DIBS: A Data Integration Benchmark Suite

Anthony M. Cabrera, Clayton J. Faber, Kyle Cepeda, Robert Derber, Cooper Epstein, Jason Zheng, Ron K. Cytron, and Roger D. Chamberlain
Washington University in St. Louis, St. Louis, Missouri, USA
{acabrera,cfaber,kcepeda,derber,cepstein,jasondzheng,cytron,roger}@wustl.edu

ABSTRACT

As the generation of data becomes more prolific, the amount of time and resources necessary to perform analyses on these data increases. What is less understood, however, is the data preprocessing steps that must be applied before any meaningful analysis can begin. This problem of taking data in some initial form and transforming it into a desired one is known as data integration. Here, we introduce the Data Integration Benchmarking Suite (DIBS), a suite of applications that are representative of data integration workloads across many disciplines. We apply a comprehensive characterization to these applications to better understand the general behavior of data integration tasks. As a result of our benchmark suite and characterization methods, we offer insight regarding data integration tasks that will guide other researchers designing solutions in this area.

CCS CONCEPTS

• **General and reference → Performance**;

KEYWORDS

Big data, data integration, data wrangling

ACM Reference Format:
Anthony M. Cabrera, Clayton J. Faber, Kyle Cepeda, Robert Derber, Cooper Epstein, Jason Zheng, Ron K. Cytron, and Roger D. Chamberlain. 2018. DIBS: A Data Integration Benchmark Suite. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering Companion , April 9–13, 2018, Berlin, Germany*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3185768.3186307

1 INTRODUCTION

Generating and analyzing big data are tasks encountered by many researchers in various disciplines. Social networks, computational biology, sensor data, and entrepreneurial records are just a small sample of the range of applications that encounter extensive data streams. It is generally well understood that big data is voluminous and prevalent in the research and industrial communities alike, and there is often a non-trivial amount of time, effort, and resources that are spent retrieving and preprocessing big data.

This problem of taking data in some initial form and transforming it into a desired one comes in several flavors. It might involve field rearranging, altering boundary notations, encryption, parsing non-record data and organizing it into a record-oriented form, etc. We define this problem, collectively, as *data integration*. In the business community, this is part of the Extract, Transform, and Load (ETL) process, specifically the transform step. Other phrases that are often used are data cleansing and pre-analytics. Frequently, the challenges here include parsing of the data to extract what structure does exist (e.g., click streams) and text processing to address unstructured data (e.g., blog posts). While the individual transforms are each mostly straightforward, the task is quickly complicated by voluminous data streams that require distinct transformation specifications. Tens to hundreds of multi-Gigabyte data streams must be concurrently integrated prior to actually doing any of the real data analysis, the ultimate goal.

Data integration manifests itself in many preliminary steps taken before analyzing any data. As an example, consider the needs of a researcher in biosequence analysis. Genomic and proteomic data sets are available from a wide variety of sources in a large number of disparate formats (e.g., FASTA, FASTQ, SAM, BAM, AB1/SCF, PDB, GTF, etc.). The data volumes are sufficiently large that simply transforming the data from its original form into that needed for analysis is becoming time prohibitive (e.g., three days are required to perform duplicate marking, base score quality recalibration, and local realignment on a 250 GB BAM file at 30× coverage [5]).

While there are a number of ways to organize data integration applications, we will consider an individual data integration job to be decomposed into one or more of the following three tasks:

- Parsing/Cleansing – the computation associated with recognizing the records, fields, and/or other components of the input data, including checking to see if it is well-formed and addressing any example inputs that aren't well-formed.
- Transformation – once parsed, the input data must be translated into the form expected by the primary computation, typically going from a file-oriented format to a memory-oriented format.
- Aggregation – any pre-analytics computations that result in aggregate information about the input.

We present the Data Integration Benchmark Suite (DIBS), a set of applications spanning several different application domains and the above three types of data integration tasks. DIBS tries to be reasonably comprehensive with respect to both applications and tasks. To help us address how comprehensive they truly are, the benchmarks are characterized through different measures in order to capture the properties (and idiosyncrasies) across the various data integration tasks represented in the suite. The goals of DIBS are to provide insight into the the nature of data integration tasks to guide research in this area, and to create a way in which different research groups can compare their work and improve performance.

ICPE '18, April 9–13, 2018, Berlin, Germany

ACM ISBN 978-1-4503-5629-9/18/04…$15.00
https://doi.org/10.1145/3185768.3186307

2 BACKGROUND AND RELATED WORK

The data integration problem has received considerable attention already in the research community. Quoting from Kandel et al. [6], "In spite of advances in technologies for working with data, analysts still spend an inordinate amount of time diagnosing data quality issues and manipulating data into a usable form. This process of 'data wrangling' often constitutes the most tedious and time-consuming aspect of analysis." Dasu and Johnson indicate that data reformatting and cleaning accounts for up to 80% of the development time and cost in data warehousing projects [3].

Customized domain specific languages and graphical user interfaces exist that are designed explicitly for describing data transformation workflows. Examples include Potter's Wheel [11] and Wrangler [7]. In addition, there are commercial systems available both to specify and execute workflows, e.g., IBM's InfoSphere and Informatica. There are also many systems aimed at scientific data (e.g., see [4, 9]). While there is significant disparity of data formats in many disciplines, other disciplines have a stronger culture of data description via XML and semantic ontologies, enabling a higher degree of automation in the specification of data transformations.

Our interest is in helping research groups compare and improve implementations of systems that execute data transformations by providing a baseline implementation and its accompanying characterization. The classic way to do this is via a benchmark suite. Poess et al. [10] have developed an enterprise-centric data integration benchmark, but do not speak to the more general data integration audience. To the best of our knowledge, we present the first benchmark suite that broadly characterizes data integration tasks.

3 OVERVIEW OF BENCHMARK SUITE

The challenges in selecting candidate applications for any benchmark suite include whether or not the candidates that are ultimately included are both representative of the field and comprehensive in their coverage of the field. To help us assure that the selected applications are representative, we consider each application across two dimensions. First, we want to capture the breadth of application domains that handle large volumes of data. Second, we include tasks that span the three composite parts of data integration.

To address this, we have selected data integration applications from a variety of disciplines that span different dimensions (i.e., 1- and 2-D data) and tasks. The relationship between the five application domains, types of integration tasks, and elements included in the benchmark are all summarized in Table 1. In all cases, the data integration applications are written in C and the input data set size is large enough such that any second-order effects caused by start-up transients can be ignored.

To assess the extent to which the benchmark suite provides comprehensive coverage, we will rely primarily on the distribution of the properties of the applications, described in Section 4 below.

4 CHARACTERIZATION

In determining what attributes to choose to characterize our benchmark suite, we want to address two specific things. The first is choosing an analysis that allows for a comprehensive look at the suite through many characteristic dimensions to capture the overall behavior of each task and capture any idiosyncrasies associated with them. Second, we wish to craft an analysis that is independent of the system that our suite is deployed on. From this, we have chosen to characterize the applications based on spatial/temporal locality, branch entropy, and instruction mix.

	Data Integration Tasks		
Domain	**Parsing/Cleansing**	**Transformation**	**Aggregation**
Computational Biology	Separate bases and meta-data Handle non-A,T,G,C bases	fa→2bit 2-bit→fa	Track total size
Image Processing	Parse FITS tags	fits→tiff idx→tiff optdigits→tiff unipen→tiff	Pixel statistics Histogram Taking log of pixels
Enterprise	Adjust non-ASCII characters	ebcdic→txt fix→float	Count number of elements
Internet of Things	Tokenize input	tstcsv→csv gotrackcsv→csv plt→csv	Running total of file size
Graph Processing	Parse edge list	edgelist→csr	Get total vertex/edge count Compute vertex edge degree

Table 1: Data Integration Task Classification.

Measures of locality allow us to examine the behavior of a program's memory access patterns. Qualitatively, a program's spatial locality is described by whether or not subsequent memory references will be located near previous memory accesses. Higher spatial locality is generally beneficial to performance because it allows contiguous chunks of data to exist in caches with less thrashing and evictions. In order to quantify this in an architecturally independent manner, we draw from work from Weinberg et al. [12]. In this metric they describe the stride of a memory access as the difference between two memory reference addresses in units of a 64-bit word size and quantify spatial locality in the following way:

$$L_{Spatial} = \sum_{i=1}^{\infty} \frac{stride_i}{i} \quad (1)$$

where $stride_i$ is the total number of memory accesses that are of stride length i. The result of this expression is a normalized score in the range [0,1] that can be used to compare the spatial locality between programs.

Temporal locality is a characteristic of a program's memory access pattern that describes the frequency of memory accesses to the same memory location. Higher temporally local programs reference the same memory locations numerous times which positively affects performance. To quantify temporal locality, we again draw from Weinberg et al. [12] and examine data reuse. Given a particular memory address, data reuse is the number of unique memory addresses that have been accessed before that particular memory address is referenced again. The formula used to quantify temporal locality is shown below:

$$L_{Temporal} = \frac{\sum_{i=0}^{\log_2(N)-1} \left[\left(reuse_{2^{i+1}} - reuse_{2^i}\right) \times \left(\log_2(N) - i\right) \right]}{\log_2 N} \quad (2)$$

where $reuse_{2^i}$ is the number of dynamic memory accesses with reuse distance less than or equal to 2^i and N is the largest reuse distance used. This metric also produces a score within the range [0,1] with which to compare temporal locality scores of programs.

The predictability of a program's control flow can be characterized by the regularity of the program's branching behavior during execution. Regularity in a program's control can dictate the performance of a program on an underlying architecture. Strong regularity in control behavior allows for more confident branch predictions. To quantify branching behavior, we draw from work

CPU	Intel Core i7 930
Clock rate	2.8 GHz
L1 d-cache size	32 KB
L1 i-cache size	32 KB
L2 cache size	256 KB
L3 cache size	8192 KB
RAM size	24 GB
Compiler	GCC 4.8.4
ISA	x86-64

Table 2: Experimental machine specifications (left).

Application	Throughput (MB/s)
fa→2bit	23.4
2bit→fa	12.2
fits→tiff	43.8
idx→tiff	13.2
optdigits→tiff	133
unipen→tiff	0.113
ebcdic→txt	139
fix→float	204
tstcsv→csv	75.8
gotrackcsv→csv	104
plt→csv	123
edgelist→csr	40

Table 3: Throughput results (right).

done by Yokota et al. [13] regarding branch entropy. Specifically, we use their formulation for table reference entropy, based on the values that a pattern history register assumes. The pattern history register acts as a shift register that either shifts in a 1 or 0 for a branch that is taken or not taken, respectively. In this case, we will make the shift register 16 bits in length. A table reference entry, then, is a resulting 16-bit value that the pattern history register takes after it is updated by a branching decision. The formula for the branch entropy metric is shown below:

$$BE = -\sum_i p(E_i) \log_2 p(E_i) \tag{3}$$

where E_i is the i-th entry of the table, and $p(E_i)$ is the probability of E_i occurring.

The instruction mix of a program is a measure of the unique instruction classes the program contains and the distribution of those instructions during execution. We classify instructions in three categories: compute, control flow/branch, and data movement. In our benchmark suite, we examine dynamic instruction mix, which is the count of how many times the aforementioned classes of machine instructions were executed. This metric can reveal hotspots of a program's execution and characterize the execution-time leanings of the program, for example, mostly compute or an equal combination of data movement and compute.

5 EXPERIMENTAL METHODS

In this work, we are only considering program execution on a single core and the working set size fits in the experimental machine's RAM. We leave to future work the deployment on multiple cores and heterogeneous/distributed systems.

To measure locality, branch entropy, and instruction mix, we use a dynamic binary instrumentation framework called Pin for IA-32, x86-64, and MIC ISAs that allows us to dynamically instrument our applications[8]. Thus, instrumentation is performed at run time, which captures dynamic application behaviors. The Pin framework allows us to mostly perform architecturally independent analyses of each program in our benchmark suite.

Since instruction mix is inherently architecture dependent, we compile all programs with the default level of optimization in GCC to prevent the exploitation of architecturally specific features at compile time. We also use a coarse categorization of instruction types to abstract away details of differing architectures. Since x86-64 is a CISC instruction set, some instructions do have implicit

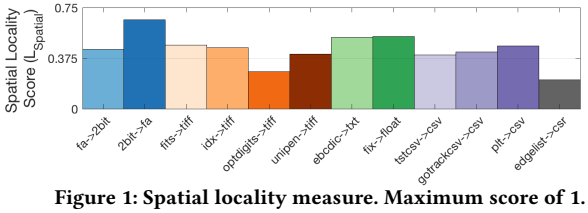

Figure 1: Spatial locality measure. Maximum score of 1.

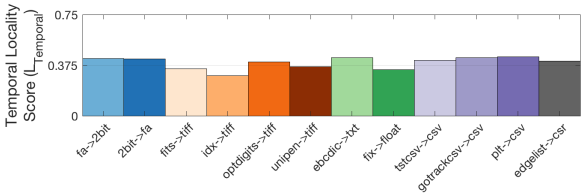

Figure 2: Temporal locality measure. Maximum score of 1.

data movement within an instruction. In these cases, we categorize the instruction based on its main function. For a subset of the applications, we provide instruction mix data for the ARM AArch64 instruction set. Since the AArch64 instruction set is not supported by the Pin framework, we use the `gem5` simulation environment [1] and cross-compile our applications to make this characterization.

To provide baseline performance numbers, we executed each benchmark application on a single core of the machine whose properties are shown in Table 2. The throughput rates ($\frac{input\ data\ size}{execution\ time}$) are shown in Table 3. In each case, the average over 100 runs is reported. These reported rates are below that achievable in a modern I/O subsystem, therefore warranting investigation of their computational performance properties.

6 RESULTS AND DISCUSSION

Spatial and temporal locality results are shown in Figures 1 and 2. Since most data records are located next to each other in memory, we expected that most programs exhibit high $L_{Spatial}$. Since data are effectively streamed, we expected $L_{Temporal}$ be generally low. Although $L_{Spatial}$ and $L_{Temporal}$ across the suite are not as high and low, respectively, as originally posited, they are consistent relative to each other. This supports the notion that most data integration applications do have a uniform degree of locality. The applications in which this does not fully hold allows this characterization to capture the idiosyncrasies associated with those tasks, which adds to the insight and comprehensiveness.

Decomposing the cumulative sums used to calculate $L_{Spatial}$ for each task, we observe that 75% of memory references occur within a stride of 80 bytes for 10 of the 12 data integration applications. Eight of the 12 applications have higher spatial locality than temporal locality. Decomposing the temporal locality scores, we observe that these integration tasks show minimal data reuse at smaller reuse distances which explains the lower temporal scores. Future approaches to performance improvement could be tailored to either exploit the current degree of locality or increase the locality present.

Figure 3 shows the branch entropy results. We observe that there is a wide swing in branch entropy, which speaks to the variability of each application with respect to control flow. However, the applications execute with some degree of control flow regularity, since no application in the benchmark suite exceeds 8 bits, where the max is 16. Achieving the maximum value of branch entropy

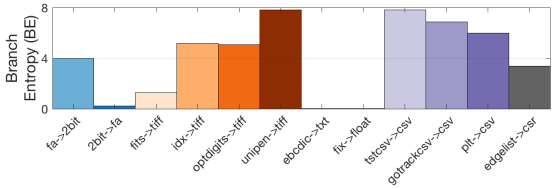

Figure 3: Branch entropy measure. Maximum score of 16.

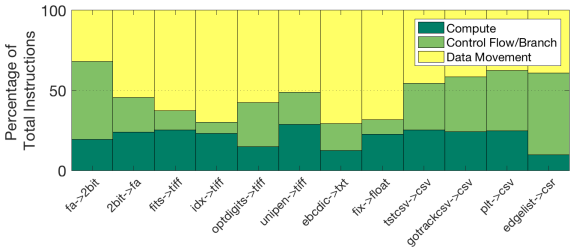

Figure 4: x86-64 dynamic instruction mix.

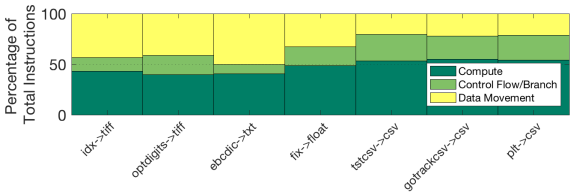

Figure 5: AArch64 dynamic instruction mix.

requires an equal probability of occurrence of all possible entries in the pattern history register. This is not the case in our application suite because at their core, all integration tasks take the form of iterating over a set of data records. This result has implications for how complex the branch predicting algorithms and hardware of a particular system needs to be for data integration tasks.

Figure 4 shows the results for the dynamic instruction mix characterization. One observation is that the instruction mix varies fairly significantly across the suite; an indication that the choice of applications is comprehensive. Additionally, we see the prominence of data movement instructions in each integration task. 8 of 12 tasks are comprised of 50% or more data movement instructions, noting that this figure could be even larger but was limited as a result of how we binned complex instructions. This follows from the fact that our benchmark suite is comprised of data-related operations. Since data movement is expensive and time-consuming, this presents opportunities towards minimizing data movement or exploring processing-in or near-memory approaches.

Figure 5 shows the results for 7 of the 12 benchmark applications when compiled to the ARM instruction set. An important observation is that these results are distinct from those reported in Figure 4, confirming that this characterization is at least somewhat architecture dependent. This necessitates a review of performance when porting and compiling data integration codes across different architectures because optimizations for one architecture may not exhibit the same gains on another architecture. Second, the fraction of data movement instructions is noticeably smaller (especially for a few of the applications), which could easily be due to the distinction between the RISC and CISC instruction set styles.

Our vision is that the future of computer architecture is heading to a situation where increased customization of execution platforms is available (e.g., big.LITTLE, reconfigurable logic accelerators, etc.). This implies that application developers can more effectively exploit properties of their applications for performance gains. Currently, we are investigating how to model performance given the resulting characterization, total number of instructions, ISA, and the relationships between these parameters in order to yield theoretical estimates of throughput and quantify the effects of each parameter. Ideally, such a model could guide the design of systems that have higher performance on data integration, generally.

7 CONCLUSIONS

The Data Integration Benchmark Suite (DIBS) is a set of data integration applications that is representative of various disciplines and integration tasks. We explore the general qualities and idiosyncrasies of the suite by applying a comprehensive and (mostly) architecturally-independent characterization to each application. From this, we observe that most data integration tasks have a consistent level of both spatial and temporal locality and usually exhibit higher spatial locality. The applications also exhibit high degrees of control flow regularity and data movement. The characterization of the applications is only the first step of a work-in-progress. We anticipate that the insights gained from our characterizations will guide both software and hardware research in exploring and exploiting the qualities of data integration tasks to improve performance. Finally, we have made these applications and datasets publicly available [2] so that researchers can compare data integration-specific solutions and systems.

ACKNOWLEDGEMENTS

This work was supported by the NSF under grant CNS-1527510.

REFERENCES

[1] Nathan Binkert et al. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7.
[2] Anthony Cabrera et al. 2018. Data Integration Benchmark Suite V1. https://doi.org/10.7936/K7NZ8715. (April 2018).
[3] Tamraparni Dasu and Theodore Johnson. 2003. *Exploratory Data Mining and Data Cleaning*. John Wiley & Sons, Inc.
[4] Ewa Deelman et al. 2005. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming* 13, 3 (2005), 219–237.
[5] Matt Massie et al. 2013. *ADAM: Genomics Formats and Processing Patterns for Cloud Scale Computing*. Technical Report UCB/EECS-2013-207. UC Berkeley.
[6] Sean Kandel et al. 2011. Research directions in data wrangling: Visualizations and transformations for usable and credible data. *Information Visualization* 10, 4 (Oct. 2011), 271–288.
[7] Sean Kandel et al. 2011. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *Proc. of SIGCHI Conf. on Human Factors in Computing Systems*. 3363–3372.
[8] Chi-Keung Luk et al. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*. 190–200.
[9] Tom Oinn et al. 2006. Taverna: Lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience* 18, 10 (2006), 1067–1100.
[10] Meikel Poess et al. 2014. TPC-DI: The first industry benchmark for data integration. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1367–1378.
[11] Vijayshankar Raman and Joseph M. Hellerstein. 2001. Potter's Wheel: An Interactive Data Cleaning System. In *Proc. of 27th Int'l Conf. on Very Large Data Bases*. 381–390.
[12] Jonathan Weinberg et al. 2005. Quantifying locality in the memory access patterns of HPC applications. In *Proc. of ACM/IEEE Supercomputing Conference*.
[13] Takashi Yokota, Kanemitsu Ootsu, and Takanobu Baba. 2008. Potentials of branch predictors: From entropy viewpoints. In *Proc. of International Conference on Architecture of Computing Systems*. 273–285.