PipeGen: Data Pipe Generator for Hybrid Analytics

Brandon Haynes Alvin Cheung Magdalena Balazinska
University of Washington
{bhaynes,akcheung,magda}@cs.washington.edu

Abstract

As the number of big data management systems continues to grow, users increasingly seek to leverage multiple systems in the context of a single data analysis task. To efficiently support such *hybrid* analytics, we develop a tool called PipeGen for efficient data transfer between database management systems (DBMSs). PipeGen automatically generates *data pipes* between DBMSs by leveraging their functionality to transfer data via disk files using common data formats such as CSV. PipeGen creates data pipes by extending such functionality with efficient binary data transfer capabilities that avoid file system materialization, include multiple important format optimizations, and transfer data in parallel when possible. We evaluate our PipeGen prototype by generating 20 data pipes automatically between five different DBMSs. The results show that PipeGen speeds up data transfer by up to 3.8× as compared to transferring using disk files.

Categories and Subject Descriptors H.2.4 [*Database Management*]: Systems—Distributed databases, Query processing; H.2.5 [*Database Management*]: Heterogeneous Databases

Keywords Hybrid analytics, heterogeneous data transfer

1. Introduction

Modern data analytics requires retrieving data stored in multiple database management systems (DBMSs). As an example, consider a scientist who collects image data in an array database [45] while storing other metadata in a relational DBMS. To analyze her data, she writes a query that retrieves data from the relational store and ingests it temporarily into the array database to join it with her image data. She further extracts features from her images that she moves to a graph database [29] to leverage its specialized capabilities such as machine learning algorithms. This type of hybrid analytics is increasingly common. Most big data system vendors today already support some type of connection between Hadoop [4] and their parallel database management system [12, 33, 46]. Several efforts focus on creating a more complete virtualized layer on top of all of an enterprise's data [19, 34, 40]. Yet other efforts build new generation systems designed specifically for hybrid analytics [1, 14]. Unfortunately, in spite of the need, hybrid analytics remains poorly supported because individual DBMSs remain poorly connected. We lack solutions for seamlessly and efficiently moving intermediate query results as an analysis crosses system boundaries. This functionality is critical to ensuring the high performance of hybrid analytics.

In general, a hybrid analytics task involves transferring data among n different DBMSs, each potentially having its own internal data format. Efficient hybrid analytics thus requires moving data efficiently among the DBMSs involved [43]. This problem is comparable to performing an extract-transform-load operation, and there are two approaches to solving this problem. First, many DBMSs support serializing data between the internal format used by the DBMS and an external one. Users can transfer data by exporting it from one DBMS and importing into another. This works well when a common data format exists between the DBMSs involved, for instance text-based data formats such as comma-separated values (CSV) or binary data formats such as Arrow [6]. Unfortunately, moving data using a text-oriented format is costly: it requires serializing the data from the source DBMS, storing the serialized data to the disk, and importing it into the internal format of the target DBMS.[1] Using binary formats alleviates some of these overheads, although data materialization via the disk still incurs unnecessary and significant time and storage space. Additionally, while support for text-oriented formats such as CSV are common, shared binary formats remain rare. For example, CSV is the only text-oriented data format supported by all five DBMSs used in our evaluation, and no common binary data format is supported by all of them.

A second approach to moving data that avoids using the disk as an intermediary is to implement dedicated

SoCC '16, October 05-07, 2016, Santa Clara, CA, USA.
© 2016 ACM. ISBN 978-1-4503-4525-5/16/10…$15.00.
DOI: http://dx.doi.org/10.1145/2987550.2987567

[1] In this paper, we use DBMS to refer to relational and non-relational stores. Also, we assume the transferred data is not persisted in the target DBMS after query execution.

data transfer programs, i.e., *data pipes*, between specific source and target DBMSs. Common data transfer protocols such as JDBC [2] and Thrift [7] are often not implemented to support reading or writing data in parallel, and because of this are impractical for efficiently moving data between systems. Other dedicated software packages exist, which do transfer data in parallel between specific systems, such as `spark-sframe` for moving data between Spark and GraphLab [48]. Unfortunately, generalizing this approach requires implementing $O(n^2)$ data pipes to transfer data between n different DBMSs. Besides being impractical, this approach requires knowledge about the internals of each DBMS, making it inaccessible to non-technical users. Even for technical professionals, implementing dedicated data pipes is often a brittle and error-prone process with quickly outdated mechanisms.

In this paper, we describe a new tool called PipeGen that retains the benefits of dedicated data pipes but without the shortcomings of manual data pipe implementation or serialization via physical storage. The key idea is to leverage a DBMS's existing data serialization functionality for commonly-used formats (e.g., CSV) to *automatically generate* data pipes from DBMS binaries. To do so, PipeGen assumes existing import and export functionality for a given DBMS and unit tests that exercise the code associated with these operations, where export tests cover code paths from reading data in the internal representation to serializing it to the disk, and import tests read serialized data from disk back into the internal data representation. Should existing tests not cover all relevant code paths, PipeGen allows users to provide additional test cases to increase coverage.

Data pipes are created in two steps. First, PipeGen executes the data export unit tests and analyzes the DBMS binary to create an export data pipe that transmits data via a network socket rather than the disk. Then, PipeGen uses the import unit tests to create an import data pipe that reads data from a network socket rather than a disk file. The same tests are used to validate the correctness of the generated data pipes. To use the generated pipes, users issue queries that export and import from disk files using a special filename, whereupon PipeGen creates a network socket and connects the generated import and export pipes to transmit data.

To further speed up data transfer, PipeGen comes with a number of optimizations. First, PipeGen attempts to transmit data using binary rather than text-oriented formats. This is done by serializing data using external binary encoders (Apache Arrow [6] in our current implementation) rather than text encoders in the generated pipes. As another optimization, PipeGen analyzes the code to eliminate textual delimiters during data transfer. Finally, PipeGen compresses the encoded data to further reduce transfer time.

We have implemented a prototype of PipeGen that generates data pipes for DBMSs written in Java by analyzing their bytecode implementations. PipeGen currently does not address the problem of schema matching and mapping. It assumes that users of the generated data pipes will insert appropriate operators to transform data as needed.

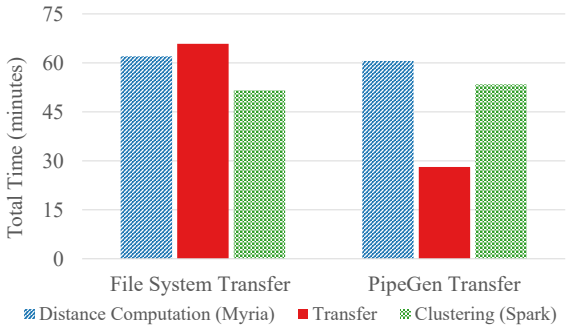


Figure 1. Total time for an astronomy workflow with and without PipeGen data pipes. Distances were computed in Myria and transferred to Spark for clustering.

In sum, this paper makes the following contributions:

- We describe an approach based on program analysis to automatically create data pipes for efficient direct data transfer between DBMSs (Section 3 and Section 4).
- We present techniques that improve the performance of the generated pipes, including encoding data using binary rather than text-oriented formats, applying compression, and transferring data in columnar form (Section 5).
- We build a prototype of PipeGen and evaluate it by generating data pipes between five different Java-based DBMSs. The optimized data pipes perform up to 3.8× faster than a naïve approach that transfers data via the file system using CSV (Section 6).

2. Motivating Example

Consider an astronomer who studies the evolution of galaxies using N-body simulations of the universe [24]. The simulation output takes the form of a series of snapshots, each one comprising particles distributed in 3D space. A typical workflow for the astronomer is to first cluster the particles for each snapshot into galaxies and then analyze the evolution of these galaxies over time. In this example, the particle clustering is a critical piece of the analysis and astronomers often experiment with different clustering algorithms [26].

In our scenario, an astronomer uses the Myria DBMS [23, 31] to analyze the output of an N-body simulation, including performing an initial particle clustering, when she learns of a novel clustering method. This new method may or may not outperform the current approach, and the astronomer is interested in evaluating the two. Critically, however, Myria *does not support this technique*, but it is available in Spark [52].

A typical approach is the following: perform data preparation in Myria, export the intermediate result to the file system, and import the files into Spark. Since the only common file format supported by both systems is CSV (they support JSON but produce incompatible documents), the intermedi-

ate result is materialized using CSV files. A second iteration is necessary to bring results back to Myria for comparison.

Unfortunately, transferring large datasets via the file system is an expensive proposition, and for some datasets there may be insufficient space available for a second materialization. The alternate approach of modifying the DBMS source code to support efficient data transfer requires deep programming and DBMS expertise.

To illustrate, we execute the workflow described above using a two-node cluster and the 100 GB present-day snapshot of an astronomy simulation stored in Myria. An initial data clustering has already been performed on the data in Myria. Our goal is to count differences in cluster assignments between this existing data clustering and those for power iteration clustering (PIC) [28] available on Spark. We compute pairwise distances in Myria between all particles having distance less than threshold $\varepsilon = 0.00024$, set by our astronomy collaborators. This yields approximately one billion pairs. We then transfer this data to Spark where we perform PIC clustering. Finally, we transfer the resulting assignments back to Myria for comparison with the existing clusters. To contrast different data transfer mechanisms, we perform the transfer using as intermediary both the file system and data pipes generated by PipeGen.

Figure 1 illustrates the performance of each of these steps. When using the file system, transferring the large set of intermediate pairwise differences is already more expensive than either of the other two steps alone. Using the data pipe generated by PipeGen, however, reduces the data transfer time from 66 to 28 minutes.

If the astronomer finds the results useful, she may repeat the comparison for the other snapshots and may even include the new clustering algorithm as part of her regular data analytics pipeline. In those scenarios, the data pipes created by PipeGen will repeatedly deliver the performance benefit.

3. The PIPEGEN Approach

PipeGen enables users to move data efficiently between DBMSs. In this section, we give an overview of PipeGen's usage and approach.

3.1 Using PipeGen

PipeGen works in two phases. First, a user invokes PipeGen to generate new bytecode for the data pipes. Second, the user writes and executes queries that use the generated pipes.

Constructing a data pipe. To generate a data pipe, the user invokes PipeGen with several inputs. These include a pair of scripts that execute unit tests associated with import and export functionality for a specific data format (e.g., CSV, JSON, or binary format), and the locations of the DBMS binaries and any external libraries used. Given these inputs, PipeGen analyzes the bytecode of the DBMS and generates an optimized data pipe.

Using the generated data pipe. To use a generated data pipe, the user executes two queries: one that exports data from a source DBMS extended by PipeGen, and one that imports data into a target DBMS extended by PipeGen, as shown in Figure 2. These queries may occur in any order; PipeGen will automatically block until both DBMSs are ready (see Section 4.3). The queries issued to each DBMS are written as if the data was moved using the original export and import code, and they use a special filename (e.g., "db://X") to identify the use of the generated data pipes.

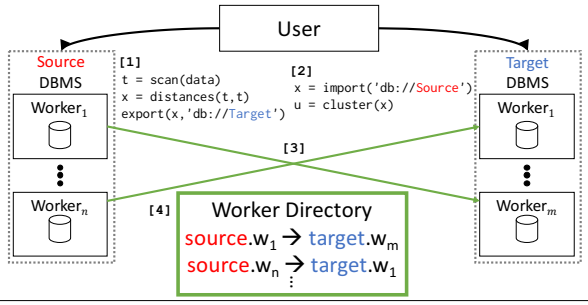


Figure 2. Using the data pipe generated by PipeGen for the hybrid analysis from Section 2: 1. User submits a query to the source DBMS (e.g., Myria) to compute distance and export to the target DBMS using data pipe. 2. User issues an import query on the target DBMS (e.g., Spark) to cluster the result. Data is transferred using the generated data pipe in 3., and in 4. a worker directory (see Section 4.3) coordinates the connection process. PipeGen-related components are highlighted in green.

After receiving the queries, PipeGen connects the generated import and export data pipes in the source and target DBMSs by passing them a network socket to transmit data. In the case where both DBMSs support multiple worker threads, they are matched with each other by a worker directory maintained by PipeGen, as shown in Figure 2.

Alternatively, the PipeGen automatically-generated data pipes can be used in other contexts such as by being integrated into an existing hybrid [14, 27] or federated DBMS [25], with the query optimizer determining when the generated pipes should be used. However, in this paper we focus on the first usage, where users explicitly submit queries for execution.

3.2 Data Pipe Generation Overview

To generate a data pipe, PipeGen takes as input the bytecode of a DBMS and tests that exercise the import and export functionality for a given format. It begins by executing and analyzing the provided tests. Using the results, it modifies the bytecode of the DBMS to generate a data pipe that can transfer data directly between that DBMS and another PipeGen-extended DBMS that uses a common data format. When the common format is text-oriented, PipeGen further optimizes the format of the transferred data to improve performance, as we will discuss in Section 5. Finally, PipeGen verifies the correctness of the new data pipe. The full process is illustrated in Figure 3.

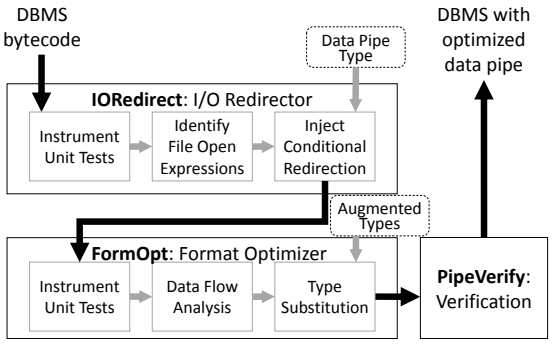


Figure 3. Compile-time components of PipeGen. The file IO redirector (IORedirect) generates a data pipe to transfer data via a socket, while the format optimizer (FormOpt) improves efficiency for text-oriented formats.

PipeGen supports single-node and parallel DBMSs. For the latter, as illustrated in Figure 2, the bytecode produced transfers data in parallel directly between individual workers. Our implementation and evaluation currently target shared-nothing systems, but the approach can be applied to other parallel architectures.

4. File IO Redirection

In this section, we discuss the redirection component of PipeGen. This component, called *IORedirect*, creates a data pipe from the DBMS's existing serialization code and modifies that code such that instead of using the file system, data are exported to (and imported from) a network socket provided by PipeGen at runtime. When the source and target DBMSs are colocated on the same machine, the socket is a local loopback one. Otherwise the socket connects to the remote machine hosting the target DBMS, with the address provided by a directory as part of PipeGen (Section 4.3).

4.1 Basic Operations

To generate a data pipe, PipeGen first locates in the DBMS's bytecode the relevant fragments that import from and export data to disk files. It does this by instrumenting the execution of the export and import unit tests to inject tracing code that enables PipeGen to identify file system operations (e.g., file open and close). It then substitutes these operations with equivalents on a network socket. This allows transmitting arbitrary size datasets (in contrast to an approach based on memory-mapped files, for example), and for file systems that do not support named pipes (e.g., HDFS).

However, it does not suffice to replace all file system operations in the bytecode with socket equivalents, as a DBMS may open multiple, unrelated files such as debugging logs. IORedirect disambiguates this by capturing the filenames that are used in all file open calls. Calls with filenames other than the target of the import or export are eliminated.

IORedirect then modifies each remaining call site by adding code that redirects I/O operations to a network socket. The new code is executed only when a user specifies a *reserved filename* ("`db://X`") for import or export. The

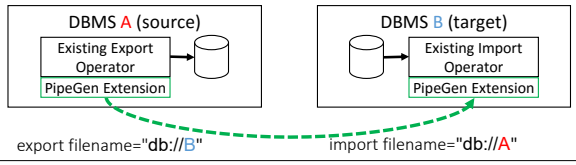


Figure 4. PipeGen creates a data pipe to transmit data via a socket. A user utilizes the data pipe by specifying a reserved filename ("`db://X`") as the import/export destination.

```
1  void export(String fname, ...) {
2    ...
3    stream = new FileOutputStream(fname);
4    ...
5    stream.close(); }
```

⇓

```
1  void export(String fname, ...) {
2    ...
3    stream = fname.matches(format)
4      ? new DataPipeOutputStream(fname);
5      : new FileOutputStream(fname);
6    ...
7    stream.close(); }
```

Figure 5. Modifications to create a Java data pipe. Unit tests indicate that line 3 (top) should be modified. The modified code below matches against the reserved filename format (bottom, line 3), and uses the generated data pipe if a match is found (bottom, line 4). Thanks to subtyping, other uses of the stream are unaffected (e.g., line 7).

conditional nature of the redirection is important because we want to preserve a DBMS's ability to import from and export to disk files. This modified architecture is shown in Figure 4.

When adding the conditional logic to each relevant call site, IORedirect introduces a specialized data pipe class into the DBMS that reads or writes to a remote DBMS via a passed-in network socket rather than the file system. This class is a subtype of its file system-oriented counterpart. Subtyping allows PipeGen to substitute all instances of the file IO classes with a data pipe instance. To illustrate, a socket descriptor can be substituted for a file descriptor in a C or C++-like language, while Java or .NET file streams can be interchanged with network socket streams.

IORedirect modifies the code to use the generated data pipe when the reserved filename is specified. Figure 5 illustrates one modification made by IORedirect for the Myria DBMS, which is written in Java. Here an instantiation of `FileOutputStream` is replaced with a ternary operator that checks for the reserved filename and substitutes a data pipe class (a subtype of `FileOutputStream`) if so. Thanks to subtyping, other IO operations (e.g., `close`) operate as before.

4.2 Verification

To verify the correctness of the modifications as well as the correctness of the optimizations that we describe in Section 5, PipeGen also includes a module called *PipeVerify*.

Following standard practices in the programming systems community [17, 32, 35, 36, 41], PipeGen defines correctness

of the generated data pipes based on passing the unit test cases that are used during pipe generation. Operationally, the correctness of the generated data pipe is verified as follows. First, PipeVerify launches a *verification proxy* that imports and exports data as if it were a remote DBMS. This proxy redirects all data received over a data pipe to the file system, and transmits data from the file system through a data pipe. Next, PipeVerify executes the unit tests for the modified DBMS using the reserved filename format that activates the data pipe for all files and connects the unit test outputs and inputs over that pipe to the the verification proxy. As data is imported and exported over the data pipe, the verification proxy reads from and writes to disk. PipeVerify then relies on existing unit test logic to verify that the contents that are read from or written to disk are correct.

Finally, IORedirect exposes a dynamic debugging mode that verifies the correctness of the generated data pipe at runtime. Under this mode, a user specifies that n elements of the transmitted data be both written to the disk using existing serialization logic, and also transmitted over the data pipe. The receiving DBMS then reads the data from the file system and compares it to the values that have been transmitted over the pipe. Any deviations trigger a failure and imply that the failing query should itself be used as an additional unit test during data pipe creation.

4.3 Parallel Data Pipes

Many multi-node DBMSs support importing or exporting data in parallel using multiple worker threads. PipeGen uses a *worker directory* to match up the worker threads in the source and target DBMSs. The directory is instantiated by PipeGen when the DBMS starts and is accessible by all DBMS worker processes.

The worker directory is used as follows. When a user exports data from DBMS A to DBMS B, as DBMS B prepares to import, each worker $b_1, \ldots, b_n$ registers with the directory to receive data from DBMS A. As data is exported from DBMS A, workers $a_1, \ldots, a_n$ query the directory to obtain the address and port of a receiving worker b_i. Each exporting worker a_i blocks until an entry is available in the directory, after which it connects to its corresponding worker using a network socket. A simplified Java implementation for the query process is shown in Figure 6, and Figure 7 describes the overall workflow.

The worker directory ordinarily assumes that the number of workers between the source and target DBMSs are identical. However, a user may embed metadata to explicitly indicate the number of exporting and importing processes. For example, an export from DBMS A using two workers to DBMS B with three workers would use the respective filenames `db://A?workers=2` and `db://A?workers=3`. When the number of importing workers exceeds the exporters, the worker directory opens a "stub" socket for the orphaned importing worker that immediately signals an end-of-file condition; under this approach the extra importing workers sit idle until the data transfer has completed. The case where the number of exporters exceeds the number of importers is handled similarly, though we leave this as a future extension. Finally, IORedirect supports simultaneous data transfer by independent queries by attaching a unique query identifier to the reserved filename (e.g., `db://A?id=n`).

```
1 class DataPipeOutStream extends
      FileOutputStream {
2   DataPipeOutStream(String fname) {
3     e = Directory.query(fname);
4     skt = new Socket(e.host, e.port);
5     ... } }
```

Figure 6. Simplified Java implementation of directory query logic. The query (line 3) blocks until an import worker has registered with the directory. The resulting directory entry is used to open a socket to the remote DBMS (line 4).

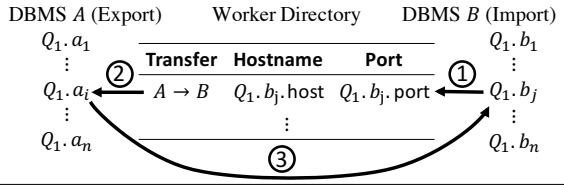


Figure 7. Using the worker directory to initiate data transfer. In (1), import worker b_j from query Q_1 registers with the directory and awaits a connection. In (2), export worker a_i queries the directory and blocks until an entry is available, after which it uses the details to connect (3).

5. Optimizations

The data pipe that IORedirect generates can *immediately* be used to transfer data between two systems using a common format covered by unit tests. For example, a user might use the data pipe to transfer data between Spark and Giraph using Hadoop sequence files, as that format is supported by both systems and is exercised by unit tests.

Unfortunately, it is rare for two DBMSs to support the same efficient binary format. For example, Apache Parquet is (natively) supported by only two of the systems we evaluate (Spark [52] and Hadoop [4]). Giraph does not support Parquet without third-party extension while Derby [3] does not support import and export of any binary format.

By contrast, support for text-oriented formats is far more pervasive: *all* of the DBMSs we evaluate (Myria [23], Spark [52], Giraph [8], Hadoop [4], and Derby [3]) support bulk import and export using a CSV format, and all but Derby support JSON. Accordingly, we expect users to revert to text-based data formats to transfer data between such systems, due to the lack of shared binary-based data formats.

Unfortunately, text-based formats are inefficient and incur substantial performance overhead, including:

- **String encoding of numeric types.** It is often the case that the size of a converted string greatly exceeds that of its original value, which increases transfer time. For example, the 4-byte floating point value $-2.2250738585072020 \cdot 10^{-308}$ requires 24 bytes to encode as a string.
- **String conversion and parsing overhead.** Export and import logic spends a substantial amount of time converting primitive values into/from their string representation.
- **Extraneous delimiters.** For attributes with fixed-width representations, many formats include extraneous separators such as delimiters or newlines.
- **Row-orientation.** DBMSs that output CSV and JSON often do so in a row-oriented manner; this is the case for all of the DBMSs we evaluate in Section 6. This makes it difficult to apply layout and compression techniques that benefit from a column-oriented layout. For instance, our experiments show that converting the exported data to column-major form before transmitting offers a modest performance benefit (see Figure 12).

To improve the performance of text-based formats, PipeGen comes with a format optimizer called *FormOpt* (Figure 3) to address these inefficiencies. FormOpt optimizes a given text format in one of two modes. First, it analyzes the DBMS bytecode to determine if the export or import logic uses an external library to serialize data to JSON or CSV (PipeGen currently supports the Jackson JSON library [16] and we plan to support Java JSONArrays and the Apache Commons library [5]). If so, FormOpt replaces use of the library with a PipeGen-aware variant, to be discussed in 5.2. The implementation of this variant is then directly responsible for performing the optimizations described above.

On the other hand, a DBMS might directly implement serialization functionality without using an external library. For example, Myria directly implements its JSON export functionality. For such systems, FormOpt supports a second mode that leverages *string decoration* to target the string production and consumption logic that occurs during data export and import. Since components of string decoration are used when applying the library extension strategy, we introduce it first and then discuss the extension process.

5.1 String Decorations

Without using external libraries, data is serialized into CSV or JSON format by first converting objects and primitive values into strings, concatenating the strings while interspersing delimiters and metadata such as attribute names, and writing the result. To optimize these steps, FormOpt modifies the DBMS bytecode such that, whenever the modified DBMS attempts to write the string representation of a fixed-width primitive, it instead writes to a compact binary representation provided by PipeGen. Ordinary strings and other non-primitive values are transmitted in their unmodified form. As we will see, doing so eliminates the transmission of unnecessary values such as delimiters and attribute names.

To accomplish this bytecode-level modification, PipeGen's data pipe (introduced in Section 4.1) must receive the primitive values of the data to transmit before their conversion into strings. The core difficulty in ensuring this property is that there may be many primitive values embedded in any given string that is written to the data pipe. For example, a particular DBMS might concatenate together all attributes before writing them to the output stream `s`: `s.write(attr1.toString() + "," + attr2.toString() + ...)`. By the time the data pipe receives the concatenated value, it will be too late as both of the attributes would have already been converted into strings.

FormOpt addresses this by introducing a new augmented string called `AString`, which is a subtype of `java.lang.String`. `AString` is backed by an array of `Objects` rather than characters, with the goal of storing the data to be transmitted in binary rather than textual form. By substituting `String` instances for `AStrings` in the appropriate locations, FormOpt avoids the problem described above by storing references to the objects to be serialized rather than their string representation.

For example, given ordinary string concatenation:

```
s = new Integer(1).toString() + "," + "a";
```

FormOpt changes the statement into one that uses `AStrings`:

```
s = new AString(1) + new AString(",") +
    new AString("a");
```

Each of the three instances maintains its associated value as an internal field (`1`, `","`, and `"a"` respectively) and the concatenated result—itself an `AString` instance—internally maintains the state `[1, ",", "a"]`. Note that the final `AString` instance need not include the concatenated string `"1,a"` in its internal state since it may easily reproduce (and memoize) it on demand. More complex types are immediately converted into strings during this aggregation process to ensure that subsequent changes to their state do not affect the internal state of the `AString` instance. However, as we shall see below, converting a complex object into a string (e.g., through a `toString` invocation) may produce an `AString` instance, which allows for nesting.

When a DBMS's implementation invokes an IO method on an injected data pipe, the data pipe implementation inspects any string parameter to see if it is an `AString`. Additionally, methods exposed by the data pipe that produce a string return an `AString`. During export, this allows the data pipe to directly utilize the unconverted values present in the internal state of an `AString`; similarly, during import the `AString` implementation efficiently executes common operations such as splitting on a delimiter and conversion to numeric values without materializing as character string.

This resolves the problems we described above, but does not address the issue of *where* to substitute an `AString` for a

Algorithm 1 String decoration

```
function TRANSFORM(T: tests)
1:  for each t ∈ T do
2:      Find relevant IO call sites C
3:      for each c ∈ C do
4:          Construct data-flow graph G
5:          for each expression e ∈ G do
6:              if e is literal v or
7:                      e is instantiation String(v) or
8:                      e is v.toString() then
9:                  Replace v with AString(v)
10:             else if e is Integer.parseInt(v) then
11:                 Replace v with AString.parseInt(v)
12:             else if e is Float.parseFloat(v) then
13:                 Replace v with AString.parseFloat(v)
14:             Similarly for other string operations
```

regular string instance. Intuitively, we want to substitute only values that are (directly or indirectly) related to data pipe operations, rather than replacing all string instances in the bytecode. To find this subset, PipeGen executes each of the provided unit tests and marks all call sites where data is written to/read from the data pipe. PipeGen then performs data-flow analysis to identify the sources of those values (for export) and conversions to primitive values (for import). This produces a data-flow graph (DFG) that identifies candidate expressions for substitutions.

Using the resulting DFG, FormOpt replaces three types of string expressions: string literals and constructors, conversion/coercion of any value to a string (for export), and conversion/coercion of any string to a primitive (for import). To illustrate this, Figure 8 shows (a) two potential implementations of an export function, (b) the code after string replacement, and (c) the accumulated values in the internal state of the `AString`s after one iteration of the loop.

Algorithm 1 summarizes the replacement process. On lines 1-2, the algorithm executes each test and identifies the relevant file IO call sites. On line 4, it uses these sites to construct a DFG. For each expression e that converts to or from a string format, it replaces e with a corresponding `AString` operation (lines 5-14). Lines 6-9 target expressions relevant for data export; for example, a string literal v is replaced with an augmented instance `AString(`v`)`. To support efficient imports, the algorithm performs a similar replacement for strings converted to primitive values (lines 10-14).

This optimization is also applied to speed up transmitting user-defined types (UDTs) using text encoding. As above, the implementation used to convert instances is contained in the DFG produced by unit test execution, and so each conversion is replaced with an `AString` instance. For example, a UDT with value `{["abc", 6.7]}` might produce an `AString` instance with the state `{"[", "abc", ",", 6.7, "]"}]` that is efficiently transferred to the destination DBMS. If the importing code later parses the string by splitting on a token, the result is directly produced using this state. On the other hand, if the UDT is converted to a byte array representation during export (e.g., `"abc\0\x40\xD6\x66\x66"`), when this array is wrapped in an `AString` it is sent unmodified to the destination system.

PipeGen verifies the correctness of these modifications by executing the specified unit tests as discussed in 4.2. PipeGen turns off this optimization if one or more unit tests fail following the modifications made by the FormOpt component in string decoration mode, although we have not encountered this in our experiments.

5.2 Using External Libraries

As mentioned, many DBMSs use external libraries to serialize data. PipeGen targets these external libraries directly by including a custom implementation for each external library (of which there are a few that are commonly used) by overriding the methods with PipeGen-aware versions. Under this mode, FormOpt replaces instantiations of a given formatting library with a PipeGen-aware implementation that avoids the overhead associated with strings and delimiters. For example, whenever the DBMS invokes a method that builds or parses the text format, PipeGen instead internally constructs or produces a binary representation. When the resulting text fragment is converted to string form, the PipeGen-aware version generates an `AString` that contains the binary representation in its internal state. During import, the PipeGen-aware library recognizes that it is interacting with a data pipe (q.v. Section 4.1) and directly consumes the intermediate binary representation. This allows the PipeGen-aware library to construct an efficient internal representation of the input.

As before, FormOpt must identify only those locations where a library is used for import and export. Our approach for doing so—using unit tests and DFGs—is similar to that of string decoration. For example, if a user specifies JSON when invoking PipeGen, FormOpt will examine the bytecode of the DBMS for instantiations of supported JSON libraries. Using the resulting DFG, FormOpt replaces string literals, constructors, and conversion/coercion expressions in a manner identical to that discussed earlier. Additionally, FormOpt replaces the instantiation of the library with an augmented variant along with any writer or stream interfaces that the library exposes. For example, consider the following simplified version of the Spark JSON export function:

```
String toJSON(RDD<String> rdd) {
  Writer w = new CharArrayWriter();
  JsonGenerator g = new JsonGenerator(w);
  foreach(Object e: rdd){generateJSON(g,e);}
  return w.toString(); }
```

The PipeGen-aware implementation overrides this to:

```
String toJSON(RDD<String> rdd) {
 Writer w = new AWriter(new ACharWriter());
 JsonGenerator g =
   new AJsonGenerator(new JsonGenerator(w));
 foreach(Object e: rdd) {generateJSON(g,e);}
 return w.toString(); } // an AString!
```

As in string decoration, FormOpt disables library call replacement if the generated code does not pass all unit test cases. If string decoration also fails to pass the tests,

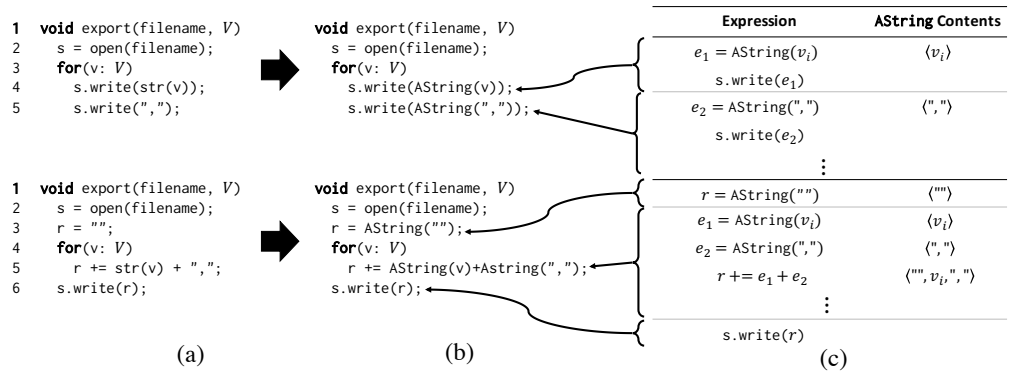


Figure 8. (a) Two ways to implement CSV export (inserting newlines instead of a comma on the last value is omitted for clarity); (b) after replacement of literals and conversions; (c) accumulated values in `AString` instances after one iteration.

then PipeGen only generates the basic data pipe as discussed in Section 4.

5.3 Intermediate Format

Using `AStrings` resolves the first two sources of overhead introduced above (encoding of numeric types and conversion/parsing overhead). In this section, we introduce optimizations designed to eliminate delimiters and avoid redundant metadata. These optimizations are implemented inside PipeGen's data pipe.

5.3.1 Delimiter Inference and Removal

Text-oriented formats such as CSV and JSON include delimiters that separate attributes and denote the start and end of composite types. In some cases these delimiters are fixed in advance; for example, square brackets are used in JSON to indicate an array. However, default delimiters often vary on a per-system basis. This is common under CSV, where some systems default to a non-comma delimiter (e.g., Hadoop uses tab separation by default) or allow the delimiter to be specified by the user (e.g., Derby). To eliminate delimiters, FormOpt needs to first infer them. FormOpt does so by first running the provided unit tests. During the execution of each test, FormOpt counts the length-one strings within the array and identifies which character is most likely to be the delimiter. For example, the array $[1,$"|","a,b","\n"$]$ contains exactly one length-one string ("|"), and FormOpt concludes that this is most likely to be the delimiter. The input $[1,$"|","a","\n"$]$ is ambiguous, since both "|" and "a" appear with equal frequency. In this case, FormOpt applies, in order, the following tie-breaking heuristics: (i) prefer non-alphanumeric delimiters, and (ii) prefer earlier (in terms of position) delimiters. Under both heuristics, FormOpt would select "|" as the final delimiter.

Note that should FormOpt infer an incorrect delimiter, invalid data will be transmitted to the remote DBMS. In the previous example, if FormOpt's selection of "|" was invalid and the character "a" was actually the correct delimiter, it would incorrectly transmit the tuple $(1,$"a"$)$ instead of the correct value ("1|","""). More importantly, this is likely to cause the unit tests to fail as discussed in Section 4. This results in FormOpt disabling the optimization until the unit tests were extended to fully disambiguate the inference.

5.3.2 Redundant Metadata Removal

More complex text formats such as JSON may not require the delimiter inference described above, but instead serialize complex composite types such as arrays and dictionaries. When producing/consuming JSON or a similar textual format, the composite types produced by a DBMS often contain values that are highly redundant. For example, consider the following document produced by the Spark `toJSON` method:

```
{"column1": 1, "column2": "value1"}
{"column1": 2, "column2": "value2"}
{"column1": 3, "column2": "value3"}
```

When such JSON documents are moved between systems, the repeated column names greatly increase the size of the intermediate transfer. To avoid this overhead, FormOpt modifies the format of the intermediate data to transmit exactly once the set of keys associated with an array of dictionaries. In the above example, FormOpt would transmit the column names $[$"column1","column2"$]$ as a *key header*, and then the values $[(1,$"value1"$),\ldots]$ as a sequence of pairs. When importing, FormOpt reverses this process to produce the original JSON document(s).

The logic for this transformation is embedded into the JSON state machine (a subcomponent of the data pipe) that is used to consume the `AString` array. When FormOpt transitions into the key state for the first dictionary in an array, it accumulates that key in the key header. Once the dictionary has been fully examined, PipeGen transmits the key header to the remote DBMS. Subsequent dictionaries in that array are transmitted without keys, so long as they are identical to the initial dictionary. While this approach may be extended to nested JSON documents, our prototype currently only optimizes top-level dictionaries.

If a new key is encountered in some subsequent dictionary after the key header has been transmitted, FormOpt adopts one of two strategies. First, if the keys from the new dictionary are a superset of those found in the key header,

FormOpt appends the new key to the existing key header and retransmits it; this is possible since keys in a JSON object are unordered [11]. This addresses the common case where the set of exported keys was not complete due to, for example, a missing value in the initial exported dictionary.

A second case occurs when the keys associated with a new dictionary are disjoint from those in the key header. This might occur during export from a schema-free DBMS, where exported elements have widely varying formats. In this case, FormOpt disables the optimization for the current dictionary and does not remove keys during its transmission.

5.4 Column Orientation & Compression

DBMSs that output text-oriented formats generally do so in a row-oriented manner. For example, a Spark RDD containing n elements that is exported to CSV or JSON will generate n lines, each containing one element in the RDD. This is also true in the other systems we evaluate, for both JSON and CSV formats. However, once FormOpt produces an efficient data representation, we no longer need to transmit data in row-major form. For example, the data pipe can accumulate blocks of exported data and transform it to column-major form to improve transfer performance. Indeed, recent work on column-oriented DBMSs suggests that some of the benefits (e.g., compacting/compression, improved IO efficiency) [44] may also improve performance for data transfer.

After examining various formats for the wire representation of our data (see Section 6.3) we settled on Apache Arrow as the format we transmit, since it performs the best. To maximize performance, our prototype accumulates blocks of rows in memory, pivots them into column-major form by embedding them into Arrow buffers, and transmits these buffers to the destination DBMS. The receiving DBMS reverses this process.

6. Evaluation

We have implemented a prototype of PipeGen in Java. PipeGen enhances Java DBMSs that make use of the local file system or HDFS for data import and export. All modifications are made on bytecode and do not require access to the source code of the DBMS. PipeGen uses `btrace` [47] for instrumentation, `javassist` [9, 10] for bytecode modification, and Soot [49] to produce dataflow graphs used to identify augmentation sites.

Our prototype assumes that a DBMS's import and export implementations are well-behaved, meaning that they do not perform large seeks, since seeks are not supported by network sockets. It also expects that a file is opened exactly once, since otherwise multiple sockets will be created when using the pipe. For text-based formats, PipeGen assumes that character strings are used to serialize data (rather than byte arrays, for instance), and that values are constructed through string concatenation rather than random access.

We evaluate PipeGen using benchmarks that show data transfer times between five Java DBMSs: Myria [23], Spark 1.5.2 [52], Giraph 1.0.0 [8], Hadoop 2.7.1 [4], and Derby 10.12.1.1 [3]. We first examine performance differences between the PipeGen data pipes and importing/exporting data through the file system (Section 6.1). Next, we analyze the performance gains from each of our optimizations (Section 6.2). We then evaluate the impact of different data formats transferred between DBMSs (Section 6.3), along with the compression method across colocated and wide-area clusters (Section 6.4). Finally, we show the number of modifications made during data pipe compilation (Section 6.5).

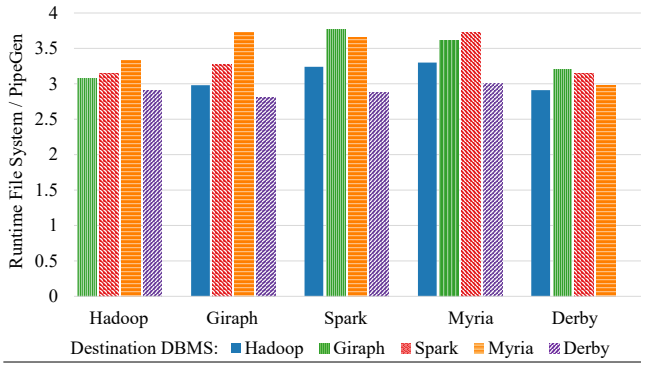


Figure 9. Total speedup between file system and PipeGen for 10^9 elements. Transfer occurred from a source DBMS (x-axis) to a destination DBMS (bar color/pattern) using CSV. The number of workers/tasks was fixed at 16.

Unless otherwise specified, all experiments utilize a 16-node cluster of `m4.2xlarge` instances in the Amazon Elastic Compute Cloud. Each node has 4 virtual CPUs, 16 GB of RAM, and a 1TB standard elastic block storage device. We deploy the most recent stable release of each DBMS under OpenJDK 1.8. Except for Derby (a single-node DBMS), we deploy each system across the cluster using YARN [50]. For each pair of DBMSs, we colocate workers and assign each YARN container 2 cores and 8 GB of RAM.

With the exception of Figure 10, the experiments in this section all involve the transfer of n elements with a schema having a unique integer key in the range $[0, n]$ followed by three (integer $\in [0, n]$, double) pairs. Each 8-byte double was sampled from a standard normal distribution. For Giraph, we interpreted the values as a graph having n weighted vertices each with three random directed edges weighted by the following double.

6.1 Paired Transfer

Figure 9 shows the total transfer time between pairs of DBMSs using an export/import through the file system using functionality provided by the original DBMS versus an export/import using PipeGen-generated data pipes. For this experiment, we transfer 10^9 elements using 16 workers, and enable all optimizations. Since CSV is the only common format supported by all DBMSs, file system transfers use this format and the PipeGen data pipes are generated from CSV export and import code.

As the results show, data pipes significantly outperform their file system-oriented counterparts. For this transfer size, the average speedup over all DBMSs is 3.2×, with maxi-

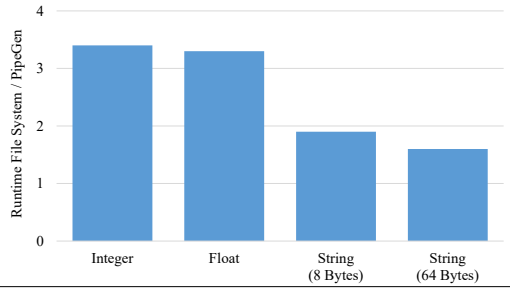


Figure 10. Overall speedup between file system and PipeGen transfer for different data types and sizes. Each transfer moves $4 \cdot 10^8$ elements of the given data type.

# Workers	1	4	8	16
Speedup	3.1	3.7	3.5	3.7

Table 1. Overall speedup (file system / PipeGen runtime) from Myria to Spark for $4 \cdot 10^8$ elements when varying the number of workers and tasks involved in the transfer.

mum speedup up to 3.8×. This speedup is approximately the same across all transfer sizes and pairs of DBMSs. As shown in Table 1, this speedup is also similar for various cluster sizes. In this experiment, we compare PipeGen with using local file system. As we show later, going through HDFS is even slower (see Figure 11).

This result emphasizes the impact that PipeGen can have on hybrid data analytics: without writing a single line of code, a user can get access to 20 optimized data pipes and speed up transfers between any combination of the five systems tested (by 3.2× on average). PipeGen produces this benefit automatically without requiring that developers agree on an efficient common data format or have access to the DBMS source code.

Meanwhile, the magnitude of the benefit does depend on the types of data transferred. Figure 10 shows the speedup between the file system and PipeGen for $4 \cdot 10^8$ elements of various data types. As the figure shows, transfer performance for fixed-width primitives is significantly better than string transfers due to the smaller amount of data transferred when using `AStrings`. While strings do not benefit from our optimizations, they still benefit from avoiding serializing to the file system.

6.2 Optimizations

Next, we drill down into the different components of the speedup afforded by PipeGen's data pipes. In this section, we evaluate the performance benefits of FormOpt's optimizations, which convert text-formatted data to binary after removing delimiters and metadata.

6.2.1 String Decoration

We first evaluate the optimizations due to using `AStrings` as described in Section 5.1. Figure 11 shows the performance of an export between Myria and Giraph. For this pair of DBMSs, FormOpt's optimizations are responsible for approximately one third of the runtime benefit beyond what IORedirect already provides.

To assess the benefits of avoiding both text-encoding and delimiters, we also compare the performance against a manually-constructed data pipe that transmits binary data and removes delimiters. To produce the manually constructed pipes, we modify each DBMS to directly transmit/receive results to/from a network socket and remove logic related to text-encoding that might degrade performance. We then transfer data in both directions and measure the total runtime. Overall, the PipeGen-generated data pipes perform closely to their manually-optimized counterparts. Transferring from Myria to Giraph is slower due to Giraph's import implementation, which materializes `AString` instances into character strings and escapes individual characters.

6.2.2 Library Extensions

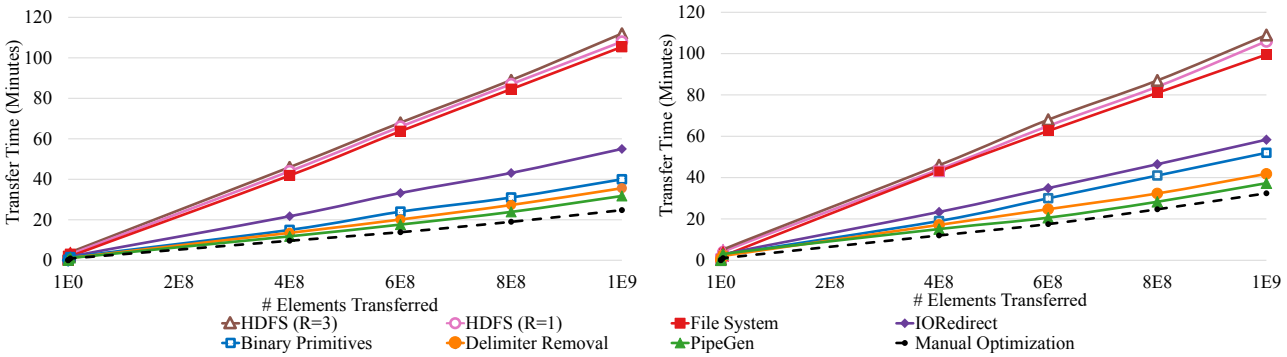
We implemented a library extension implementation for the Jackson JSON library and evaluate its performance under the library extension mode of FormOpt. Interestingly, for the pairs of DBMSs that we examine, most do not support mutually-compatible exchange using JSON as an intermediate format. Myria produces a single JSON document, Spark and Giraph both expect a document-per-line, and Derby does not natively support bulk import and export of JSON at all.

Figure 12 shows the performance of using library extensions with Jackson between Spark and Giraph. We use a mutually-compatible JSON adjacency-list format for the schema of transmitted data. We find that the relative performance benefit closely matches that of the string decorations.

6.3 Intermediate Format

Once FormOpt captures the transferred data in an `AString`, PipeGen can use any intermediate format to transfer the data between DBMSs. As expected, the choice of that intermediate format significantly impacts performance. This observation is important as a key contribution of PipeGen is to free developers from the need to add new data import and export code to the DBMS every time a new data format becomes available.

Our experiments include two third-party formats: protocol buffers [20] and Arrow [6]. We also evaluate the basic custom format from the previous section, which transmits schema information as a header, values in binary form, and uses length-prefixing for strings. We examine protocol buffers using a version where message templates are fixed at compile time and another where they are dynamically constructed. Figure 13 shows the results. Protocol buffers, depending on whether message formats are statically or dynamically generated, perform approximately as well as PipeGen's custom binary format. Arrow offers a substantial boost in performance due primarily to its optimized layout and efficient allocation and iteration process [6]. The column-oriented format offers a further modest advantage over its row-oriented counterpart.



(a) Giraph→Myria DBMS (b) Myria DBMS →Giraph

Figure 11. Transfer time between the Myria and Giraph using PipeGen and a manually-optimized variant. In (a) we export tuples from Myria and import them as vertices in Giraph. In (b) we reverse the direction of transfer. We show as baseline transfer through the file system, and HDFS with replication factors of 1 and 3. We then activate PipeGen optimizations as follows. First, we apply the IORedirect component. Next, we transmit fixed-width values in binary form. We then activate delimiter removal. The PipeGen series shows all optimizations, which additionally include column pivoting.

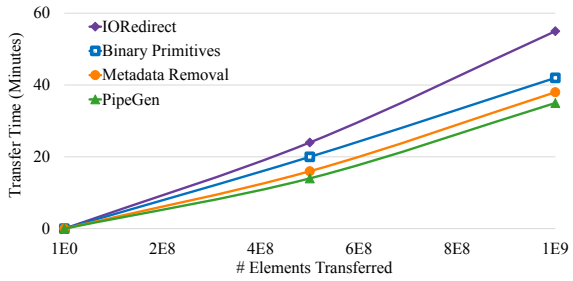


Figure 12. Runtime of a transfer from Spark to Giraph using the library extension mode for the Jackson JSON library. We show a baseline application of only IORedirect. We then transmit fixed-width values in binary form. Next, we remove repeated column names and delimiters. The PipeGen series shows these optimizations plus column pivoting (i.e., all optimizations).

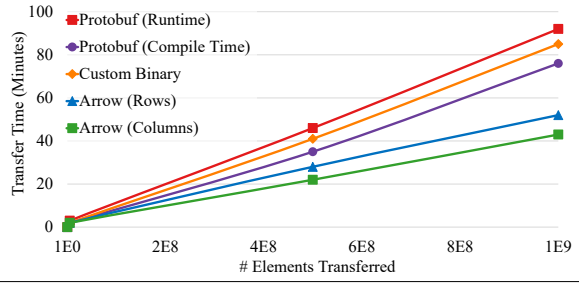


Figure 13. Transfer performance by intermediate format between Hadoop and Spark. Message templates for protocol buffers were generated both at compile time and dynamically at runtime.

Overall, Arrow yields the highest performance as an intermediate format. Since we preallocate a buffer when pivoting blocks of data into a columnar format, we must select an appropriate size for this intermediate buffer. In Figure 14 we show transfer performance between Myria and Giraph for various `ArrowBuf` sizes. Note that since Arrow is column-oriented, we allocate one buffer for each attribute. As long as the buffer is not too small, the buffer size has only a negligible impact on performance.

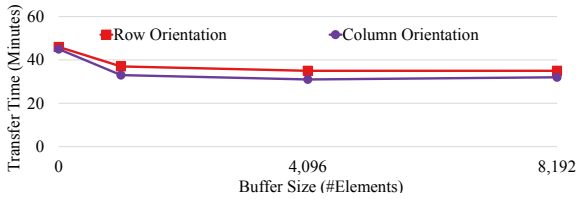


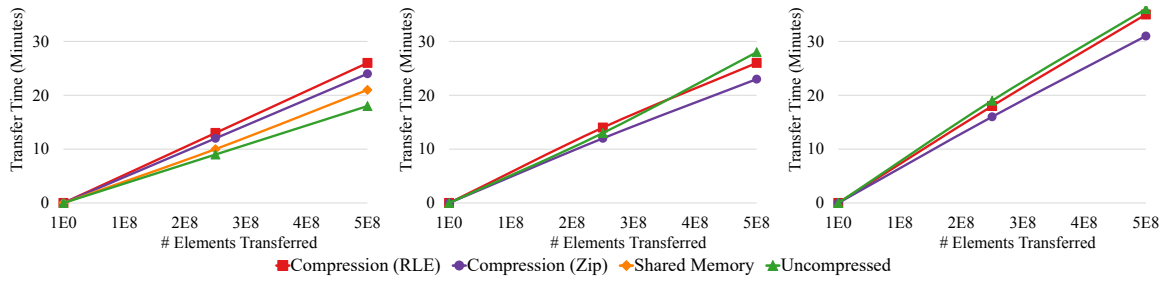
Figure 14. Transfer time from Myria to Giraph for 10^9 elements with various Arrow buffer sizes. Each column buffer was sized to hold the number of values listed on the x-axis.

6.4 Compression & Inter-cluster Transfer

Orthogonal to the previous choices is PipeGen's ability to compress the data transferred between pairs of DBMSs. Utility of this approach depends on the geographical distance between DBMS workers, with nearby DBMSs being less likely to benefit than distant ones.

Figure 15 shows Myria-to-Giraph transfer performance when applying compression. We show three techniques: run-length encoding (RLE), dictionary-based compression (zip), and uncompressed transfer. We separately show transfer performance for colocated workers (Figure 15(a)), workers with a 40ms artificial latency introduced into the network adapter (Figure 15(b)), and workers located in different data centers with ∼100ms latency (Figure 15(c)). For colocated workers, we also show transfer performance using shared memory; all other transfers utilize sockets.

For colocated nodes, both compression techniques add modest overhead to the transfer process that yields a loss in performance. For nodes with higher latency, our results shows a modest benefit for dictionary-oriented compres-



(a) Colocated workers (b) 40ms artificial latency (c) Inter-datacenter, ∼100ms latency

Figure 15. Transfer time between Myria and Giraph using compression. We show transfer for (a) workers on the same physical node, (b) 40ms artificially-introduced latency, and (c) separate datacenters (approximately 100ms latency). For RLE, zip, and uncompressed formats we transfer data over a socket. For colocation we also show transfer using uncompressed shared memory.

sion. This suggests that this strategy may be beneficial for physically-separated DBMSs.

6.5 Code Modifications

Table 2 shows modifications made by the IORedirect and FormOpt components in terms of the number of classes and lines of code affected. These results show that the number of changes is modest across all of the DBMSs we evaluate. The total number of modifications is small for the IORedirect component, suggesting that DBMS implementations rarely open an import or export file at more than one call site. The FormOpt component modifies more, with library extension requiring fewer changes than string decoration. Even with string decoration, for the DBMSs that we evaluate, primitive values are converted to/from a string in close proximity in the code where they are written/read. This reduced the number of modifications required for this optimization.

Mode	DBMS	Execution	IORedirect		FormOpt	
		Time (sec)	#Classes	LOC	#Classes	LOC
String Decoration	Hadoop	245	3	6	6	36
	Myria	160	2	8	5	54
	Giraph	223	2	9	4	47
	Spark	187	5	18	8	38
	Derby	130	2	5	2	67
Lib. Extension	Spark	178	5	18	2	6

Table 2. Number of classes and lines of code modified by the IORedirect and FormOpt phases and execution time.

7. Related Work

PipeGen automatically embeds data pipes into a DBMS. Previous work in data integration has explored similar manual embedding for data transfer. For example, Rusinkiewicz et al. proposed a common transfer format mediated via STUB operators [38]. These operators are similar to data pipes, but are manually generated and do not address direct transfer. Prior work has also explored automatic generation of transformation operators that are similar to data pipes. For example, Mendell et al. used code generation under an XML streaming engine [30]. Other tools target automatic parsing of semi-structured data for ad hoc processing [18]. While these efforts shares some commonality with PipeGen, they do not address data transfer performance between DBMSs.

Similarly, data exchange involves transfer between DBMSs. However, while work exists related to generating mappings between schemata across heterogeneous systems [15], optimization has focused primarily on inference performance [22, 39], or does not address data shipping performance [37]. In contrast, PipeGen focuses on optimizing this and assumes that a user (or optimizer) can generate queries that reconcile schemata.

One concurrent effort [13] investigates optimizations that select data transfer formats based on the source and target engines, query, data, and cluster configuration. This line of work could serve to further improve the performance of the data pipes that PipeGen generates (e.g., by using it instead of Apache Arrow buffers).

Finally, PipeGen's IORedirect component redirects specific file system calls to sockets. This is similar to previous work such as the CDE tool, which uses `ptrace` for redirection [21]. Similar work uses redirection for prototyping [42], sandboxing [51] and information flow analysis [53].

8. Conclusion

In this paper we described PipeGen, a tool that automatically generates efficient data pipes between DBMSs. PipeGen leverages DBMSs' support for transmitting data via a common data format and unit test cases to create data pipes from existing bytecode. As future work, we plan to target other common formats (e.g., Apache Parquet) and apply our techniques to non-Java systems.

We implemented a prototype of PipeGen and evaluated it by generating data pipes across relational and graph-based DBMSs. Our experiments show that the generated data pipes can speed up data transfers by up to 3.8× as compared to transferring data via the file system.

Acknowledgments

This work is supported in part by the National Science Foundation through grants IIS-1247469, IIS-1110370, IIS-1546083, and CNS-1563788; DARPA award FA8750-16-2-0032; DOE award DE-SC0016260; and gifts from the Intel Science and Technology Center for Big Data, Adobe, Amazon, Facebook, and Google.

References

[1] D. Agrawal, M. Lamine, L. Berti-Equille, S. Chawla, A. Elmagarmid, H. Hammady, Y. Idris, Z. Kaoudi, Z. Khayyat, S. Kruse, M. Ouzzani, P. Papotti, J. Quiane, N. Tang, and M. Zaki. Rheem: Enabling multi-platform task execution. In *SIGMOD*, 2016.

[2] L. Andersen. Jdbc 4.2. Technical Report JSR 221, Oracle, March 2014.

[3] Apache Software Foundation. Derby. `https://db.apache.org/derby`, 2015.

[4] Apache Software Foundation. Hadoop. `https://hadoop.apache.org`, 2015.

[5] Apache Software Foundation. Apache Commons CSV. `https://commons.apache.org/proper/commons-csv/`, 2016.

[6] Apache Software Foundation. Apache arrow. `https://arrow.apache.org/`, 2016.

[7] Apache Software Foundation. Apache Thrift. `https://thrift.apache.org/`, 2016.

[8] C. Avery. Giraph: Large-scale graph processing infrastructure on Hadoop. In *Hadoop Summit*, Santa Clara, 2011.

[9] S. Chiba. Load-time structural reflection in java. In *ECOOP*, pages 313–336. Springer, 2000.

[10] Chiba, S. Javassist. `http://www.javassist.org`.

[11] D. Crockford and T. Bray. The JavaScript object notation (JSON) data interchange format. *IETF RFC*, 7159:1–15, 2006.

[12] D. J. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. Split query processing in Polybase. In *SIGMOD*, pages 1255–1266, 2013.

[13] A. Dziedzic, A. Elmore, and M. Stonebraker. Data Transformation and Migration in Polystores. In *HPEC*. IEEE, 2016.

[14] A. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, U. Cetintemel, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska, et al. A demonstration of the BigDAWG polystore system. *VLDB*, 8(12):1908–1911, 2015.

[15] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

[16] FasterXML. Jackson JSON Processor. `http://wiki.fasterxml.com/JacksonHome/`, 2016.

[17] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, pages 229–239, 2015.

[18] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From dirt to shovels: Fully automatic tool generation from ad hoc data. In *POPL*, page 421, 2008.

[19] I. Gog, M. Schwarzkopf, N. Crooks, M. P. Grosvenor, A. Clement, and S. Hand. Musketeer: all for one, one for all in data processing systems. In *EuroSys*, page 2, 2015.

[20] Google. Protocol Buffers. `https://developers.google.com/protocol-buffers/`, 2016.

[21] P. J. Guo and D. R. Engler. CDE: Using system call interposition to automatically create portable software packages. In *USENIX ATC*, 2011.

[22] L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth. Clio grows up: from research prototype to industrial tool. In *SIGMOD*, page 805, 2005.

[23] D. Halperin, V. T. de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, et al. Demonstration of the Myria big data management service. In *SIGMOD*, pages 881–884, 2014.

[24] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. Quinn. Massively parallel cosmological simulations with ChaNGa. In *IPDPS*, pages 1–12. IEEE, 2008.

[25] V. Josifovski, P. Schwarz, L. Haas, and E. Lin. Garlic: a new flavor of federated query processing for DB2. In *SIGMOD*, pages 524–532, 2002.

[26] A. Knebe, F. R. Pearce, H. Lux, Y. Ascasibar, P. Behroozi, J. Casado, C. C. Moran, J. Diemand, K. Dolag, R. Dominguez-Tenreiro, et al. Structure finding in cosmological simulations: the state of affairs. *MNRAS*, 435(2): 1618, 2013.

[27] H. Lim, Y. Han, and S. Babu. How to fit when no one size fits. In *CIDR*, volume 4, page 35, 2013.

[28] F. Lin and W. W. Cohen. Power iteration clustering. In *ICML*, page 655, 2010.

[29] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *VLDB*, 5(8): 716–727, 2012.

[30] M. Mendell, H. Nasgaard, E. Bouillet, M. Hirzel, and B. Gedik. Extending a general-purpose streaming system for XML. In *EDBT*, page 534, 2012.

[31] Myria: Big Data Management as a Cloud Service. `http://myria.cs.washington.edu/`.

[32] P.-M. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, pages 619–630, 2015.

[33] F. Özcan, D. Hoa, K. S. Beyer, A. Balmin, C. J. Liu, and Y. Li. Emerging trends in the enterprise data analytics: Connecting Hadoop and DB2 Warehouse. In *SIGMOD*, pages 1161–1164, 2011.

[34] A. Pan, J. Raposo, M. Álvarez, P. Montoto, V. Orjales, J. Hidalgo, L. Ardao, A. Molano, and Á. Viña. The Denodo data integration platform. In *VLDB*, pages 986–989, 2002.

[35] D. Perelman, S. Gulwani, D. Grossman, and P. Provost. Test-driven synthesis. In *PLDI*, pages 408–418, 2014.

[36] M. Raza, S. Gulwani, and N. Milic-Frayling. Compositional program synthesis from natural language and examples. In *ICAI*, pages 792–800, 2015.

[37] T. Risch, V. Josifovski, and T. Katchaounov. Functional data integration in a distributed mediator system. In *The Functional Approach to Data Management*, pages 211–238. Springer, 2004.

[38] M. Rusinkiewicz, K. Loa, and A. K. Elmagarmid. Distributed operation language for specification and processing of multidatabase applications. 1988.

[39] K. Saleem, Z. Bellahsene, and E. Hunt. Porsche: Performance oriented schema mediation. *Information Systems*, 33(7):637–657, 2008.

[40] J. Sirosh. Microsoft acquires Metanautix to help customers connect data for business insights. http://blogs.microsoft.com/blog/2015/12/18/microsoft-acquires-metanautix-to-help-customers-connect-data-for-business-insights/, 2016.

[41] C. Smith and A. Albarghouthi. Mapreduce program synthesis. In *PLDI*, pages 326–340, 2016.

[42] R. P. Spillane, C. P. Wright, G. Sivathanu, and E. Zadok. Rapid file system development using ptrace. In *ExpCS*, page 22, 2007.

[43] M. Stonebraker. ACM SIGMOD blog: The case for polystores. http://wp.sigmod.org/?p=1629.

[44] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-Store: a column-oriented dbms. In *VLDB*, pages 553–564, 2005.

[45] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. The architecture of SciDB. In *SSDBM*, pages 1–16, 2011.

[46] X. Su and G. Swart. Oracle in-database Hadoop: when MapReduce meets RDBMS. In *SIGMOD*, pages 779–790, 2012.

[47] Sun Microsystems. BTrace. https://kenai.com/projects/btrace, 2016.

[48] Turi. Spark unity codebase. https://github.com/turi-code/spark-sframe, 2015.

[49] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON*, page 13, 1999.

[50] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache Hadoop YARN: Yet another resource negotiator. In *SOCC*, page 5, 2013.

[51] D. Wagner, I. Goldberg, and R. Thomas. A secure environment for untrusted helper applications. In *USENIX Security*, 1996.

[52] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, page 2, 2012.

[53] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *OSDI*, pages 263–278, 2006.