BlowFish: Dynamic Storage-Performance Tradeoff in Data Stores

Anurag Khandelwal
UC Berkeley

Rachit Agarwal
UC Berkeley

Ion Stoica
UC Berkeley

Abstract

We present BlowFish, a distributed data store that admits a smooth tradeoff between storage and performance for point queries. What makes BlowFish unique is its ability to navigate along this tradeoff curve efficiently at fine-grained time scales with low computational overhead.

Achieving a smooth and dynamic storage-performance tradeoff enables a wide range of applications. We apply BlowFish to several such applications from real-world production clusters: (i) as a data recovery mechanism during failures: in practice, BlowFish requires 6.7× lower bandwidth and 3.5× lower repair time compared to state-of-the-art erasure codes, while reducing the storage cost of replication from 3× to 1.8×; and (ii) data stores with spatially-skewed and time-varying workloads (*e.g.*, due to object popularity and/or transient failures): we show that navigating the storage-performance tradeoff achieves higher system-wide utility (*e.g.*, throughput) than selectively caching hot objects.

1 Introduction

Random access and *search* are the two fundamental operations performed on modern data stores. For instance, key-value stores [2, 4, 11, 15, 16, 18, 23, 25] and NoSQL stores [1, 3, 7, 12, 13, 17, 21, 30] support random access at the granularity of records. Many of these [1, 3, 7, 17, 21, 22] also support search on records. These data stores typically store an amount of data that is larger than available fast storage[1], *e.g.*, SSD or main memory. The goal then is to maximize the performance using caching, that is, executing as many queries in faster storage as possible.

The precise techniques for efficiently utilizing cache vary from system to system. At a high-level, most data stores partition the data across multiple *shards*, with each server potentially storing multiple shards (partitions) [1, 7, 21, 23]. Shards may be replicated and cached across multiple servers and the queries are load balanced across shard replicas [1, 3, 7, 12, 21].

[1]To support search, many of these systems store indexes in addition to the input, which further adds to the storage overhead. We collectively refer to the indexes combined with the input as "data".

To cache more shards, many systems use compression [1, 3, 7, 21]. Unfortunately, compression leads to a hard tradeoff between throughput and storage for the cached shards — when stored uncompressed, a shard can support high throughput but takes a larger fraction of available cache size; and, when compressed, takes smaller cache space but also supports lower throughput. Furthermore, switching between these two extreme points on the storage-performance tradeoff space cannot be done at fine-grained time scales since it requires compression or decompression of the entire shard. Such a hard storage-performance tradeoff severely limits the ability of existing data stores in many real-world scenarios when the underlying infrastructure [28, 29], workload [9, 10, 14, 26, 31], or both changes over time. We discuss several such scenarios from real-world production clusters below (§1.1).

We present BlowFish, a distributed data store that enables a *smooth* storage-performance tradeoff between the two extremes (uncompressed, high throughput and compressed, low throughput), allowing fine-grained changes in storage and performance. What makes BlowFish unique is that applications can navigate from one operating point to another along this tradeoff curve *dynamically* over fine-grained time scales. We show that, in many cases, navigating this smooth tradeoff has higher system-wide utility (*e.g.*, throughput per unit of storage) than existing techniques. Intuitively, this is because BlowFish allows shards to increase/decrease the storage "fractionally", just enough to meet the performance goals.

1.1 Applications and summary of results

BlowFish, by enabling a dynamic and smooth storage-performance tradeoff, allows us to explore several problems from real-world production clusters from a different "lens". We apply BlowFish to three such problems:

Storage and bandwidth efficient data repair during failures. Existing techniques either require high storage (replication) or high bandwidth (erasure codes) for data repair, as shown in Table 1. By storing multiple replicas at different points on tradeoff curve, BlowFish can achieve the best of the two worlds — in practice, Blow-

Table 1: Storage and bandwidth requirements for existing techniques for recovering failed data.

	Erasure Codes	Replication	BlowFish
Storage	1.2×	3×	1.8×
Repair Bandwidth	10×	1×	1×

Fish requires storage close to state-of-the-art erasure codes while requiring repair bandwidth close to replication. System state is restored by copying one of the replicas and navigating along the tradeoff curve. We explore the corresponding storage-bandwidth-throughput tradeoffs in §4.2.

Skewed workloads. Existing data stores can benefit significantly using compression [1, 3, 7, 12, 21]. However, these systems lose their performance advantages in case of dynamic workloads where (i) the set of hot objects changes rapidly over time [9, 14, 26, 31], and (ii) a single copy is not enough to efficiently serve a hot object. Studies from production clusters have shown that such workloads are a norm [9, 10, 14, 26, 31]. Selective caching [8], that caches additional replicas for hot objects, only provides coarse-grained support to handle dynamic workloads — each replica increases the throughput by 2× while incurring an additional storage overhead of 1×.

BlowFish not only provides a finer-grained tradeoff (increasing the storage overhead fractionally, just enough to meet the performance goals), but also achieves a better tradeoff between storage and throughput than selective caching of compressed objects. We show in §4.3 that BlowFish achieves 2.7–4.9× lower storage (for comparable throughput) and 1.5× higher throughput (for fixed storage) compared to selective caching.

Time-varying workloads. In some scenarios, real-world production clusters delay additional replica creation to avoid unnecessary traffic (*e.g.*, for 15 minutes during transient failures [28, 29]). Such failures contribute to 90% of the failures [28, 29] and create high temporal load across remaining replicas. We show that BlowFish can adapt to such time-varying workloads even for spiked variations (as much as by 3×) by navigating along the storage-performance tradeoff in less than 5 minutes (§4.4).

1.2 BlowFish Techniques

BlowFish builds upon Succinct [7], a system that supports queries on compressed data[2]. At a high-level, Succinct stores two *sampled* arrays, whose sampling rate acts as a proxy for the compression factor in Succinct. BlowFish introduces *Layered Sampled Array* (LSA), a new data structure that stores sampled arrays using multiple layers of sampled values. Each combination of layers in LSA correspond to a static configuration of Succinct. Layers in LSA can be added or deleted transparently, independent of existing layers and query execution, thus enabling dynamic navigation along the tradeoff curve.

Each shard in BlowFish can operate on a different point on the storage-performance tradeoff curve. This leads to several interesting problems: how should shards on a single server share the available cache? How should shard replicas share requests? BlowFish adopts techniques from scheduling theory, namely back-pressure style Join-the-shortest-queue [19] mechanism, to resolve these challenges in a unified manner. Shards maintain request queues that are used both to load balance queries as well as to manage shard sizes within and across servers.

In summary, this paper makes three contributions:

- Design and implementation of BlowFish, a distributed data store that enables a smooth storage-performance tradeoff, allowing fine-grained changes in storage and performance for each individual shard.
- Enables dynamic adaptation to changing workloads by navigating along the smooth tradeoff curve at fine-grained time scales.
- Uses techniques from scheduling theory to perform load balancing and shard management within and across servers.

2 BlowFish Overview

We briefly describe Succinct data structures in §2.1, with a focus on how BlowFish transforms these data structures to enable the desired storage-performance tradeoff. We discuss the target workloads for BlowFish in Appendix C.1. Finally, we provide a high-level overview of BlowFish design (§2.2).

2.1 Succinct Background

Succinct internally supports random access and search on flat unstructured files. Using a simple transformation from semi-structured data to unstructured data [7], Succinct supports queries on semi-structured data, that is, a collection of records. Similar to other key-value and NoSQL stores [1–3, 12, 15, 21, 23], each record has a unique identifier `key`, and a potentially multi-attribute

[2]Unlike Succinct, BlowFish does not enforce compression; some points on the tradeoff curve may have storage comparable to systems that store indexes along with input data.

value. Succinct supports random access via get, put and delete operations on keys; in addition, applications can search along individual attributes in values.

Succinct supports random access and search using four data structures — Array-of-Suffixes (AoS), Input2AoS, AoS2Input and NextCharIdx. AoS stores all suffixes in the input file in lexicographically sorted order. Input2AoS enables random access by mapping offsets in the input file to corresponding suffixes in the AoS. AoS2Input enables search by mapping suffixes in AoS to corresponding offsets in the input file (see Figure 1). Finally, NextCharIdx allows computing the value at any index in Input2AoS and AoS2Input given a few sampled values in the respective arrays. The description of AoS, NextCharIdx, and their compact representations is not required to keep the paper self-contained; we refer the reader to [7]. We provide necessary details on representation of Input2AoS and AoS2Input below.

Sampled Arrays: Storage versus Performance. The AoS and the NextCharIdx arrays have certain structural properties that enable a compact representation. The Input2AoS and AoS2Input arrays do not possess any such structure and require $n\lceil \log n \rceil$ space each for a file with n characters (each entry being an integer in range 0 to $n-1$).

Succinct reduces the space requirements of Input2AoS and AoS2Input using *sampling* — only a few sampled values (*e.g.*, for sampling rate α, value at indexes $0, \alpha, 2\alpha, ..$) from these two arrays are stored. NextCharIdx allows computing unsampled values during query execution.

> The tradeoff is that for a sampling rate of α, the storage requirement for Input2AoS and AoS2Input is $2n\lceil \log n \rceil / \alpha$ and the number of operations required for computing each unsampled value is α.

Succinct thus has a fixed storage cost (for AoS and NextCharIdx), and the sampling rate α acts as a proxy for storage and performance in Succinct.

2.2 BlowFish Design Overview

BlowFish enables the same set of functionality as Succinct — support for random access and search queries on flat unstructured files, with extensions for key-value stores and NoSQL stores.

BlowFish uses a system architecture similar to existing data stores, *e.g.*, Cassandra [21] and ElasticSearch [1]. 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, comprising of multiple data shards (§3.1), a *request queue* per shard that

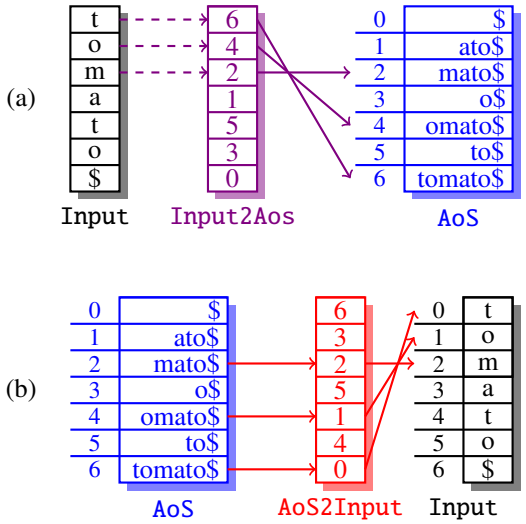


Figure 1: AoS stores suffixes in the input in lexicographically sorted order. (a) The Input2AoS stores the mapping from each index in the input to the index of the corresponding suffix in AoS. (b) The Aos2Input stores the inverse mapping from each suffix index in AoS to the corresponding index in the input.

keeps track of outstanding queries, and a special module *server handler* that triggers navigation along the storage-performance curve and schedules queries (§3.2).

Each shard admits the desired storage-performance tradeoff using *Layered Sampled Array* (LSA), a new data structure that allows transparently changing the sampling factor α for Input2AoS and AoS2Input over fine-grained time scales. Smaller values of α indicate higher storage requirements, but also lower latency (and vice versa). Layers can be added and deleted without affecting existing layers or query execution thus enabling dynamic navigation along the tradeoff curve. We describe LSA and the layer addition-deletion process in LSA in §3.1.

BlowFish allows each shard to operate on a different operating point on the storage-performance tradeoff curve (see Figure 3). Such a flexibility comes at the cost of increased dynamism and heterogeneity in system state. Shards on a server can have varying storage footprint and as a result, varying throughput. Moreover, storage footprint and throughput may vary across multiple replicas. How should different replicas of the same shard share queries? When should a shard trigger navigation along the storage-performance tradeoff curve? How should different shards on the same server share the limited cache?

BlowFish adopts techniques from scheduling theory, namely back-pressure scheduling style Join-the-shortest-queue [19] mechanism, to resolve the above questions. BlowFish servers maintain a *request queue* per shard,

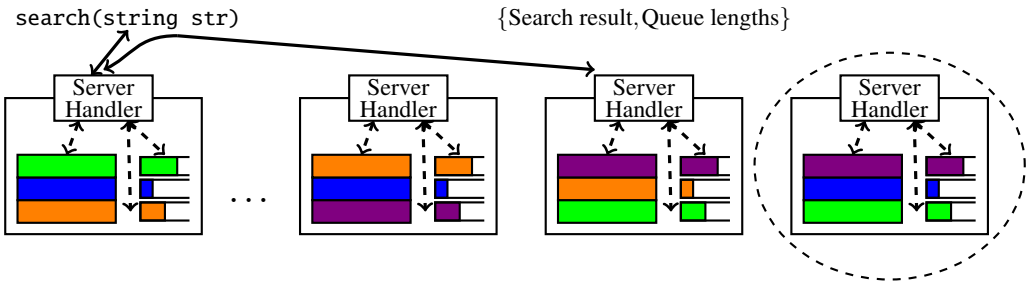


Figure 2: Overall BlowFish architecture. Each server has an architecture similar to the one shown in Figure 3. Queries are forwarded by Server Handlers to appropriate servers, and query respones encapsulate both results and queue lengths at that server.

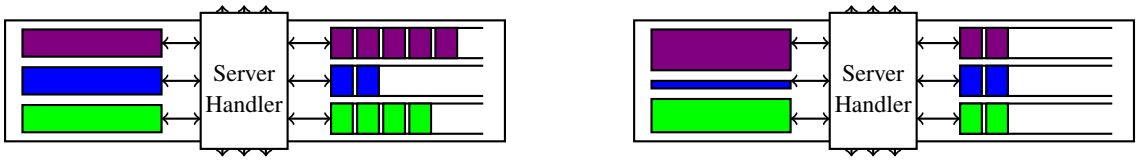


Figure 3: Main idea behind BlowFish: (left) the state of the system at time t; (right) the state of the shards after BlowFish adapts — the shards that have longer outstanding queue lengths at time t adapt their storage footprint to a larger one, thus serving larger number of queries per second than at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

that stores outstanding requests for the respective shard. A server handler module periodically monitors request queues for local shards, maintains information about request queues across the system, schedules incoming queries and triggers navigation along the storage-performance tradeoff curve.

Upon receiving a query from a client for a particular shard, the server handler forwards the query to the shard replica with shortest request queue length. All incoming queries are enqueued in the request queue for the respective shard. When the load on a particular shard is no more than shard's throughput at the current operating point on the storage-performance tradeoff curve, the queue size remains minimal. On the other hand, when the load for a particular shard increases beyond the supported throughput, the request queue length for this shard increases (see Figure 3 (left)). Once the request queue length crosses a certain threshold, the navigation along the 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 tradeoff curve, storage management during navigation and ensuring correctness in query execution during the navigation. We discuss these design details in §3.2.

3 BlowFish Design

We now provide design details for BlowFish. We start with the description of Layered Sampled Array (§3.1) and then discuss the system details (§3.2).

3.1 Layered Sampled Array

BlowFish enables a smooth storage-performance tradeoff using a new data structure, Layered Sampled Array (LSA), that allows dynamically changing the sampling factor in the two sampled arrays — Input2AoS and AoS2Input. 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[idx]` stores `A` value at index $\alpha \times$`idx`. For instance, if `A = {6, 4, 3, 8, 9, 2}`, the sampled-by-index array with sampling rate 4 and 2 are SA_4 `= {6, 9}` and SA_2 `= {6, 3, 9}`, respectively.

LSA emulates the functionality of the SA, but stores the sampled values in multiple *layers*, together with a few auxiliary structures that enable two properties (see Figure 4). First, layers in LSA can be added or deleted transparently, without affecting the existing layers or query execution, thus enabling a dynamic storage-performance tradeoff. Addition of layers results in higher storage overhead (lower sampling rate α) and lower query latency; deletion of layers, on the other hand, reduces the storage overhead but also increases the query latency. Second,

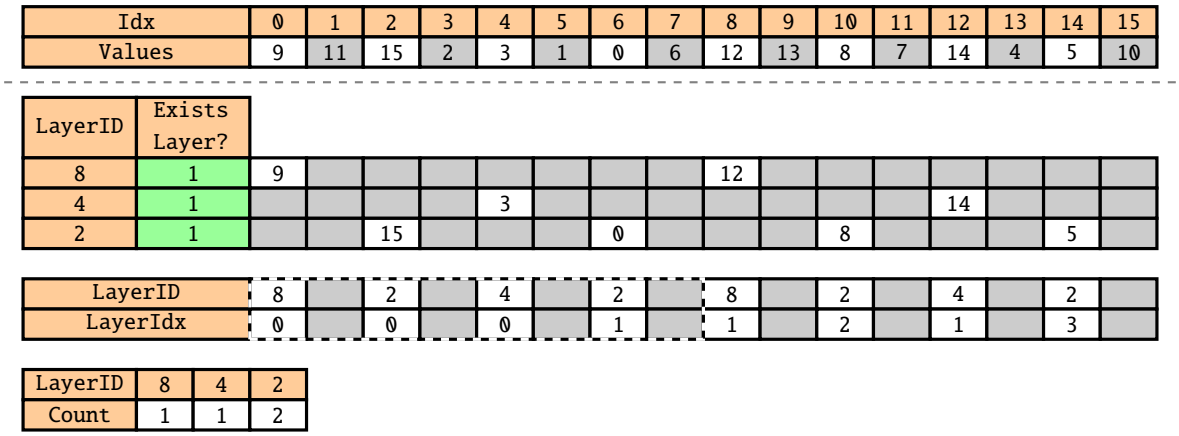


Figure 4: Illustration of *Layered Sampled Array* (LSA). The original unsampled array is shown above the dashed line (gray values indicate unsampled values). In LSA, each layer stores values for sampling rate given by `LayerID`, modulo values that are already stored in upper layers (in this example, sampling rates $8, 4, 2$). Layers are added and deleted at the bottom; that is, `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 not (0). `LayerID` and `ExistsLayer` allow checking whether or not value at any index `idx` is stored in LSA — we find the largest existing `LayerID` that is a proper divisor of `idx`. Note that among every consecutive 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.

looking up a value in LSA is *agnostic* to the existing layers, independent of how many and which layers exist (pseudo code in Appendix A). This allows BlowFish to navigate along the storage-performance tradeoff curve without any change in lookup semantics. The query execution also remains unchanged compared to Succinct.

Layer Addition. The design of LSA allows arbitrary layers (in terms of sampling rates) to coexist; furthermore, layers can be added or deleted in arbitrary order. However, our implementation of LSA makes two simplifications. First, layers store sampled values for indexes that are *power of two*. Second, new layers are always added at the bottom. The rationale is that these two simplifications induce a certain structure in LSA, that makes the increase in storage footprint as well as time taken to add the layer very predictable. In particular, under the assumption that the unsampled array is of length $n = 2^k$ for some integer k, the number of sampled values stored at any layer is equal to the cumulative number of sampled values stored in upper layers (see Figure 4). If the sampling rate for the new layer is α, then this layer stores precisely $n/2\alpha$ sampled values; thus, the increase in storage becomes predictable. Moreover, since the upper layers constitute sampling rate 2α, computing each value in the new layer requires 2α operations (§2.1). Hence, adding a layer takes a fixed amount of time independent of the sampling rate of layer being added.

BlowFish supports two modes for creating new layers. In *dedicated layer construction*, the space is allocated for a new layer[3] and dedicated threads populate values in the layer; once all the values are populated the `ExistsLayer` bit is set to 1. The additional compute resources required in dedicated layer construction may be justified if the time spent in populating the new layer is smaller than the period of increased throughput experienced by the shard(s). However, such may not be the case for many scenarios.

The second mode for layer creation in BlowFish is *opportunistic layer construction*. This mode exploits the fact that the unsampled values for the two arrays are computed on the fly during query execution. A subset of the these values are the ones to be computed for populating the new layer. Hence, the query execution phase can be used to populate the new layer without using dedicated threads. The challenge in this mode is when to update the `ExistsLayer` flag — if set during the layer creation, the queries may incorrectly access values that have not yet been populated; on the other hand, the layer may remain unused if the flag is set after all the values are populated. BlowFish handles this situation by using a bitmap that stores a bit per sampled value for that layer. A set bit indicates that the value has already been populated and vice versa. The algorithm for opportunistic layer construction is outlined in Algorithm 2 in Appendix A.

[3] using free unused cache or by deleting layers from relatively lower loaded shards, as described in §3.2.4.

It turns out that opportunistic layer construction performs really well for real-world workloads that typically follow a zipf-like distribution (repeated queries on certain objects). Indeed, the required unsampled values are computed during the first execution of a query and are thus available for all subsequent executions of the same query. Interestingly, this is akin to caching the query results without any explicit query result caching implementation.

Layer Deletion. Deleting layers is relatively easier in BlowFish. To main consistency with layer additions, layer deletion proceeds from the bottom most layer. Layer deletions are computationally inexpensive, and do not require any specialized strategies. Upon the request for layer deletion, the `ExistsLayer` bitmap is updated to indicate that the corresponding layer is no longer available. Subsequent queries, thus, stop accessing the deleted layer. In order to maintain safety, we delay the memory deallocation for a short period of time after updating the `ExistsLayer` flag.

3.2 BlowFish Servers

We now provide details on the design and implementation of BlowFish servers.

3.2.1 Server Components

Each BlowFish server has three main components (see Figure 2 and Figure 3):

Data shards. Each server stores multiple data shards, typically one per CPU core. Each shard stores the two sampled arrays — Input2AoS and AoS2Input — using LSA, along with other data structures in Succinct. This enables a smooth storage-performance tradeoff, as described in §3.1. The aggregate storage overhead of the shards may be larger than available main memory. Each shard is memory mapped; thus, only the most accessed shards may be paged into main memory.

Request Queues. BlowFish servers maintain a queue of outstanding queries per shard, referred to as *request queues*. The length of request queues provide a rough approximation to the load on the shard — larger request queue lengths indicate a larger number of outstanding requests for the shard, implying that the shard is observing more queries than it is able to serve (and vice versa).

Server Handler. Each server in BlowFish has a server handler module that acts as an interface to clients as well as other server handlers in the system. Each client connects to one of the server handlers that handles the client query (similar to Cassandra [21]). The server handler interacts with other server handlers to execute queries and to maintain the necessary system state. BlowFish server handlers are also responsible for query scheduling and load balancing, and for making decisions on how shards share the cache available at the *local* server. We discuss these functionalities below.

3.2.2 Query execution

Similar to existing data stores [1, 3, 21], an incoming query in BlowFish may touch one or more shards depending on the sharding scheme. The server handler handling the query is responsible for forwarding the query to the server handler(s) of the corresponding shard(s); we discuss query scheduling across shard replicas below. Whenever possible, the query results from multiple shards on the same server are aggregated by the server handler.

BlowFish does *not* require changes in Succinct algorithms for executing queries at each shard, with the exception of looking up values in sampled arrays[4]. In particular, since the two sampled arrays in Succinct — Input2AoS and AoS2Input — are replaced by LSA, the corresponding lookup algorithms are replaced by lookup algorithms for LSA (§2.2, Figure 4). We note that, by using `ExistsLayer` flag, BlowFish makes LSA lookup algorithms transparent to existing layers and query execution.

3.2.3 Scheduling and Load Balancing

BlowFish server handlers maintain the request queue lengths for each shard in the system. Each server handler periodically monitors and records the request queue lengths for local shards. The request queue lengths for non-local shards are collected during the query phase — each server handler encapsulates the request queue lengths for its local shards in its query responses. Upon receiving a query response, a server handler decapsulates the request queue lengths and updates its local metadata to record the new lengths for the corresponding shards.

Each shard (and shard replica) in BlowFish may operate on a different point on the storage-performance curve (Figure 3). Thus, different replicas of the same shard may have different query execution time for the same query. To efficiently schedule queries across such a heterogeneous system, BlowFish adopts techniques from scheduling theory literature — a back-pressure scheduling style Join-the-shortest-queue [19] mechanism. An incoming query for a shard is forwarded to the replica with the smallest request queue length. By conceptually modeling this problem as replicas having the same speed but varying job sizes (for the same query), the analysis for

[4]The description of these algorithms is not required to keep the paper self-contained; we refer the reader to [7] for details.

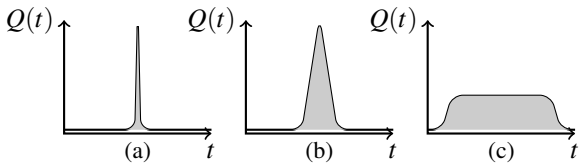


Figure 5: Three different scenarios of queue length ($Q(t)$) variation with time (t). (a) shows a very short-lasting "spike", (b) shows a longer lasting spike while (c) shows a persistent "plateau" in queue-length values. BlowFish should ideally ignore spikes as in (a) and attempt to adapt to the queue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

Join-the-shortest-queue [19] applies to BlowFish, implying close to optimal load balancing.

3.2.4 Dynamically Navigating the Tradeoff

BlowFish uses the request queues not only for scheduling and load balancing, but also to trigger navigation along the storage-performance tradeoff curve for each individual shard. We discuss below the details on tradeoff navigation, and how this enables efficient cache sharing among shards within and across servers.

One challenge in using request queue lengths as an approximation to load on the shard is to differentiate short-term spikes from persistent overloading of shards (Figure 5). To achieve this, BlowFish server handlers also maintain exponentially averaged queue lengths for each local shard — the queue lengths are monitored every δ time units, and the exponentially averaged queue length at time t is computed as:

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

The parameters β and δ provide two knobs for approximating the load on a shard based on its request queue length. β is a fraction ($\beta < 1$) that determines the contribution of more recent queue length values to the average — larger β assigns higher weight to more recent values in the average. δ is the periodicity at which queue lengths are averaged — smaller values of δ (i.e., more frequent averaging) results in higher sensitivity to bursts in queue length. Note that a small exponentially average queue length implies a persistently underloaded shard.

We now describe how shards share the available cache within and across servers by dynamically navigating along the storage-performance tradeoff curve. We start with the relatively simpler case of shards on the same server, and then describe the case of shards across servers.

Shards on the same server. Recall that BlowFish implementation adds and deletes layers in a bottom-up fashion, with each layer storing sampled values for powers of two. Thus, at any instant, the sampling rate of LSA is a power of two ($2, 4, 8, \ldots$). For each of these sampling rates, BlowFish stores two *threshold* values. The *upper threshold* value is used to trigger storage increase for any particular shard — when the exponentially averaged queue length of a shard S crosses the upper threshold value, S must be consistently overloaded and must increase its throughput.

However, the server may not have extra cache to sustain the increased storage for S. For such scenarios, BlowFish stores a *lower threshold* value which is used to trigger storage reduction. In particular, if the exponentially averaged queue length *and* the instantaneous request queue length for one of the other shards S' on the same server is below the lower threshold, BlowFish reduces the storage for S' before triggering the storage increase for S. If there is no such S', the server must already be throughput bottlenecked and the navigation for S is not triggered.

We make two observations. First, the goals of exponentially averaged queue lengths and two threshold values are rather different: the former makes BlowFish stable against temporary spikes in load, while the latter against "flap damping" of load on the shards. Second, under stable loads, the above technique for triggering navigation along the tradeoff curve allows each shard on the same server to share cache proportional to its throughput requirements.

Shard replicas across servers. At the outset, it may seem like shards (and shard replicas) across servers need to coordinate among themselves to efficient share the total system cache. It turns out that local cache sharing, as described above, combined with BlowFish's scheduling technique implicitly provides such a coordination.

Consider a shard S with two replicas R1 and R2, both operating at the same point on the tradeoff curve and having equal queue lengths. The incoming queries are thus equally distributed across R1 and R2. If the load on S increases gradually, both R1 and R2 will eventually experience load higher than the throughput they can support. At this point, the request queue lengths at R1 and R2 start building up at the same rate. Suppose R2 shares the server with other heavily loaded shards (that is, R2 can not navigate up the tradeoff curve). BlowFish will then trigger a layer creation for R1 only. R1 can thus support higher throughput and its request queue length will decrease. BlowFish's scheduling technique kicks in here: incoming queries will now be routed to R1 rather than equal load balancing, resulting in lower load at R2. It is easy to see that at this point, BlowFish will load balance queries to R1 and R2 proportional to their respective throughputs.

4 Evaluation

BlowFish is implemented in ≈ 2K lines of C++ on top of Succinct [7]. We apply BlowFish to application domains outlined in §1.1 and compare its performance against state-of-the-art schemes for each application domain.

Evaluation Setup. We describe the setup used for each application in respective subsections. We describe here what is consistent across all the applications: dataset and query workload. We use the TPC-H benchmark dataset [5], that consists of records with 8 byte keys and roughly 140 byte values on an average; the values comprise of 15 attributes (or columns). We note that several of our evaluation results are independent of the underlying dataset (*e.g.*, bandwidth for data repair, time taken to navigate along the tradeoff curve, etc.) and depend only on amount of data per server.

We use a query workload that comprises of 50% random access queries and 50% search queries; we discuss the impact of varying the fraction of random access and search queries in §4.1. Random access queries return the entire value, given a key. Search queries take in an (attribute, value) pair and return all keys whose entry for the input attribute matches the value. We use three query distributions in our evaluation for generating queries over the key space (for random access) and over the attribute values (for search). First, *uniform distribution* with queries distributed uniformly across key space and attribute values; this essentially constitutes a worst-case scenario for BlowFish[5]. The remaining two query workloads follow *Zipf distribution with skewness* 0.99 *(low skew) and* 0.01 *(heavily skewed)*, the last one constituting the best-case scenario for BlowFish.

All our distributed experiments run on Amazon EC2 cluster comprising of c3.2xlarge servers, with 15GB RAM backed by two 80GB SSDs and 8 vCPUs. Unless mentioned otherwise, all our experiments shard the input data into 8GB shards and use one shard per CPU core.

4.1 Storage Performance Tradeoff

We start by evaluating the storage-performance tradeoff curve enabled by BlowFish. Figure 6 shows this tradeoff for query workload comprising of 50% random access and 50% search queries; Appendix A presents the curves for other workloads, as well as for throughput variation with sampling rates. We make two observations. First, BlowFish achieves storage footprint varying from 0.5× to 8.7× the input data size (while supporting search functionality; the figure shows only up to 1.5× the data size for clarity)[6]. BlowFish, thus, does not enforce compression. Second, increase in storage leads to super-linear increase in throughput (moving from ≈ 0.5 to ≈ 0.75 leads to 20× increase in throughput) due to non-linear computational cost of operating on compressed indexes [7].

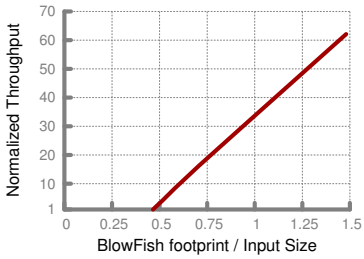


Figure 6: Storage-Throughput tradeoff curves (per thread) enabled by BlowFish. The y-axis is normalized by the throughput of smallest possible storage footprint (71Ops) in BlowFish.

Note that the tradeoff for mixed workload has characteristics similar to 100% `search` workload (Appendix A) since, similar to other systems, execution time for search is significantly higher than random access. The throughput is, thus, dominated by search latency.

4.2 Data Repair During Failures

We now apply BlowFish to the first application: efficient data recovery upon failures.

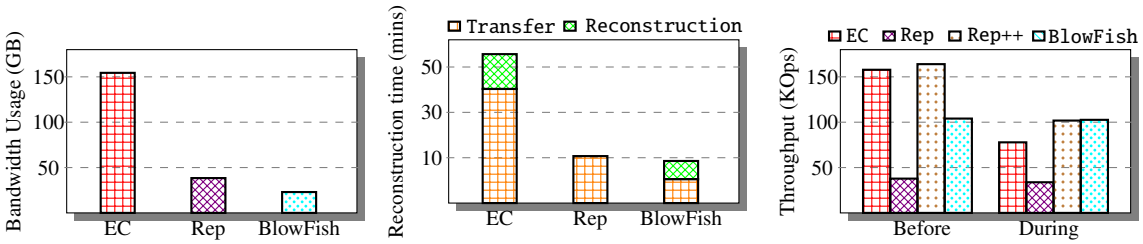
Existing techniques and BlowFish tradeoffs. Two techniques exist for data repair during failures: replication and erasure codes. The main tradeoff is that of storage and bandwidth, as shown in Table 1. Note that this tradeoff is hard; that is, for both replication and erasure codes, the storage overhead and the bandwidth for data repair is fixed for a fixed fault tolerance. We discuss related work in §5, but note that erasure codes remain inefficient for data stores serving small objects due to high repair time and/or bandwidth requirements.

4.2.1 Experimental Setup

We perform evaluation along four metrics: storage overhead, bandwidth and time required for data repair, and throughput before and during failures. Since none of the open-source data stores support erasure codes, we use an implementation of Reed-Solomon (RS) codes [6]. The code use 10 data blocks and 2 parity blocks, similar to those used at Facebook [24, 29], but for two failure case. Accordingly, we use 3× replication. For BlowFish, we use an instantiation that uses three replicas with storage 0.7×, 0.55× and 0.55×, aggregating to 1.8× storage — an operating point between erasure codes and replication.

[5] Intuitively, queries distributed uniformly across shards and across records alleviates the need for shards having varying storage footprints.

[6] The smallest footprint is 0.5× since TPC-H data is not very compressible, achieving compression factor of 3.1 using gzip.



(a) Bandwidth (b) Restoration Time (c) Throughput

Figure 7: Comparison of BlowFish against RS erasure codes and replication (discussion in §4.2.2). BlowFish requires 6.7× lower bandwidth for data repair compared to erasure codes, leading to 3.5× faster repair time. BlowFish achieves 1.5× lower throughput than erasure codes and replication under no failures, and comparable throughput during failures.

We use 12 server EC2 cluster to put data and parity blocks on separate servers; each server contains both data and parity blocks, but not for the same data. Replicas for replication and BlowFish were also distributed similarly. We use 160GB of total raw data distributed across 20 shards. The corresponding storage for BlowFish, replication and erasure codes is, thus, 288, 480 and 192GB. Note that the cluster has 180GB main memory. Thus, all data shards for erasure codes fit in memory, while a part of BlowFish and replication data is spilled to disk (modeling storage-constrained systems).

We use uniform query distribution (across shards and across records) for throughput results. Recall that this distribution constitutes a worst-case scenario for BlowFish. We measure the throughput for the mixed 50% random access and 50% search workload.

4.2.2 Results

Storage and Bandwidth. As discussed above, BlowFish, replication and RS codes have a storage overhead of 1.8×, 3× and 1.2×. In terms of bandwidth, we note that the three schemes require storing 24, 40 and 16GB of data per server, respectively. Figure 7(a) shows the corresponding bandwidth requirements for data repair for the three schemes. Erasure codes require 10× the amount of failed data compared to replication and BlowFish, but also store lesser data per server (best case scenario for erasure codes in terms of all metrics).

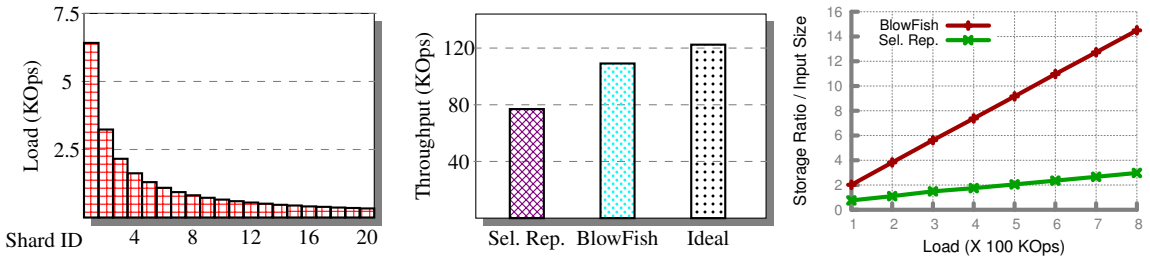
Repair time. The time taken to repair the failed data is a sum of two factors — time taken to copy the data required for recovery (transfer time), and computations required by the respective schemes to restore the failed data (reconstruction time). Figure 7(b) compares the data repair time for BlowFish against replication and RS codes.

RS codes require roughly 3.5× higher transfer time compared to BlowFish. Although erasure codes read the required data in parallel from multiple servers, the access link at the server where the data is being collected becomes the network bottleneck. This is further exacerbated since these servers are also serving queries. The decoding time of RS codes is similar to reconstruction time for BlowFish. Overall, BlowFish is roughly 3× faster than RS codes and comparable to replication in terms of time taken to restore system state after failures.

Throughput. The throughput results for the three schemes expose an interesting tradeoff (see Figure 7(c)).

When there are no failures erasure codes achieve higher throughput compared to both BlowFish and replication (`Rep`). This is rather non-intuitive since replication has three copies to serve queries while RS codes only one. However, recall that the cluster is bottlenecked by the capacity of faster storage. Since the queries in replication and in BlowFish are load balanced across multiple replicas, many of these queries have to be executed off SSD, thus reducing the overall system throughput (much more for replication since many more queries are executed off SSD). To that end, we also evaluated the case of replication where queries are load balanced to only one replica (shown by `Rep++`); in this case, as expected, replication achieves throughput comparable to erasure codes; BlowFish achieves throughput roughly 33% less than the two schemes under no failure scenario.

During failures, the throughput for both erasure codes and `Rep++` reduces significantly. For RS codes, 10 out of 11 servers are used to both read the data required for recovery as well as to serve queries. This severely affects the overall RS throughput (reducing it by 2×). For replication, note that the amount of failed data is 40GB (five shards). Recovering these shards results in replication creating two kinds of interference: interfering with queries being answered on data unaffected by failures *and* queries in `Rep++` answered on failed server now being answered off-SSD from remaining servers. This in-



(a) Load distribution across shards (b) Throughput for a fixed storage (c) Storage required to sustain load

Figure 8: Comparison of BlowFish and Selective caching for Skewed Workload application. See §4.3 for discussion.

terference reduces the throughput of Rep++ by almost 33%. Note that both these interferences are minimal in BlowFish: fewer shards need be constructed, thus fewer servers are interfered with, and fewer queries go to SSD. It turns out that the interference is minimal, and BlowFish maintains its throughput during failures.

4.3 Skewed Workloads

We now apply BlowFish to the problem of efficiently utilizing the system cache for workloads with skewed query distribution across shards (*e.g.*, more queries on hot data and fewer queries on warm data). The case of skew across shards varying with time is evaluated in next subsection.

State-of-the-art. The state-of-the-art technique for spatially-skewed workloads in Selective caching [8] that caches, for each object, number of replicas proportional to the load on the object. Selective caching is no worse than Succinct [7], that will store equal number of replicas irrespective of the load.

4.3.1 Experimental Setup

We use 20 data shards, each comprising of 8GB of raw data, for this experiment. We compare BlowFish and Selective caching using two approaches. In the first approach, we fix the cluster (amount of fast storage) and *measure* the maximum possible throughput that each scheme can sustain. In the second approach, we vary the load for the two schemes and *compute* the amount of fast storage required by each scheme to sustain that load.

For the former, we use a cluster with 8 EC2 servers. A large number of clients generate queries with a Zipf distribution with skewness 0.01 (heavily skewed) across the shards. As shown in Figure 8(a), the load on the heaviest shard using this distribution is 20× the load on the lightest shard — this models the real-world scenario of a few shards being "hot" and most of the shards being "cold". For selective caching, each shard has number of replicas proportional to its load (recall, total storage is fixed); for BlowFish, the shard operates at a point on the tradeoff curve that can sustain the load with minimal storage overhead. We distribute the shards randomly across the available servers. For the latter, we vary the load and compute the amount of fast storage required by the two schemes to meet the load assuming that the entire data fits in fast storage. Here, we increase the number of shards to 100 to perform computations for a more realistic cluster size.

4.3.2 Results

For fixed storage. The storage required for selective caching and BlowFish to meet the load is 155.52GB and 118.96GB, respectively. Since storage is constrained, some shards in selective caching can not serve queries from faster storage. Intuitively, this is because BlowFish provides a finer-grained tradeoff (increasing the storage overhead fractionally, just enough to meet the performance goals) compared to the coarse-grained tradeoff of selective replication (throughput can be increased only by 2× by adding another replica requiring 1× higher storage overhead). Thus, BlowFish utilizes the available system cache more efficiently. Figure 8(b) shows that this leads to BlowFish achieving 1.5× higher throughput than selective caching. Rather interestingly, we note that BlowFish achieves 89% of the ideal throughput — the load cluster can sustain for this skewed workload. We show in Appendix B how to compute the ideal throughput.

Fixed load. Figure 8(c) shows that, as expected, BlowFish requires 2.7 – 4.9× lower amount of fast storage compared to selective caching to sustain the load.

4.4 Time-varying workloads

We now evaluate BlowFish's ability to *adapt* to time-varying load, in terms of time taken to adapt, queue stability. We also evaluate the performance of BlowFish's

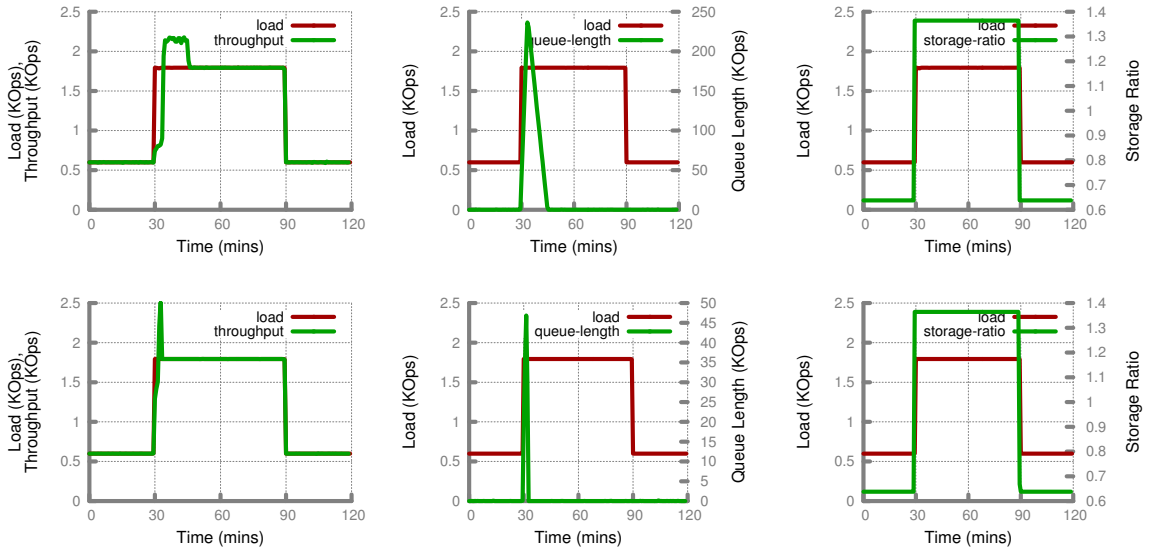


Figure 9: Opportunistic layer construction with spiked changes in load for uniform workload (top three) and skewed workload (bottom three). The figures show variation in throughput (left), request queue length (center) and storage footprint (right).

scheduling technique during such time-varying loads.

4.4.1 Experimental Setup

We perform micro-benchmarks to focus on adaptation time, queue stability and per-thread shard throughput for time-varying workloads. We use a number of clients to generate time-varying load on the system. We performed four sets of experiments: uniform and skewed (Zipf with skewness 0.01) query distribution (across queried keys and search terms); and, gradual and spiked variations in load. It is easy to see that (uniform, spiked) and (skewed, gradual) are the worst-case and the best-case scenario for BlowFish, respectively. We present results for spiked variations in load (*e.g.*, due to transient failures) for both uniform and skewed query distribution; the remaining results are in Appendix C. We perform micro-benchmarks by increasing the load on the shard from 600Ops to 1800Ops suddenly (3× increase in load models failures of two replicas, an extremely unlikely scenario) at time $t = 30$ and observe the system for an hour before dropping down the load back to 600Ops at time $t = 90$.

4.4.2 Results

BlowFish adaptation time and queue stability. As the load is increased from 600Ops to 1800Ops, the throughput supported by the shard at that storage ratio is insufficient to meet the increased load (Figures 9(a) and 9(d)). As a result, the request queue length for the shard increases (Figures 9(b) and 9(e)). At one point, BlowFish triggers *opportunistic layer creation* — the system immediately allocates additional storage for the two sampled arrays (note the increased storage ratio in Figures 9(c) and 9(f)); the sampled values are filled in gradually as queries are executed.

At this point, the results for uniform and skewed query distribution differ. For the uniform case, the already filled sampled values are reused infrequently. Thus, it takes BlowFish longer to adapt (≈ 5 minutes) before it starts draining the request queue (the peak in Figure 9(b)). BlowFish is able to drain the entire request queue within 15 minutes, making the system stable at that point.

For the skewed workload, the sampled values computed during query execution are reused frequently since queries repeat frequently. Thus, BlowFish is able to adapt much faster (≈ 2minutes) and drain the queues within 5minutes. Note that this is akin to caching of results, explicitly implemented in many existing data stores [1, 3, 21] while BlowFish provides this functionality inherently.

BlowFish scheduling. To evaluate the effectiveness and stability of BlowFish scheduling, we turn our attention to a distributed setting. We focus our attention on three replicas of the same shard. We make the server storing one of these replicas storage constrained (replica #3); that is, irrespective of the load, the replica cannot trigger navigation along the storage-performance tradeoff curve. We then gradually increase the workload from 3KOps to 8KOps in steps of 1KOps per 30 minutes (Figure 10) and

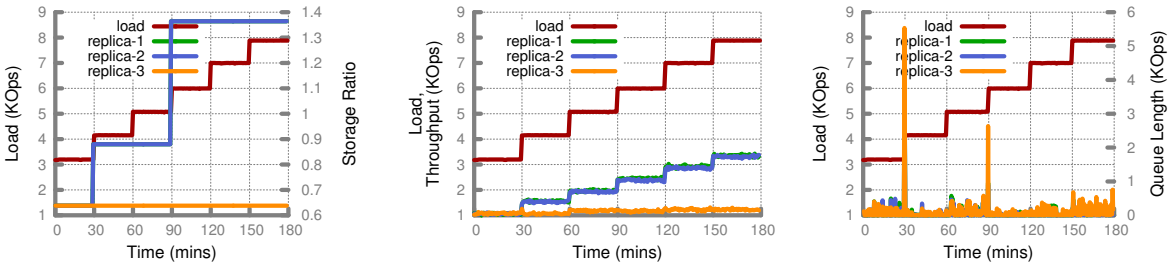


Figure 10: The effect of query scheduling. Performance when queries are scheduled across different shards based on the load seen by them. Variation in storage-footprints (left), response-rates (center) and request queue-lengths (right).

observe the behavior of request queues at three replicas.

Initially, each of the three replicas observe a load of 1KOps since queue sizes are equal, and BlowFish scheduler equally balances the load. As the load is increased to 4KOps, the replicas are no longer able to match the load, causing the request queues at the replicas to build up (Figure 10(c)). Once the queue lengths cross the threshold, replica #1 and #2 trigger layer construction to match higher load (Figure 10(a)).

As the first two replicas opportunistically add layers, their throughput increases; however, the throughput for the third replicas remains consistent (Figure 10(b)). This causes the request queue to build up for the third replica at a rate higher than the other two replicas (Figure 10(c)). Interestingly, the BlowFish reduces quickly adapts, and stops issuing queries to replica#3, causing its request queue length to start dropping. We observe a similar trend when the load increases to 5KOps. BlowFish does observe queue length oscillations during adaptation, albeit of extremely small magnitude.

5 Related Work

We compare and contrast BlowFish against three key areas of related work:

Storage-performance tradeoff. Existing data stores typically support only two operating points on the storage-performance tradeoff curve — compressed but low throughput, and uncompressed but high throughput. As shown in the paper, a smooth storage-performance tradeoff can enable a wide range of new applications, as well as, provide benefits for existing applications. Several compression techniques (*e.g.*, gzip) can allow achieving different compression factors by changing parameters. However, these are also require decompressing and re-compressing the entire data on the server.

Fault Tolerance in Data Stores. Replication and Erasure Codes are the two main schemes to achieve fault tolerance. Replication achieves high throughput and requires close to optimal bandwidth and time to recover failed data. Perhaps the only shortcoming of replication is high storage requirements. Erasure codes achieve similar fault tolerance using significantly lower storage, but require extremely high bandwidth and repair time. Studies have shown that the bandwidth requirement of traditional erasure codes is simply too high to use them in practice [29]. Several research proposals [20, 27, 29] reduce the bandwidth requirements of traditional erasure codes for batch processing jobs. These codes remain inefficient for data stores serving small objects.

As shown in §4, BlowFish achieves storage close to erasure codes, while maintaining the bandwidth and repair time advantages of replication.

Selective Caching. As discussed in §1 and §4, selective caching can achieve good performance for workloads skewed towards a few popular objects. However, it only provides a coarse-grained support — increasing the throughput by 2× by increasing the storage overhead by 1×. BlowFish, instead, provides a much finer-grained control allowing applications to increase the storage fractionally, just enough to meet the performance goals. This provides BlowFish performance benefits when the system is not network bottlenecked.

6 Conclusion

We have presented BlowFish, a distributed data store that provides a smooth storage-performance tradeoff between the two extremes — compressed data that supports low throughput and uncompressed data that supports high throughput. BlowFish is unique, in that, not only it exposes a flexible tradeoff but also allows applications to navigate along this tradeoff curve over fine-grained time scales. Using this flexibility, we explored several problems from real-world production clusters from a different "lens" and showed that the tradeoff exposed by BlowFish can offer significant benefits.

References

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

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

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

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

[5] TPC-H. http://www.tpc.org/tpch/.

[6] TPC-H. https://github.com/catid/longhair.

[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] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] V. Gupta, M. H. Balter, K. Sigman, and W. Whitt. Analysis of join-the-shortest-queue routing for web server farms. *Performance Evaluation*, 64(9):1062–1081, 2007.

[20] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, et al. Erasure coding in windows azure storage. In *USENIX Annual Technical Conference*, pages 15–26, 2012.

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

[22] 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.

[23] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *USENIX Sympo-*

sium on Networked Systems Design and Implementation (NSDI), 2014.

[24] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. f4: Facebook's warm blob storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 383–398, Broomfield, CO, Oct. 2014. USENIX Association.

[25] 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.

[26] 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.

[27] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 331–342. ACM, 2014.

[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] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring elephants: Novel erasure codes for big data. In *Proceedings of the VLDB Endowment*, volume 6, pages 325–336. VLDB Endowment, 2013.

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

[31] 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.

A Storage-Performance Tradeoff Curve

We outline how lookups are performed on the LSA (§3.1) in Algorithm 1. At a high level, we obtain the layer ID and index into the corresponding layer for an index into the LSA, and perform a lookup on that layer. The layer ID and index values are obtained by consulting the auxiliary structures stored along with the layers (see Figure 4).

Algorithm 2 shows how BlowFish creates layers *opportunistically* by using the computations done during query execution to populate the values in a layer.

Algorithm 1 `LookupLSA`

```
1:  procedure LookupLSA(Idx)   ▷ Procedure to perform lookups on the LSA. If the value is sampled at the index, perform a lookup.
2:      if IsSampled(Idx) then
3:          l_id = LayerID(Idx)                    ▷ Get layer ID.
4:          l_idx = LayerIdx(Idx)                  ▷ Get index into layer.
5:          return SampledArray[l_id][l_idx]
6:      end if
7:  end procedure
8:  procedure LayerID(Idx)   ▷ Get the layer ID from the index into the sampled array; α is the sampling rate.
9:      return LayerID[Idx % α]
10: end procedure
11: procedure LayerIdx(Idx) ▷ Get the layer ID from the index into the sampled array; α is the sampling rate.
12:     count = Count[LayerID(Idx)]
13:     PeriodIdx = LayerIdx[Idx % α]
14:     return count × (Idx / α) + PeriodIdx
15: end procedure
```

The layered structure of the LSA, together with efficient layer addition and deletion algorithms, enable a storage-performance tradeoff, wherein higher storage footprint (i.e., more layers in LSA) leads to higher throughput for `random access` and `search` query workloads. Figure 11 summarizes the tradeoffs enabled by BlowFish, and Figure 12 correlates the variation in throughput with the effective sampling rates of the LSA.

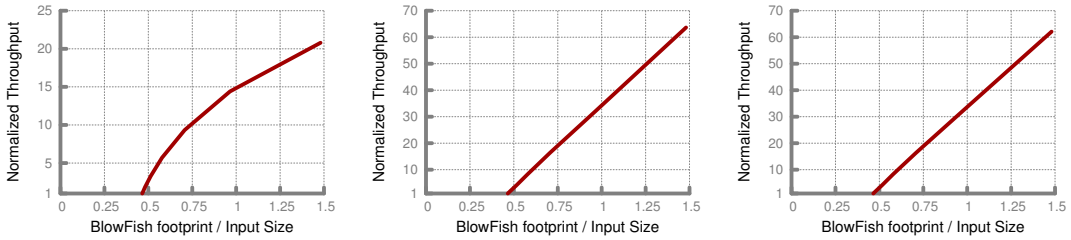
Algorithm 2 `CreateLayerOpportunistic`

```
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 = GetValue(Idx)   ▷ Get the unsampled value.
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
```

B Computing maximum System Load and Ideal Throughput

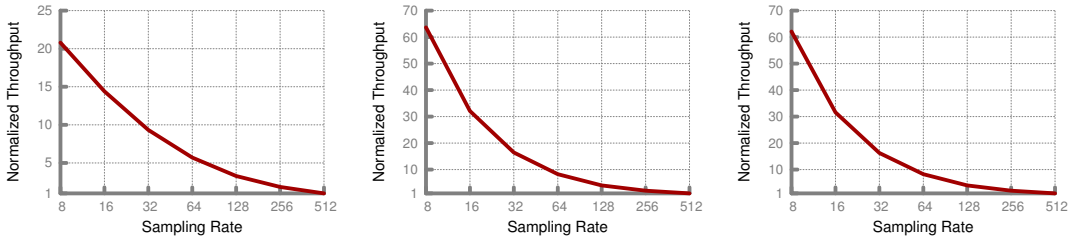
Recall, from §4.1, that increase in storage footprint leads to non-linear throughput increase. Thus, the highest per-server throughput is achieved for the rightmost point on the tradeoff curve for which a single 8GB shard fits entirely in memory (shown by 1.5× storage overhead, achieving throughput of 4210Ops for mixed query workload). We use 8 EC2 servers for this experiment, giving us the maximum possible system throughput to be roughly 134720 queries per second. We consider this to be the load on the cluster.

In order to compute the maximum throughput that can be acheived by the system, we first take the system overheads into account. In particular, the Server Handler module in BlowFish recieves a dedicated core for handling client requests and load balancing queries across shards. Since each EC2 server has 8 cores, we are left with 7 cores dedicated to executing queries. We then compute the fraction of queries handled by each shard, given the skew in queries. The compute resources allocated to each shard is proportional to the number of queries seen by it; therefore, for each server, we compute the amount compute resource allocated to each shard, from the remaining 7 cores. We then obtain the maximum throughput for each shard, given its storage footprint by consulting Figure 11(c). The sum of throughputs for different shards in the system, weighted by the amount of compute resources allocated to them, gives us the ideal throughput for the system; we find this value to be 120687 Ops.



(a) `random access` (b) `search` (c) `50% random access + 50% search`

Figure 11: Storage-Throughput tradeoff curves (per thread) for three workloads with varying fraction of `random access` and `search` queries. The x-axis is normalized by shard size and the y-axis is normalized by the throughput of smallest possible storage footprint in BlowFish. The normalization factors are 3874 OPS for random access only, 37 OPS for search only, and 71 OPS for the mixed workload.



(a) `random access` (b) `search` (c) `50% random access + 50% search`

Figure 12: Sampling Rate vs. Throughput for workloads with varying fractions of `random access` and `search` queries. The x-axis values show the sampling rates for the two sampled arrays (see §2.1). The y-axis is normalized as described in Figure 11

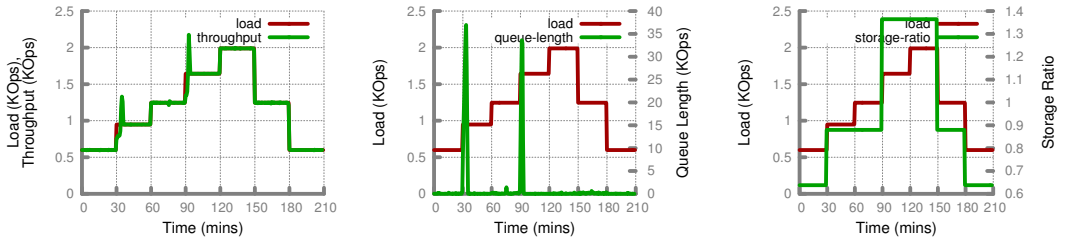


Figure 13: Opportunistic layer construction with gradual changes in request-rate for uniform workload; Variation in throughput (left), storage-footprint (center) and request queue-length (right).

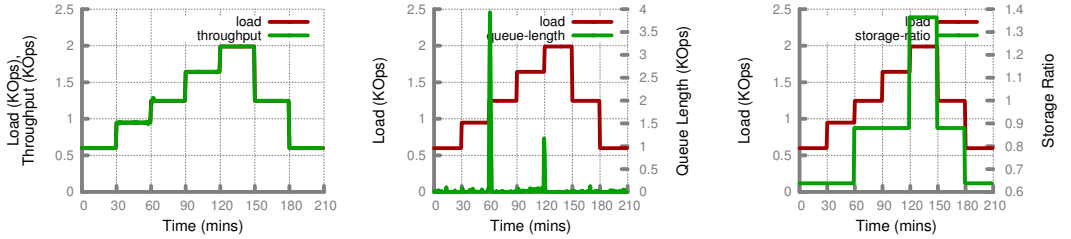


Figure 14: Opportunistic layer construction with gradual changes in request-rate for skewed workload; Variation in throughput (left), storage-footprint (center) and request queue-length (right).

C Gradual Workload Variation

We present the results for how BlowFish adapts to changin workloads with a setup identical to §4.4, but for a slightly modified workload. In particular, instead of increasing the load on the shard from 600Ops to 1800Ops in a single step, we increase the load from 600Ops to 2000Ops per thread, with a gradual increase of 350Ops per thread at 30 minute intervals. This granularity of request rate increase is similar to those reported in real-world production clusters [9].

C.0.3 Results

Uniform workload (Figure 13). As the load increases from 600Ops to 950Ops, the throughput increases to 800Ops, the throughput supported at that storage ratio (Figure 13(a)). Since the load is higher than the system can sustain, there is a corresponding buildup at the request queue (Figure 13(b)), and BlowFish triggers a layer addition by allocating space for the new layers (Figure 13(c)). As before, incoming queries opportunistically fill up values in the new layers, and gradually increase the throughput for the shard. This continues until the throughput matches the load on the shard; the throughput increases even beyond the load 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 load is increased to 1650Ops.

Skewed workload (Figure 14). The trends observed for the skewed workload are similar to those for the uniform worklod, with two key differences. First, we observe that BlowFish triggers layer creation at different points for this workload. In particular, the throughput for the skewed workload at the same storage footprint (0.8 in Figure 13(c) and 14(c)) is higher than that for the uniform workload. To see why, note that the performance of `search` operations varies significantly based on the queries; while the different queries contribute equally for the uniform workload, the throughput for the skewed workload is shaped by the queries that occur more frequently. This effect attributes for the different throughput characteristics for the two workloads at the same storage footprint.

Second, as noted before (§4.4), BlowFish adaptation benefits from the repetitive nature of queries in the skewed workload. Intuitively, repeated queries can reuse the values populated during their previous exection, significantly speeding up the queries. This leads to a faster adaptations to increase in load and quicker depletion of the built up queues.

C.1 BlowFish data model and assumptions

BlowFish enables the same set of functionality as Succinct — support for random access and search queries on flat unstructured files, with extensions for key-value stores and NoSQL stores.

Assumptions. BlowFish makes three 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. Indeed, indexes to support search queries along with the input data makes it hard to fit the entire data in fastest storage especially for purely in-memory data stores (*e.g.*, Redis [4], MICA [23], RAMCloud [25]). Second, as performance benchmarks over existing data stores [23] show, most *systems are not bottlenecked by the network capacity*. Third, BlowFish assumes that data can be sharded in a manner that a query does not require touching each server in the system. Many real-world datasets and query workloads admit such sharding schemes [14, 26, 31].