Benchmarking of Big Data Architecture Trade-Offs

Tim Horgas

Hamburg University of Applied Sciences, Dept. Computer Science, Berliner Tor 7, 20099 Hamburg, Deutschland,
tim.horgas@haw-hamburg.de

Abstract. Big data technology influences current data management and software stacks. Different patterns for building data pipelines are proposed to address the challenges brought by big data. This paper describes current big data architecture patterns and it presents benchmarking solutions to analyze arising trade-offs when implementing big data pipelines.

Keywords: big data, architecture trade-offs, benchmarking

1 Introduction

In the past years the number of use cases and applications driven by big data increased [1]. Social media and other trends like Internet of Things are producing huge amounts of data which can be stored and analyzed relatively cheap due to the still decreasing prices of storage and computation [1],[2]. Furthermore, there is a considerable number of open source software available for analyzing the data (often referred as "Hadoop ecosystem"). This encourages developers and entrepreneurs to build data pipelines and model new business cases based on data.

The architecture of data pipelines and the configuration of individual frameworks is one of the main challenges of current software stacks which are able to store and process big data [3],[4]. The integration of different types of processing systems (for example batch and streaming applications) in a big data architecture requires measurement of performance to choose the best system for the specified use case. Examples for this kind of integration patterns are the Lambda architecture [5], the Kappa architecture [6] and Liquid [7].

Provisioning and management of infrastructure is crucial for big data applications, bare metal or virtualization technology could be used as basic infrastructure. Containers are one of the most promising technologies in high performance computing. They provide scalability, what makes them also interesting for big data applications [8]. Containers have a interface for configuration parameters to optimize resource allocation, for example with machine learning algorithms [4]. Additionally meta-containers are an important concept, because they provide an extra layer for managing and controlling containers and their included applications [9].

The performance of a big data stack in general can only be measured based on a use case to be relevant in the proper sense of Jim Gray's four criteria for database benchmarks [10]. Another important objective to be benchmarked is the connection between big data frameworks. A big data architecture can be implemented as a sequence of frameworks to build a data pipeline for an application (e.g. object recognition of images). More specific, the big data architecture has an strong influence on the general application performance, on cluster management and on configuration of included software stacks.

SMACK is an implementation of such an big data pipeline, which consists of a sequence of frameworks [11]. In most cases alternatives could be found for each of the frameworks which are building the data pipeline, so a continuous measurement of performance is necessary to select the best fitting solution and to optimize the final applications. The performance of frameworks and databases for big data applications can be measured with benchmarks. Examples for well known big data benchmarks are TPC-HS [12], BigDataBench [13], HiBench [14] and YCSB [15]. These goal of this paper is to analyze current big data architectures and to find big data benchmarks that identify trade-offs between different proposed architectures.

This paper is organized as follows: Section 3 presents an general overview about big data architectures. Section 3 provides solutions for autoscaling with

emphasis on container technology. Section 4 compares actual big data benchmark suites and analyses their applicability to find trade-offs between big data architectures. Section 5 introduces implications for a self-developed benchmark, 6 provides a conclusion.

2 Big Data Architectures

In context of big data architectures it is necessary to define the terms big data and architecture separately. Subsection 2.1 explains the term big data in general. Then concepts for big data systems and their architecture are described in 2.2.

2.1 Definition of Big Data

The most popular definition categorizes big data in three dimensions, which are volume, velocity and variety [16]. Short named as three V's, they were first introduced by Laney in context of e-commerce [17].

Volume refers to data which is too big to be stored or processed by conventional database systems [18]. Due to the ongoing technical evolution, the volume dimension of big data could not reasonable be quantified in sense of units of digital information, for example petabytes. The volume of big data is usually as large, that it can only be stored in a distributed system. Analyzing these massive data sets could ether be SQL or complex analytics like machine learning [16]. The need for complex analytics on big data goes beyond the features of traditional database systems and is therefore one of the main drivers for developing big data systems.

The speed of incoming data and accordingly the need to compute results in real time is defined as velocity [19], [17]. Applications like social media and Internet of Things are examples that produce data streams in real time. Managing this high-speed data is a main challenge for big data applications [1].

Variety of data involves a broad range of data types, data structures and data inconsistencies [17], [1]. This factor implies a need for multiple data models when storing big data. It also leads to a "no one size fits all" observation, that multiple big data frameworks or databases are necessary to fulfill the job along a data pipeline [20].

2.2 Characteristics of Big Data Systems

The 3 V's of big data directly influence the characteristics of big data systems. The following characteristics are essential for the design of a big data application and the implemented software stack:

- distributed data storing
- computation at place of data
- distributed algorithms
- scaling up and down of computing resources

- fault tolerance
- excessive utilization of RAM for real time applications
- load balancing

The volume dimension of big data constrains that data has to be stored on multiple machines. For storage different solutions like distributed databases, distributed file systems or a distributed messaging system could be used. Accordingly, the data store needs to distribute the incoming data inherently. Network connection could be a bottleneck of big data systems, which generally can be addressed by moving the computation to data, as it is implemented with MapReduce paradigm [21]. The quantity of files and the size of single files is too big to be transferred to a specific node, so computation needs to be executed at the place where data is stored. Then algorithms have to compute in a distributed and parallel approach to perform well with big data.

Workloads that redistribute data result in poor performance of big data processing systems [22]. The benchmark from [22] shows that the execution time of network intensive operations can be reduced by up to 36% installing a faster network connection. Accordingly intelligent techniques of data distribution can be a possible solution to increase performance of big data analytics workloads.

The ability to scale computing resources targets a reduction of processing time and resource usages [23]. It can also reduce costs when using cloud infrastructures with pay-as-you-go payment model. Executing big data applications in cloud environments heavily depend on pricing and cost of ownership models, this case can additionally influence the design of big data applications.

Fault tolerance is a important feature that needs to be satisfied by big data systems. Usually automatic replication of data is used to fulfill this requirement, as it is implemented in HDFS [24]. Analytical frameworks like Apache Spark use transformation logging to avoid network traffic, which could occur through replication [25].

Complex analytics with incremental processing can be speed up with different memory management techniques as implemented in Apache Spark or Apache Flink [26], [27].

A main requirement of big data systems is loadbalancing. Loadbalancing can be either application-internal resource management like YARN [28] or external management of a complete big data software stack. For external management, currently the most popular solutions are Docker Swarm, Google Kubernetes and Apache Mesos [29], [30], [31].

All characteristics of big data systems have to be considered when implementing a big data architecture. Depending on use case, different big data architectures can be deployed to satisfy the needs of a specific big data application. Additionally, framework specific characteristics have to be considered. Finding these trade-offs could be done with benchmarks and will be analyzed in Section 5. The next Section describes Big Data Architectures in general and presents most recent approaches.

2.3 Definition of Big Data Architecture

An architecture in general models the basic arrangement and connectivity of parts of a system [32]. A big data architecture describes the data architecture of a big data application system [33]. The parts of a big data system are usually a bundle of frameworks, which taken all together building an application. This bundle and the connection of data pipelines could be referred as big data architecture. Big data applications consist of diverse data pipelines to fulfill the requirements of different types of storing and processing. The description and management of big data architectures is aimed solve two main problems:

1. overview of used functional entities and frameworks, modular extensibility and replacement of individual components (performance measurement desirable).
2. better fit into organizational master data management.

The next Subsections presents patterns for big data architectures and discusses advantages and disadvantages of these solutions. Section 5 describes trade-offs of big data architectures and how they can be benchmarked.

2.4 Lambda Architecture

The following explanation of Lambda architecture is mainly based on original work by Nathan Marz [5]. The Lambda architecture defines a big data system as a series of layers to address the challenge of processing arbitrary functions on a arbitrary dataset in real time ([5], p. 14). It is composed of a batch layer, a serving layer and a speed layer.

The batch layer has a precomputing function to compute arbitrary views, which are stored in a serving layer. It computes all incoming data, so the time of computation usually took multiple hours. The results of computation are stored in serving layer as views, additionally the master data remains unchanged. Big data systems that are processing in batch layer should be high throughput and high latency frameworks, for example Apache Hadoop can be used for this kind of computation. Because of the long-running manner of batch layer the views can not be computed with the latest data. This missing case is implemented in speed layer.

The speed layer also computes arbitrary views for serving layer, but it processes only recent data in real time. It does fast and incremental processing and uses databases supporting random reads and random writes. Accordingly latency-optimized big data systems are deployed for speed layer, for example Apache Spark or Apache Flink for analytical processing. Due to random writes, Apache Cassandra can be used as database for real time views.

The views generated by batch layer and speed layer got stored in serving layer, where results also were merged. The serving layer generates indexed views of data for fast querying, they must be updated very quickly. Real time usually means the range between an a few milliseconds to a few seconds. ElephantDB is

proposed as database for serving layer. It is a key-value database, where data is stored in distributed shards.

The Lambda architecture features batch and real time, to the cost of complexity when reconciling both layers [7]. Accordingly multiple big data systems need to be maintained in parallel to satisfy the need of a single big data application. This can be evaluated as implementation overhead and results in an architecture trade-off.

2.5 Kappa Architecture

The Kappa architecture was initially described in a blog post by Jay Kreps, the following description is mainly based on this source [6]. This architecture addresses the problem of Lambda architecture to maintain code in two different big data frameworks to compute a single result. Kappa architecture unifies batch and streaming into one layer. When reprocessing is necessary, a instance of the job is launched which processes all data, specified with a longer time period as job configuration. The second job instance writes outcomes as batch computation to a new result in service layer, for example a table. A big data application switches to read from this result, when the second job instance caught up the first instance of the job. Technically a framework is like Apache Flink is needed, that executes batch computations and streaming computations on the same runtime [27].

The parallel computation of Kappa architecture induces a kind of racing condition, because the second job instance need to catch up the first instance of the job. Accordingly the second job instance needs more computational resources then the first instance of the job to catch up with it. This increases the need for specific loadbalancing and it also doubles the storage space during time of "pursuit race" [7]. The doubled storage space taken by these two results can be denoted as short time backup, which can be restored when the second jobs produces bad results.

2.6 Liquid

Liquid is a nearline data integration stack to provide low latency data access, it can be implemented as architecture for batch and near real time applications [7]. The following description of Liquid architecture is mainly based on initial publication by [7].

Liquid is composed of two layers, a processing layer and a messaging layer. The messaging layer is responsible for data input from sources (primary data) and it also provides processed output to the big data application. Data is replicated and stored in messaging layer with a publish-subscribe model. Storing data in RAM is used by preference due to file system caching of operating systems. More specifically, data is persisted into distributed topics. They are implemented as partitioned, append-only commit logs with a inherent natural order per partition for finegrained data access. Each partition keeps a sequence of messages and an offset, which is implemented as unique identifier. A offset manager enables

metadata-based access, because it can maintain annotations on data. These annotations could be metadata such as timestamps or the software version that consumed a given offset. The offset manager allows to checkpoint offsets and therefore it is possible for processing layer to reprocess the last consumed data after failure. Apache Kafka is proposed to be used as framework for messaging layer.

The processing layer allows batch and near real time processing. It can access data in messaging layer through metadata (annotations) and it can produce such metadata when storing data in messaging layer. Computation is based on streams. A job processes data from input feeds, they can be primary data or already processed data. The job-results are named derived data feeds. They can be source for another processing job or being directly queried by big data applications. A job is divided into tasks when parallel processing is necessary. In this case tasks process different, distributed partitions of a topic. Liquid allows computation of both stateful and stateless jobs, also a communication between jobs is possible. A stateful job must combine previous results with new incoming data to work properly, for example stream processing operators like join or aggregate [34], [35]. In Liquid architecture, the state is published in a changelog which is stored as derived data feed in messaging layer.

The Liquid data integration stack supplies low latency, incremental processing, high availability, resource isolation and cost effectiveness. The focus on meta data management is a new approach to fit into organizational master data management. One critical aspect of big data and NoSQL technologies assumed by traditional SQL supporters is the lack of available metadata for multiple applications. Liquid could be a step towards this cumbersome.

2.7 SMACK

The acronym SMACK stands for the frameworks Apache Spark, Apache Mesos, Akka, Apache Cassandra and Apache Kafka [11]. The following description is mainly based on [11].

The SMACK stack processes data in real time and aims to output in shortest possible time, which is usually in milliseconds. It therefore builds a data pipeline to store and analyze big data. The stack consists of four layers which are usually processed in the following order: ingestion layer, aggregation layer, analysis layer and storage layer. The ingestion layer takes data from data producers and publishes them into SMACK stack. Data is collected in form of queues with a distributed messaging system. The message queue system allows partitioning, replication and sorting of data. Apache Kafka is used as message broker in SMACK stack.

In Aggregation layer the data is processed with technology based on actor model. The actor model allows lightweight, parallel and fault tolerant computation, data filtering and data enrichment. The actor model is implemented to execute distributed algorithms, for example the Akka framework. Akka can additionally be included in analysis layer, depending on the complexity of analysis [36]. More complex analysis is implemented with a specialized processing engine,

which is also based on actor model. The processing engine usually implements stateful and stateless transformations on data, generally different systems could be used for processing. Apache Spark is proposed as processing engine in analysis layer, but it can also be included in aggregation layer when preprocessing of data is necessary.

In storage layer a aggregate-oriented NoSQL database is used to persist data. SMACK uses a database which implements key-value and column-family data models, particularly Apache Cassandra is used as persistent data storage due to the easy integration with Apache Spark in analysis layer. Cassandra implements a ring architecture, where all nodes have the same role. Accordingly there is no single point of failure, which leads to high availability in context of CAP theorem. Cassandra could be used as data source for the final big data applications.

All frameworks or databases that process or store data are managed with a cluster manager. The cluster manager abstracts computer resources away from machines to enable fault-tolerant and elastic computing. Apache Mesos is used in SMACK stack to orchestrate components and managing resources. The targets of this cluster manager beside fault-tolerance and resource allocation are scalability and isolation, latter is implemented based on container technology [31].

In conclusion, the SMACK is rather a possible software stack than an architecture pattern which incorporates real time analytics with a classical ETL process. From a architectural perspective, ingestion layer is implemented with a message broker and storage layer relies on NoSQL solutions. In between, aggregation layer and analysis layer are implemented based on actor model. The next chapter describes solutions for autoscaling, which could be deployed with Apache Mesos and container technology as it is proposed in SMACK stack.

3 Solutions for autoscaling of Big Data Systems

Virtualization is a technology to provide isolation and efficiency for computation [8]. Traditional hypervisors prefer isolation over resource sharing, with less efficiency due to the emulation of hardware and a running operating system [9]. Container-based virtualization provides a shared, virtualized image of a operating system which consists of a root file system and a set of system libraries and executables [8]. In this environment, applications could be bundled and executed. Application virtualization bundles a set of processes to a container, that can be moved between different systems [9]. The transfer of applications due to containers can be executed between all systems that have the container environment installed.

According to [4], virtualization of the big data framework Apache Hadoop offers benefits including shorter provisioning time, better performance isolation and higher resource utilization. Especially higher resource utilization and a short startup time of containers are useful to automatically scale workloads of big data systems. Autoscaling needs the management of available computing resources and the provisioning and scheduling of containers itself and their included applications. For this purpose the concept of meta-containers is invented, which

allows signaling for workload characteristics and communication between containers [37]. Meta-containers can be deployed in two layers, which are cluster resource management and application scheduling. Cluster manager like Apache Mesos, Google Kubernetes and Docker Swarm are technologies that aim to manage cluster resources. Additionally application schedulers like YARN [28] fulfill the property of scheduling workload characteristics and managing application specific resources.

Containers are a good solution to run legacy code within a modern microservice architecture [38]. Big data systems and their configurations are changing permanently, which makes benchmark solutions quickly obsolete. Containerization of big data benchmark solutions can simplify installation and configuration big data systems and their benchmark solutions. Additionally, the possible cost-effectiveness of container-based big data applications can influence big data benchmarks to introduce more metrics like cost of ownership.

4 Benchmark suites for Big Data Architectures

This Section describes benchmarking solutions and analyzes trade-offs between big data architectures presented in Section 2. Subsection 4.1 describes big data benchmarking solutions. These solutions can compare trade-offs between big data architecture patterns. Following this, a mapping of big data benchmarks to big data architecture patterns is presented in Subsection 4.2.

4.1 Big Data Benchmarks

Current approaches for big data architecture patterns tackle the problem of real time applications. These use cases require systems that can handle data streams. Accordingly benchmarks for streaming applications are necessary to analyze the performance when comparing different frameworks as a particular component in a big data architecture. The TPC-HS benchmark is the first industry standard big data benchmark, but it only benchmarks batch computation with Apache Hadoop as processing system [39]. With batch computation, especially the Volume dimension of big data characteristics is targeted. This aspect of big data could easily be generated, TPC-HS and BigDataBench provide data generators which could produce appropriate big data [12], [40]. Generating data to simulate streaming applications in a benchmark is more difficult [41].

StreamBench One of the first approaches to benchmark streaming applications in context of big data is StreamBench, which is a synthetic benchmark that address typical stream computing scenarios and core operations [41]. The following description of StreamBench is mainly based on the original publication [41].

In StreamBench, data streams are generated based on real world query log data and statistical information of an hour-long Internet package trace collected

once a month. The workloads of StreamBench covering three dimensions: the target data type of computation, the complexity of computation and the integration of stored data and streaming data. The integration dimension is related to the architectural need of processing stateful jobs. The implemented algorithms for stateful computation are WordCount, DistinctCount and Statistics. The latter calculates the maximum, minimum, sum and average of a numeric input.

Generated input records are transfered into a messaging system which serves as data source for the streaming framework. The metrics of StreamBench are throughput and latency [42]. The throughput metric is measured as average count of records and data size processed per second. Latency is measured as average time frame from the arrival of a record till the end of processing of this record.

The benchmark measures the performance of stream computing frameworks by identifying four different categories: 1. Performance with a single node recipient; 2. Performance with multi node recipients; 3. Performance of fault tolerance by having a failure of a single node; 4. Performance comparison of durability when executing a 48 hour run with constant data scales. The first proposed execution of StreamBench compares Apache Storm and Apache Spark [41]. A second publication adds Apache Samza, but only a workload addressing throughput could be measured with this framework [43], [42]. The results showing that Apache Spark and Apache Samza can handle larger amount of data than Apache Storm. But Apache Storm has shorter latency.

Yahoo Streaming Benchmarks Yahoo Streaming Benchmarks is a open source real-world streaming benchmark suite [44], [45]. The following description is mainly based on the original publication [44].

This benchmark suite simulates a real world application using the example of a advertisement analytics pipeline. The benchmark includes Apache Kafka as public-subscribe messaging system for data fetching and Redis as key-value NoSQL database for storage [46],[47]. It builds a full data pipeline implemented as middle layer between event sources and data storage. The computation is performed on structured data streams and it includes following operators: filter, projection, join and windowed count. These operators have to be implemented for each framework individually, because of the lack of a standardized language for streaming frameworks. The implemented metrics are throughput and latency, the final results are calculated as percentile.

The benchmark is executed comparing the frameworks Apache Spark, Apache Storm and Apache Flink [48], [49], [50]. The results show that Apache Storm and Apache Flink have a lower latency when running with high throughput. But Apache Spark can handle a higher maximum throughput rate in comparison to Apache Storm and Apache Flink.

4.2 Mapping of Big data Benchmarks on Big Data Architecture patterns

When benchmarking Lambda architecture, it is necessary to compare the costs of maintaining processing logic in two different layers to the performance benefits implementing a Kappa architecture solution. For this case, a benchmark that compares stream processing frameworks and batch processing frameworks both executing batch jobs could be deployed. BigDataBench can be used for this purpose, especially having metrics for performance. Other metrics for durability complexity of programming jobs should be added to make a fair decision when implementing a Lambda architecture. Durability metrics could be measured with a long time execution. Metrics like lines of could be integrated to give hints for measuring complexity of implementing jobs, referring as soft indicators for costs of implementation. According to Nathan Marz batch layer of Lambda architecture needs to read a lot of data at onces and it concedes to have a higher latency ([5], p. 56, 109). With these characteristics Apache Spark would be a good choice for streaming applications according to StreamBench. It could be benchmarked if Apache Spark has the same capabilities when doing batch computation.

Additionally the performance of big data frameworks needs to be measured when doing incremental reprocessing, which is a crucial feature in all presented big data architecture patterns. As mentioned in Section 2, all analyzed big data architectures require the ability to incremental reprocessing. That is defined as a merging process that combines previously computed data with new incoming data. Benchmarking of big data architectures requires to analyze this feature to find performance trade-offs between them. StreamBench can measure this trade-off with the integration dimension. A interesting measurement would be the time frame that frameworks need to catch up with jobs in situations denoted as "pursuit race" in Kappa architecture. The implemented analytical pipeline of Yahoo Streaming Benchmark could also be used for benchmarking incremental reprocessing. It would be necessary to add a cluster manager for provisioning of resources to let the batch job catch up the streaming job.

For choosing the best big data system for data cleansing, a benchmark that measures the ability to absorb different data types and filtering functions is necessary. This would address the Variety dimension of big data. A data cleansing feature is needed in all presented big data architecture patterns, but it is implemented with different types of systems depending on the specific big data architecture pattern. To benchmark this feature, it would be necessary to put special emphasis on filtering functions and throughput measurement. Another challenge would be a benchmark measuring the hardware footprint of Lambda architecture and Kappa architecture in comparison to Liquid, where Liquid should perform better according to [7]. A benchmark measuring the efficiency of metadata management in Liquid architecture would also be interesting to measure practically

5 Implications for a self-developed benchmark solution

Benchmarking of big data architectures could be initiated by analyzing two theoretical dimensions as guide for choosing a benchmark solution. These dimensions can be used to choose the right system for a individual purpose, answering the main question of domain-specific big data benchmarking similar to Jim Gray's proposed question ([10], p.1): "Which system should I choose for my big data architecture?". The two dimensions are:

- computation type: Does the big data system has the same performance characteristics when choosing other types of computation?
- competition: How does the big data system perform when benchmarking it in comparison to other systems on the same type of computation?

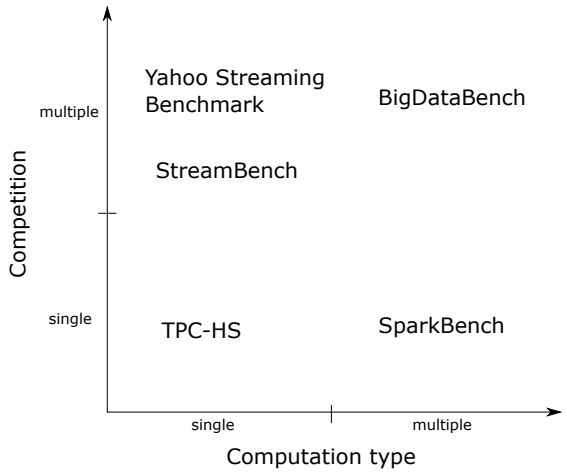

Fig. 1. Classification of big data benchmarks to compare architectural decisions

Figure 1 shows an approach to characterize a few big data benchmarks. The classification could be used to select components of big data benchmarks to compare architectural decisions. For example, the classification leads to following hypothesis which needs to be benchmarked when comparing solutions for big data architectures:

- Hypothesis: A streaming framework can compute batch jobs at least as efficient as a batch framework.

This hypothesis addresses a single computation type, for example when benchmarking Lambda architecture. But multiple competitors (big data frameworks) must be benchmarked to make the best decision. Accordingly Yahoo Streaming Benchmark or StreamBench could be used to test this hypothesis. When making

a decision to implement ether Lambda architecture or Kappa architecture, multiple computation types and multiple competitors must be benchmarked. BigDataBench could be used to decide this specific architecture problem. Included test could be the cost of implementation and the performance of a streaming framework running batch jobs.

To my best knowledge, there is no open source big data benchmark solution that is completely containerized and addresses all challenges when benchmarking big data architectures. Implementing a containerized big data benchmark solution could accomplish the four criteria relevance, portability, scalability and simplicity introduced by Jim Gray ([51], p.3f). Current solutions are scalable. However, they sometimes lack relevance due to a legacy software stack included in benchmark. The configuration is often too complicated and not simple enough to make quick decisions when choosing systems for a big data architecture.

6 Conclusion and Future Work

This paper describes big data architectures and benchmarking solutions to measure the performance of these architectures. Furthermore, it presents an approach for using current big data benchmarks to decide on the best big data architecture. Therefore current definition of big data and architecture patterns are presented. Additionally, several benchmarks are analyzed whether they can measure big data architecture trade-offs. A simple classification matrix is introduced which can help to choose a benchmark for measuring a big data architecture trade-off. In future work, the matrix could be extended with more big data benchmark suites to have a better overview of current solutions. Also big data benchmarks could be executed to check if they can verify proposed hypotheses. Finally approaches to improve current big data benchmarks are presented.

References

1. D. Abadi, R. Agrawal, A. Ailamaki, M. Balazinska, P. A. Bernstein, M. J. Carey, S. Chaudhuri, J. Dean, A. Doan, M. J. Franklin, J. Gehrke, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, D. Kossmann, S. Madden, S. Mehrotra, T. Milo, J. F. Naughton, R. Ramakrishnan, V. Markl, C. Olston, B. C. Ooi, C. Ré, D. Suciu, M. Stonebraker, T. Walter, and J. Widom, "The beckman report on database research," *Commun. ACM*, vol. 59, pp. 92–99, Jan. 2014.
2. L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.
3. P. Pääkkönen and D. Pakkala, "Reference Architecture and Classification of Technologies, Products and Services for Big Data Systems," *Big Data Research*, vol. 2, no. 4, pp. 166–186, 2015.
4. R. Zhang, M. Li, and D. Hildebrand, "Finding the big data sweet spot: Towards automatically recommending configurations for hadoop clusters on docker containers," *Proceedings - 2015 IEEE International Conference on Cloud Engineering, IC2E 2015*, pp. 365–368, 2015.

5. N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Greenwich, CT, USA: Manning Publications Co., 1st ed., 2015.
6. J. Kreps, "Questioning the lambda architecture," 2014.
7. R. C. Fernandez, P. Pietzuch, J. Kreps, N. Narkhede, J. Rao, J. Koshy, D. Lin, C. Riccomini, and G. Wang, "Liquid: Unifying Nearline and Offline Big Data Integration," *Cidr*, 2015.
8. S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 275–287, 2007.
9. E. Eberbach and A. Reuter, "Toward el dorado for cloud computing: Lightweight vms, containers, meta-containers and oracles," *ACM International Conference Proceeding Series*, vol. 07-11-Sept, 2015.
10. J. Gray, *Benchmark Handbook: For Database and Transaction Processing Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.
11. R. Estrada and I. Ruiz, *Big Data SMACK: A Guide to Apache Spark, Mesos, Akka, Cassandra, and Kafka*. Apress, 2016.
12. Transaction Processing Performance Council, "Tpc express benchmark(tm) hs." `http://www.tpc.org/tpc_documents_current_versions/pdf/tpcx-hs_v1.3.0.pdf/`, 02 2015.
13. L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "Bigdatabench: A big data benchmark suite from internet services," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pp. 488–499, Feb 2014.
14. S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pp. 41–51, March 2010.
15. B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*, pp. 143–154, 2010.
16. M. Stonebraker, S. Madden, and P. Dubey, "Intel "big data" science and technology center vision and execution plan," *SIGMOD Rec.*, vol. 42, pp. 44–49, May 2013.
17. D. Laney, "3D Data Management: Controlling Data Volume, Velocity, and Variety.," *Application Delivery Strategies*, vol. 949, no. February 2001, p. 4, 2001.
18. A. H. B. James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, "Big data: The next frontier for innovation, competition, and productivity," *McKinsey Global Institute*, no. June, p. 156, 2011.
19. U. Cetintemel, N. Tatbul, K. Tufte, H. Wang, S. Zdonik, J. Du, T. Kraska, S. Madden, D. Maier, J. Meehan, A. Pavlo, M. Stonebraker, and E. Sutherland, "S-Store: a streaming NewSQL system for big velocity applications," *Proceedings of the VLDB Endowment*, vol. 7, no. 13, pp. 1633–1636, 2014.
20. J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik, "The BigDAWG Polystore System," *ACM Sigmod Record*, vol. 44, no. 3, pp. 11–16, 2015.
21. J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.
22. X. Lu, M. W. U. Rahman, N. Islam, D. Shankar, and D. K. Panda, "Accelerating Spark with RDMA for Big Data Processing: Early Experiences," *2014 IEEE 22nd Annual Symposium on High-Performance Interconnects*, pp. 9–16, 2014.

23. S. Padhy, J. Alameda, R. Kooper, R. Liu, S. P. Satheesan, I. Zharnitsky, G. Jansen, M. C. Dietze, P. Kumar, J. Lee, R. Marciano, L. Marini, B. Minsker, C. Navarro, M. Slavenas, W. Sullivan, and K. McHenry, "An Architecture for Automatic Deployment of Brown Dog Services at Scale into Diverse Computing Infrastructures," *Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale*, pp. 33:1—-33:8, 2016.
24. K. Shvachko, R. Chansler, H. Kuang, and S. Radia, "The Hadoop Distributed File System," pp. 1–13, 2011.
25. M. Zaharia, M. Chowdhury, T. Das, and A. Dave, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," *NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, p. 14, 2012.
26. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark : Cluster Computing with Working Sets," in *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, p. 10, 2010.
27. P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas, "Apache Flink: Unified Stream and Batch Processing in a Single Engine," *Data Engineering*, pp. 28–38, 2015.
28. Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwali, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler, "Apache Hadoop YARN : Yet Another Resource Negotiator," in *ACM Symposium on Cloud Computing*, p. 16, 2013.
29. Docker Inc., "Docker swarm," 2016.
30. B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Lessons learned from three container- management systems over a decade," *Acmqueue*, no. february, p. 24, 2016.
31. B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pp. 295–308, 2011.
32. International Organization for Standardization, "Industrial automation systems - requirements for enterprise-reference architectures and methodologies," 2000.
33. H. Mcheick and E. Grant, "Big Data Architecture Evolution : 2014 and Beyond," *ACM Association for Computing Machinery*, pp. 139–144, 2014.
34. D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum, "Stateful bulk processing for incremental analytics," *SoCC*, pp. 51–62, 2010.
35. R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management," p. 12, 2013.
36. R. Estrada, *Fast Data Processing Systems with SMACK Stack*. Packt Publishing Ltd. Livery, 2016.
37. G. Morana and R. Mikkilineni, "Scaling and self-repair of Linux based services using a novel distributed computing model exploiting parallelism," *Proceedings of the 2011 20th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2011*, pp. 98–103, 2011.
38. A. Slominski, V. Muthusamy, and R. Khalaf, "Building a multi-tenant cloud service from legacy code with docker containers," *Proceedings - 2015 IEEE International Conference on Cloud Engineering, IC2E 2015*, pp. 394–396, 2015.

39. R. Nambiar and M. Poess, "Performance characterization and benchmarking: Traditional to big data: 6th tpc technology conference, TPCTC 2014 Hangzhou, China, September 1???5, 2014 revised selected papers," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8904, no. June 2015, 2015.
40. Z. Ming, C. Luo, W. Gao, R. Han, Q. Yang, L. Wang, and J. Zhan, "BDGS: A scalable big data generator suite in big data benchmarking," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8585, pp. 138–154, 2014.
41. R. Lu, G. Wu, B. Xie, and J. Hu, "Stream bench: Towards benchmarking modern distributed stream computing frameworks," in *Proceedings - 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, UCC 2014*, pp. 69–78, 2014.
42. S. Qian, G. Wu, J. Huang, and T. Das, "Benchmarking modern distributed streaming platforms," *Proceedings of the IEEE International Conference on Industrial Technology*, vol. 2016-May, pp. 592–598, 2016.
43. Apache Software Foundation , "Apache samza," 2017.
44. S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1789–1792, May 2016.
45. Yahoo Inc. , "Yahoo streaming benchmarks," Initial Commit 15 Dec. 2015.
46. Apache Software Foundation , "Apache kafka," 2016.
47. "Redis project," 2017.
48. The Apache Software Foundation, "Apache spark," 2017.
49. The Apache Software Foundation, "Apache storm," 2017.
50. The Apache Software Foundation, "Apache flink," 2017.
51. J. Gray, ed., *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.