figure 19 that Hyracks runs the TPC-H job to completion faster even when failures are injected into the cluster very frequently. Note also that, since a given Hyracks job runs to completion much faster than the "fault-tolerant" Hadoop job, a given Hyracks job is less likely to be hit by a failure during its execution in the first place.

As we mentioned earlier, a strategy that requires a complete restart cannot be the only fault recovery strategy in a system.

If the mean time between failures is less than the mean time to complete a job, then such a job may never complete. Work is currently underway in Hyracks to limit the part of a job that needs to be restarted while not losing the performance benefits of pipelining.

VI. Related Work

Hyracks builds on the bodies of prior work in two areas: parallel database systems and data-intensive computing. As much of Hyracks' relationship to other work has already been mentioned, we review related work only briefly here.

Early parallel database systems such as Gamma [14], Teradata [24], and GRACE [25] applied partitioned-parallel processing to data management, particularly query processing, over two decades ago. In [26], DeWitt and Gray explained how shared-nothing database systems can achieve very high scalability through the use of partitioned and pipelined parallelism, and many successful commercial products today are based on these principles. Hyracks essentially ports the same principles to the world of data-intensive computing. Hyracks' operator/connector division of labor was motivated in particular by the Gamma project's architecture; Gamma's operators implemented single-threaded data processing logic, and its connections (split-tables) dealt with repartitioning data. The modeling of operator trees by Garofalakis and Ioannidis [27] motivated the expansion of Hyracks operators into activities in order to extensibly capture blocking dependencies.

The introduction of Google's MapReduce system [1] has led to the recent flurry of work in data-intensive computing. In contrast to MapReduce, which offers a simple but rigid framework to execute jobs expressed in specific ways (i.e., using mappers and reducers), Microsoft proposed Dryad [3], a generalized computing platform based on arbitrary computational vertices and communication edges. Dryad is a much lower-level platform than MapReduce; e.g., we could have built Hyracks on top of Dryad if it had been available under an open source license. Hadoop [2], of course, is a popular open source implementation of the MapReduce platform; Hyracks targets the same sorts of applications that Hadoop is used for. The recent Nephele/PACTS system [28] extends the MapReduce programming model by adding additional second-order operators (e.g., match, cross, and co-group) with "your function goes here" plug-in points. Such an extension could be implemented as a layer on top of Hyracks similar to our Hadoop compatibility layer. Finally, declarative languages such as Pig [29], Hive [7], and Jaql [6]) all seek to compile higher-level languages down to MapReduce jobs on Hadoop. These languages could potentially be re-targeted at a platform like Hyracks with resulting performance benefits. (We are currently doing that for Hive, as one proof point, in addition to our own language, AQL, in the ASTERIX project at UCI.)

ETL (Extract-Transform-Load) systems have also used the DAG model to describe the transformation process. The DataStage ETL product from IBM [30] is similar to Hyracks in the way that its graph of operators and data-redistribution it expresses. DataStage does not provide automatic fault recovery, which is understandable given the smaller sized clusters it targets. Also, although DataStage generalizes the implementation of operators and data-redistribution logic, it provides no way for implementers to provide details about behavioral aspects that the infrastructure could use for scheduling purposes.

Recently, there has been work in making Hadoop jobs more performant through clever placement and organization of data [31] or by sharing intermediate computations across jobs [32]. Such work is orthogonal to the way that Hyracks achieves its performance improvements. Thus, these techniques could be used in conjunction with Hyracks, leading to further performance improvements. One other project that improves the performance of relational queries using Hadoop, essentially using it as a distributed job dispatcher, is HadoopDB [33].

VII. Conclusion and Future Work

This paper introduced Hyracks, a new partitioned-parallel software platform that we have designed and implemented to run data-intensive computations on large shared-nothing clusters of computers. As described, Hyracks allows its users to express a dataflow job as a DAG of operators and connectors. Operators operate on partitions of input data and produce partitions of output data, while connectors repartition operators' outputs to make the newly produced partitions available at the inputs of their consuming operators. We described the Hyracks end user model, for authors of Hyracks jobs, as well as the extension model for users who wish to augment the Hyracks built-in library with new operator and/or connector types. We also described the initial Hyracks implementation, including its approaches to job control, scheduling, fault-tolerance, and the efficient handling of data.

Lastly, we presented results from an initial performance evaluation of the current implementation of Hyracks on a 40-core/40-disk cluster at UCI. Since Hyracks is in same space as the popular Hadoop platform, our experiments compared Hyracks with Hadoop. The results indicate that Hyracks' fusion of database-derived processing techniques with the world of data-intensive computing has promise: The Hadoop compatibility layer of Hyracks consistently outperformed Hadoop for several different classes of applications, and relaxing the MapReduce programming model was shown to provide significant further gains. We also demonstrated that a less pessimistic approach to fault-tolerant job execution can be advantageous. We have released a preliminary version of Hyracks via open source (http://code.google.com/p/hyracks/) and are ready to begin welcoming inquiries and trial uses of Hyracks (and feedback!) from other data-intensive computing researchers.

In terms of future work, we have just begun to scratch the surface of the areas that we hope to explore in the next few years. Our plans include much more work related to parallel query compilation and execution on large clusters, using Hyracks as the runtime platform, as well as the integration of resource- and load-based query scheduling with fault-tolerant query execution and recovery techniques. We are also working on large-scale storage and indexing of semistructured data in the context of our ASTERIX project, again with Hyracks providing the runtime glue. Our eventual goal is to offer two open source software platforms to the community: Hyracks, for data-intensive computation, and ASTERIX, for large-scale, next-generation information management.

Acknowledgements: This project is supported by NSF IIS awards 0910989, 0910859, 0910820, and 0844574, a grant from the UC Discovery program, and a matching donation from eBay. We wish to thank Tyson Condie for making JOL available as a robust implementation of OverLog. Vinayak Borkar is supported in part by a Facebook Fellowship award.

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI '04*, December 2004, pp. 137–150. [Online]. Available: http://www.usenix.org/events/osdi04/tech/dean.html

[2] "Apache Hadoop website," http://hadoop.apache.org.

[3] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *EuroSys*, 2007, pp. 59–72.

[4] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with sawzall," *Scientific Programming*, vol. 13, no. 4, pp. 277–298, 2005.

[5] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: a not-so-foreign language for data processing," in *SIGMOD*, 2008, pp. 1099–1110.

[6] "Jaql Website," http://jaql.org/.

[7] "Hive Website," http://hadoop.apache.org/hive.

[8] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language." in *OSDI*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 1–14.

[9] R. Chaiken, B. Jenkins, P. A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "SCOPE: easy and efficient parallel processing of massive data sets," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1265–1276, 2008.

[10] Wikipedia, "Accidental complexity — Wikipedia, the free encyclopedia," 2010, [Online; accessed 23-July-2010].

[11] "TPC-H," http://www.tpc.org/tpch/.

[12] "ASTERIX Website," http://asterix.ics.uci.edu/.

[13] G. Graefe, "Query evaluation techniques for large databases," *ACM Comput. Surv.*, vol. 25, no. 2, pp. 73–169, 1993.

[14] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen, "The Gamma database machine project," *IEEE TKDE*, vol. 2, no. 1, pp. 44–62, 1990.

[15] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears, "BOOM analytics: exploring data-centric, declarative programming for the cloud," in *EuroSys*, 2010, pp. 223–236.

[16] "Declarative languages and systems – Trac," https://trac.declarativity.net/.

[17] "Apache Mahout: Scalable machine-learning and data-mining library," http://mahout.apache.org/.

[18] C. Chu, S. K. Kim, Y. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun, "Map-Reduce for machine learning on multicore," in *Proc. of NIPS*, 2007, pp. 281–288.

[19] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce online." in *NSDI*. USENIX Association, 2010, pp. 313–328.

[20] R. Vernica, M. J. Carey, and C. Li, "Efficient parallel set-similarity joins using MapReduce," in *SIGMOD*, 2010, pp. 495–506.

[21] "DBLP/xml," http://dblp.uni-trier.de/xml/dblp.xml.gz.

[22] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *SIGMOD*, 2009, pp. 165–178.

[23] G. Graefe and P.-Å. Larson, "B-tree indexes and CPU caches," in *ICDE*, 2001, pp. 349–358.

[24] J. Shemer and P. Neches, "The genesis of a database computer," *Computer*, vol. 17, no. 11, pp. 42 –56, Nov. 1984.

[25] S. Fushimi, M. Kitsuregawa, and H. Tanaka, "An overview of the system software of a parallel relational database machine GRACE," in *VLDB*, 1986, pp. 209–219.

[26] D. DeWitt and J. Gray, "Parallel database systems: the future of high performance database systems," *Commun. ACM*, vol. 35, no. 6, pp. 85–98, 1992.

[27] M. N. Garofalakis and Y. E. Ioannidis, "Parallel query scheduling and optimization with time- and space-shared resources," in *VLDB '97*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, pp. 296–305.

[28] 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*. New York, NY, USA: ACM, 2010, pp. 119–130.

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

[30] "IBM DataStage website," http://www-01.ibm.com/software/data/infosphere/datastage/.

[31] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, "Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing)," *PVLDB*, vol. 3, no. 1, pp. 518–529, 2010.

[32] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas, "MR-Share: Sharing across multiple queries in MapReduce," *PVLDB*, vol. 3, no. 1, pp. 494–505, 2010.

[33] A. Abouzeid, K. B. Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz, "HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads," *PVLDB*, vol. 2, no. 1, pp. 922–933, 2009.