Enabling Global Big Data Computations

Damianos Chatziantoniou
Athens University of Economics and Business
Athens, Greece
damianos@aueb.gr

Panos Louridas
Athens University of Economics and Business
Athens, Greece
louridas@aueb.gr

ABSTRACT

Most analytics projects focus on the management of the 3Vs of big data and use specific stacks to support this variety. However, they constrain themselves to "local" data, data that exists within or "close" to the organization, or external data imported to local systems. And yet, as it has been recently pointed out, "the value of data explodes when it can be linked with other data." In this paper we present our vision for a global marketplace of analytics—either in the form of per-entity metrics or per-entity data, provided by globally accessible data management tasks—where a data scientist can pick and combine data at will in her data mining algorithms, possibly combining with her own data. The main idea is to use the *dataframe*, a popular data structure in R and Python. Currently, the columns of a dataframe contain computations or data found within the data infrastructure of the organization. We propose to extend the concept of a column. A column is now a collection of key-value pairs, produced anywhere by a remotely accessed program (e.g., an SQL query, a MapReduce job, even a continuous query.) The key is used for the outer join with the existing dataframe, the value is the content of the column. This whole process should be orchestrated by a set of well-defined, standardized APIs. We argue that the proposed architecture presents numerous challenges and could be beneficial for big data interoperability. In addition, it can be used to build mediation systems involving local or global columns. Columns correspond to attributes of entities, where the primary key of the entity is the key of the involved columns.

1 INTRODUCTION

Currently, most big data deployments follow a highly ad hoc, non-disciplined approach, entailing a high degree of data replication and heterogeneity, both in terms of storing options and analysis tasks. The system administrator has to choose one (or more) data management systems from a plethora of alternatives and facilitate the enterprise's reporting needs utilizing a wide range of query languages and analysis techniques. Data management systems involve traditional RDBMSs, Hadoop clusters, NoSQL databases, and others. Reporting and analysis tasks include plain SQL, spreadsheet scripts, MapReduce jobs, R/Java/Python programs, complex event processing queries, machine learning algorithms, and others. A not-so-new challenge resurfaces: interoperability. How can these systems interact? How can these systems interoperate?

This necessity has been identified by the current authors in [7, 8] and more recently by the Beckman report [1]. The Beckman report recognized the problems the "diversity in the data management landscape" creates and asserted "the need for coexistence of multiple Big Data systems and analysis platforms is certain" and that in order "to support Big Data queries that span systems, platforms will need to be integrated and federated." Data integration involves combining data residing in different sources and providing users with a unified view of them [15]. Data integration can be seen as constructing a data warehouse, or creating a virtual database (federated/mediated systems). While data warehousing was the way to go in the past—mainly due to the dominance of relational systems in data management—there are well-thought arguments to reconsider a federated approach in big data applications [20]. Polystores [10], closely related to federated databases, address the need for managing information represented in different data models. This is similar to this paper's motivation: using the *answer* of computations defined over different data models and query languages. However, we focus on standardizing the output of a computation and use it in a conceptual model, rather than integrating data model and query capabilities in the system. It is worth mentioning that defining global views over heterogeneous data sources is not a big data-era issue and has been extensively discussed in the past (e.g., [2]).

We argue that a standardized and protocol-based approach can significantly facilitate the unified dissemination, federation and analysis of data. Once the output of big data computations (from simple SQL queries to complex predictive models) can be standardized and accessed globally, anyone can use it in his own analysis framework.

Section 2 presents an example from the telecom domain and motivates the paper. It introduces the concept of global dataframes: dataframes constructed by columns that are globally accessible and represent a data management task. Section 3 describes the big picture: a dataframe composed of globally addressed columns. Section 4 presents the architecture and the necessary APIs to support the management and usage of these global columns. The challenges of such an architecture (performance, transactionality issues, distribution, etc.) are introduced in Section 5. We conclude with conclusions in Section 6.

2 MOTIVATION

Consider the churn prediction problem in a telecom environment in the presence of structured and unstructured data. For this purpose, a predictive model had to be designed and implemented taking into account the many possible variables (features) characterizing the customer. The goal was to equip the data analyst with a simple tool that enables fast and interactive experimentation by using features from multiple data sources, involving different data management systems and data formats. In our case, the company had a variety of data sources, such as:

- A traditional RDBMS containing basic customer-related data such as gender, age, address and various demographics.
- A relational data warehouse storing billing, usage and traffic activity per contract key—a contract may involve several customer IDs.

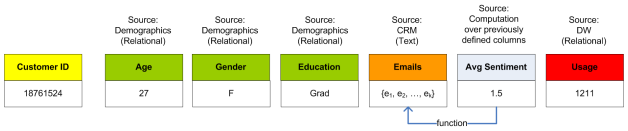

Figure 1: Defining a Tabular View over Multiple Data Sources

- Flat files produced by statistical packages such as SAS and SPSS, containing data transformations and precomputed measures per contract key on different datasets.
- CRM data stored in a relational database, containing metadata of customer-agent interactions, including agent's notes (text) on the call.
- Email correspondence between customers and the customer service center of the company (text).
- Audio files stored in the file system, containing conversations between customers and agents.

The ultimate goal was to provide the data scientist with a simple way (a tool of some kind) allowing her to choose and experiment in an ad-hoc manner with multiple tabular views of customer-related data. Each different combination of columns yields a different view. Intuitively, there are two simple yet ineffective ways of achieving that: a) by collecting all data in a single data repository and performing reporting tasks on top of it (a data lake approach), or b) by programmatically producing it by using an RDBMS for example as an intermediate storage point. Both of those are impractical. The first one imposes significant costs of moving data around and data lakes have received criticism in terms of governance, security, and lack of consistency [13], while research tries to mitigate these problems leading to a "data swamp" [11, 12]. The second one requires significant manual intervention and is not flexible to schematic updates (changes) in the underlying data sources.

The solution we chose was to keep data in their respective host systems and define "tabular" views over these systems (a mediator approach): start from one or more "base" columns (e.g., contract ID, customer ID, area code) and incrementally extend this "basic" schema with columns containing data or computations coming from different data sources. For example, one could define the first column (base) to contain the IDs of customers. Then, she could add columns corresponding to the age, gender and educational background for each customer, coming from the Demographics RDBMS. Then, she could add a column containing, for each customer, the set of emails the customer has sent in the last six months. This is a set of texts coming from the CRM database. Then, she could add a column that computes the average sentiment of these emails, using some Python script. Finally, she could add a column corresponding to the customer's monthly average usage in the last six months, coming from the Billing Data Warehouse (DW). This process is depicted in Figure 1.

The idea is similar to Multi-Feature SQL [3, 6], MD-Joins [5], Grouping variables [4] and Associated Sets [9]. In these papers, one can express a series of outer-joins combined with aggregation, *possibly correlated*, in a succinct and concise manner. At the same time, several efficient evaluation techniques (based on parallel, distributed and in-memory processing) are presented for these queries. More recently, the same idea is expressed using dataframes in R and Python pandas (without correlation between columns), as well as in Spark. The fundamental difference between our proposal and these approaches is that we treat, in principle, our data as columns coming from heterogeneous sources, and dataframes are composed on the fly from them. In R and pandas dataframes are typically created from data imported from different, possibly heterogeneous sources, but the dataframe reflects the structure of the underlying data. One could adopt a columnar key-value approach, for example by using exclusively Series in pandas or atomic vectors and lists in R, but that would probably defeat the purpose and the philosophy of these frameworks. Concerning Spark, again dataframes reflect the structure of underlying data; moreover, most likely data are copied to an underlying HDFS substrate.

This is the pattern in most big data tasks: building a dataframe over different datasets, to serve as input to data mining algorithms, visualization tools and reporting systems. Essentially this could be modeled conceptually as a single tabular data representation of joined results coming from *different* data management systems. Each result consists of the keys used for the outer join and the corresponding values to be added as a column; it can therefore be represented as a set of key-value pairs. This formalism is quite appropriate to represent columnar data in a dataframe. It is simple, prone to distributed, fault-tolerant and scalable implementations, and can easily, naturally in a way, represent most well-known data models. At the same time, key-value engines, such as Redis, have been organically developed, through analysis of real applications at Google, Amazon, Facebook, LinkedIn and elsewhere.

Going one step further, one can imagine *globally available* key-value structures, produced *outside* the organization and used in the same manner. For example, an *analytics provider* could generate for each Facebook user some social metrics (e.g., number of checkins in the last month), useful in a data scientist's analysis. We envision a global "environment" for these key-value structures that analysts pick and try into their data mining algorithms or embed into their visualizations and reporting. We call these widely available key-value structures *globalized analytics*. The challenge is to provide a framework for this environment, to make this process simple, useful and efficient. In the next section we present our proposal for how such a framework can be build. The framework will leverage the idea of distributed key-value pairs used to compose dataframes, over heterogeneous data, allowing users to compose and manipulate tabular views of their data on the fly.

3 THE BIG PICTURE

The main abstraction for representing data is a *column*, consisting of a set of key-value pairs, called a *key-valued structure* (KVS). Columns can be joined to create *dataframes*. Columns have the following characteristics:

- Columns may be distributed among different machines. That means that a dataframe can comprise data residing in different machines, and the data is joined on the fly to create an integrated dataframe.
- The column keys must be unique, but the value associated with a key need not be atomic, so that values can be lists or sets. Therefore, a column can represent both a vector of atomic values, as well as associations between keys and collection of values. In this way a column can act as the stage between a mapping and a reduce stage in a typical MapReduce job.

- The values of a column may exist in three states: *defined*, *not available*, and *observer*. The defined and not available states correspond to known and known unknowns, respectively. The observer state corresponds to values that we expect to be filled in the future, possibly more than once.

The observer status allows us to handle data dynamically, from disparate sources, without requiring that all data be available and frozen at each source. In effect, this extends the semantics of our proposed model with *reactive programming concepts*, e.g., see [17], and facilitates the handling of streams. Streams are collections of multiple values that are pushed to their destinations—in contrast to a pull model, where we request values from a stream, we can also use a push model, where the source of the stream emits values that enter the stream. In this way, a stream is an *observable*. A column can be an observer that receives the values emitted by the observable.

Columns can be joined based on the values of their keys. A dataframe is a collection of columns that have the same set of keys. The set of keys must be determined at column creation time, so that the values will be filled in from the underlying data. Dataframes themselves do not contain data. Their constituent columns do. That means that when we are presented with a dataframe and we interact with it, we are in fact interacting with the underlying columns. Some data may be local, if the corresponding columns local, but in general data may be remote, even not yet present when in the observable state.

Note that there is no assumption that the keys of the columns are consistent. Indeed, keys over heterogeneous data sources are not consistent in most cases. However, at a certain point there must be a mapping step, which can either be transparent to our system, or it can be handled through an intermediate column. Similarly, values need not be consistent: imagine two monetary columns being joined in the dataframe, but being expressed in different currencies [1]. Such issues could be tackled with transformation tasks.

4 ARCHITECTURE

We propose a layer of columns backed by a commonly-referenced memory space for establishing global views in a tabular data format. The columns contain data, indexed by their keys. The underlying key-value structures contain minimal schema information:

- The values may be atomic, or they may be lists or sets. Lists are ordered, while sets are not; lists may contain multiple times the same value (at different positions), while sets obey the usual set semantics.
- Lists and sets may be composed by other lists and sets, interchangeably, and atomic items. Therefore, lists and sets can represent arbitrary complex nested structures.
- Atomic values do not need any schema assumptions, so a column could contain both numeric and string data. For practical and efficiency reasons, however, an implementation could choose to represent columns using specific underlying datatypes. For example, if a column is known to contain integers, the column could be declared to be of integer type to speed up calculations. If a column contains multiple types then it would be represented as a generic object type.

[1] Our thanks to the anonymous reviewer who provided this example.

The column data could be stored by any underlying mechanism, which could be a relational database, or a NoSQL key-value store, or a dynamic data source. The underlying data source is only relevant to the data scientist at column creation time, where she has to describe the column; it then remains invisible. The data source does not need to physically host the data. It may produce the data that will be filling in the column. In this scenario, the data in the column are in the observable state as explained in Section 3.

The data of a column are handled by a *Column Manager* (CM, or simply *manager*). A manager is responsible for providing the data in the form of key-value pairs. Managers accept incoming requests from *Column Consumers* (CCs, or simply *consumers*). Consumers and managers communicate according to a CM-CC protocol that defines the location where the underlying KVS will be stored. The location of the KVSs is independent of both the consumer and the manager. They may be stored at either of them, or somewhere else entirely; moreover, they may be produced programmatically or be saved in a distributed, redundant manner among different machines. The location can be negotiated: for example, the consumer may offer a location that the manager will accept, the consumer may ask the manager to respond with a location, or the consumer may suggest a location and the manager may respond with a different one. In this way, KVSs can be reused among different consumers. To cover different scenarios, KVSs reside in a globally addressable storage space.

The idea is simple: A consumer wants to use data from a manager. The data could be result of an SQL statement, or a MapReduce job, or a script, etc. The consumer communicates with the manager and passes to it the address of the KVS. The consumer also communicates with the KVS and passes to it the set of keys whose values will be filled in by the manager; the communication between the consumer and the KVS may take place well before the communication between the consumer and the manager. The manager finds the keys in the KVS and fills the corresponding values.

To create a dataframe, a consumer communicates with the managers that handle the columns it wants. It passes to each of the managers the same set of keys and agrees with them on the addresses of the KVSs that will contain the data. Then the consumer can obtain the data from the KVS and present them to the data scientist as an integrated dataframe. Note that the consumer may access the KVSs at any time, asynchronously, even before the managers complete the KVSs. This way, dataframes can incorporate columns corresponding to stream computations.

The above can be implemented with a two-layered architecture. At the upper layer we have the consumer-manager communication. At the lower level we have the set of KVSs. Consumers and managers communicate with the KVSs using a column-to-KVS communication protocol. Figure 2 summarizes the aforementioned discussion.

Currently there are efforts pointing towards a separate addressable memory layer, such as RAMCloud [18] and Piccolo [19], which both share the notion of in-memory addressable "tables" supporting key-value operations. Clarifications, challenges and opportunities of the proposed architecture are presented below.

5 CHALLENGES AND OPPORTUNITIES

Commonly-Referenced Memory Space. The architecture implies the presence of a well-defined API so CCs can create and manage KVSs. The development of such an API requires careful

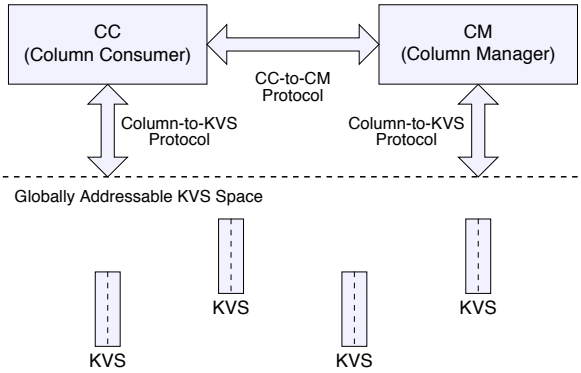

Figure 2: Defining a Tabular View over Multiple Data Sources.

consideration. While CRUDE operations are clearly understood, a discussion is required for the exact format and behavior of each. In particular, the read operation should allow some filtering of the KVS, either through a simple predicate over keys and values or by providing a set of keys to be selected. Our first implementation [7] allows filtering conditions over just the key, but in many industrial applications complex expressions involving values are not uncommon. Other issues that have to be addressed are: what if a CC does not delete a KVS that has created? Should the corresponding KVS management system implement garbage collection? Who is the "owner" of a KVS? What is the "lifetime" of a created KVS? Which CCs are allowed to access this KVS and in what mode (read/write)? For example consider Webdis, an HTTP interface for Redis that provides some insights on these issues. A similar kind of middleware between data producers and consumers in the form of publish-subscribe is suggested in [14]. A commonly-referenced memory layer is also proposed in Tachyon system [16], constrained however within a cluster.

Globally Addressable Key-Value Sets. This is a conceptual layer, consisting of systems that provide KVS management according to the proposed framework. To do so, it should implement the column-to-KVS API mentioned above, and allow access to a KVS through an address, internet-wide, following some standardized addressing scheme. The scheme should capture location hierarchies (e.g., domains, sub-domains, etc.) and identify the position in the memory hierarchy of a KVS. There is no restriction on what such a system could be. It could store KVSs anywhere in the memory hierarchy: main-memory, distributed-cache, disk; it could guarantee (or not) fault-tolerance, availability, etc. In addition, it should provide answers on how it handles ownership, lifetime and access control of KVSs.

Suitability for Stream Engines. The layered architecture essentially introduces a referencing layer (i.e., indirection) between communicating programs (the CC and the CM). This is particularly appropriate for collaborating applications involving stream data: a stream management CM can continuously produce aggregated data (e.g., the average stock price over a sliding window of 10 minutes) consumed by the CC. The asynchronous access to the shared KVS allows the data consumer to retrieve data whenever it deems appropriate (e.g., [18]); alternatively, the data can be observed continuously by the consumer in a reactive approach (recall Section 3).

Transactionality Issues. A potentially challenging aspect in the proposed architecture is the issue of transactional consistency at the KVS layer. We currently consider non-materialized dataframes, so we do not have to deal with transactions and isolation at the data sources. However, transactionality issues still arise, in the case of complex workflows where multiple CCs constantly request execution from a CM. For example, consider two separate CC-to-CM connections where both CMs populate the same KVS. As another example, if a CM is using a remote address to store a continuously running query that returns a huge set of key-value pairs, is that large result updated atomically or incrementally? If some data feeds are slow and some are fast, one might get an inconsistent (nonserializable) view of the KVS layer. What (if anything) can the framework do to manage transactional requirements across systems? For instance, when a CC creates a KVS (and thus becomes "owner" of the KVS), it could also specify the required isolation level for that KVS.

Query Response Times. Mediation approaches do not have, in general, good query response times. This is one of the main reasons for building data warehouses and having Extraction, Transformation, and Loading (ETL) processes: storing data into one system, using a single data model, and having an efficient query processing engine. However, the goal of this work is not performance, but functionality and interoperability. We want to enable users to easily construct data frames in heterogeneous environments, employing multiple programming languages. Usually, this data frame will feed some learning algorithm, instead of being used for online queries. In this context, some of the columns of a dataframe could be formed as queries over a relational system using a metadata catalog like Hive Metastore. However, statistical information existing in the Metastore is relevant to the query that produces the column and is transparent to our architecture that will simply use the end result in an outer join.

Opportunities. The proposed architecture can also be used to generalize various existing distributed data management frameworks, such as distributed relational query processors, MapReduce evaluation algorithms and column-oriented processing engines. However, given the diversity in data management systems, it opens up a wide range of interesting possibilities, both in terms of infrastructures and optimization opportunities. The interested reader can refer to [8].

6 DISCUSSION AND CONCLUSIONS

In this paper, we presented a layered architecture to data interoperability based on a ubiquitous universe of remotely accessibly key-value sets. The architecture uses a number of concepts that can and will be formalized in an extended version of this paper—doing so here would be beyond the scope of a vision paper. In essence, with the proposed architecture we completely decouple the computation and memory layer of any data management scenario. By doing so we are able to generalize, abstract and effectively encapsulate all the key components of distributed data computation, storage and management. We believe that such an approach is a first step towards an interoperable universe of big data systems. Along this way however there are numerous challenges able to serve as fruitful research directions.

REFERENCES

[1] Daniel Abadi, Rakesh Agrawal, Anastasia Ailamaki, Magdalena Balazinska, Philip A. Bernstein, Michael J. Carey, Surajit Chaudhuri, Jeffrey Dean, AnHai

Doan, Michael J. Franklin, Johannes Gehrke, Laura M. Haas, Alon Y. Halevy, Joseph M. Hellerstein, Yannis E. Ioannidis, H. V. Jagadish, Donald Kossmann, Samuel Madden, Sharad Mehrotra, Tova Milo, Jeffrey F. Naughton, Raghu Ramakrishnan, Volker Markl, Christopher Olston, Beng Chin Ooi, Christopher Ré, Dan Suciu, Michael Stonebraker, Todd Walter, and Jennifer Widom. 2016. The Beckman report on database research. *Commun. ACM* 59, 2 (2016), 92–99. https://doi.org/10.1145/2845915

[2] Silvana Castano, Valeria De Antonellis, and Sabrina De Capitani di Vimercati. 2001. Global Viewing of Heterogeneous Data Sources. *IEEE Trans. Knowl. Data Eng.* 13, 2 (2001), 277–297. https://doi.org/10.1109/69.917566

[3] Damianos Chatziantoniou. 1999. Evaluation of Ad Hoc OLAP : In-Place Computation. In *ACM/IEEE International Conference on Scientific and Statistical Database Management (SSDBM)*. 34–43.

[4] Damianos Chatziantoniou. 2007. Using grouping variables to express complex decision support queries. *Data Knowl. Eng.* 61, 1 (2007), 114–136. https://doi.org/10.1016/j.datak.2006.05.001

[5] Damianos Chatziantoniou, Michael Akinde, Ted Johnson, and Samuel Kim. 2001. The MD-Join: An Operator for Complex OLAP. In *IEEE International Conference on Data Engineering*. 524–533.

[6] Damianos Chatziantoniou and Kenneth Ross. 1996. Querying Multiple Features of Groups in Relational Databases. In *22nd International Conference on Very Large Databases (VLDB)*. 295–306.

[7] Damianos Chatziantoniou and Florents Tselai. 2014. Introducing Data Connectivity in a Big Data Web. In *Proceedings of the Third Workshop on Data analytics in the Cloud, DanaC 2014, June 22, 2014, Snowbird, Utah, USA, In conjunction with ACM SIGMOD/PODS Conference*. 7:1–7:4. https://doi.org/10.1145/2627770.2627773

[8] Damianos Chatziantoniou and Florents Tselai. 2016. The Data Management Entity: A Simple Abstraction to Facilitate Big Data Systems Interoperability. In *Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference, EDBT/ICDT Workshops 2016, Bordeaux, France, March 15, 2016*. http://ceur-ws.org/Vol-1558/paper38.pdf

[9] Damianos Chatziantoniou and Elias Tzortzakakis. 2009. ASSET queries: a declarative alternative to MapReduce. *SIGMOD Record* 38, 2 (2009), 35–41. https://doi.org/10.1145/1815918.1815926

[10] Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, Magdalena Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stanley B. Zdonik. 2015. The BigDAWG Polystore System. *SIGMOD Record* 44, 2 (2015), 11–16. https://doi.org/10.1145/2814710.2814713

[11] Mina Farid, Alexandra Roatis, Ihab F. Ilyas, Hella-Franziska Hoffmann, and Xu Chu. 2016. CLAMS: Bringing Quality to Data Lakes. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 2089–2092. https://doi.org/10.1145/2882903.2899391

[12] Rihan Hai, Sandra Geisler, and Christoph Quix. 2016. Constance: An Intelligent Data Lake System. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 2097–2100. https://doi.org/10.1145/2882903.2899389

[13] Nick Heudecker and Andrew White. 2014. The Data Lake Fallacy: All Water and Little Substance. Gartner report. (July 23 2014).

[14] Rajive Joshi. 2007. Data-Oriented Architecture: A Loosely-Coupled Real-Time SOA. (2007). Real-Time Innovattions, Inc., Technical Report.

[15] Maurizio Lenzerini. 2002. Data Integration: A Theoretical Perspective. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*. 233–246. https://doi.org/10.1145/543613.543644

[16] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2014. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, November 3–5, 2014*. 6:1–6:15. https://doi.org/10.1145/2670979.2670985

[17] Erik Meijer. 2012. Your Mouse is a Database. *Commun. ACM* 55, 5 (May 2012), 66–73. https://doi.org/10.1145/2160718.2160735

[18] John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru M. Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. 2009. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *Operating Systems Review* 43, 4 (2009), 92–105. https://doi.org/10.1145/1713254.1713276

[19] Russell Power and Jinyang Li. 2010. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4–6, 2010, Vancouver, BC*. 293–306. http://www.usenix.org/events/osdi10/tech/full_papers/Power.pdf

[20] Michael Stonebraker. 2015. The Case for Polystores. ACM SIGMOD Blog. (July 13 2015). http://wp.sigmod.org/?p=1629