RepBun: Load-Balanced, Shuffle-Free Cluster Caching for Structured Data

Minchen Yu*, Yinghao Yu†, Yunchuan Zheng*, Baichen Yang†, Wei Wang*
Hong Kong University of Science and Technology
*{myuaj, yzhengbj, weiwa}@cse.ust.hk, †{yyuau, byangak}@connect.ust.hk

***Abstract*—Cluster caching systems increasingly store *structured data objects* in the *columnar format*. However, these systems routinely face the *imbalanced load* that significantly impairs the I/O performance. Existing load-balancing solutions, while effective for reading *unstructured data objects*, fall short in handling columnar data. Unlike unstructured data that can only be read through a full-object scan, columnar data supports direct query of specific columns with two distinct access patterns: (1) columns have the *heavily skewed popularity*, and (2) hot columns are likely *accessed together* in a query job. Based on these two access patterns, we propose an effective load-balancing solution for structured data. Our solution, which we call RepBun, groups hot columns into a bundle. It then copies multiple replicas of the column bundle and stores them uniformly across servers. We show that RepBun achieves improved load balancing with reduced memory overhead, while avoiding data shuffling between cache servers. We implemented RepBun atop Alluxio, a popular in-memory distributed storage, and evaluate its performance through EC2 deployment against the TPC-H benchmark workload. Experimental results show that RepBun outperforms the existing load-balancing solutions with significantly shorter read latency and faster query completion.**

I. Introduction

Cluster caching systems are widely deployed in front of the cloud storage to provide low-latency data access at memory speed [1]–[6]. However, the routinely observed data popularity skew in cluster caches results in *imbalanced load* across cache servers, creating hot spots with excessive I/O latency [7], [8].

Existing load-balancing solutions employ three techniques to mitigate hot spots: (1) *selective replication* which copies multiple replicas of hot data objects [9]–[11], (2) *erasure coding* which creates parity chunks of data objects [7], and (3) *selective partition* which splits hot objects into multiple small partitions so as to spread their load across servers [8]. These solutions are proven effective for caching *unstructured* data objects which are simply dumped as data blobs that mandate a *full-object scan* for data access (e.g., text files in HDFS).

Compared to unstructured data blobs, *structured* data objects are increasingly stored in cluster caches. These objects have clear semantics of data schema and are usually stored as *columnar tables* (e.g., Parquet [12] and Apache Arrow [3]) that can be efficiently queried by analytics frameworks such as Spark SQL [13], Hive [14], and Presto [15]. Unlike unstructured data that require full-object scans, columnar data supports direct query of specific columns using SQL (e.g., SparkSQL querying Parquet files). Our characterization study against the TPC benchmark suites [16]–[18] shows that columns of a table typically have *heavily skewed popularity* (Sec. II-D). That is, a small number of hot columns contribute a large fraction of the data access. In addition, hot columns are likely queried together by an analytic job, indicating a strong *co-access pattern*.

These two distinct access patterns from unstructured data render existing load-balancing solutions inefficient in handling columnar data. In particular, selective replication falls short with high memory overhead as copying the entire hot object also replicates the cold columns, which is unnecessary; creating parity chunks using erasure code precludes direct access to columns, forcing an inefficient full-table scan; selective partition may retain hot columns in one partition, failing to spread their load to mitigate hot spots.

Moreover, the two access patterns of columnar data suggest an unpleasant tradeoff between load balancing and communication overhead. One one hand, to improve load balancing, hot columns should be distributed *uniformly* across servers. On the other hand, as hot columns are often queried together, caching them across servers results in heavy data shuffling (e.g., `join` two columns on two servers), which, in turn, delays query completion in spite of the improved load balancing.

In this paper, we address this unpleasant tradeoff with a simple, yet effective solution called *selective bundling and replication*. Our key idea is to *bundle* the hot columns of each structured object with strong co-access patterns and copy multiple replicas of the bundle across servers. For a query job whose working columns can be *fully served* by the bundle, we randomly choose a replica and use it to serve the query. Otherwise, we serve it using the original object containing all columns. This solution offers three benefits. First, it spreads the load of hot columns into multiple bundle replicas, leading to improved load balancing. Second, it significantly reduces the memory overhead as only a small amount of hot columns are copied. Third, it avoids data shuffling as queries are always served using the *local data*.

Critical to our solution is to judiciously determine which columns should be grouped into a bundle and how many replicas should a bundle be copied. This requires the knowledge of column co-access patterns, which cannot be directly obtained from cluster caching systems like Alluxio [2], [4]. As low-level storage, cluster caches are *agnostic* to the high-level query semantics and can only track the access count of each column. We address this challenge with a simple, yet effective algorithm that accurately infers the co-access patterns between

columns based on their individual popularity. Based on this information, we formulate and solve a stochastic optimization problem that minimizes load variance across servers, under the constraint of a given memory budget.

We have implemented our solution, termed RepBun, atop Alluxio using Parquet [12] as the columnar storage substrate. We evaluated RepBun against the TPC-H benchmark in a 21-node EC2 cluster. Experimental results show that compared with selective replication, RepBun achieves better load balancing and reduces the mean and tail read latency by up to 30% and 36%, respectively. When deployed with Spark SQL, RepBun leads to faster query completion, accelerating 40% of SQL queries in the TPC-H benchmark by over 15%.

II. Background and Motivation

In this section, we briefly survey the existing load-balancing solutions for cluster caches, and motivate the need to have a new approach for handling structured data through a characterization study on their access patterns.

A. Cluster Caching and Load Imbalance

Data-intensive clouds are increasingly bottlenecked on the storage I/O, and hence rely on cluster caching systems to meet the I/O performance demands [2]–[6], [19]. By deploying a cluster of cache servers in front of the cloud storage, I/O-intensive jobs can access data at memory speed, leading to significant performance improvement. In this paper, we primarily focus on the *compute-collocated* cluster caches, in which the cache storage is deployed in collocation with the compute node for improved memory locality—a common practice in production environments according to our contacts in Alluxio [4].

However, cluster caching systems routinely face severe load imbalance. Prior study shows that the popularity of data objects in production clusters follows a Zipf-like distribution [1], [7]–[9], [20], [21]. Meaning, a small number of hot objects account for a large number of access requests. Such heavy popularity skew creates hot spots among cache servers, which not only delays data access that substantially impairs the I/O performance, but also necessitates over-provisioning in order to accommodate the peak demands [7], [8].

B. Existing Load-Balancing Solutions

Existing load-balancing solutions for cluster caches can be broadly categorized into three approaches: *selective replication*, *erasure coding*, and *selective partition*.

Selective replication mitigates hot spots by copying multiple replicas of data objects based on their popularity [9]–[11]. That is, the more popular an object is, the more replicas it is copied. These replicas are stored uniformly in cache servers in order to spread the load of hot objects. Although replication improves load balancing, it results in the high memory overhead, given that hot objects are usually large in size [7], [8]. Therefore, selective replication usually does not lead to the optimal caching performance [7], [8].

TABLE I: Statistics of the three TPC benchmark suites.

Benchmark	Table #	SQL Query #
TPC-DS [17]	24	99
TPC-H [16]	8	22
TPC-xBB [18]	19	30

Erasure coding comes as an improved solution over replication with reduced memory overhead [7]. In a nutshell, a (k, r) coding scheme splits an object into k data chunks and computes r parity chunks. Any k of the $k + r$ chunks are sufficient to decode the original object. This allows the data read to be performed on k servers in parallel, leading to more balanced load across the cluster [7].

Selective partition is an alternative approach that adaptively splits data objects based on their size and popularity [8], where large, hot objects are divided into multiple small partitions, and the load of their access requests is evenly spread to multiple servers. Compared with replication and erasure coding, it results in better load balancing at no expense of high memory overhead and coding complexity [8].

All three load-balancing approaches assume a *full-object scan* for data access. However, this is not the case for reading structured data in cluster caches, as explained below.

C. Structured and Unstructured Data Objects

In cluster caches, data are usually stored in two forms: *structured objects* and *unstructured objects*. The former organize data records in multiple fields (columns) each having a clear semantics, known as the *data schema*. Structured objects are usually stored as *columnar tables* for analytical workloads, where columns can be directly retrieved using SQL-like queries. In contrast, unstructured objects have no data schema. They are simply dumped as data blobs (e.g., text files in HDFS) that require a full-object scan for data access.

Compared with unstructured objects, structured data are increasingly stored in production clusters, as they can be more conveniently and efficiently processed by analytics jobs. For example, Apache Parquet [12] is a popular columnar storage format for structured data that are widely supported in many analytics frameworks such as Spark SQL [13] and Hive [14]. In Parquet, a columnar table is *horizontally divided* into multiple partitions called *row groups*. A row group consists of a *column chunk* for each column in the table. A column chunk stores the compressed data of the column and is highly efficient in storage. To read a Parquet table, the framework schedules multiple parallel query tasks, each handling a row group (partition). A task only retrieves the required columns, without performing a full-partition scan, which significantly improves the read performance.

D. Characterizing the Access Patterns of Structured Data

As the full-table scan is usually avoided for reading structured data, we expect the uneven popularities across columns. To validate this, we characterize the column access patterns of

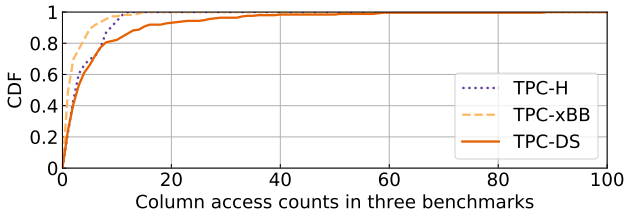
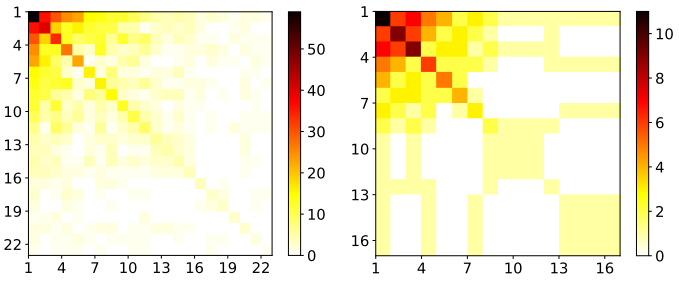


Fig. 1: CDF of column access counts for the three benchmarks.



(a) `store_sales` in TPC-DS. (b) `store_sales` in TPC-xBB.

Fig. 2: Heat map of column co-access count in two representative tables. Columns are sorted in a descending order of popularity. Grid (i, j) illustrates the co-access count of columns i and j, where dark color indicates a high count.

the structured data in the standard TPC benchmark suites. We run three representative query workloads provided by TPC-DS [17], TPC-H [16], and TPC-xBB [18] in Spark SQL. Table I summarizes the number of source tables and standard queries included in the three benchmark suites. For each benchmark, we generate 1 GB columnar data in a source table and store it in Parquet files.[1] We execute all standard queries in sequence and measure the access count of each column in a source table. Our characterization highlights the following two distinct access patterns for columnar data.

P1: Heavily skewed column popularity. In all three benchmarks, we observe the significant popularity skew among columns. Fig. 1 depicts the distribution of the column access counts measured in the three benchmarks. While the majority of the columns are cold and are rarely accessed, a small number of hot columns have very high access counts. Notably, in the TPC-DS benchmark, the hottest column is requested in 89 out of the 99 queries. In comparison, over 80% of columns are accessed less than 10 times.

P2: Strong co-access pattern between hot columns. Our characterization also suggests that hot columns have a high chance to be accessed *together* in a query job. To illustrate this, we refer to Fig. 2 which depicts the heat map of the co-access count of two columns in two representative tables. In a heat map, columns are sorted in descending order of popularity, and grid (i, j) illustrates the times that the i^{th} hottest column and the j^{th} hottest column are both requested by a query. As a special case, gird (i, i) is simply the access count of column i. We observe the hot spots mostly concentrated in the top left corner of the heat map, indicating high co-access counts between popular columns.

The two distinct access patterns of structured data pose new challenges to load balancing cluster caches, as we shall see in the next section.

[1]Our observations do not depend on the input data size, as the data access pattern in each source table does not change with the size of the table.

III. Challenges

In this section, we show that existing load-balancing solutions fall short in handling the structured data. We also show through experiments that simply achieving load balancing without accounting for the frequent co-accesses of columns may result in significant communication overhead caused by data shuffling, leading to even slower I/O.

A. Inefficiency of Existing Solutions

Selective replication copies multiple replicas of hot objects [9]–[11]. Applying it to the structured data means that the entire table gets copied. This, however, is *highly inefficient* as the majority of columns are cold in the table (see ***P1*** in Sec. II-D), and copying those columns only adds memory overhead without improving load balancing. In fact, we have implemented selective replication as a baseline and confirmed its inefficiency through experimental evaluations (see Sec. VI).

Erasure coding creates parity chunks and requires reading the amount of data *no less* than the original object to decode it [7], i.e., any k of $k + r$ chunks (see Sec. II-B). Should erasure coding be applied to the structured data, we would not be able to directly retrieve columns but have to wait for the *entire table* to be decoded. This not requires reading a large amount of data but also incurs the decoding overhead, which can be expensive for large objects even using an optimized implementation [8].

Selective partition splits hot objects into multiple small partitions. Applying it to the columnar data, we could vertically divide a table into multiple *column groups*. To achieve better load balancing, column groups should have approximately equal load. This requires hot columns to be evenly distributed across the groups. However, because hot columns are often queried together (see ***P2*** in Sec. II-D), doing so results in heavy data shuffling. As we show next, this would substantially impair the read performance.

B. Load Balancing vs. Data Shuffling

For the compute-collocated cluster caches storing columnar data, simply improving load balancing at the expense of increased communication may result in even worse performance. We illustrate this problem through two simple experiments.

Query slowdown caused by data shuffling. In the first experiment, we evaluate how the data shuffling can delay a query job in various network conditions. We deploy Spark SQL atop Alluxio in a 2-node Amazon EC2 [22] cluster. Each node is a `c5.4xlarge` instance with 16 CPU cores and 32 GB memory. We run the TPC-H benchmark workload, where we generate 10 GB source data in Parquet format and execute all

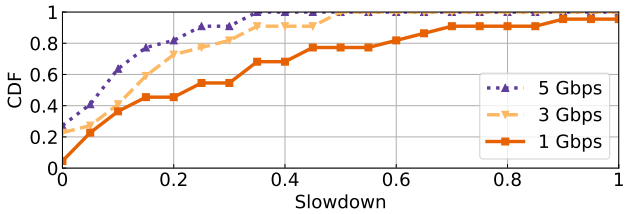
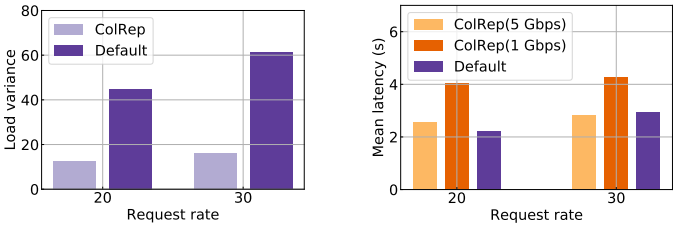


Fig. 3: Distribution of query slowdown caused by data shuffling in various bandwidth configurations.



(a) Load variance (b) Mean read latency

Fig. 4: Comparison of default Alluxio and column-wise replication in terms of load variance and read latency.

the 22 queries in the benchmark, one after another. We evaluate the query performance in two cases, with and without data shuffle. In the *shuffle* case, we place all 10 GB source data in one server, causing data shuffling between two nodes during query execution.[2] In the *non-shuffle* case, we copy the source data to two servers so that the query jobs can be executed locally without communication.

We configure various network bandwidth in the shuffle case and compare the *end-to-end completion time* of each query with that in the non-shuffle case. In particular, we measure the *slowdown* as the normalized query completion delay due to shuffling, i.e.,

$$\text{Slowdown} = \frac{L_{\text{S}} - L_{\text{N}}}{L_{\text{N}}},$$

where L_{S} and L_{N} respectively denote the query completion time in the shuffle and non-shuffle cases.

Fig. 3 depicts the distributions of query slowdown in various network settings. We see that data shuffling is expensive even in a good network condition with 5 Gbps bandwidth, in which 20% of queries are delayed by over 20%. The slowdown becomes more salient as the bandwidth contention increases: when the available bandwidth drops to 1 Gbps, data shuffling slows down 55% of TPC-H queries by more than 20%. While production clusters typically have much higher network bandwidth (i.e., 40-100 Gbps), they usually serve hundreds of queries at the same time, leaving each query even less bandwidth than that in our experiments. We therefore expect an even more significant slowdown caused by shuffling in production environments.

Load balancing at the expense of data shuffling. In our second experiment, we show that load balancing should not be achieved without accounting for the shuffle overhead—failing to do so may result in even slower I/O. Specifically, we consider selective replication *at the granularity of columns* which copies multiple replicas of hot columns (more details in Sec. VI-A). This approach improves load balancing but incurs shuffled data when hot columns are queried together as they are uniformly cached across servers.

We have implemented this *column-wise replication* scheme in Alluxio and evaluated its performance in a 20-node EC2 cluster. Each node is an `m5.xlarge` instance with 4 CPU cores and 16 GB memory. We generate 20 GB columnar data using the TPC-H benchmark and cache them in Alluxio as Parquet files. We submit the read requests following Poisson arrivals at *low* (20 reqs per min) and *high* (30 reqs per min) rates. A read request retrieves multiple columns of data following the column access pattern of the TPC-H queries.

We evaluate column-wise replication (ColRep) against Alluxio's default load-balancing strategy (Default) with 1 Gbps and 5 Gbps network bandwidth. Fig. 4a compares the *load variance* across servers under the two strategies at low and high request rates, where a smaller variance indicates better load balancing. As expected, column-wise replication leads to more balanced load and is robust to the high request rate. Note that we do not differentiate between 1 Gbps and 5 Gbps bandwidth in the figure as the network has no impact on load balancing.

Fig. 4b compares the mean read latency of the two strategies measured in 1 Gbps and 5 Gbps network. Note that in Alluxio, data locality is by default respected, in that a read request is always scheduled onto the server containing the requested data. As no data is shuffled over the network, the mean read latency of Default remains unchanged in the two bandwidth configurations, which we do not differentiate in the figure. However, this is no longer the case for column-wise replication. In Fig. 4b, despite the much improved load balancing, the frequent data shuffling results in > 1.5x longer mean read latency than Default with 1 Gbps bandwidth. In fact, even with 5 Gbps network, column-wise replication still performs worse than Default at a low request rate.

IV. Selective Bundling and Replication

In this section, we present our solution, which we call *selective bundling and replication* (RepBun), that achieves load balancing for structured data without the shuffle overhead. We start with an overview of our key idea followed by a detailed design of the solution.

A. Solution Overview

In a nutshell, given a structured data object stored as a columnar table, RepBun identifies the hot columns with strong co-access patterns and groups them into a *bundle* for selective replication. That is, it copies multiple bundle replicas based on its popularity and stores them uniformly in cache servers. Upon receiving a query job, RepBun checks if the requested columns can be *fully covered* by a bundle replica. If so, it randomly chooses a replica and uses it to serve the query

[2]We disabled passive caching in Spark and Alluxio to enforce remote read.

request. Otherwise, it redirects the query to the original table. In either case, data shuffle is avoided as a query request is always served using the locally cached columns.

Compared with the existing solutions, RepBun achieves three performance benefits. First, by copying the bundle replicas of hot columns, RepBun spreads the load of the read requests evenly to multiple servers, effectively mitigating the hot spots in the cluster. Second, unlike selective partition [8] and column-wise replication (see Sec. III-B), RepBun meets the memory locality requirements of the query jobs, obviating the need for data shuffling. Third, compared with full-table replication, RepBun significantly reduces the memory overhead as it only copies a small number of hot columns.

B. Algorithm Design

Key to realizing the benefits of RepBun is to judiciously determine (1) *which columns* should be grouped into a bundle, and (2) *how many replicas* should a bundle be copied. Given the strong co-access patterns between hot columns (***P2*** in Sec. II-D), we choose to bundle the *top-k hottest columns* for replication. To compute the optimal k and the number of replicas, we formulate an optimization problem and show that it can be efficiently solved using binary search.

Problem formulation. We assume that there are m tables persisted in a cluster of N cache servers, where table i contains n_i columns $\left\{c_i^1, c_i^2, \ldots, c_i^{n_i}\right\}$. For column c_i^j, let s_i^j and p_i^j respectively denote its size[3] and popularity. The expected load of column c_i^j is measured by $l_i^j = p_i^j s_i^j$. Without loss of generality, we assume columns are sorted in descending order of load, i.e., $l_i^1 > l_i^2 > \cdots > l_i^{n_i}$ for all table i.

In RepBun, we group the *hottest* k_i columns of table i into a bundle and copy r_i bundle replicas that are stored *uniformly* in the cluster. This incurs additional memory overhead which is measured by the amount of cache space used to hold all bundle replicas, i.e.,

$$o = \sum_{i=1}^{m} r_i \sum_{j=1}^{k_i} s_i^j. \tag{1}$$

Since data shuffle is avoided in RepBun, achieving load balancing is sufficient to optimize the caching performance. Therefore, our goal is to *minimize the load variance* across servers by determining k_i and r_i for each table i, under the constraint that the incurred memory overhead o must be contained within a given *budget* B. Formally, let X be a *random variable* denoting the total load on any particular server in RepBun. We solve the following optimization problem to minimize the load variance:

$$\begin{aligned} \min_{\{k_i, r_i\}} \quad & \mathrm{Var}(X), \\ \text{s.t.} \quad & o \le B. \end{aligned} \tag{2}$$

Equalizing the load contribution. Recall that in RepBun, the replicas of column bundles and the original table objects are stored *uniformly* in cache servers. Therefore, to attain the minimum load variance, each replica and table should ideally contribute an *equal amount of load*. This can be achieved by configuring the number of replicas of a bundle *in proportion to its load contribution*.

Formally, for table i, let b_i be the load of the query requests that are served using the bundle replicas. In RepBun, these queries only retrieve columns among the hottest k_i and can be fully served using a bundle replica without referring to table i (see Sec. IV-A). We configure a *global scale factor* α and copy r_i replicas of the column bundle, where

$$r_i = \lceil \alpha b_i \rceil . \tag{3}$$

This results in a *uniform load contribution* across bundle replicas, i.e., $b_i / r_i \approx \alpha^{-1}$.

In RepBun, bundle replicas are used to serve only a fraction of the query requests. The remainder of the requests are redirected to the original object containing the entire table i, as their working columns cannot be fully covered by a bundle replica (see Sec. IV-A). Let t_i be the load of those query requests. Both t_i and b_i constitute the entire query loads of table i, i.e.,

$$t_i + b_i = \textstyle\sum_j l_i^j. \tag{4}$$

To minimize the load variance, the table object should contribute an equal amount of load as a bundle replica, i.e.,

$$t_i = b_i / r_i \approx \alpha^{-1}. \tag{5}$$

Key insight. We see from Eqs. (4) and (5) that $b_i \approx \sum_j l_i^j - \alpha^{-1}$. Since the load served by a column bundle (b_i) critically depends on how that bundle is constructed (k_i), we can establish the connection between the scale factor α and k_i. This suggests that to solve the load-balancing problem (2), it is sufficient to *configure the optimal scale factor* α, with which both k_i and r_i can be determined accordingly. In the following, we tackle problem (2) in two steps. We first establish the connection between b_i and k_i. We then configure the optimal scale factor through a binary search.

Step-1: **Quantifying the load served by bundles.** We stress that the query load served by a column bundle cannot be obtained by simply adding the load of its constituent columns, i.e., $b_i \neq \sum_{j=1}^{k_i} l_i^j$. Consider a simple example: a query that requests both the hottest and the coldest columns is served by the original table, and its access to the hottest column should not be included in b_i. In fact, exactly quantifying b_i requires the full knowledge of the *co-access* patterns between columns, which is, however, *unavailable* to cluster caching systems such as Alluxio. As low-level storage, these systems are *agnostic* to the high-level query semantics, and can only track the load of read requests *per-column*, i.e., l_i^j.

We sidestep this limitation with an *indirect* approach that *infers* the load of bundle replicas b_i based on the per-column load information l_i^j. We consider a simple query model. Suppose that in a time window, the caching system has logged q_i queries accessing the columns of table i, with column c_i^j having popularity p_i^j. We assume that column c_i^j is *uniformly* requested by all q_i queries. Meaning, each query has probability p_i^j / q_i to request column c_i^j, and each column is

[3]We normalize the size of all columns to 1, i.e., $\sum_{i=1}^{m} \sum_{j=1}^{n_i} s_i^j = 1$.

requested *independently*. In this simple query model, we derive b_i, the expected load of query requests that can be fully served using a bundle consisting of the hottest k_i columns.

Lemma 1: Assuming each column is requested independently and uniformly by q_i queries, we have

$$b_i = \sum_{j=1}^{k_i} l_i^j \prod_{j'=k_i+1}^{n_i} \left(1 - \frac{p_i^{j'}}{q_i}\right). \quad (6)$$

Proof: Consider a particular query q. Let X_q be a binary indicator where $X_q = 1$ if query q only requests the hottest k_i columns in table i and can be served by the column bundle, and $X_q = 0$ otherwise. We have $\Pr(X_q = 1) = \prod_{j=k+1}^{n_i}\left(1 - \frac{p_i^j}{q_i}\right)$. Let Q be the total number of queries served by the column bundle. We have $E(Q) = \sum_{q=1}^{q_i} E(X_q) = q_i \prod_{j=k+1}^{n_i}\left(1 - \frac{p_i^j}{q_i}\right)$. Let D be the load contribution of a query that is served by the column bundle. We have $E(D) = \sum_{j=1}^{k_i} \frac{p_i^j}{q_i} s_i^j = \frac{\sum_{j=1}^{k_i} l_i^j}{q_i}$.

Given that Q and D are independent, we compute the expected load contributed by the column bundle:

$$b_i = E(QD) = E(Q)E(D) = \sum_{j=1}^{k_i} l_i^j \prod_{j'=k+1}^{n_i} \left(1 - \frac{p_i^{j'}}{q_i}\right).$$

■

While Eq. (6) is derived assuming a simplified query model, our evaluations against the TPC benchmark workloads show that it closely approximates the actual load of requests served by column bundles (more in Sec. VI-E). We therefore use Eq. (6) to estimate b_i based on the per-column load that can be easily tracked in cluster caches.

Combining Eqs. (4), (5) and (6), we establish the relationship between k_i and α, i.e.,

$$\sum_{j=1}^{n_i} l_i^j - \sum_{j=1}^{k_i} l_i^j \prod_{j'=k_i+1}^{n_i} \left(1 - \frac{p_i^{j'}}{q_i}\right) \approx \alpha^{-1}. \quad (7)$$

Eq. (7) suggests that solving the load-balancing problem (2) boils down to finding the optimal scale factor α. Once it is determined, we can compute both the number of columns included in a bundle (k_i) and the number of replicas the bundle is copied (r_i in Eq. (3)).

Step-2: **Configuring the optimal scale factor.** Intuitively, configuring a large scale factor results in more bundle replicas, leading to an increased memory overhead. On the other hand, the more replicas are copied, the better load balancing can be achieved. Based on this intuition, we should configure the *largest* scale factor provided that the memory overhead remains within a given budget. We next show that this intuition indeed holds in problem (2) with the following two theorems.

Theorem 1: The memory overhead grows as scale factor α increases.

Proof: By Eqs. (1) and (3), we rewrite the memory overhead as

$$o \approx \alpha \sum_{i=1}^{m} b_i \sum_{j=1}^{k_i} s_i^j.$$

To prove the statement, we show that both b_i and k_i increase with α. By Eqs. (4) and (5), we have $b_i \approx \sum_{j=1}^{n_i} l_i^j - \alpha^{-1}$, and b_i increases with α. To show that k_i also increases with α, we refer to Eq. (6), from which we see that b_i increases with k_i. Therefore, as α increases, b_i grows, so does k_i. ■

Theorem 2: Assuming independent query access to each table, the load variance decreases as scale factor α increases.

Proof: Let X be the load on any particular server, and L_i be the load of the read quests accessing columns in table i, including both the contributions of the column replicas and the full table. We have $X = \sum_i L_i$ and $\mathrm{Var}(X) = \sum_i \mathrm{Var}(L_i)$ assuming independent query accesses to each table.

Let A_i be a binary indicator where $A_i = 1$ if there is a bundle replica of table i cached in this particular server, and $A_i = 0$ otherwise. Let B_i be similarly defined to indicate if the full-table object is stored in the server. Both A_i and B_i are random variables following the Bernoulli distribution with parameters $\frac{r_i}{N}$ and $\frac{1}{N}$, respectively, where N is the number of cache servers. We rewrite L_i as

$$L_i = A_i \frac{b_i}{r_i} + B_i t_i,$$

and derive its variance as

$$\mathrm{Var}(L_i) = (\frac{b_i}{r_i})^2 \frac{r_i}{N}(1 - \frac{r_i}{N}) + (t_i)^2 \frac{1}{N}(1 - \frac{1}{N}) \approx \frac{b_i + t_i}{\alpha N}. \quad (8)$$

Summing up the variances of all L_i and plugging (4), we have

$$\mathrm{Var}(X) = \sum_i \mathrm{Var}(L_i) \approx \frac{\sum_i \sum_j l_i^j}{\alpha N}.$$

This suggests that $\mathrm{Var}(X)$ decreases with larger α. ■

Theorems 1 and 2 suggest that to solve the optimization problem (2), it suffices to configure the largest scale factor without exceeding the memory budget, i.e., maximizing α subject to $o \leq B$. As the memory overhead increases with α, we apply *binary search* to find the largest α. We show in our evaluations that the optimal scale factor can be efficiently configured in sub-seconds (see Sec. VI-E).

V. Implementation

We have implemented RepBun atop Alluxio [4]. While our implementation assumes Parquet [12] as the columnar storage layer, it can be easily extended to support other columnar data formats such as Apache Arrow [3].

Architecture overview. Fig. 5 depicts the architecture overview of our implementation. RepBun consists of two components: a `master` and multiple `clients`. The `master` keeps track of the per-column popularity, configures the optimal scale factor, and makes the bundling and replication decisions using the algorithms described in Sec. IV. It informs each cache server for local enforcement of column bundling, which then copies the bundle replicas uniformly across the cluster. RepBun runs multiple `clients`, which interact with the query applications (e.g., Spark SQL) through Alluxio API. As illustrated in the figure, a `client` inquires the `master` for a list of locations of the requested columns. It then contacts the corresponding cache servers for local reads. We next elaborate on a few implementation details.

Tracking the per-column popularity. As low-level caching storage for general data objects, Alluxio is agnostic to the

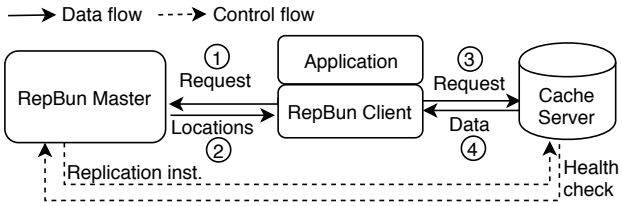


Fig. 5: Architecture overview of RepBun.

Parquet semantics and provides no native support for obtaining the per-column access information. We address this problem by directly retrieving the metadata of Parquet files using tools provided by `Parquet-format` [23]. This allows RepBun to obtain the file offset of each column in a table, which can then be used to differentiate accesses to different columns, enabling RepBun to track the per-column popularity.

Working with row groups. In Parquet, a table is *horizontally* divided into multiple *row groups*, each having a column chunk for every column. RepBun supports working at the level of row groups. That is, it bundles the hot column chunks in a row group and copies multiple bundle replicas. Doing so offers two benefits over working with the entire table. First, as each row group is handled by a separate query task (e.g., Spark SQL), the increased read parallelism leads to improved I/O performance. Second, even in the same table, row groups may have *uneven* popularities as the query jobs would filter out those groups containing no record that satisfies the given predicates (known as *predicate pushdown* [13]). RepBun ignores those cold row groups but copies hot column chunks in hot groups only, leading to reduced memory overhead.

Handling temporal popularity shift. In production environments, data access patterns may change over time. Therefore, RepBun *periodically* performs load balancing to handle the popularity shift. Similar to the prior work [8], [9], RepBun makes new bundling and replication decisions every 12 hours based on the collected per-column load information in the past 24 hours. This is sufficient to maintain load balancing as data popularity is usually stable in a short period of time, e.g., days [8], [9]. Since bundling and replication are only needed once per 12 hours, the overhead of periodic load balancing is negligible (more in Sec.VI-E).

VI. Evaluation

We evaluate RepBun through EC2 deployment against both the table read requests in the TPC-H [16] benchmark and real Spark SQL queries. We summarize the highlights of our evaluations as follows:

- Compared with selective replication, RepBun achieves better load balancing using 50% less memory space; when configured with the same memory budgets, RepBun reduces the average read latency by 30% and the tail by 36% (Sec. VI-B).
- RepBun is resilient to intensive stragglers, improving the tail latency by up to 38% over the replication strategy (Sec. VI-C).
- When deployed together with Spark SQL, RepBun accelerates 40% of SQL queries in TPC-H benchmark by over 15% (Sec. VI-D).
- RepBun configures the optimal scale factor α in subseconds and completes bundling and replication for 50 GB data in less than 90 seconds (Sec. VI-E).

A. Methodology

Cluster settings. We deploy RepBun in a 21-node Amazon EC2 cluster with 1 Gbps network. Each node is an `m5.xlarge` instance with 4 CPU cores and 16 GB memory. We use 20 nodes as the cache servers and one as the master.

Workloads. We use the TPC-H benchmark and generate 20 GB columnar data in Parquet format. We configure Poisson arrivals of the synthesized read requests, each retrieving multiple columns of data in a table following the co-access patterns of the TPC-H queries. We configure the arrival rate of 20-40 requests per minute (rpm). We use this synthesized read requests to evaluate the I/O performance of our solution. We also use the real Spark SQL queries to evaluate how the improved I/O leads to faster query completion when RepBun is deployed with Spark SQL.

Memory budget for replication. We normalize the memory budget for replication by the amount of cache space used to hold the original data. Unless otherwise specified, we set the budget to be 0.5, meaning, the data amount of bundle replicas should be no more than 50% of the size of the original data.

Metrics. We use the mean and tail (95^{th} percentile) read latencies as the primary performance metrics. In addition, we measure the degree of load imbalance by the *load variance* among cache servers. In each server, the load is measured by the total amount of data reads.

Baselines. We benchmark RepBun against three baselines.

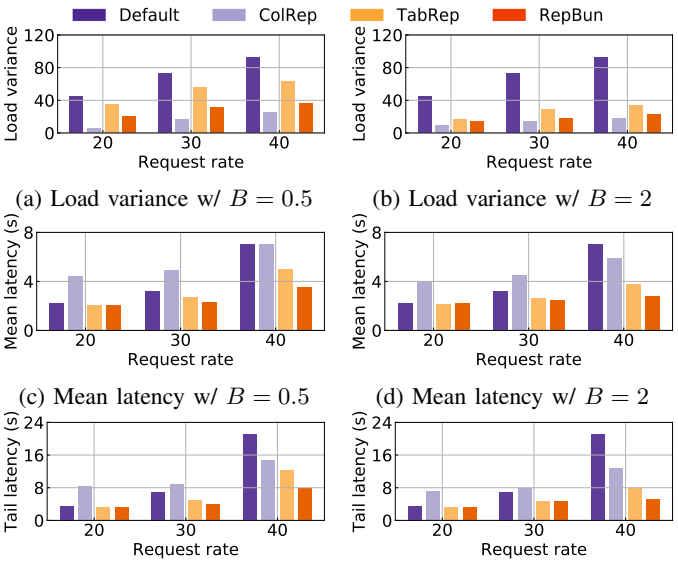
(1) Alluxio's default load-balancing strategy (***Default***) divides data objects into partitions of a fixed size, where partitions are placed on servers in a *round-robin* manner, irrespective of their popularity. In the experiments, we employ the default setting with the partition size of 512 MB.

(2) Table-wise selective replication (***TabRep***) copies multiple replicas of hot tables. For a fair comparison, we determine the number of replicas of a table by configuring the optimal scale factor α within a given replication budget (***Step-2*** in Sec. IV-B). This ensures an equal load contribution of the original table and its replicas, and uniformly caching them in the cluster improves load balancing.

(3) Column-wise selective replication (***ColRep***) copies multiple replicas of hot columns based on their popularity. Similar to TabRep, the number of replicas of a column is determined by configuring the optimal α. Replicas of columns are uniformly cached in the cluster for improved load balancing.

B. Load Balancing and Read Performance

We evaluate RepBun against the three baselines using the read requests synthesized from the TPC-H benchmark under two replication budgets: 0.5 (low) and 2 (high).



(a) Load variance w/ $B = 0.5$ (b) Load variance w/ $B = 2$

(c) Mean latency w/ $B = 0.5$ (d) Mean latency w/ $B = 2$

(e) Tail latency (95^{th}) w/ $B = 0.5$ (f) Tail latency (95^{th}) w/ $B = 2$

Fig. 6: Load variance and read latency (mean and 95^{th} percentile) of the four load-balancing solutions with low ($B = 0.5$) and high ($B = 2$) replication budgets.

Load balancing. Figs. 6a and 6b compare the load variance of the four solutions with low and high budgets for replication, respectively. In general, higher request rate leads to more severe load imbalance. RepBun consistently outperforms TabRep and Default with more balanced load, yet falls behind ColRep which copies replicas at the finest granularity (i.e., columns). However, as we show next, ColRep results in heavy data shuffle, leading to even longer read latency.

Read latency. Figs. 6c-6f depict the mean and tail (95^{th}) latencies of the four solutions with low and high replication budget. RepBun consistently results in the shortest latency and achieves even more prominent speedup over the three baselines as the request rate increases. In particular, compared with TabRep (ColRep), RepBun improves the mean and tail latencies by up to 30% (55%) and 36% (60%), respectively. Even at high budget (e.g., $B = 2$) where TabRep achieves comparable latencies as RepBun, the latter uses 50% less cache space than the former. We attribute the performance advantage of RepBun to its capability of improving load balancing without data shuffling and high memory overhead.

C. Resilience to Stragglers

Replication-based caching solutions are usually resilient to *stragglers* [7], [9], i.e., servers that run much slower than others. We therefore compare RepBun with the other two replication baselines. Specifically, we manually inject stragglers by sleeping an I/O thread with probability 0.05 and delaying the read completion by a factor randomly drawn from the distribution profiled in the Microsoft Bing cluster trace [24]. Fig. 7 shows the box plots of read latencies of RepBun, TabRep and ColRep, where the request rate is 40 per minutes. RepBun outperforms TabRep and ColRep, reducing the tail latency by up to 38% and 51%, respectively.

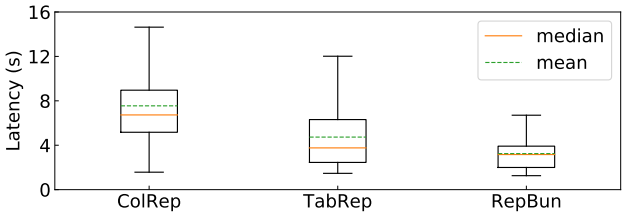


Fig. 7: Comparison of read latencies of ColRep, TabRep, and RepBun with injected stragglers. The request rate is 40 per minute. Boxes depict the 25^{th}, 50^{th}, and 75^{th} percentiles, and whiskers depict the 5^{th} and 95^{th} percentiles.

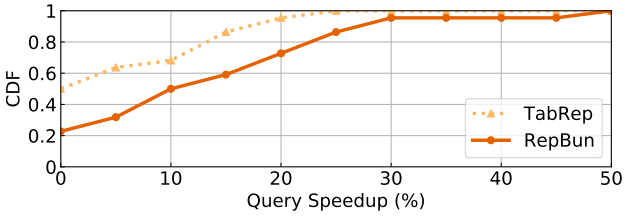


Fig. 8: Distribution of SQL query speedup over Default.

D. End-to-end Query Performance

Our previous experiments mainly focus on evaluating the I/O performance using synthesized read requests from the TPC-H benchmark. We next evaluate how the improved I/O leads to faster completion of SQL queries. We deployed Spark SQL in the same cluster and run all TPC-H queries in sequence. For each query, we measure its completion time in RepBun and TabRep and calculate the query *speedup* of the two solutions over Default. Here, the *speedup* is defined as:

$$\text{Speedup} = \frac{L - L_{\text{Default}}}{L_{\text{Default}}} \times 100\%, \tag{9}$$

where L and L_{Default} denote the end-to-end query completion time under the concerned solution (i.e., RepBun and TabRep) and Default, respectively.

Fig. 8 depicts the distributions of query speedup of RepBun and TabRep over Default. While both solutions leads to faster query completion, the speedup of RepBun is more salient: 40% of TPC-H queries are accelerated by over 15%.

E. System Overhead and Load Estimation

After showing the performance advantage of RepBun over the other solutions, we turn to its overhead and load estimation.

Configuration overhead. We start to show that RepBun can quickly configure the optimal scale factor α. We measure the time to find the optimal α using binary search (see Sec. IV-B) with 1-10k data objects. Fig. 9 depicts the mean configuration time in three trials, where the error bar measures the variance. We see that even with 10k objects, the optimal scale factor can be quickly configured in 400 ms.

Replication overhead. We next evaluate the overhead of creating bundle replicas. Fig. 10 shows the mean completion time for replication under various memory budgets in three

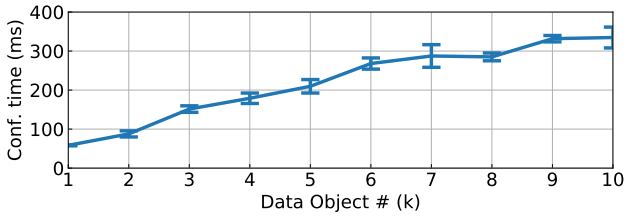


Fig. 9: The configuration time for finding the optimal scale factor α with various numbers of data objects.

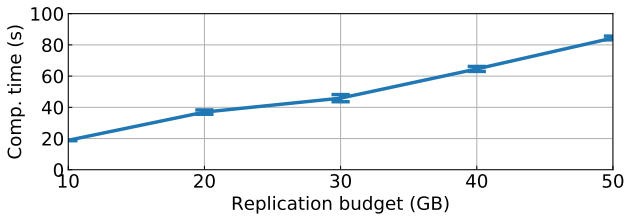


Fig. 10: The completion time for bundle replication under various budgets.

trials, where the error bar measures the variance. With more data to copy, the replication time grows linearly. Nevertheless, RepBun can still finish copying 50 GB replicas in less than 90 seconds. Since copying bundle replicas is only needed every 12 hours (see Sec. V), its overhead is less of a concern.

Accuracy of load estimation. Recall that in our algorithm (see Sec. IV-B), we use Lemma 1 to estimate the load of column bundles. To evaluate how accurate such load estimation is, we compare it with the *actual load* of requests served by bundle replicas, which we obtained by manually logging the co-accessed columns for each query. Fig. 11 compares our estimation and the actual load of requests that can be served by a bundle consisting of the hottest k columns in `lineitem`, a representative table in TPC-H. Our estimation closely approximates the actual load, leading to *almost identical* decisions for the number of columns a bundle should contain and the number of replicas a bundle should copy. In Fig. 12, we further compare the performance of two RepBun implementations that make the bundling and replication decisions based on the actual load and the estimation. Both implementations achieve the similar performance in load balancing and read latency, suggesting that our load estimation is fairly accurate.

VII. Related Work

Structured data have clear semantics of schema and are widely used in both the traditional web applications (e.g., sale transactions) and big data analytics. While structured data are usually organized as rows for traditional transactional workloads [25]–[27], they are more commonly stored in columnar formats such as Parquet [12] and Arrow [3] for analytics workloads. Popular data analytics frameworks, such as Spark [13], Hive [14], and Impala [28], are highly optimized for columnar storage with significantly improved I/O performance [10], [29], [30]. RepBun focuses on columnar data for analytical workloads.

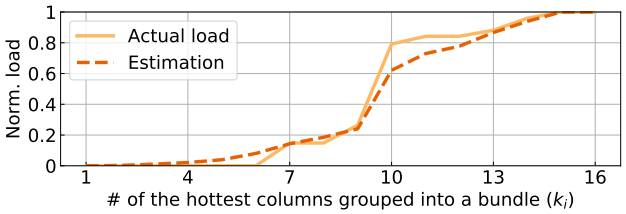
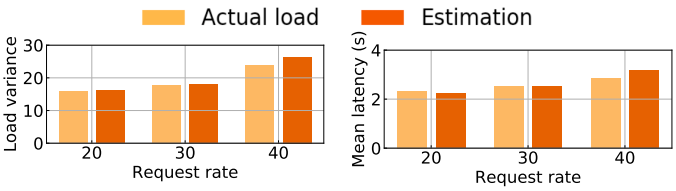


Fig. 11: Comparison of our estimation and the actual load of requests that can be served by a bundle consisting of the hottest k columns in table `lineitem`.



(a) Load variance (b) Mean latency

Fig. 12: Performance comparison of two implementations that make the bundling and replication decisions based on the actual load and the estimation.

Load balancing. In addition to replication, partition, and erasure coding (Sec. II-B), *in-network caching* is another technique for load-balancing storage clusters [31]–[33]. By caching hot objects in the memory of network devices (e.g., programmable switches [33]), the read requests can be directly served in network without reaching the storage nodes. However, it requires expensive hardwares (e.g. programmable switches with high-speed cache) and is only effective for small data objects due to the limited cache space of network devices.

Reducing data shuffle. Querying multiple tables from different servers incurs heavy communication overhead due to data shuffle. Existing solutions employ graph-based partition to reduce shuffle operations [27], [34]–[36]. RepBun mainly focuses on data analytics queries accessing columns in a single table and is hence orthogonal to these works.

VIII. Conclusion

In this paper, we have presented RepBun, a load-balanced, shuffle-free cluster caching for structured data in columnar format. RepBun groups hot columns of a table into a bundle and creates multiple replicas of that bundle based on its popularity. We have proposed an efficient algorithm that judiciously determines which columns should be grouped into a bundle and how many replicas a bundle should be copied. We have implemented RepBun atop Alluxio with Parquet as the columnar storage layer. Extensive evaluations show that RepBun leads to improved load balancing and faster query completion, significantly outperforming the existing solutions.

Acknowledgements

This research was supported by RGC ECS (contract number 26213818). Minchen Yu and Yunchuan Zheng were supported in part by the Huawei PhD Fellowship Scheme.

REFERENCES

[1] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab *et al.*, "Scaling Memcache at Facebook," in *Proc. USENIX NSDI*, 2013.

[2] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proc. ACM SoCC*, 2014.

[3] Apache Arrow. [Online]. Available: https://arrow.apache.org

[4] Alluxio. [Online]. Available: https://www.alluxio.io/

[5] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The RAMCloud storage system," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 7:1–7:55, Aug. 2015.

[6] Redis. [Online]. Available: https://redis.io

[7] K. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran, "EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding," in *Proc. USENIX OSDI*, 2016.

[8] Y. Yu, R. Huang, W. Wang, J. Zhang, and K. B. Letaief, "SP-Cache: Load-balanced, redundancy-free cluster caching with selective partition," in *Proc. IEEE/ACM International Conference for High Performance Computing, Networking, Storage, and Analysis (SC18)*, 2018.

[9] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: Coping with skewed content popularity in mapreduce clusters," in *Proc. ACM EuroSys*, 2011.

[10] Y. Cheng, A. Gupta, and A. R. Butt, "An in-memory object caching framework with adaptive load balancing," in *Proc. ACM EuroSys*, 2015.

[11] Y.-J. Hong and M. Thottethodi, "Understanding and mitigating the impact of load imbalance in the memory caching tier," in *Proc. ACM SoCC*, 2013.

[12] Apache Parquet. [Online]. Available: https://parquet.apache.org/

[13] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational data processing in spark," in *Proc. ACM SIGMOD*, 2015.

[14] Apache Hive. [Online]. Available: https://hive.apache.org/

[15] Presto. [Online]. Available: http://prestodb.github.io/

[16] TPC-H. [Online]. Available: http://www.tpc.org/tpch/

[17] TPC-DS. [Online]. Available: http://www.tpc.org/tpcds/

[18] TPCx-BB. [Online]. Available: http://www.tpc.org/tpcx-bb/

[31] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky, "Small cache, big effect: provable load balancing for randomly partitioned cluster services," in *Proc. ACM SoCC*, 2011.

[19] Memcached. [Online]. Available: https://memcached.org/

[20] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "PACMan: Coordinated memory caching for parallel jobs," in *Proc. USENIX NSDI*, 2012.

[21] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. ACM SIGMETRICS*, 2012.

[22] Amazon EC2. [Online]. Available: https://aws.amazon.com/ec2/

[23] Apache Parquet Format. [Online]. Available: https://github.com/apache/parquet-format

[24] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri," in *Proc. USENIX OSDI*, 2010.

[25] A. Pavlo, C. Curino, and S. Zdonik, "Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems," in *Proc. ACM SIGMOD*, 2012.

[26] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker, "E-store: fine-grained elastic partitioning for distributed transaction processing systems," in *Proc. VLDB Endow.*, 2014.

[27] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: A workload-driven approach to database replication and partitioning," in *Proc. VLDB Endow.*, 2010.

[28] Apache Impala. [Online]. Available: https://impala.apache.org

[29] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-stores vs. row-stores: how different are they really?" in *Proc. ACM SIGMOD*, 2008.

[30] H. Plattner, "The impact of columnar in-memory databases on enterprise systems: Implications of eliminating transaction-maintained aggregates," in *Proc. VLDB Endow.*, 2014.

[32] X. Jin, X. Li, H. Zhang, R. SoulÃl', J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing key-value stores with fast in-network caching," in *Proc. ACM SOSP*, 2017.

[33] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica, "DistCache: Provable load balancing for large-scale storage systems with distributed caching," in *Proc. USENIX FAST*, 2019.

[34] K. A. Kumar, A. Quamar, A. Deshpande, and S. Khuller, "SWORD: Workload-aware data placement and replica selection for cloud data management systems," *VLDB*, vol. 23, no. 6, pp. 845–870, Dec. 2014.

[35] Y. Nam, M. Kim, and D. Han, "A graph-based database partitioning method for parallel OLAP query processing," in *Proc. IEEE ICDE*, 2018.

[36] E. Zamanian, C. Binnig, and A. Salama, "Locality-aware partitioning in parallel database systems," in *Proc. ACM SIGMOD*, 2015.