BlowFish: Dynamic Storage-Performance Tradeoff in Data Stores

Paper #85, 14 Pages

Abstract

We present BlowFish, a distributed data store that admits a seamless and dynamic tradeoff between storage and throughput for point queries on unstructured and semi-structured data. To achieve this, BlowFish uses a new data structure that stores a compressed representation of the input data, enables queries directly on this compressed representation, and in addition, allows dynamically changing the compression factor. We show that navigating along this storage-performance curve can be done at fine grained time scales and is often more effective than selectively replicating hot objects.

We evaluate BlowFish over workloads that have a time-varying query distribution across data shards, for both gradual and spiked variations. Evaluation over a distributed Amazon EC2 cluster shows that BlowFish requires 1.8–4.5× less storage compared to selective replication, while navigating along the storage-performance curve significantly faster than the rate at which real-world workloads usually change.

1 Introduction

Random access and *search* are the two fundamental operations performed on modern data storage systems. For instance, most data stores, *e.g.* document stores [3], key-value stores [2, 5, 11, 16, 17, 19, 25, 26] and multi-attribute NoSQL stores [1, 3, 7, 12, 14, 18, 22, 32] support random access at the granularity of records. Many of these data stores [1, 3, 7, 18, 22] also support search on underlying records. Similarly, storage systems for genomics (*e.g.*, [24]) need to perform a large number of search operations.

Most of the above systems often store an amount of data much larger than available fast storage[1], *e.g.*, SSD or main memory. Since these fast storage medium are at least an order of magnitude faster than slower magnetic disks drive, most of the above data stores attempt to maximize the performance (throughput or latency at high percentiles) using caching, that is, pushing as much data as possible into faster storage.

One way to push more data in faster storage is to use compression. However, traditional compression techniques are not suitable for random access and search workloads since the overhead of data decompression during query execution is comparable to the overhead of answering queries off slower storage. To overcome this limitation, a number of recent systems [4, 6, 7, 24] execute queries directly on compressed data. These systems have shown that, for random access and search workloads, executing queries on compressed data leads to significant improvement in system throughput.

However, existing systems that execute queries on compressed data lose their performance advantages in case of dynamic workloads where (i) the set of hot objects changes rapidly over time [9, 15, 27, 33], and (ii) a single copy of the compressed object is not enough to efficiently server a hot object. Several studies from real-world production clusters [9, 10, 15, 27, 33] have shown that such dynamic workloads are a norm in large-scale clusters. Moreover, transient failures [28] that render data unavailable temporarily can create high temporal skew since queries are rerouted to remaining data replicas. See [15, 27, 33] for examples. For such workloads, systems that execute queries on compressed data suffer from suboptimal performance since changing the compression factor requires reconstructing the data structures from scratch, making it infeasible to match the rate at which workloads change.

The traditional way to overcome this limitation is using selective replication [8], that is, creating additional replicas for hot objects. These replicas can be stored across different machines, thus exploiting available fast storage efficiently. However, selective replication suffers from two disadvantages. First, it only provides coarse-grained support to handle dynamic workloads — each replica increases the throughput by 2× while incurring an additional storage overhead of 1×. Second, rather non-intuitively and as discussed later, this tradeoff of increasing throughput by 2× using 1× additional storage overhead may not be optimal since executing queries on compressed objects have non-linear overhead in terms of compression factors [4, 6, 7, 20, 24].

[1] To support search without scanning the dataset, many of the above stores construct secondary indexes. These indexes can be significantly larger than the input data itself [7]. The size of these indexes, combined with the size of the input data to support random access, can be much larger than the available fast storage. Throughput the paper, we collectively refer the indexes combined with the input as "data".

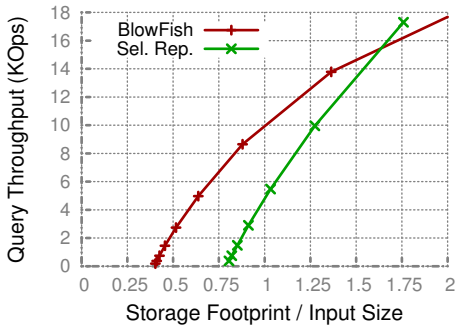

Figure 1: Navigating the storage-performance curve using BlowFish is more effective than Selective Replication since the former provides a super-linear increase in throughput as the storage footprint is increased (until a certain point).

We present BlowFish, a distributed data store that allows seamless and dynamic tradeoff between compression ratio and performance, and show that in many cases navigating this tradeoff is more effective (that is, throughput increase per unit of storage) than selective replication. Specifically, BlowFish allows applications to navigate the storage-performance tradeoff along a smooth curve (see Figure 1), and has three desirable properties. First, BlowFish achieves a better tradeoff between storage and throughput than selective replication of compressed objects. For instance, in Figure 1, BlowFish can increase the storage overhead by $1\times$ (going from storage overhead of 0.4 to 0.8 the data size), while increasing the *per-thread* throughput by almost $5\times$, much better than selective replication. Second, BlowFish allows a finer grained tradeoff. While with selective replication we can only add one replica at a time, BlowFish allows one to increase the storage overhead fractionally, just enough to meet the performance goals. Finally, navigating from one operating point to another along this curve can be done fast enough to meet the demands of many dynamic workloads.

We apply BlowFish to two interesting use cases from real-world production clusters: (1) gradual workload variations in over time [9]; and (2) spiked variations in workloads over time (*e.g.*, due to transient machine failures [28]). We compare the performance of BlowFish against Selective Replication [8] and show that: (1) for skewed workloads, BlowFish can achieve throughput similar to Selective Replication using 1.8–$4.5\times$ lower storage overhead; (2) BlowFish can adapt to time-varying gradual workloads at a rate much faster than the rate at which real-world workloads usually change; (3) even for spiked variations in query workloads (as much as by $3\times$), BlowFish can navigate along the storage-performance tradeoff in less than 5 minutes.

2 BlowFish Overview

In this section, we give an overview of BlowFish. We start with a description of storage model for existing techniques that execute queries on compressed data (§2.1) and how BlowFish achieves the storage-performance tradeoff for these techniques. We then outline target applications and workloads for BlowFish, including a set of assumptions that BlowFish makes (§2.2). Finally, we provide a high-level overview of BlowFish design (§2.3).

2.1 Executing Queries on Compressed Data

Traditional compression techniques allow pushing more data in faster storage, but fail to provide throughput advantages for point queries since the cost of data decompression for each query is comparable to the overhead of executing queries off slower storage. Several recent systems [4, 6, 7, 24] have shown that, for point queries, it is much more efficient to execute queries directly over the compressed representation. These systems can, thus, achieve as much as an order of magnitude higher throughput when data does not fit in faster storage [7, 24]. However, these systems are unable to adapt to spatial and temporal variations in query workloads since the underlying compression scheme is essentially static. BlowFish is designed to overcome this limitation. We give a high-level overview of how systems above execute queries on compressed data, focusing on details relevant to BlowFish design; see [4, 6, 7, 24] for a detailed description.

Storage Model. BlowFish uses a data storage model similar to above systems [4, 6, 7, 24]. In particular, input data is sharded across the set of servers, and each server may store a number of shards. Each shard is compressed independently using the compression technique in the respective systems. Moreover, each shard may be replicated along multiple servers to provide fault tolerance as well as load balancing. What differentiates BlowFish from above systems is that the compression factor in BlowFish can vary across shards (or, even across shard replicas) and across time. This "heterogeneous" compression factor across shards and across time allows BlowFish to adapt to dynamic workloads.

Intuition: Compression via Sampled Arrays. All known techniques for executing queries directly on compressed data [21, 29–31], including the ones used in existing systems [4, 6, 7, 24], achieve compression using a similar idea. These techniques require storing three arrays `A1`, `A2` and `A3`. `A1` and `A2` enable search and random access, respectively, but are large in size. To achieve a compressed representation, only a few sampled values (*e.g.*, for sampling rate α, value at indexes $0, \alpha, 2\alpha, ..$) from these two

arrays are stored. However, unsampled values are required to execute queries. The array `A3` has two interesting properties (1) it allows to compute any unsampled value for the above two arrays on the fly; and (2) it is small in size. Thus, `A3` is stored as is, enabling queries on compressed data. These techniques thus have a fixed storage cost (corresponding to the leftmost point in Figure 1) due to storing `A3`, and a variable storage cost that depends on the sampling rate of `A1` and `A2`.

Intuition: Storage versus Performance Tradeoff. We observe that the storage versus performance tradeoff for systems that execute queries on compressed data (Figure 1) can be achieved by varying the sampling rate α for the two sampled arrays. Specifically, a small value of α indicate that more sampled values are stored thereby increasing the storage cost but also improving the performance; on the other hand, a large value of α leads to a smaller storage overhead but lower system throughput.

Data Model and API. Most of the systems [4, 6, 24] that execute queries on compressed data support storing, retrieving and querying flat (unstructured) files. Succinct [7] showed that this interface can be used to model many powerful abstractions for semi-structured data like document stores [3], key-value stores [5, 16, 25] and multi-attribute NoSQL stores [12, 18, 22]. BlowFish uses the same data model and supports queries on both unstructured and semi-structured data. BlowFish also supports the same set of queries as above systems, namely, random access using `extract(offset, length)` and searching for arbitrary strings using `search(str)`.

2.2 Target Applications and Workloads

BlowFish makes three fundamental assumptions regarding system bottlenecks and query workloads. First, *systems are limited by capacity of faster storage*, that is operate on data sizes that do not fit entirely into the fastest storage medium. The rationale is that, for real-world datasets, indexes to support search queries can be as much as 8× larger than the input data itself [7, 23] making it hard to fit the entire data in fastest storage especially for purely in-memory data stores (*e.g.*, Redis [5], MICA [25], RAMCloud [26]). Second, as performance benchmarks over existing data stores [25] show, BlowFish assumes that most of the existing data stores are *not* bottlenecked by the network capacity. Third, BlowFish assumes that the data can be sharded in a manner than most search queries do not require touching each server in the system. Most real-world datasets and query workloads admit such sharding schemes [15, 27, 33]. Finally, BlowFish naturally assumes that the query workloads vary across data items [9, 10, 15, 27, 33] and over time [9, 15, 27, 28, 33].

2.3 BlowFish Design Overview

We now provide a high-level overview of BlowFish design. BlowFish uses a system architecture similar to existing data stores, *e.g.*, Cassandra [22], ElasticSearch [1] and Succinct [7]. Specifically, BlowFish comprises of a set of servers that store the data as well as execute queries (see Figure 2). Each server shares a similar design (§3), comprising of compressed data shards, a *request queue* per shard that keeps track of outstanding queries, and a special module *server handler* that triggers navigation along the storage-performance curve and schedules queries.

Each data shard is stored using a compressed representation from systems that execute queries on compressed data [4, 6, 7, 24]. However, the data structures used in existing systems are essentially static, that is, changing the compression factor over time is very expensive. To overcome this limitation, we introduce a new data structure, *Layered Sampled Array* (LSA), that enables a *dynamic storage-throughput tradeoff* while maintaining the other desirable properties of above systems. In particular, LSA enables navigating the tradeoff depicted in Figure 1 at fine-grained time scales thus allowing a seamless and dynamic tradeoff between storage and system throughput. We describe LSA design in detail in §3.1.

To trigger navigation along the storage-performance tradeoff curve, BlowFish servers also maintain a request queue per data shard. All incoming queries are enqueued in the queue for the respective shards, and the queue length for each shard is monitored periodically. When the request rate for a particular shard is within the response rate[2], the queue size remains minimal. On the other hand, when the request rate for a particular shard increases beyond the supported rate, the request queue starts building up and the request queue length for this shard increases (see Figure 3). Once the request queue crosses a certain threshold, the navigation along the storage-performance tradeoff curve is triggered either using the remaining storage on the server or by reducing the storage overhead of a relatively lower loaded shard. BlowFish internally implements a number of optimizations for selecting navigation triggers, maintaining request hysteresis to avoid unnecessary oscillations along the storage-performance tradeoff curve, storage management during navigation and ensuring correctness in query execution during the navigation. We discuss these design details in §3.2 and §3.3.

BlowFish uses the server handler module to monitor request queues and trigger navigation along the storage-performance tradeoff curve. The server handler module also maintains and distributes information about request

[2] system throughput at the current operating point on the storage-performance tradeoff curve

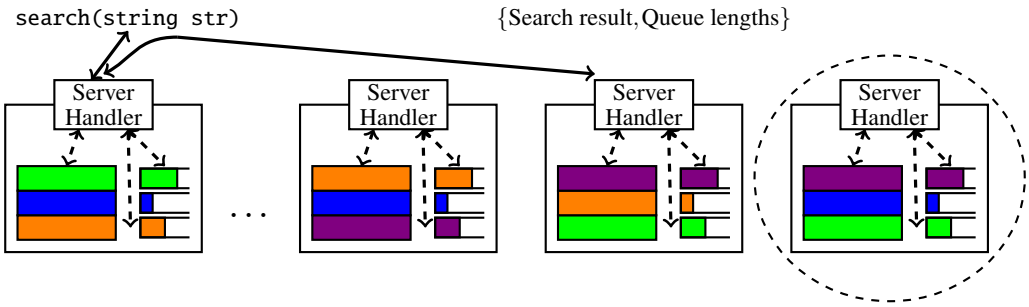

Figure 2: Overall BlowFish architecture. Each server has an architecture similar to the one shown in Figure 3. Queries are forwarded by Server Handlers to appropriate servers, and query respones encapsulate both results and queue lengths at that server.

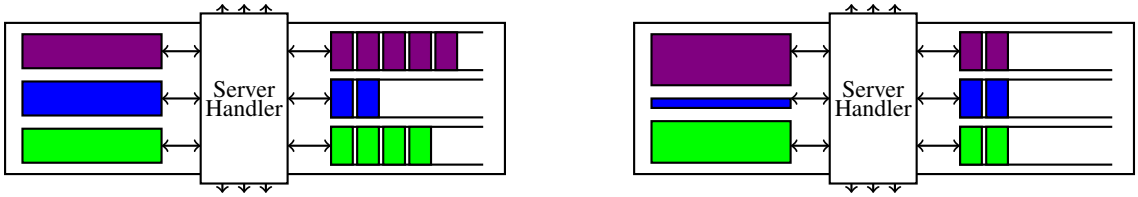

Figure 3: Main idea behind BlowFish: (left) the state of the system at time t; (right) the state of the shards after BlowFish adapts — the shards that have longer outstanding queue sizes adapt their storage footprint to a larger one, thus serving larger number of queries per second than at time t; the shards that have smaller outstanding queues, on the other hand, adapt their storage footprint to a smaller one thus matching the respective request rates.

queues across the system, and uses this information for scheduling queries across shards and across shard replicas. In particular, spatial and temporal skew in query distribution across data shards (and across replicas within a shard) introduce a significant amount of dynamism and heterogeneity in BlowFish's system state (see Figure 3 (right)). Shards on a server can have varying storage footprint and as a result, varying response rates. Moreover, storage footprint and response rates may vary across multiple replicas. To schedule queries across such a dynamic and heterogeneous system, each server handler uses the request queues. Specifically, each server handler encapsulates the "local" request queue lengths in its query responses; a server handler, upon receiving a query response, decapsulates the queue lengths and updates its local metadata to record the new queue lengths. BlowFish uses a simple scheduling algorithm that takes these queue lengths into account, performs close to optimal load balancing and avoids unnecessary hotspots. We discuss this scheduling algorithm in detail in §3.3.2.

3 BlowFish Design

BlowFish servers consist of three main modules. First, storing data shards using the Layered Sampled Array that allows seamless and dynamic navigation along the storage-performance tradeoff curve. Second, request queues for local shards, and metadata that holds request queue lengths for all shards (and replicas) in the system. And third, a server handler that triggers navigation along the tradeoff and schedules queries.

3.1 Layered Sampled Array

BlowFish stores each data shard using a compressed representation that enables queries directly on the compressed representation. We describe the BlowFish ideas using the compression technique from Succinct [7]. However, the compression technique in Succinct is essentially static. BlowFish adapts the technique in Succinct using a new Layered Sampled Array (LSA) data structure, that enables seamless and dynamic navigation along the storage-performance tradeoff curve of Figure 1 while maintaining all the properties of Succinct data structures. We describe LSA below.

Consider an array `A`, and let `SA` be another array that stores a set of *sampled-by-index* values from `A`, that is, for sampling rate α, `SA` value at index `idx` stores `A` value at index $\alpha \times$ `idx`. We call α the *sampling rate*. For instance, for an array `A = {6, 4, 3, 8, 9, 2}`, the sampled-by-index array with sampling rate 4 and 2 are given by SA_4 = `{6, 9}` and SA_2 = `{6, 3, 9}`, respectively. Note that modifying the sampling rate from 4 to 2 (or, vice versa) for SA is heavy-weight — it requires allocating memory, computing the newly sampled values, re-writing the pre-

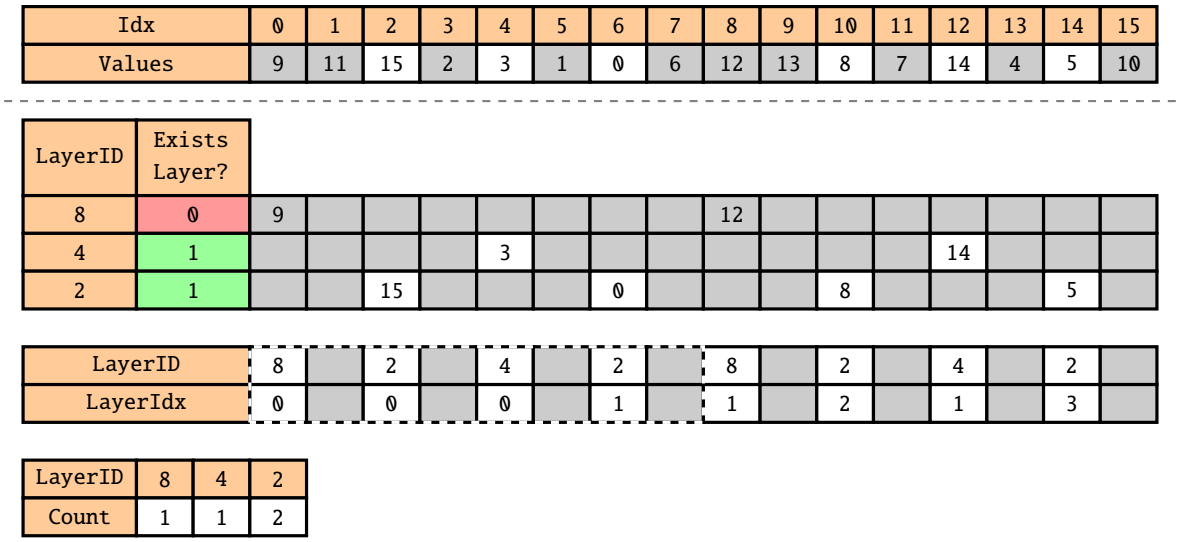

Figure 4: Illustration of *Layered Sampled Array* (LSA). The array above the dashed line is the original unsampled array. In LSA, each layer stores values for sampling rate given by `LayerID`, modulo values that are already stored in upper layers (in this example, sampling rates $8, 4, 2$). Layers are added and deleted at the bottom; that is, layer `LayerID=2` will be added if and only if all layers with sampling rate $4, 8, 16, ..$ exist. Similarly, `LayerID=2` will be the first layer to be deleted.The `ExistsLayer` bitmap indicates whether a particular layer exists (1) or is deleted (0). Gray values indicate unsampled values. `LayerID` and `ExistsLayer` allow us to check whether or not value at any given index `idx` is stored in LSA — we find the largest existing `LayerID` that is a proper divisor of `idx`. Note that among first 8 values in original array, 1 is stored in topmost layer, 1 in the next layer and 2 in the bottommost layer. This observation allows us to find the index into any layer `LayerIdx` where the corresponding sampled value is stored.

viously sampled values along with the new ones into the new array, and finally deleting the old array.

LSA emulates the functionality of the SA, but stores the sampled values in multiple *layers*, together with a few auxiliary structures (see Figure 4). Storing sampled values across multiple layers allows LSA to efficiently change the sampling rate since new layers can be created and filled in with newly sampled values independent of existing layers; increasing the sampling rate is even easier since the desired layers can be deleted without affecting the other layers. Fine-grained addition and deletion of layers allows fine-grained changes in the sampling rate, which on the other hand, affects the storage footprint and query latency — higher sampling rates require higher storage but allow lower query latency since most values would already be sampled whereas smaller sampling rates require lower storage but higher query latency. LSA, thus, enables a smooth tradeoff between query latency and storage footprint at fine-grained time intervals.

The auxiliary data structures stored in LSA enable a simple and low overhead algorithm to emulate the functionality of SA on top of LSA, as shown in Figure 4. Note that looking up a value in LSA is *agnostic* to the layers in the system, that is, the lookup can proceed irrespective of how many and which particular layers exist.

3.2 Request Queues

Each BlowFish server maintains a queue of outstanding queries per shard stored locally at the server, referred to as *request queues*. At each server, the length of request queues provide a close approximation to the load on the shard. Larger request queue lengths indicate that a larger number of the requests for the shard are being queued up, implying that the shard is observing more queries than it is able to serve. Similarly, smaller request queue lengths indicate an underloaded data shard. BlowFish uses these request queues to trigger navigation along the storage-performance tradeoff curve. In particular, each request queue is monitored periodically for its length, and a decision is made whether or not to trigger increase or decrease of storage overhead based on the queue length.

Increasing (decreasing) a shard's storage footprint requires reducing (increasing) the sampling rate for the corresponding LSA (§3.1). While increasing the sampling rate simply requires deleting layers and hence is extremely fast, decreasing the sampling rate is much slower. To that end, BlowFish needs to carefully react to changes in queue lengths. If the system reacts to transient spikes in request rates, it would result in wasted computation and added load on already loaded servers due to wasteful creation of layers. In addition, we do not necessarily want to

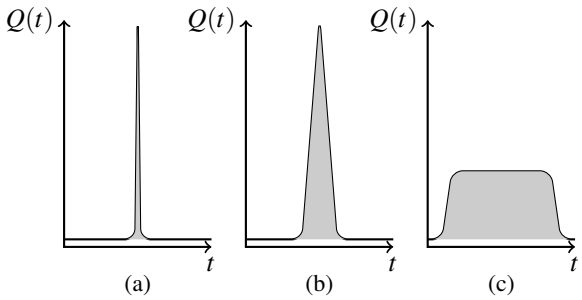

Figure 5: Three different scenarios of queue length ($Q(t)$) variation with time (t). (a) shows a very short-lasting "spike", (b) shows a longer lasting spike while (c) shows a persistent "plateau" in queue-length values. BlowFish should ideally ignore spikes as in (a) and attempt to adapt to the queue length variations depicted in (b) and (c).

reduce a shard's storage overhead if the queue lengths are empty since this may also be transient affect. In order to avoid such cases, the queue lengths are exponentially averaged over time to remove noise in the perceived request rate at a shard.

The exponential averaging scheme used in BlowFish monitors the queue length after every δ time units, and computes the average queue length at time t as:

$$Q_t^{avg} = \alpha \times Q_t + (1-\alpha) \times Q_{t-\delta}^{avg} \quad (1)$$

The parameters α and δ provide two knobs to set BlowFish's sensitivity to changes in queue length. α is a fraction ($\alpha < 1$) that determines the contribution of more recent queue length values to the average — larger values of α assign higher weightage to more recent values in the average. δ is the periodicity at which queue length values are averaged — smaller values of δ (i.e., more frequent averaging) results in higher sensitivity to bursts in queue length, but provides a more fine-grained feedback to BlowFish for triggering transitions to different storage-footprints. For instance, consider the two cases depicted in Figure 5; we typically do not want to react to short spikes in queue length as in (a), but want to react to longer lasting spikes and persistent increase in queue lengths as in (b) and (c).

To trigger navigation along the storage-performance tradeoff curve based on the exponentially averaged queue lengths, BlowFish uses the curve shown in Figure 1. In particular, BlowFish triggers an increase (decrease) in sampling rate of LSA if the exponentially averaged queue length crosses a certain threshold beyond that can be served using the current storage overhead. This threshold values play an important role in how well the system adapts to changes in request rates. Triggering navigation at a fine granularity allows the system to quickly adapt to the changes in request rates, avoiding high latency due to queue build up during the period when request rate does not match the response rate. However, if the granularity is too fine, the system would be prone to being too sensitive to small changes in the request rate, *e.g.*, oscillations in request rates around the threshold.

In fact, a single threshold value could potentially lead issues when there are oscillations in the request rates experienced by a shard. Consider a case where the request queue lengths keep oscillating about the threshold queue length value for that storage footprint with a very small amplitude of oscillation. Under such circumstances, the system would react each time the queue length increases beyond or falls below the threshold value, and the partition would not be able to stabilize at a single storage footprint. BlowFish overcomes this by maintaining two queue length threshold values for each storage footprint that the partition can occupy — a high-mark, which the queue length must increase beyond to trigger increase in storage footprint, and a low-mark that the queue length must fall below to trigger decrease in storage footprint. These two marks are kept sufficiently apart to prevent oscillations in storage footprint.

3.3 Server Handlers

BlowFish uses a server handler module to monitor request queues, trigger navigation along the storage-performance tradeoff curve and schedule queries across the system. We now describe these functionalities in detail.

3.3.1 Dynamically Navigating the Tradeoff

BlowFish enables dynamic navigation along the storage-performance tradeoff by addition and deletion of layers on Layered Sampled Arrays (LSA) from §3.1. To trigger navigation along the tradeoff, BlowFish uses the request queues from §3.2. We now describe this process.

We start by making two observations. First, lookups on the LSA are agnostic to the layers present in LSA. This allows BlowFish to add and delete layers transparently without any change in lookup semantics. Second, LSA has an interesting structure (see Figure 4) — the storage footprint of any layer is equal to the sum of the storage footprints of all the layers above it (i.e., all the layers with a sampling-rate *larger* than that layer). Although layers can be added in arbitrary order, we add layers sequentially from the top-down, i.e., new layers are added below all existing layers, and each newly created layer adds a storage-footprint equal to the sum of the storage-footprints of layers that already exist. The amount of time spent in computing an unsampled value is dependent on the number of values that are already sampled — fewer sampled val-

ues implies more time to compute an unsampled value. As layers progressively are added to the bottom, more values must be computed for the subsequent layers, but fewer computations are required since more values are sampled than during the previous layer creation. This makes the cost of each layer addition *constant* for the top-down layer addition approach.

Dedicated Layer Addition. Once BlowFish determines which layers to add to meet the request rate experienced by a partition, it creates the corresponding layers and updates the `ExistsLayer` bitmap to indicate availability of new layers. Layers are created in the background by dedicated threads that populate values in the layer. Due to the use of dedicated threads for layer construction, additional compute resources are required at the node on which the layer creation is requested.

Opportunistic Layer Addition. This additional resource consumption is justified if the time spent in creating the layer is smaller than the period of increased throughput experienced by the data partition. However, this scheme may not be very well suited for cases where the nodes are persistently overloaded, or where the increase in query request rates is not sustained. In the actual implementation of BlowFish, we use an opportunistic layer creation strategy, which exploits the computation performed by queries to populate values in the layer, minimizing the overheads for layer creation.

The query algorithms in Succinct when adapted to BlowFish enable an interesting means of layer creation for the Layered Compressed Storage. In this layer creation scheme, the values of the layers are computed *on-the-fly* during query-execution, i.e.queries that incur computations of certain unsampled values are used to fill-up the newly added layers. This is outlined in Algorithm 1.

This scheme is particularly useful when the compute resources are heavily contended – at the added cost of slightly more computation during query execution, opportunistic layer creation avoids using dedicated resources for creating layers to increase throughput, allowing a smoother transition from low throughput capacity to a higher throughput capacity that matches the request rate.

Additionally, this transition is much faster for real-world workloads, which typically follow a zipf-like distribution. Under such access patterns, certain hot objects are repeatedly queried, while the frequency of accesses for less popular objects is significantly smaller. With opportunistic layer addition, the first execution of the popular queries fills in the layer values, speeding up all future executions. This results in much faster adaptations to even spiked changes in query request-rates (See 4).

Algorithm 1 `CreateLayerOpportunistic`

```
1: procedure CreateLayerOpportunistic (l_id)        ▷ Only marks a layer (l_id) for creation, delegating actual computations to Lookup queries.
2:     MarkLayerForCreation(l_id)
3:     for l_idx in (0, InputSize) do
4:         IsLayerValueSampled[l_id][l_idx] = 0
5:     end for
6: end procedure
7: procedure OpportunisticLookup (Idx)        ▷ Exploit Lookup queries to fill-up layers opportunistically if the layer is marked for creation.
8:     result = Lookup(Idx)
9:     l_id = LayerID(Idx)                    ▷ Get layer ID.
10:    if IsMarkedForCreation(l_id) then
11:        l_idx = LayerIdx(Idx)   ▷ Get index into layer.
12:        SampledArray[l_id][l_idx] = result
13:        IsLayerValueSampled[l_id][l_idx] = 1
14:    end if
15:    return result
16: end procedure
```

Layer Deletion. When a reduction in memory footprint by a certain size (in bytes) is requested, we first determine the layers that would need to be deleted. Layer deletions follow a bottom-up pattern to complement the layer addition scheme, i.e., layers are always deleted form the bottom to maintain consistency with layer additions. As opposed to layer creation, layer deletions are computationally inexpensive, and do not require any specialized strategies. Upon the deletion of a layer, the `ExistsLayer` bitmap is updated to indicate that the corresponding layer is no longer available for lookups. Queries would eventually notice that the layer no longer exists, and stop accessing it. In order to maintain safety, we ensure that all queries that were accessing the deleted layer have completed before deleting the layer.

3.3.2 Scheduling Queries

BlowFish uses the server handler module to monitor request queues and trigger navigation along the storage-performance tradeoff curve. The server handler module also maintains and distributes information about request queues across the system, and uses this information for scheduling queries across shard replicas. In particular, spatial and temporal skew in query distribution across data shards (and across replicas within a shard) introduce a significant amount of dynamism and heterogeneity in BlowFish's system state (see Figure 3 (right)). Shards on a server can have varying storage footprint and as a re-

sult, varying response rates. This can result in varying storage and response rates across multiple replicas over time. To schedule queries across such a dynamic and heterogeneous system, each server handler uses the request queues. Specifically, each server handler encapsulates the "local" request queue lengths in its query responses; a server handler, upon receiving a query response, decapsulates the queue lengths and updates its local metadata to record the new queue lengths.

The server handler schedules queries based on the request queue lengths of each of the replicas. In particular, the fraction of queries scheduled on any of the replicas is inversely proportional to its queue-length — the goal being to prevent "hot-spots" on any one of the replicas. The request queue lengths are indicatorss of the amount of load on a particular replica, and scheduling queries based on the queue-lengths enables low-overhead and fair load-balancing across the replicas, particularly in cases when the throughput capacities of the replicas may vary with time.

Consider a case with two replicas, where both replicas are initially at the same storage-footprint (and hence throughput capacity), and both have nearly empty request queues. Under these circumstances, the server handler would distribute incoming queries equally across the two replicas. Suppose that the node on which the second replica resides is memory-constrained, and the replica cannot increase its storage-footprint. If the request rate to the shard is increased gradually, the replicas would eventually reach a point where they cannot answer incoming queries at the same rate that they arrive. At this point, the request-queues at both the replicas would start building up at the same rate. BlowFish would trigger a layer creation at the first replica, but not the second replica due to limited memory on the corresponding node. Once the layer creation completes at the first replica, it would be capable of answering queries faster and its request queue would start draining. The request queue would still keep growing at the same rate as before at the second replica. In response, the Server Handler would schedule *fewer* queries at the second replica and *more* queries at the first one, in inverse proportion of their respective request-queue lengths. This would cause the request queue at the second replica to be drained as well, until the lengths of both request-queues stabilize to roughly equal values. Thus, the scheduling mechanism ensures that all replicas see queries at rates that they can handle.

3.3.3 Server Handler Implementation

The server handler module is implemented as a composition of two sub-modules, namely the Layer Manager and the Query Scheduler. The Layer Manager monitors the local Request Queues, and maintains exponential averages for all local Request Queues lengths. Whenever the length exceeds a high-mark or falls below a low-mark value, the Layer Manager triggers layer additions or deletions respectively. The Query Scheduler maintains the Request Queue lengths for all of the shards in the system, based on the queue-length values embedded in query responses. It distributes queries queries across different replicas of a data-partition based on their queue-length values. The Server Handler is multi-threaded, and all of its data-structures are kept lock-free wherever possible, to minimize overheads of query scheduling and layer management.

4 Evaluation

In this section, we use macro- and micro-benchmarks to evaluate BlowFish's performance to to evaluate how its system design, data structure, and algorithms meet its goals. Specifically, we evaluate the performance of BlowFish over dynamic workloads in terms of (1) time taken to adapt to workload variations; (2) system stability in terms of queue lengths during workload variations; and (3) comparison against selective replication.

Implementation. BlowFish is implemented in roughly 2K lines of C++ on top of Succinct [7]. The implementation requires several major changes in Succinct architecture. First, the data structures in Succinct that store sampled arrays are replaced by BlowFish's LSA (§3.1), and query algorithms are modified accordingly. Second, request queues (§3.2) per shard are maintained to estimate load across each shard. Finally, a server handler module (§**??**) is implemented to periodically monitor the shard queues, to trigger navigation along the storage-performance tradeoff curve, to append the length of local shard queues along with each query response and to schedule the queries based on queue lengths.

Dataset and Cluster. We use a dataset from Conviva customers comprising of 16 attribute records which include integers and strings. Each key is 8 bytes long and each record is 140 bytes long; since our queries are search queries that return keys whose corresponding value attribute matches the search term, the length of the record does not affect the evaluation results.

Since our goal is to evaluate the *per-thread* throughput of BlowFish over dynamic workloads, we execute our micro-benchmarks using a single-core, single-thread implementation on an Amazon EC2 m2.4xlarge machine. For our macro-benchmarks, we use a distributed Amazon

EC2 cluster consisting of m2.4xlarge machines; a number of clients are used to generate time-varying request rate (system load) over the network.

Workload Generation. We use two synthetic query workloads, each comprising of search queries with output cardinality varying from 1 to 1000. Each of the two workloads generates query search terms with popularity following a Zipf distribution with skewness 0.99, similar to YCSB [13].

The first workload models changes in request rate that are *gradual*. Specifically, we increase the request rate from 2.5Kops per thread to 6.5Kops per thread, with a gradual increase of 1Kops per thread at 30 minute intervals. This granularity of request rate increase is similar to those reported in real-world production clusters [9]. Our second workload models sudden changes in request rates, *e.g.*, due to transient machine failures. In this case, we increase the request rate from 2.5Kops per-thread to 7.5Kops per-thread in a single step. Note that the latter workload constitutes a worst-case scenario for BlowFish.

Performance Metrics and State-of-the-art. We evaluate the performance of BlowFish in terms of time taken to adapt to workloads described above and system stability in terms of queue sizes over time. We evaluate two version of BlowFish: one with dedicated layer construction and the other with more practical opportunistic layer construction, as described in §**??**. Note that the dedicated layer construction is the worst-case scenario for BlowFish since queries can not exploit the new layer until the construction is complete. Opportunistic layer construction, on the other hand, allows queries to exploit even partially constructed layers. Finally, we compare BlowFish against Selective replication that creates, for each shard, exactly as many replicas as required to meet the request rate for that shard.

4.1 Micro-benchmarks

We now present evaluation results for micro-benchmarks using the set up described above. We start by discussing the results for dedicated layer construction, followed by opportunistic layer construction results.

4.1.1 Gradual Workload Variation

Recall that the request rate for this workload is gradually increased from 2.5Kops per thread to 6.5Kops per thread, with each increase being 1Kops at 30 minute intervals.

Dedicated Layer Creation (Figure 6). As the request rate increases from 2.5Kops to 3.5Kops, BlowFish first increases the response rate to 2.7Kops, the throughput supported at that storage ratio (Figure 6(b)). As the request rate increases beyond 2.7Kops, the storage ratio of BlowFish changes in two steps (Figure 6(a)), one for the array that supports search queries and the other that supports random access. This is triggered due to queue sizes increasing beyond the threshold since the request rate is much higher than the response rate that the system can sustain; the queue build-up is depicted in Figure 6(c). The queue length grows to the point till which more requests cannot be buffered anymore; at this point, the system stops receiving queries at a rate greater than the system can handle. This corresponds to the drop in request rate in Figure 6(b) to match the response rate. When the layer addition completes, the response rate increases beyond the request rate to deplete the saturated Request Queue, until the queue length reduces to zero and the system resumes normal operation. A similar trend can be seen when the request rate is increased beyond 5.5Kops.

When the request-rate drops (*e.g.*, at 6500 ops/sec), BlowFish can adapt to the decrease much faster since layer deletions are computationally inexpensive and finish within seconds. This is characterized by the reduction in storage footprint in Figure 6(a).

Opportunistic Layer Construction (Figure 7). In this case, when the request rate is increased to 2.5Kops, BlowFish triggers layer creation *opportunistically*. At this point, the system immediately allocates storage for the two arrays, and marks them for creation without performing any computation. This corresponds to the increase in storage ratio in Figure 7(a). The values of the layers are filled in during query execution. As more queries come in, more values are filled in. These values are reused by other queries, as well as by the same queries in the future. Since the query patterns follow Zipf distribution, popular queries observe performance gains since values filled in by the first execution of the query speed up all subsequent executions. This also increases the overall throughput since popular queries complete faster and contribute more to the throughput, whereas less popular queries are relatively slower but have a smaller contribution to the system throughput. This is depicted in Figure 7(b). Note that, in contrast to the dedicated layer construction, the response rate in opportunistic layer construction increases within minutes of the layer creation trigger, and drains the Request Queues much faster than in the dedicated case (Figure 7(c)).

Similar trend is seen when the request rate increases to 5.5Kops. However, note that since the response rate is higher, opportunistic layer construction is faster — more queries are answered at higher response rates and hence

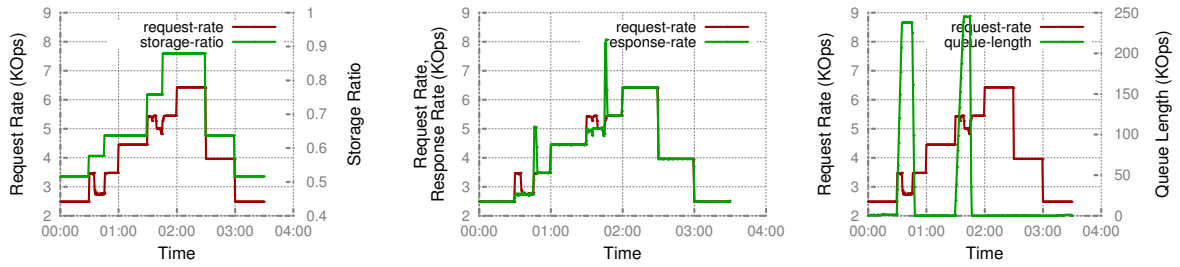

Figure 6: Dedicated layer construction with gradual changes in request-rate; Variation in storage-footprint (left), response-rate (center) and request queue-length (right).

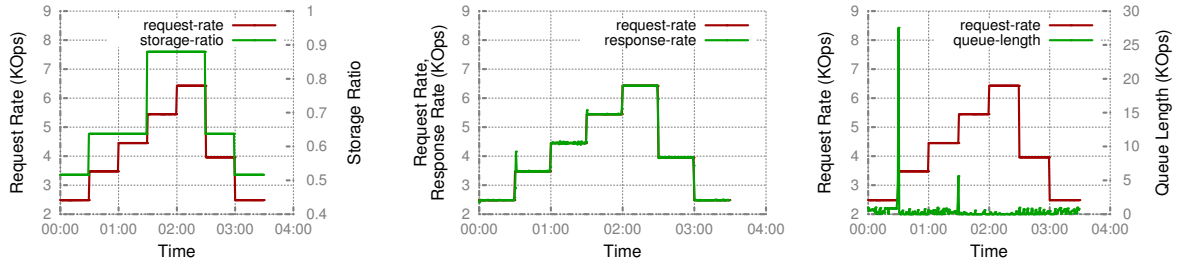

Figure 7: Opportunistic layer construction with gradual changes in request-rate; Variation in storage-footprint (left), response-rate (center) and request queue-length (right).

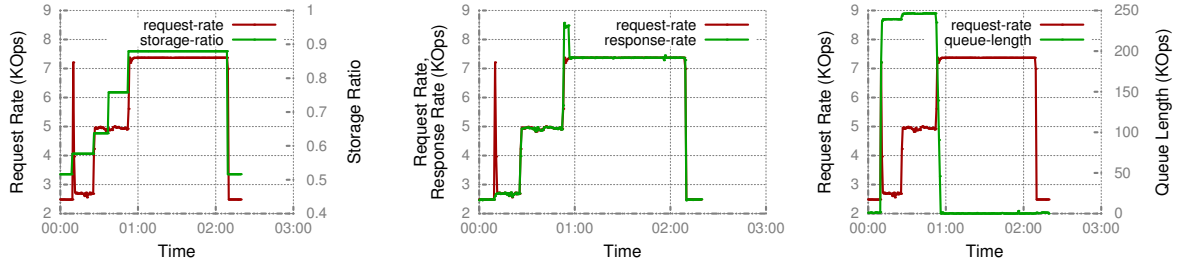

Figure 8: Dedicated layer construction with spiked changes in request-rate; Variation in storage-footprint (left), response-rate (center) and request queue-length (right).

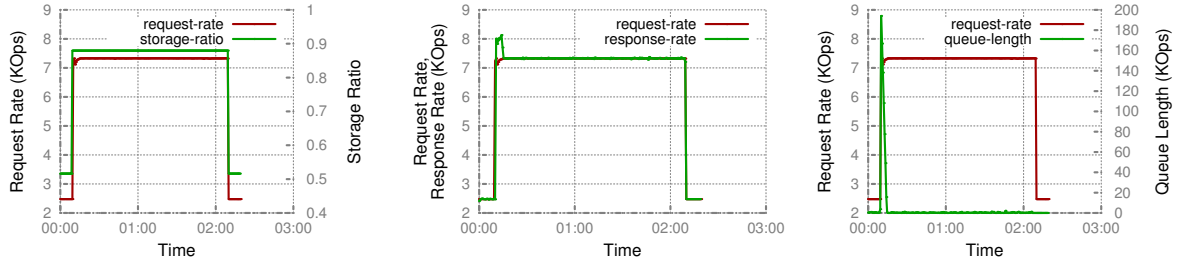

Figure 9: Opportunistic layer construction with spiked changes in request-rate; Variation in storage-footprint (left), response-rate (center) and request queue-length (right).

more values are computed opportunistically. As a result, the queue lengths are significantly smaller than the case of 2.5Kops transition.

4.1.2 Spiked Workload Variation

Recall that the request rate for this workload is increased from 2.5Kops per thread to 7.5Kops per thread instantaneously.

Dedicated Layer Construction (Figure 8). The case of dedicated layer construction is very similar for the two workloads. There are two main differences. First, the system is unable to sustain the request rate for dedicated construction due to instantaneous increase in request rate (drop in request rate soon after the increase). Second, the queues (rightmost figure) stay saturated for a much longer while since BlowFish needs to create two layers and during this period, the request rate is higher than BlowFish's response rate.

Opportunistic Layer Construction (Figure 9). Rather interestingly, oppportunistic layer construction performs well even in case of spiked increases in request rates, a worst-case scenario for BlowFish. Intuitively, due to queries have Zipf distribution, the increase in request rate renders most queries repeating. In this case, opportunistic construction is able to answer the queries since the required sampled values were already computed opportunistically. Note, however, that in the case of spiked workloads even opportunistic layer construction observes much higher queue sizes compared to gradual workload. Nonetheless, BlowFish is able to adapt in ≈ 5 minutes and is able to drain the queues in no time.

4.2 Macro-Benchmarks

For macro-benchmarks, we create three replicas for each shard and distribute them across three different machines on the cluster. To evaluate BlowFish scheduling, we make one of these replicas (machines) storage constrained; that is, increasing the storage footprint for this replica is impossible. We focus only on one shard since all shards have similar behavior. We present the results for gradual workloads and for opportunistic layer construction (Figure 10). In particular, we increase the request rate from 6Kops to 16Kops in steps of 2Kops per 30 minutes. Note that we use higher request rates compared to micro-benchmarks since the shards are replicated and the requests are load balanced across the three replicas.

Initially, each of the three replicas observe a request rate of 2Kops since queue sizes are equal, and BlowFish scheduler equally balances the load. As the request rate is increased to 8Kops, the replicas are no longer able to match the request rate since each replica can only perform $\approx$ 2.5Kops with this storage ratio. This causes the request queues at all of the three replicas to build up. Once the queue lengths cross the threshold, two of the three replicas trigger an opportunistic layer construction to sustain higher request rates (the third one is storage constrained) (Figure 10(a)).

As the first two replicas opportunistically add layers, their response rate increases; however, the response rate for the third replicas remains at $\approx$ 2.5Kops (Figure 10(b)). This causes the request queue to build up for the third replica at a much faster rate than the request queues for the first two replicas (Figure 10(c)). Since the number of queries scheduled at a replica depends inversely on the request queue length, BlowFish schedules more queries to the first two replicas and fewer queries to the third replica. Since the first two replicas can sustain the increased request rate, this causes the request queues at all three replicas to decrease proportionately until it stabilizes at a smaller value. Note that BlowFish still observes queue length oscillations, albeit of much smaller magnitude.

On increasing the request-rate further, the system is able to cope up with the increased rates until 10Kops. At this point, the same trend repeats, where two of the replicas increase their storage footprint to sustain higher request rates, while the third remains at its original storage footprint. BlowFish's query scheduler is still able to ensure that the request queue occupancy at all the three replicas is minimized and the system is able to sustain the increased request rate (Figures 10(b) and 10(c)).

4.3 Comparison with Selective Replication

We compare BlowFish's performance with Selective Replication of compressed objects. In order to evaluate the two schemes, we measure the amount of storage that is required to serve queries at a certain rate. We fix the amount of data to 200GB distributed across 100 shards, and apply a zipf distribution of queries to the system.

Figure 11 shows the amount of storage required for the two schemes as the request rates are increased from 100 Kops to 800 Kops. BlowFish requires $1.6\times$ to $4.2\times$ lesser storage to serve queries at the same rate. This is mainly due to two reasons; first, Selective Replication provides a very coarse tradeoff between storage and performance as compared to BlowFish. As a result, BlowFish is able to "pack" the shards into the storage-footprint required by them to be able to answer queries much more efficiently, while Selective Replication ends up increasing the storage footprint in multiples of the original shard size. Second,

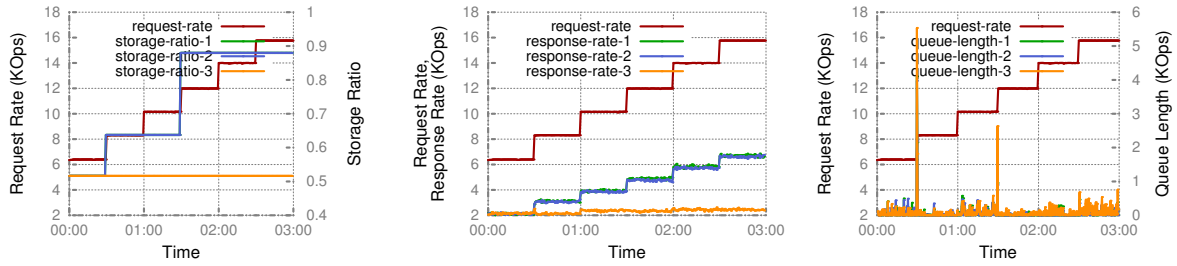

Figure 10: The effect of query scheduling. Performance when queries are scheduled across different shards based on the load seen by them. Variation in storage-footprints (left), response-rates (center) and request queue-lengths (right).

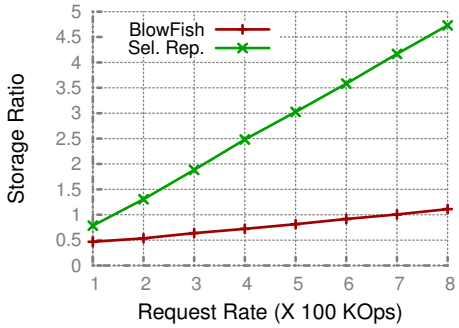

Figure 11: BlowFish requires 1.6× to 4.2× lesser storage to serve queries at the same rate as Selective Replication.

BlowFish provides a super-linear gains in throughput on increasing the storage footprint until a point, while Selective Replication provides linear gains for the same.

5 Related Work

Queries on Compressed Data. A number of recent systems [4, 6, 7, 24] have emerged that attempt to maximize performance by compressing the data, thereby pushing larger volumes of data into memory, and enabling queries *directly* on the compressed data (i.e.without decompression). Since these systems do not store secondary indexes, they can store much more data in memory and see significant gains in performance for random access and search workloads. The performance for these systems is typically characterized by the compression factor of the compressed data — heavily compressed data offers lesser performance as opposed to smaller compression factors. However, in the presence of dynamic workloads where the popularity of data-items changes with time, these systems are unable to adapt since changing the compression factor requires reconstructing the compressed data-structures from scratch.

Selective replication of Compressed Objects. Since a single copy of the compressed object may become insufficient to efficiently support queries for "hot" data, these objects can be *selectively* replicated [8]. This enables these systems to cope with increase in load for some compressed objects with time. However, selective replication provides a very coarse grained control over the amount of storage and the corresponding performance gains. For instance, by selectively replicating a single compressed object, it's storage footprint is doubled along with the throughput capacity, even though a 2× increase may be unnecessary. Additionally, a 1× increase in throughput for a 1× increase in storage-footprint may not be optimal for these compressed objects, since these compression schemes typically have non-linear performance-storage tradeoffs [4, 6, 7, 20, 24].

6 Conclusion

We have presented BlowFish, a distributed data store that provides a seamless and dynamic tradeoff between compression and ratio and system performance. BlowFish efficiently navigates this tradeoff space to adapt to temporal variations of query distributions across data items and spiked workload variations due to transient machine failures. BlowFish uses a new data structure that stores a compressed representation of the input data, enables queries directly on the compressed representation, and permits dynamic changes to the compression factor for the representation. BlowFish also exploits a back-pressure style query scheduling mechanism to balance queries across different replicas with dynamically varying performance characteristics. Our evaluations have shown that BlowFish is able to adapt to query-rate increments of 200% within a span of 5 minutes, while increasing the storage footprint from 51% to 89% of the original input.

References

[1] Elasticsearch. http://www.elasticsearch.org.

[2] MemCached. http://www.memcached.org.

[3] MongoDB. http://www.mongodb.org.

[4] Pizza&Chili Corpus: Compressed Indexes and their Testbeds. http://pizzachili.dcc.uchile.cl/indexes/Compressed_Suffix_Array/.

[5] Redis. http://www.redis.io.

[6] SDSL. https://github.com/simongog/sdsl-lite.

[7] R. Agarwal, A. Khandelwal, and I. Stoica. Succinct: Enabling Queries on Compressed Data. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[8] 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 *ACM European Conference on Computer Systems (EuroSys)*, 2011.

[9] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64, 2012.

[10] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[11] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, et al. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX Technical Conference (ATC)*, 2013.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *ACM Symposium on Cloud Computing (SoCC)*, 2010.

[14] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's Globally-distributed Database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[15] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[17] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[18] R. Escriva, B. Wong, and E. G. Sirer. HyperDex: A Distributed, Searchable Key-value Store. In *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2012.

[19] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[20] R. Grossi, A. Gupta, and J. S. Vitter. High-order Entropy-compressed Text Indexes. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2003.

[21] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[22] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[23] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica Analytic Database: C-store 7 Years Later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, 2012.

[24] B. Langmead, C. Trapnell, M. Pop, S. L. Salzberg, et al. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biol*, 10(3):R25, 2009.

[25] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[26] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, et al. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[27] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 61–72. ACM, 2012.

[28] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster. In *Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'13, pages 8–8, Berkeley, CA, USA, 2013. USENIX Association.

[29] K. Sadakane. Compressed Text Databases with Efficient Query Algorithms Based on the Compressed Suffix Array. In *International Conference on Algorithms and Computation (ISAAC)*. 2000.

[30] K. Sadakane. Succinct Representations of Lcp Information and Improvements in the Compressed Suffix Arrays. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2002.

[31] K. Sadakane. New Text Indexing Functionalities of the Compressed Suffix Arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[32] S. Sivasubramanian. Amazon dynamoDB: A Seamlessly Scalable Non-relational Database Service. In *ACM International Conference on Management of Data (SIGMOD)*, 2012.

[33] C. B. Walton, A. G. Dale, and R. M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *VLDB*, volume 91, pages 537–548, 1991.