Honeycomb: Fine-grained Sharing of Ephemeral Storage for Serverless Analytics

Paper #653

Abstract

Existing ephemeral storage systems for serverless analytics allocate storage resources at the job granularity — jobs specify their storage demands at the time of the submission; and, the system allocates storage resources equal to the job's demand for the entirety of its lifetime. This leads to resource underutilization and/or performance degradation when intermediate data sizes vary during job execution.

Honeycomb is an ephemeral storage system for serverless analytics that meets the instantaneous storage demand of a job at seconds timescales. Honeycomb thus efficiently multiplexes faster storage capacity across concurrently running jobs, reducing the overheads of reads and writes to slower storage. As a result, job execution time improves by $1.6-2.5\times$ over production workloads. Honeycomb implementation currently runs on Amazon EC2, and natively supports a large class of analytics applications on AWS Lambda.

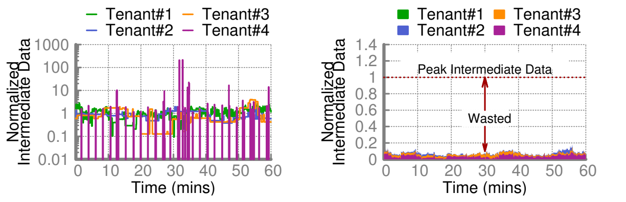
1 Introduction

Serverless architectures offer on-demand elasticity of compute and persistent storage, while charging users for resources consumed by their jobs at fine-grained timescales [1–3]. While originally deemed useful only for web microservices, IoT applications and ETL workloads [4, 5], recent work on serverless analytics has demonstrated the benefits of serverless architectures for resource- and cost-efficient data analytics [6–20].

The core idea in serverless analytics is to use a remote low-latency, high-throughput *ephemeral storage system* for: (1) inter-task[1] communication via shared memory; and (2) for multi-stage jobs, storing intermediate data beyond the lifetime of the task that produced the data (until it is consumed by downstream tasks). Ephemeral storage systems thus allow decoupling storage, communication and lifetime management of intermediate data from individual compute tasks, enabling serverless analytics frameworks to exploit the on-demand compute elasticity offered by serverless architectures.

Existing ephemeral storage systems [7, 8], however, suffer from a fundamental limitation: they allocate storage resources at the job granularity. That is, jobs specify their storage demands at the time of the submission; and, the system allocates and reserves storage resources equal to the job's demand (potentially by elastic scaling of total ephemeral storage system capacity) for the entirety of its lifetime [8, Figure 1].

[1]Existing distributed programming frameworks, while different in underlying programming models and semantics, share a common structure (Figure 2, Figure 3) — the job is split into multiple *tasks*, possibly organized along multiple stages or a directed acyclic graph. Each task generates *intermediate* data during its execution; upon completion, each task partitions its intermediate data and exchanges it with tasks in the next stage.



(a) Intermediate data (normalized by mean usage) (b) Cumulative intermediate data (normalized by peak usage)

Figure 1. Analysis of production workloads from Snowflake [20] for four tenants over a 1 hour window: (a) the ratio of peak to average storage demand for a job can vary by an order of magnitude, or more, during its execution; and (b) provisioning for peak usage results in average utilization less than 10%. Across all ($>$ 2000) tenants, the average utilization is ~19%.

The problem of performance degradation and/or resource underutilization for such job-level resource allocation in storage systems is well-understood [20, 21]. On the one hand, if jobs specify their average demand, their performance degrades when instantaneous demand is higher than their average demand (due to read/write requests being executed on slower secondary storage, *e.g.*, S3), as shown in Figure 1(a). On the other hand, if jobs specify their peak demand, the system suffers from resource underutilization when their instantaneous demand is lower than the peak demand, as shown in Figure 1(b). Indeed, the problem worsens as the difference between the peak demand and the average demand increases. Unfortunately, the target use case of ephemeral storage systems — intermediate data — is a bad-case scenario for the difference between peak and average demands: recent deployment studies have reported that intermediate data sizes can vary over multiple orders of magnitude during the lifetime of the job [20]. For instance, Figure 1 presents our analysis of the publicly-released dataset of $>$ 2000 tenants from Snowflake [20]: it shows that the ratio of peak to average demands in Snowflake production workloads can vary by two orders of magnitude over a period of minutes! As a result, job-level resource allocation in existing ephemeral storage systems can lead to significant performance degradation and/or resource underutilization (our evaluation results in §6 show as much as $4.1\times$ performance degradation and 60% resource underutilization for production workloads).

This paper presents Honeycomb, an ephemeral storage system for serverless analytics that allocates storage resources at the granularity of small fixed-size storage blocks — multiple storage blocks store intermediate data for individual tasks within a job. Honeycomb design is motivated by virtual memory design in operating systems that also does memory allocation to individual processes at the granularity of fixed-size memory blocks (pages); indeed, Honeycomb adapts this design to ephemeral storage systems for serverless analytics. Performing resource allocation at the granularity of small storage blocks allows Honeycomb to elastically scale storage resources allocated to individual jobs *without* a priori knowledge of intermediate data sizes, and to meet the instantaneous job demands at seconds timescales. As a result, Honeycomb can efficiently multiplex the available faster ephemeral storage capacity across concurrently running jobs, thus minimizing the overheads of reads and writes to slower secondary storage (*e.g.*, S3). We show that such fine-grained resource allocation allows Honeycomb to achieve $1.6 - 2.5\times$ improvement in application-level performance, when compared to Pocket [8], across a variety of workloads and cluster configurations.

Enabling fine-grained resource allocation requires resolving four unique challenges introduced by serverless analytics:

- First, each serverless analytics job can be organized around multiple stages (or a directed acyclic graph), with tens to thousands of individual tasks in each stage [6–20]. Thus, performing fine-grained resource allocation requires an efficient mechanism to keep an up-to-date mapping between tasks and storage blocks allocated to individual tasks.
- Second, the number of tasks reading and writing to the shared ephemeral storage can change rapidly in serverless analytics. Thus, task-level isolation becomes critical: arrival and departure of new tasks should not impact the performance of existing tasks.
- Third, decoupling of serverless tasks from their intermediate data means that the tasks can fail independent of the intermediate data. Thus, we need mechanisms for explicit lifetime management of intermediate data.
- Fourth, decoupling of tasks from their intermediate data also means that data partitioning upon elastic scaling of storage capacity becomes challenging, especially for certain data types used in serveless analytics (*e.g.*, key-value stores [6, 6–8, 11, 13, 15, 19]). Indeed, naïvely delegating this to applications would require large network transfers (between compute tasks and ephemeral storage system) and data read/write operations every time the capacity is scaled (§3). Thus, we need new mechanisms to efficiently enable data partitioning within the ephemeral storage system.

Honeycomb resolves these challenges by integrating several mechanisms into an end-to-end system. First, in a sharp contrast to classical distributed shared memory systems [22–26] and recent in-memory stores [27–30] that use a global address space, Honeycomb exposes a hierarchical address space that captures the structure of the analytics job (*e.g.*, directed-acyclic graphs with individual tasks) [20, 21]. Such a hierarchical addressing mechanism allows Honeycomb to both efficiently manage the mapping between storage blocks and tasks, and provide task-level isolation. Second, Honeycomb ties the hierarchical addresses with a lease-based mechanism for efficient lifetime management of intermediate data. Finally, similar to function shipping, Honeycomb supports partition-function shipping — analytics jobs can offload data repartitioning upon resource allocation/deallocation to Honeycomb, that performs seamless data repartitioning. We discuss in §3 how, for each of these techniques, the aforementioned unique challenges introduced by serverless architectures require Honeycomb to make different design decisions than original realizations of these mechanisms. Honeycomb integrates these mechanisms into an end-to-end ephemeral storage system that provides resource elasticity at the granularity of seconds, matching the compute elasticity timescales of serverless architectures.

We have realized an end-to-end implementation of Honeycomb, which we will open-source along with the paper. Honeycomb's data plane enables compute tasks to read-/write intermediate data to their blocks via an intuitive, programmable, API (§4.1). We demonstrate the expressiveness of Honeycomb's API by realizing serverless incarnations of several powerful distributed programming frameworks on top of Honeycomb (§5): MapReduce [31], Dryad [32], StreamScope [33] and Piccolo [34]. We compare Honeycomb against five state-of-the-art ephemeral storage systems over a variety of workloads. Our evaluation suggests that fine-grained resource allocation in Honeycomb allows it to improve resource utilization by as much as $3\times$ and distributed analytics performance by a factor of $1.6 - 2.5\times$ compared to peak provisioning.

2 Pocket Background

Honeycomb builds upon Pocket, a distributed low-latency, high-throughput ephemeral storage system for serverless analytics. Pocket already resolves a number of interesting challenges pertinent to serverless analytics, as we describe next.

Scalable centralized management. Pocket architecture (Figure 2) comprises decoupled control, metadata and data planes. While the data storage itself is distributed across multiple storage servers, the storage management functionalities via control and metadata planes are logically centralized. This greatly simplifies management since the controller and metadata servers have a global view of the entire system. Specifically, the controller allocates storage resources to analytics jobs and decides where to place the data for different jobs based on its global view of load across storage servers. The metadata plane, in turn, organizes job data into buckets across the storage allocated by the controller, and stores the mapping

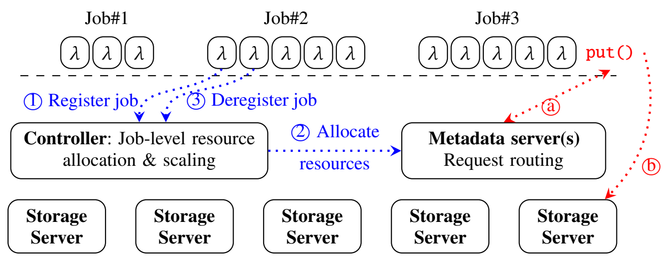


Figure 2. Pocket Architecture. Steps ①-③ show how jobs are registered, memory is allocated and jobs are deregistered, while ⓐ, ⓑ show how operations are routed from serverless compute tasks to storage servers via metadata servers (Figure adapted from [8]).

from buckets to their physical locations at the data plane for directing client requests appropriately. A single centralized metadata server can support 90K requests per second per core (sufficient to support thousands of serverless tasks).

Multi-tiered data storage. Pocket data plane simply stores the job data in a bucket across multiple storage servers and serves them via a key-value API (i.e., `get()`, `put()`). Once a job obtains the physical resource locations from the metadata server, it can read and write data directly from the storage servers. Pocket supports multi-tiered storage: jobs can store data across DRAM, Flash or HDD tiers at the data plane, based on their performance and/or cost constraints.

Adding/removing storage servers. If the aggregate demand of jobs storing data on Pocket grows so much that the system storage capacity is insufficient to serve all of them, then Pocket can scale up the storage capacity by adding more storage servers at the data plane. Similarly, the controller can also scale down the capacity if it falls below a low threshold.

Analytics execution with Pocket. We now describe how a serverless analytics job interacts with Pocket using Figure 2. When the job first starts, it registers itself with the control plane (①), either specifying the amount of storage resources it expects to use, or providing hints that Pocket can use to estimate it. The controller uses this information to allocate resources for the job, and informs the metadata plane regarding the resource placement at the data plane (②). When the job's serverless compute tasks first attempt to read or write intermediate data on the job's allocated storage, it contacts the metadata service to get the IP addresses for the storage servers with its allocated resources (ⓐ). The serverless tasks can subsequently access data directly from the storage servers (ⓑ). Once the job is finished, it deregisters itself at the control plane to release its resources (③).

For the remainder of the paper, we do not focus on the issues that Pocket has already addressed — Honeycomb uses the same centralized management mechanism, supports multi-tiered data storage, can add or remove storage servers when all the capacity is utilized and uses the same analytics execution pipeline as Pocket. Instead, we focus on the specific problems arising out of Pocket's resource allocation mechanisms, which we describe next. We note that while Honeycomb is implemented on top of Pocket, Honeycomb's design can be incorporated into a wide range of multi-tenant data stores that have similar architectures [20, 35–38].

2.1 Limitations of Pocket Resource Allocation

The core challenge in Pocket's resource allocation mechanism is that it allocates storage resources at the granularity of jobs. Upon submission, the job specifies its storage demands; and, Pocket allocates and reserves storage resources equal to the job's demand for the entirety of its lifetime (see Figure 1 in [8]), only releasing them when the job explicitly deregisters.

Such job-level resource allocation is problematic due to two reasons. First, accurately predicting intermediate data sizes is hard for many analytics jobs. Analysis of production analytics workloads [20] have shown that intermediate data sizes have little or no correlation with the amount of persistent data read, or the expected execution time of the query. Indeed, the intermediate data size during job execution depends on the execution plan, which, in turn, can be adapted dynamically by a query planner [21].

Second, even if one could accurately predict the intermediate data sizes, Pocket's resource allocation mechanism requires jobs to specify their demands at the time of the submission (note that hints in Pocket are only used for sharing resources across jobs, not for dynamically changing the storage capacity allocated to individual jobs). This leads to the standard tradeoff between performance degradation and/or resource underutilization: if jobs specify their average demand, their performance degrades when instantaneous demand is higher than their average demand (due to read/write requests being executed on slower secondary storage) and if jobs specify their peak demand, the system suffers from resource underutilization when their instantaneous demand is lower than the peak demand. Since intermediate data sizes naturally increase and decrease over time (as tasks in different stages are executed), Pocket's job-level resource allocation will result in either performance degradation or resource underutilization. While we have already seen this for the Snowflake workload in Figure 1(b), similar observations have been made in prior studies for other workloads [7, 39] as well, *e.g.*, the intermediate data size across various stages in a typical TPC-DS query [40] ranges from 0.8MB to 66GB, a difference of 5 orders of magnitude!

3 Honeycomb Design

Honeycomb enables fine-grained sharing of ephemeral storage capacity across concurrently running serverless analytics jobs. Inspired by virtual memory, Honeycomb partitions the storage capacity into fixed-sized blocks (akin to virtual memory pages), and performs storage allocations at the granularity

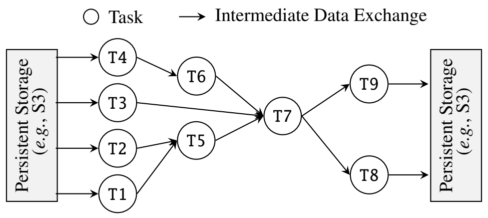

Task
Intermediate Data Exchange
Persistent Storage (e.g., S3)
T4
T3
T2
T1
T6
T5
T7
T9
T8
Persistent Storage (e.g., S3)

Figure 3. Execution DAG example for a typical analytics job. Intermediate data exchange across tasks occurs via Honeycomb.

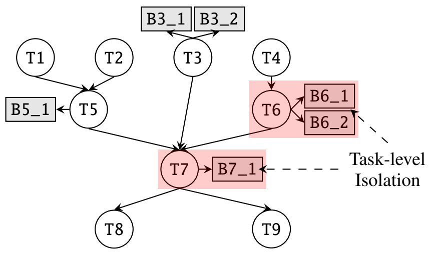


Figure 4. Hierarchical addressing for the job in Figure 3. Honeycomb provides task-level resource isolation for ephemeral storage under each task address-prefix (§3.1). Note that block addresses are only assigned to address-prefixes with currently allocated blocks.

of these blocks. This allows Honeycomb to achieve two desirable properties. First, multiplexing the available capacity at block granularity allows Honeycomb to match instantaneous job demands at seconds timescales. Second, Honeycomb does not require jobs to know (even an estimate of) intermediate data sizes a priori — as tasks write/delete data, Honeycomb dynamically allocates/deallocates resources at block granularity.

Remark. Multiplexing available storage capacity is different from scaling the capacity of the storage pool. Prior systems, including Pocket, focus on the latter: since resource allocation is done at job granularity, as jobs arrive or finish, these systems add and remove the storage servers to elastically scale the capacity of the ephemeral storage system. However, existing capacity may be underutilized since a job may not be using the storage allocated to it. Honeycomb focuses on the former: efficiently sharing the capacity available at any given instant of time across concurrently running jobs. When the ephemeral storage capacity utilization is high (i.e., many jobs are actually using the storage capacity), Honeycomb can add storage servers to scale up the capacity similar to Pocket. Interestingly, by efficiently multiplexing the available storage capacity across concurrently running jobs, Honeycomb also reduces the frequency at which storage servers need to be added/removed from the pool.

As discussed in §1, enabling fine-grained resource allocation requires resolving four unique challenges introduced by serverless analytics (primarily due to decoupling of compute tasks from their intermediate data). In this section, we describe how Honeycomb employs hierarchical addressing (§3.1), intermediate data lifetime management (§3.2) and flexible data repartitioning (§3.3) to resolve these challenges. To assist our discussion, we will use the example in Figure 3, which shows the execution plan for a representative analytics job. The plan is organized as a directed acyclic graph (DAG) where nodes correspond to computation tasks (implemented as serverless functions[2]), while edges denote intermediate data exchange between them via Honeycomb.

[2]Functions refer to a basic computation unit in serverless architectures, *e.g.*, Amazon Lambdas [1], Google Functions [3], Azure Functions [2], etc.

3.1 Hierarchical Addressing

Analytics job are usually organized around multiple stages or a directed acyclic graph. In serverless analytics, where compute elasticity is a first-class primitive, each job may execute tens to thousands of individual tasks [6–20]. Thus, performing fine-grained resource allocation requires an efficient mechanism to keep an up-to-date mapping between tasks and storage blocks allocated to individual tasks. Moreover, the number of tasks reading and writing to the shared ephemeral storage can change rapidly. Under such high concurrency and churn, it becomes important to provide isolation at the granularity of individual tasks: arrival and departure of a task should not affect the resources allocated to other tasks, even from the same job (since it can degrade the *overall* job performance). In this subsection, we describe Honeycomb's hierarchical addressing — a simple, effective, mechanism that enables Honeycomb to maintain a mapping between individual tasks and storage blocks allocated to these tasks, as well as provide isolation at individual task granularity.

Motivated by the Internet hierarchical IP addressing mechanism that captures *network structure*, Honeycomb employs a software-based hierarchical addressing mechanism that captures *execution structure* in analytics jobs. Specifically, Honeycomb organizes intermediate data for analytics jobs within a "virtual" address hierarchy to capture the dependencies between intermediate data for different tasks. We provide an example below, but conceptually, internal nodes in the hierarchy correspond to tasks in the DAG, while leaf nodes correspond to Honeycomb blocks storing intermediate data generated by the tasks. Blocks form the final layer of the hierarchy: block addresses are defined by the path used to reach it in the hierarchy. The immediate address prefix of a block, therefore, identifies the task that generated it. Finally, the edges between internal nodes capture the dependencies between the intermediate data generated by them. To construct the address hierarchy, Honeycomb uses the execution plan for a job (*e.g.*, using AWS Step Function and Azure Durable Function, or via explicit workflow specification from the job). Otherwise, Honeycomb initializes the hierarchy to a single node, and *deduces* the rest on-the-fly based on the intermediate data dependencies between the job's tasks (during registration of individual tasks using Honeycomb API §4.1); this allows Honeycomb to support dynamic query plans, where the DAG is not known a priori.

Example. Figure 4 shows the address hierarchy for the job from Figure 3. The internal nodes T1-T9 correspond to tasks in the DAG, while leaf nodes B3_1, B3_2, etc., correspond to the data blocks allocated to them by Honeycomb for storing their intermediate data. Edges (T1, T5) and (T2, T5) in the address hierarchy indicate that the intermediate data in T5 depends on the intermediate data from both T1 and T2. The complete address of block B6_2 under T6 would be T4.T6.B6_2, while the address-prefix T4.T6 identifies all blocks allocated to T6. To construct the address hierarchy, Honeycomb either uses the execution plan from Figure 3 if the job provides it, or deduces it on the fly. For instance, Honeycomb can deduce that since all sub-tasks in T7 access intermediate data produced by T3, T5, and T6, T7 must have them as parents in the hierarchy.

Organizing intermediate data across an address hierarchy allows Honeycomb to manage resource allocations for an address-prefix independent of other prefixes. Specifically, if the storage in a specific address-prefix spills over to persistent storage (using mechanisms from Pocket), it does not affect the performance of other address-prefixes. Moreover, Honeycomb ensures that once a block is allocated to an address-prefix, it will not be reclaimed until the application either explicitly reclaims it, or stops renewing leases for it (§3.2), affording *isolation* at address-prefix granularity. Since address-prefixes correspond to tasks in Honeycomb address hierarchy, this enables task-level isolation regardless of task concurrency and churn. This is similar to virtual memory, where each process is is assigned its own virtual address space that enables isolation at process granularity; Honeycomb does this at individual task granularity using hierarchical addressing that captures the execution structure of the job.

We outline two important design issues. First, Honeycomb's fine-grained resource allocation should be decoupled from the policies required to enforce desired system behavior. For instance, algorithms to achieve fairness in resource allocation across various jobs or tenants can be easily integrated on top of Honeycomb allocation mechanism. This is orthogonal to Honeycomb's goals of enabling fine-grained sharing (where the overall goal is high resource utilization). Second, address translation — the mapping from virtual addresses to physical storage blocks — is performed similar to Pocket [8] at the centralized metadata server. Thus, unlike hardware address translation that imposes a limit on the size (depth and breadth) of the execution DAG, Honeycomb can easily perform addressing for arbitrary DAGs. Honeycomb's hierarchical addressing and fine-grained resource allocation does introduce additional complexity at the controller; we evaluate the scalability of our controller implementation in

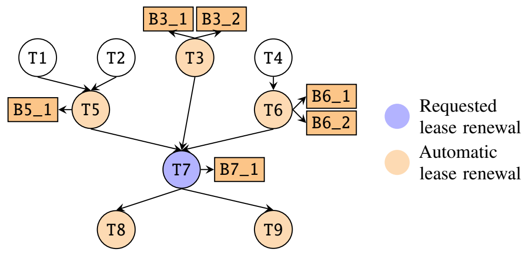


Figure 5. Lease Renewal via Address Hierarchy. Hierarchical addressing simplifies lease renewal in Honeycomb (§3.2), since lease renewal for an address-prefix automatically implies renewals for all parent and descendent address-prefixes in the hierarchy.

the evaluation section and demonstrate that Honeycomb can still scale to ~45K requests per second per core, which is large enough for most realistic deployments.

Block sizing. Similar to page-size in traditional virtual memory, block-size in Honeycomb exposes well-known tradeoffs between the amount of metadata that needs to be stored at the control plane, and memory utilization. In particular, larger block-sizes reduce the amount of per-block metadata at the control plane, at the cost of potentially reduced memory utilization from data fragmentation within blocks, and vice versa. We note that Honeycomb does not suffer the traditional overheads of higher I/O with larger blocks, since it supports fine-grained access within blocks via its data structure interface (§4.1). Moreover, Honeycomb also controls under-utilization within a block via data repartitioning, as described in §3.3. While Honeycomb block sizes can be configured, it employs the block sizes typically used for files in analytics frameworks (*e.g.*, 128MB in HDFS [36]) for compatibility.

Isolation granularity. Since the nodes in the address hierarchy correspond to tasks, Honeycomb provides task-level isolation. It is, however, possible to provide finer or coarser-grained isolation by simply adding another layer to the hierarchy (*e.g.*, for isolation at the granularity of tables in data lake queries) or removing a layer from the hierarchy (*e.g.*, for stage-level isolation in MapReduce frameworks). We chose task-level isolation as our default since most analytics frameworks stand to benefit from task-level isolation, but do not require finer-grained isolation. Individual applications can, however, chose to create custom hierarchies using Honeycomb API, as we outline in §4.1.

3.2 Data Lifetime Management

Existing ephemeral storage systems that perform job-level resource allocation also manage data lifetimes at job granularity — reclaiming storage when the job explicitly deregisters. A unique feature of serverless analytics is that a task's intermediate data storage is decoupled from its execution: while its execution occurs at the serverless compute platform, the data generated and consumed by it would reside at the ephemeral storage system. This also results in the decoupling of their fault domains: with standard mechanisms (*e.g.*, reference counting approaches [41–43]), when the task fails, its corresponding intermediate data becomes dangling state at the ephemeral storage system. To avoid the resulting inefficiency, we need additional mechanisms to efficiently perform task-level data lifetime management.

Honeycomb achieves this by integrating well-known lease management mechanisms [44–46] with hierarchical addressing to enable lifetime management of intermediate data. In particular, Honeycomb associates each address-prefix in a job's hierarchical addressing with a lease, and only keeps its data in memory as long as its lease is renewed. Consequently, a job periodically renews leases for the address-prefixes of tasks that are currently running (serverless environments already monitor function execution for billing purposes; these functions could be used to trigger lease renewals). Honeycomb tracks the time a lease was last renewed for each node in the address hierarchy, and updates it for the relevant nodes when a new renewal request for a particular address-prefix is received. Finally, on lease expiry for a particular address-prefix, Honeycomb reclaims all memory resources allocated to it.

The new aspect of lease management in Honeycomb is that it exploits the DAG-based hierarchical addressing to determine dependencies between leases. On receiving a lease renewal request from a task that is currently running, Honeycomb renews leases not only for its address-prefix, but also for the prefixes of tasks that they depend on (i.e., parent nodes in the hierarchy), and for all the prefixes of tasks that depend on it (i.e., all descendant nodes in the hierarchy). This ensures that while a task is running, not only is its own intermediate data kept in memory, but also the data for all the tasks that it depends on, and all the tasks that depend on it. Moreover, this also significantly reduces the number of lease renewal messages that a job has to send.

Honeycomb's leasing mechanism finds a favorable tradeoff between age-based eviction (*e.g.*, in caching approaches, where jobs have no control on intermediate data lifetime) and explicit acquisition and release (where jobs have full control, but job failures could lead to orphaned state). Honeycomb's mechanism not only provides jobs control over the lifetime of their storage resources (via explicit leases), but also ties the fate of the allocated resources to the job — if a lease is not renewed (*e.g.*, due to job or task failure, or if resources are no longer needed), Honeycomb reassigns resources to other jobs or tasks upon lease expiry.

Example. In Figure 3, during task `T7`'s execution, the job periodically renews leases for the prefix `T4.T6.T7`[3] — Honeycomb keeps the intermediate data for the blocks under it in memory as long as the job renews leases for it. Moreover, a lease renewal for task `T7`'s prefix also renews leases for its parent tasks prefixes (i.e., for tasks `T3`, `T5`, `T6`) and for its descendent task prefixes (i.e., for tasks `T8`, `T9`), as shown in Figure 5. Thus, renewing task `T7`'s lease ensures that `T7` can still use the intermediate data generated by its parent tasks; moreover, if any of `T7`'s downstream tasks are active, their intermediate data is automatically kept in memory as well.

Lease duration. Lease duration in Honeycomb exposes a tradeoff between control plane bandwidth and system utilization *over time*. Specifically, longer lease durations reduce the network traffic to the control plane since jobs renew their leases at coarser granularities, but reduce system utilization since Honeycomb does not reclaim (potentially unused) resources from jobs until their leases expire. The problem of picking the right lease duration based on these constraints has been well studied in prior work [44, 45], and can be configured in Honeycomb to meet deployment-specific goals.

3.3 Flexible Data Repartitioning

Decoupling compute tasks from their intermediate data in serverless analytics makes it challenging to efficiently achieve ephemeral storage elasticity at fine granularities. Specifically, as storage is allocated/deallocated to a task, the intermediate data needs to be repartitioned across the remaining blocks. However, decoupling of compute tasks from ephemeral storage, and large number of concurrent tasks means that this repartitioning should not be offloaded to the application. For instance, many existing serverless analytics approaches [7, 8] employ key-value stores for intermediate data storage. In such a setting, if the compute task were to repartition the intermediate data on memory scaling, it would have to first read the key-value pairs from the store over the network, compute the data partitions across the new memory allocation, and write back the data to the store. This would incur significant network latency and bandwidth overheads for the task.

As we discuss in §5, Honeycomb already implements standard data structures used in data analytics frameworks — *e.g.*, files [10, 16–18, 20], to key-value pairs [6, 6–8, 11, 13, 15, 19] to queues [9, 12]. Analytics jobs using these data structures can offload repartitioning of intermediate data upon resource allocation/deallocation to Honeycomb. Each block allocated to a Honeycomb data structure tracks the fraction of the block memory capacity that is currently being used to store data. Whenever the usage grows above a high threshold, Honeycomb, in turn, allocates a new block to the corresponding address-prefix[4]. Subsequently, the overloaded block triggers data structure-specific repartitioning to move part of its data to the new block. Similarly, when the block usage drops below a low threshold, Honeycomb identifies another block in the address-prefix with low-usage with which the block can merge its data. The block then conducts the

[3]Note that task `T7` has four different address-prefixes: the job can renew leases for its data using any of them.

[4]Similar to existing systems [8, 28, 30, 47], Honeycomb can trivially scale its cluster capacity: if the number of free blocks available increase/decrease beyond a certain threshold, Honeycomb adds/removes servers to adjust physical memory resources. Here, we focus only on fine-grained elasticity.

Table 1. Honeycomb User-facing API. See §4.1 for details.

	API	Description
	`connect(honeycombAddress)`	Connect to Honeycomb.
Address Hierarchy	`createAddrPrefix(addr, parent, optionalArgs)` `createHierarchy(dag, optionalArgs)` `flushAddrPrefix(addr, externalPath)` `loadAddrPrefix(addr, externalPath)`	Create address-prefix `addr` with given `parent` address-prefix and `optionalArgs` (*e.g.*, initial capacity), or, create address hierarchy from execution plan provided as a DAG `dag`. Flush/load data in address-prefix to external persistent store.
Address Hierarchy	`leaseDuration = getLeaseDuration(addr)` `renewLease(addr)`	Get the lease duration associated with address-prefix `addr`. Send lease renewal request for address-prefix `addr`.
Data Structure	`ds = initDataStructure(addr, type)`	Initialize data structure of given `type` in address-prefix `addr` and get handle `ds` that encapsulates physical locations of allocated blocks.
Data Structure	Data structure-specific interface implemented using block API (Figure 6).	See Table 2 in §5 for examples.
Data Structure	`listener = ds.subscribe(op)` `notif = listener.get(timeout)`	Subscribe to notifications for operations of type `op` on `ds`. Get latest notification; waits `timeout` seconds for response.

required repartitioning, after which Honeycomb deallocates it. Note that by having the target block conduct the repartitioning instead of the compute task, Honeycomb avoids the network and computational overheads for task itself. Finally, we note that data repartitioning occurs asynchronously in Honeycomb: data access operations across data structure blocks can proceed even while repartitioning is in progress. This allows Honeycomb to ensure application performance is minimally impacted due to data repartitioning (§6.3).

Data structures included in Honeycomb already allow us to implement serverless incarnations of several powerful distributed programming frameworks on top of Honeycomb: MapReduce [31], Spark [48], Dryad [32], StreamScope [33] and Piccolo [34]. We note that data structures used in analytics frameworks — files, queues, key-value stores — require very simple repartitioning mechanisms (unlike data structures like B-trees or other ordered trees that are not used in data analytics frameworks). As such, serverless applications employing these programming models can run on Honeycomb and leverage its flexible data repartitioning without any modification.

Thresholds for elastic scaling. The high and low thresholds for elastic scaling in Honeycomb expose a tradeoff between the data plane network bandwidth and task performance on one hand, and system utilization on the other. Specifically, if the high and low thresholds are set high and low enough, respectively, then elastic scaling is triggered rarely, reducing the amount of network traffic due to data repartitioning. At the same time, extreme threshold values also negatively affect system utilization within the blocks, *e.g.*, lower low-thresholds result in larger number of nearly empty blocks.

We note that appropriate values for these thresholds depend on the workload characteristics, *e.g.*, prior work [49, 50] has explored them for key-value stores. While Honeycomb's contribution is to enable flexibility in data repartitioning, selecting threshold values is complementary to its design. As such, Honeycomb exposes them as configurable parameters.

4 Honeycomb Implementation

Honeycomb implementation builds on Pocket (§2), and as such, inherits its scalable and fault-tolerant metadata plane, multi-tiered data storage, system-wide storage capacity scaling, analytics execution model, etc. However, Honeycomb implements hierarchical addressing, lease management and efficient data repartitioning (§3) to resolve unique challenges introduced by serverless environments. We now describe Honeycomb interface (§4.1) and implementation (§4.2), focusing on these new features.

4.1 Honeycomb Interface

We describe Honeycomb interface in terms of its user-facing API (Table 1) and internal API (Figure 6).

User-facing API. Honeycomb's user-facing interface (Table 1) is divided along its two core abstractions: *hierarchical addresses* and *data structures*. Jobs add a new address-prefix to their address hierarchy using `createAddrPrefix`, specifying the parent address-prefix, along with optional arguments such as initial capacity. Honeycomb also provides a `createHierarchy` interface to directly generate the complete address hierarchy from the application's execution plan (i.e., DAG), and `flush`/`load` interfaces to persist/load address-prefix data from external storage (*e.g.*, S3). Honeycomb provides three built-in data structures that can be associated with an address-prefix (via `initDataStructure`), and a way to define new data structures using its internal API.

Similar to existing systems [30, 51], data structures also expose a notification interface, so that tasks that consume intermediate data can be notified on data availability. For instance, a task can `subscribe` to write operations on its parent task's data structure, and obtain a `listener` handle. Honeycomb asynchronously notifies the `listener` upon a write to the data structure, which the task can get via `listener.get()`.

```
block = ds.getBlock(op, args) // Get block
block.writeOp(args) // Perform write
data = block.readOp(args) // Perform read
block.deleteOp(args) // Perform delete
```

Figure 6. Honeycomb Internal API. The block interface is used internally in Honeycomb to implement the data structure APIs (§5).

Internal API. The data layout within blocks in Honeycomb is unique to the data structure that owns it. As such, Honeycomb blocks expose a set of data structure *operators* (Figure 6) which uniquely defines how data structure requests are *routed* across their blocks, and how data is *accessed* or *modified*. These operators are used internally within Honeycomb for its built-in data structures (§5) and not exposed to jobs directly.

The `getBlock` operator determines which block an operation request is routed to based on the operation type and operation-specific arguments (*e.g.*, based on key hashes for a KV-store), and returns a handle to the corresponding block. Each Honeycomb block exposes `writeOp`, `readOp` and `deleteOp` operators to facilitate data structure-specific access logic (*e.g.*, `get`, `put` and `delete` for KV-store). Honeycomb executes individual operators *atomically* for consistency across data accesses to a block using sequence numbers, but does not support atomic transactions that span multiple operators.

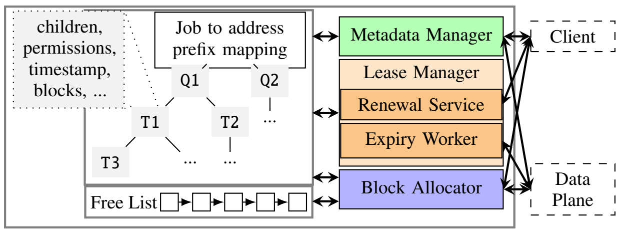


Figure 7. Honeycomb controller. See §4.2 for details.

4.2 System Implementation

Since Honeycomb design builds on Pocket, its high-level design components are also similar, except for one difference: Honeycomb combines the control and metadata planes into a unified control plane. We found this design choice allowed us to significantly simplify interactions between the control and metadata components, without affecting their performance. While this does couple their fault-domains, standard fault-tolerance mechanisms (*e.g.*, the one outlined in [8]) are still applicable to the unified control plane.

Control plane. The Honeycomb controller (Figure 7) maintains two pieces of system-wide state. First, it stores a *free block list*, which lists the set of blocks that have not been

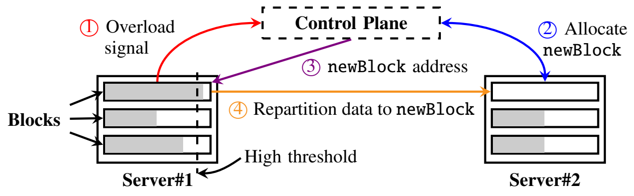


Figure 8. Data repartitioning on scaling up capacity. Scaling down capacity employs a similar approach (§4.2).

allocated to any job yet, along with their corresponding physical server addresses. Second, it stores an address hierarchy per-job, where each node in the hierarchy stores variety of metadata for its address-prefix, including access permissions (for enforcing access control), timestamps (for lease renewal), a block-map (to locate the blocks associated with the address-prefix in the data plane), along with metadata to identify the data structure associated with the address-prefix and how data is partitioned across its blocks. The mapping between jobIDs (which uniquely identify jobs) and their address hierarchies is stored in a hash-table at the controller.

While the block allocator and metadata manager are similar to their counterparts in Pocket, the lease manager implements lifetime management in Honeycomb. It comprises a lease renewal service that listens for renewal requests from jobs and updates the lease renewal timestamp of relevant nodes in its address hierarchy, and a lease expiry worker that periodically traverses all address hierarchies, marking nodes with timestamps older than the associated lease period as expired. Finally, Honeycomb adopts mechanisms from Pocket to facilitate control plane scaling and fault tolerance; we refer the reader to [8] for details.

Data plane. Honeycomb data plane is responsible for two main tasks: providing jobs with efficient, data-structure specific atomic access to data, and repartitioning data across blocks allocated by the control plane during resource scaling. It partitions the resources in a pool of storage servers across fixed sized blocks. Each storage server maintains, for the blocks managed by it, a mapping from unique blockIDs to pointers to raw storage allocated to the blocks, along with two additional metadata: data structure-specific operator implementations as described in §4.1, and a subscription map that maps data structure operations to client handles that have subscribed to receive notifications for that operation.

Data repartitioning for a Honeycomb data structure is implemented as follows: when a block's usage grows above the high threshold, the block sends a signal to the control plane, which, in turn, allocates a new block to the address-prefix and responds to the overloaded block with its location. The overloaded block then repartitions and moves part of its data to the new block (see Figure 8); a similar mechanism is used when the block's usage falls below the low threshold.

Table 2. Honeycomb Data Structure Implementations. See §5 for details.

Data Structure		Operators				
		`writeOp`	`readOp`	`deleteOp`	`getBlock`	`repartition`
Built-in	File (§5.1)	`write`	`read`	-	Route to block based on file offsets.	Not required
Built-in	FIFO Queue (§5.2)	`enqueue`	`dequeue`		`enqueue` to tail, `dequeue` to head block.	Not required
Built-in	KV-Store (§5.3)	`put`	`get`	`delete`	Route to block based on key hash.	Hash-based repartitioning
Custom data structures.						

For applications that require fault tolerance and persistence for their intermediate data, Honeycomb supports chain replication [52] at block granularity, and synchronously persisting data to external stores (*e.g.*, S3) at address-prefix granularity.

5 Programming Models on Honeycomb

We now describe how Honeycomb's built-in data structures (Table 2) enable many distributed programming frameworks atop serverless platforms (§5.1-§5.3).

5.1 Map-Reduce Model

A Map-Reduce (MR) program [31] comprises map functions that process a series of input key-value (KV) pairs to generate intermediate KV pairs, and reduce functions that merge all intermediate values for the same intermediate key. MR frameworks [31, 48, 53] parallelize map and reduce functions across multiple workers. Data exchange between map and reduce workers occurs via a shuffle phase, where intermediate KV pairs are distributed in a way that ensures intermediate values belonging to the same key are routed to the same worker.

MR on Honeycomb executes map/reduce tasks as serverless tasks. A master process launches, tracks progress of, and handles failures for tasks across MR jobs. Honeycomb stores intermediate KV pairs across multiple shuffle files, where shuffle files contain a partitioned subset of KV pairs collected from all map tasks. Since multiple map tasks can write to the same shuffle file, Honeycomb's strong consistency semantics ensures correctness. The master process handles explicit lease renewals. We describe Honeycomb files next.

Honeycomb Files. A Honeycomb file is a collection of blocks, each storing a fixed-sized chunk of the file. The controller stores the mapping between blocks and file offset ranges managed by them at the metadata manager; this mapping is cached at clients accessing the file, and updated whenever the number of blocks allocated to the file is scaled in Honeycomb. The `getBlock` operator forwards requests to different file blocks based on the offset-range for the request. Files support sequential `reads`/`writes` via append-only semantics. For random access, files support `seek` with arbitrary offsets. Honeycomb uses the provided offset to identify the corresponding block, and forwards subsequent `read`/`write` requests to it. Finally, since files are append-only, blocks can only be added to it (not removed), and do not require repartitioning when new blocks are added.

5.2 Dataflow and Streaming Dataflow Models

In the dataflow programming model, programmers provide DAGs to describe an application's communication patterns. DAG vertices correspond to computations, while data channels form directed edges between them. We use Dryad [32] as a reference dataflow execution engine, where channels can be files, shared memory FIFO queues, etc. Dryad runtime schedules DAG vertices across multiple workers based on their dataflow dependencies. A vertex is scheduled when all its input channels are ready: a file channel is ready if all its data items have been written, while a queue is ready if it has any data item. Streaming dataflow [33] employs a similar approach, except channels are continuous event streams.

Dataflow on Honeycomb maps each DAG vertex to a serverless task, while a master process handles vertex scheduling, fault tolerance, and lease renewals for task address-prefixes. We use Honeycomb FIFO queues and files as data channels. Since queue-based channels are considered ready as long as some vertex is writing to it, Honeycomb allows downstream tasks to efficiently detect availability of items produced by upstream tasks via notifications, as described below.

Honeycomb Queues. The FIFO queue in Honeycomb is a continuously growing linked-list of blocks, where each block stores multiple data items, and a pointer to the next block in the list. The queue size can be upper-bounded (in number of items) by specifying a `maxQueueLength`. The controller only stores the head and the tail blocks in the queue's linked list, which the client caches and updates whenever blocks are added/removed. The FIFO queue supports `enqueue`/`dequeue` to add/remove items. The `getBlock` operator routes `enqueue` and `dequeue` operations to the current tail and head blocks in the link-list, respectively. While, blocks can be both added and removed from a FIFO queue, queues do not need subsequent data repartitioning. Finally, the FIFO queue leverages Honeycomb notifications to asynchronously detect when the there is data in the queue to consume, or space in the queue to add more items via subscriptions to `enqueue` and `dequeue`, respectively.

5.3 Piccolo

Piccolo [34] is a data-centric programming model that allows distributed compute machines to share distributed, mutable state. Piccolo kernel functions specify sequential application logic and share state with concurrent kernel functions via a KV interface, while centralized control functions create and coordinate both shared KV stores and kernel function instances. Concurrent updates to the same key in the KV store are resolved using user-defined accumulators.

Piccolo on Honeycomb runs kernel functions across serverless tasks, while control tasks run on a centralized master. The shared state is stored across Honeycomb's KV-store data structures (described below). KV-stores may be created per kernel function, or shared across multiple functions, depending on the application needs. The master periodically renews leases for Honeycomb KV-stores. Like Piccolo, Honeycomb checkpoints KV-stores by flushing them to an external store.

Honeycomb KV-store. The Honeycomb KV-store hashes each key to one of H hash-slots in the range $[0, H\text{-}1]$ (H=1024 by default). The KV-store shards key-value pairs across multiple Honeycomb blocks, such that each block owns one or more hash-slots in this range. Note that a hash-slot is completely contained in a single block. The controller stores the mapping between the blocks and the hash slots managed by them; this metadata is, again, cached at the client and updated during resource scaling. Each block stores key-value pairs that hash to its slots as a hash-table. The KV-store supports typical `get`, `put`, and `delete` operations as implementations of `readOp`, `writeOp` and `deleteOp` operators. The `getBlock` operator routes requests to KV-store blocks based on key-hashes.

Unlike files and queues, data needs to repartitioned for the KV-store when a block is added or removed. On block addition, Honeycomb reassigns half of the overloaded block's hash-slots to the new block, moves the corresponding key-value pairs to it, and updates the block-to-hash-slot mapping at the controller. Similarly, when a block is underloaded, its hash-slots are merged with another block.

6 Evaluation

Honeycomb is implemented in 25K lines of C++, with client libraries in C++, Python and Java (~1K LOC each); this is in addition to the original Pocket code. In this section, we evaluate Honeycomb to demonstrate its benefits (§6.1, §6.2) and to understand the contribution of individual Honeycomb mechanisms to its overall performance (§6.3). Due to space constraints, we present additional results for Honeycomb in the Appendix, including evaluation for additional workloads (Appendix A.1, Honeycomb's controller scalability (Appendix A.2), and sensitivity analysis against various Honeycomb parameters like block size, lease duration and elasticity threshold (Appendix A.3).

Experimental setup. Unless otherwise specified, each intermediate storage system in our experiments is deployed across 10 m4.16xlarge EC2 [54] instances, while serverless applications are deployed across AWS Lambda [54] instances. Since Honeycomb builds on Pocket design, it supports addition of new instances to increase the system-wide capacity. However, our experiments do not evaluate overheads for doing so, since, it is orthogonal to Honeycomb's goals — Honeycomb specifically focuses on multiplexing available system storage capacity for higher utilization, and reduce the need for adding more capacity. Honeycomb employs 128MB blocks, 1s lease duration and 5% (low) and 95% (high) as thresholds for data repartitioning. As mentioned above, an analysis of Honeycomb's sensitivity to these parameters is presented in Appendix A.3.

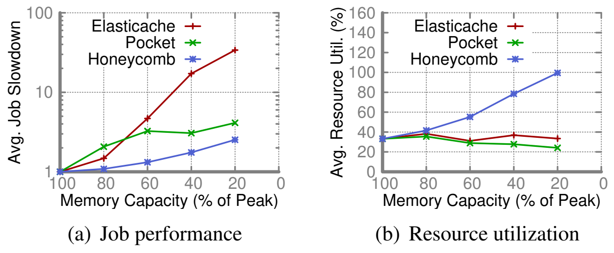


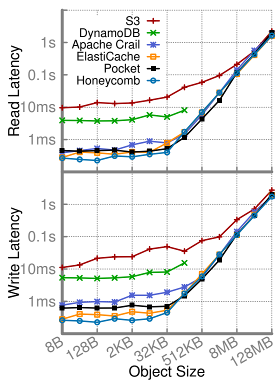
Figure 9. Fine-grained (task-level) elasticity in Honeycomb enables (a) better job performance, and (b) higher resource utilization under constrained capacity. In (a), the slowdown is computed relative to the job completion time with 100% capacity (for this data point, Elasticache performance was ~30% worse than Pocket, and Pocket performance was ~5% worse than Honeycomb). See §6.1 for details.

6.1 Benefits of Honeycomb

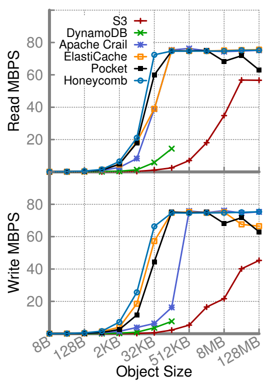
Honeycomb enables fine-grained resource allocation for serverless analytics. We demonstrate the benefits of this approach to job performance and resource utilization for roughly 50,000 jobs across 100 randomly chosen tenants over a representative 5 hour window in the Snowflake workload[5] [20]. We compare Honeycomb (with the MR programming model, §5) against Elasticache [47] and Pocket [8]. Elasticache represents systems that provision resources for *all* jobs. Since Elasticache does not support multiple storage tiers, if available capacity is insufficient, jobs must write their data to external stores like S3 [55]. Pocket, on the other hand, reserves and reclaims resources at *job* granularity; if available capacity is insufficient, Pocket allocates resources on secondary storage (SSD). Note that Pocket's utilization can sometimes be *lower* than Elasticache, since it provisions for the peak of each job *separately*, sacrificing utilization for job-level isolation. Finally, our evaluation for Pocket places its control and metadata services on the same server to ensure a fair comparison with Honeycomb's unified control plane.

Impact of fine-grained elasticity on job performance. We demonstrate this impact by constraining the amount of available capacity at the intermediate store for the Snowflake workload. Figure 9(a) shows the average job slowdown as the

[5]We were unable to evaluate the entire 14 day window with > 2000 tenants due to intractable cost overheads.



(a) Latency



(b) MBPS

Figure 10. Honeycomb performance comparison with existing storage systems (§6.2). Despite providing the additional benefits demonstrated in §6.1, Honeycomb performs as well as or outperforms state-of-the-art storage systems for serverless analytics.

capacity is reduced to a certain percentage of the peak utilization for the workload within the evaluated time window (i.e., across all jobs). With Elasticache, job performance suffers significantly when the intermediate data grows larger than capacity (34× slowdown at 20% capacity), since the data must now be accessed from S3. With Pocket, the data spills to SSD when the allocated capacity at the DRAM-tier (during job registration) is insufficient. While the slowdown is less severe due to its efficient tiered-storage, jobs still experience a $>$ 4.1× slowdown at 20% capacity. Finally, Honeycomb observes much lower job performance degradation with constrained capacity ($<$ 2.5× at 20% capacity). This is because task-level elasticity and lease based reclamation of faster storage capacity allows Honeycomb to efficiently multiplex capacity across multiple jobs at a much finer granularity than Pocket. This, in turn, significantly reduces data spilling over to a slower storage tier in Honeycomb compared to Pocket. We confirm this intuition further by studying the impact of fine-grained elasticity on resource utilization next.

Impact of fine-grained elasticity on resource utilization. Figure 9(b) shows the resource utilization across the compared systems under constrained capacity. While the resource utilization for Elasticache and Pocket either decreases or remains the same as the system capacity is reduced, resource utilization *improves* for Honeycomb. This is because Pocket and Elasticache provision capacity (at job or coarser granularity), and the unused capacity is wasted, regardless of the total system capacity. In contrast, Honeycomb is able to better multiplex the available capacity across various jobs owing to its fine-grained elasticity and lease-based reclamation of unused capacity. By making better use of available capacity, Honeycomb ensures that a much smaller fraction of data spills over to SSD, attributing to its better performance in Figure 9(a).

6.2 Performance Benchmarks for Six Systems

We now compare Honeycomb performance (using its KV-Store data structure) against five state-of-the-art systems commonly used for intermediate data storage in serverless analytics: S3, DynamoDB, Elasticache, Apache Crail and Pocket. Since only a subset of the compared systems support request pipelining, we disable pipelining for all of them.

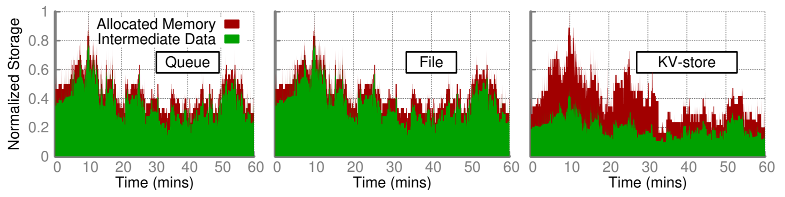
To measure latency and throughput for the above systems, we profiled synchronous operations issued from an AWS Lambda instance using a single-threaded client. Figure 10 shows that in-memory data stores like Elasticache, Pocket and Apache Crail achieve low-latency (sub-millisecond) and high-throughput. In contrast, persistent data stores like S3 and DynamoDB observe significantly higher latencies and lower throughput; note that DynamoDB only supports objects up to 128KB. Honeycomb matches the performance achieved by state-of-the-art in-memory data stores, while additionally providing the benefits outlined in §6.1.

6.3 Understanding Honeycomb Benefits

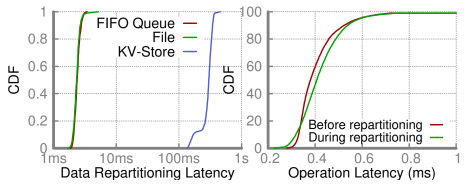
Figure 9 already shows how fine-grained elasticity in Honeycomb allows it to achieve performance and resource utilization gains over the compared state-of-the-art systems. As noted earlier, this fine-grained elasticity is enabled by hierarchical virtual addressing combined with with flexible data lifetime management and data repartitioning in Honeycomb. In this section, we evaluate their impact in isolation.

Fine-grained elasticity via data lifetime management. Unlike traditional storage systems, Honeycomb's lease-based data lifetime management allows it to reclaim unused resources from jobs, and potentially assign them to other jobs that might need them. Coupled with fine-grained resource allocations and efficient data repartitioning, this enables fine-grained elasticity for serverless jobs running on Honeycomb. To understand how, we evaluate storage allocation across different Honeycomb data structures (Figure 11(a)) when subjected to Snowflake workload from Figure 1.

FIFO queue and file observe seamless elasticity in allocated resources as intermediate data is written to them since they do not require repartitioning. The allocated capacity exceeds the intermediate data size for the data structures by only a small amount; this accounts for the additional metadata stored at each of the blocks (*e.g.*, object metadata for the items enqueued in the FIFO queue, etc.), along with unused space within the head/tail blocks. For the KV-store, the inserted keys were sampled from a Zipf distribution over the keyspace since the Snowflake dataset does not provide access patterns. Due to the skew, a few Honeycomb blocks receive most of the key-value pairs, and repeatedly split across newly allocated blocks when their used capacity grows too high. The allocated capacity is therefore higher than the dataset size, since the used capacity is low for most blocks owing to the Zipf key sampling; this corresponds to the worst-case for the KV-Store.



(a) Efficient lifetime management



(b) Efficient data repartitioning

Figure 11. Honeycomb data lifetime-management and data repartitioning. (a) Honeycomb enables fine-grained elasticity via lease-based lifetime management for its built-in data structures, FIFO Queue (left), File (center) and KV-store (right), reclaiming resources from tasks as soon as their leases expire. (b) Honeycomb facilitates efficient data repartitioning for its data structures when their allocation is scaled up, with repartitioning for a single block completing in 2-500ms (left). Moreover, Honeycomb latency for 100KB `get`s is minimally impacted during KV-store repartitioning. Note: plots in (a), (b) share a common y-axis; x-axis for (c, left) is in log-scale.

However, Honeycomb's lease mechanism reclaims resources allocated to the data structures soon after their utility is over, ensuring that the overheads are short-lived.

Efficient elastic scaling via flexible data repartitioning. A key contributor for the fine-grained resource elasticity achieved by Honeycomb is its flexible but efficient data repartitioning approach. Figure 11(b) shows the CDF of data repartitioning latency per block across the three data structures, when subjected to the Snowflake workload from above. The data repartitioning latency shown here corresponds to the total time taken from the detection of an overloaded/underloaded block to the end of data repartitioning. The storage server takes ~1-1.5ms to connect to the controller, and two round-trips (100-200μs in EC2) to trigger allocation/reclamation of data blocks and update for partitioning metadata at the controller. Unlike FIFO Queue and File, KV-Store also requires repartitioning data across blocks. However, since repartitioning a single block only requires moving only half the block capacity (~64MB), Honeycomb is able to repartition the data in a few hundred milliseconds over 10Gbps links. As such, Honeycomb repartitions data within a block for its built-in data structures with very low latency (2-500ms).

Finally, we also note that Honeycomb does not block data structure operations during data repartitioning. In fact, these operations are minimally impacted during this period: Figure 11(b) shows that the CDF for 100KB `get` operations on the KV-Store prior to and during scaling are almost identical.

7 Related Work

We discussed intermediate storage systems for serverless analytics in §1 and §6.2; we now discuss other related systems. Pocket [8] has shown how existing designs for in-memory key-value stores [27, 28, 30, 56–60], distributed memory systems [61–64], remote/disaggregated memory systems [29, 65, 66], etc.), and storage systems with flexible interfaces [67–74] can be extended to facilitate three key goals for intermediate data storage in serverless analytics: low-latency and high-throughput, storage resource sharing and resource elasticity. Honeycomb strives for complementary goals of resolving specific challenges arising from adapting virtual memory-based allocation to serverless environments (§3) to achieve task-level elasticity, isolation and lifetime management. However, Honeycomb is flexible enough to be implemented atop most such systems to achieve these goals.

Our evaluation employs publicly released datasets from Snowflake's production clusters [20]. Snowflake itself neither performs task-level resource allocation, nor does it provide isolation across tasks. In fact, Snowflake's ephemeral storage is not shared across tenants, or even tasks running on separate compute nodes. Due to the above reasons, Snowflake does not need to perform data lifetime management either. In contrast, Honeycomb is designed for multi-tenant environments, and provides lifetime management for serverless analytics.

Several other recent storage systems have also explored fine-grained resource sharing. Pisces [37] provides per-tenant *performance* isolation in a multi-tenant cloud storage system, but does not cater to sharing *storage capacity* across tenants or applications. Memshare [38] facilitates memory sharing across multiple tenants, but operates under a key-value cache setting, i.e., when sharing memory across applications, it evicts KV pairs that contribute less to overall system hit-rate. In contrast, Honeycomb focuses on more general data models that support fine-grained memory elasticity via efficient data repartitioning, and allows applications to control over what data resides in memory and for how long via leasing.

8 Conclusion

We have presented Honeycomb, an ephemeral storage system that matches the instantaneous capacity demands for serverless analytics jobs. Honeycomb resolves unique challenges introduced by serverless environments using a combination of hierarchical addressing to enable fine-grained elastic scaling of resources *with* task-level isolation, efficient data lifetime management via leasing, and flexible data repartitioning. Honeycomb supports rich data models that enable several powerful distributed programming frameworks on serverless platforms, and currently runs several them on AWS Lambda. Our evaluation shows that Honeycomb improves job execution time by $1.6 - 2.5\times$ for production workloads.

References

[1] AWS Lamda. `https://aws.amazon.com/lambda/`.

[2] Azure Functions. `https://azure.microsoft.com/en-us/services/functions`.

[3] Google Cloud Functions. `https://cloud.google.com/functions`.

[4] State of the Serverless Community Survey Results. `https://serverless.com/blog/state-of-serverless-community`.

[5] 2018 Serverless Community Survey: huge growth in serverless usage. `https://bit.ly/2Mu5TCR`.

[6] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud function services. In *ACM International Conference on Management of Data (SIGMOD'20)*.

[7] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, pages 193–206.

[8] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, 2018.

[9] Youngbin Kim and Jimmy Lin. Serverless data analytics with Flint. In *IEEE International Conference on Cloud Computing (CLOUD '18)*, pages 451–455. IEEE.

[10] Qubole Announces Apache Spark on AWS Lambda. `https://www.qubole.com/blog/spark-on-aws-lambda`.

[11] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *ACM Symposium on Cloud Computing (SoCC'19)*.

[12] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, pages 363–376.

[13] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 445–451. ACM, 2017.

[14] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. numpywren: serverless linear algebra. *arXiv preprint arXiv:1810.09679*, 2018.

[15] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *USENIX Annual Technical Conference (USENIX ATC'19)*.

[16] Amazon. Amazon Athena. `https://aws.amazon.com/athena`.

[17] Amazon. Amazon Aurora Serverless. `https://aws.amazon.com/rds/aurora/serverless`.

[18] Azure. Azure SQL Data Warehouse. `https://azure.microsoft.com/en-us/services/sql-data-warehouse`.

[19] Vikram Sreekanti, Chenggang Wu Xiayue Charles Lin, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592*, 2020.

[20] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *USENIX Networked Systems Design and Implementation (USENIX NSDI'20)*.

[21] Kshiteej Mahajan, Mosharaf Chowdhury, Aditya Akella, and Shuchi Chawla. Dynamic query re-planning using QOOP. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[22] Kai Li. Ivy: A shared virtual memory system for parallel computing. *ICPP (2)*, 88:94, 1988.

[23] Brett Fleisch and Gerald Popek. *Mirage: A coherent distributed shared memory design*, volume 23. ACM, 1989.

[24] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):109–133, 1988.

[25] Partha Dasgupta, Richard J LeBlanc, Mustaque Ahamad, and Umakishore Ramachandran. The clouds distributed operating system. *Computer*, 24(11):34–44, 1991.

[26] John B Carter, Dilip Khandekar, and Linus Kamb. Distributed shared memory: Where we are and where we should be headed. In *Proceedings 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 119–122. IEEE, 1995.

[27] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*.

[28] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[29] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*.

[30] Redis. `http://www.redis.io`.

[31] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[32] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, volume 41, pages 59–72. ACM, 2007.

[33] Wei Lin, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. Streamscope: continuous reliable distributed processing of big data streams. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*, pages 439–453.

[34] Russell Power and Jinyang Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 293–306, 2010.

[35] Apache Crail. `http://crail.apache.org`.

[36] Hadoop Distributed File System. `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`.

[37] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 349–362, 2012.

[38] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *USENIX Annual Technical Conference (USENIX ATC'17)*.

[39] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ACM Symposium on Cloud Computing (SoCC '12)*.

[40] TPC-DS. `http://www.tpc.org/tpcds/`.

[41] Paul R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management (IWMM)*, 1992.

[42] David I Bevan. Distributed garbage collection using reference counting. In *International Conference on Parallel Architectures and Languages Europe*. Springer, 1987.

[43] K G Cassidy. Feasibility of automatic storage reclamation with concurrent program execution in a lisp environment. master's thesis. 1985.

[44] Cary Gray and David Cheriton. *Leases: An efficient fault-tolerant mechanism for distributed file cache consistency*, volume 23. ACM, 1989.

[45] Mike Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 335–350, 2006.

[46] R. Droms. RFC 2131: Dynamic Host Configuration Protocol. `https://www.ietf.org/rfc/rfc2131.txt`, 1997.

[47] Amazon ElastiCache. `https://aws.amazon.com/elasticache`.

[48] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[49] MongoDB: Sharded Cluster Balancer. `https://docs.mongodb.com/manual/core/sharding-balancer-administration/#sharding-migration-thresholds`.

[50] Ceph: Dynamic Bucket Index Resharding. `https://docs.ceph.com/docs/mimic/radosgw/dynamicresharding/`.

[51] Amazon Simple Notification Service (SNS). `https://aws.amazon.com/sns`.

[52] Robbert van Renesse and Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 7–7, 2004.

[53] Apache Hadoop. `https://hadoop.apache.org/`.

[54] Amazon EC2. `https://aws.amazon.com/ec2/`.

[55] Amazon S3. `https://aws.amazon.com/s3`.

[56] MemCached. `http://www.memcached.org`.

[57] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*.

[58] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. Autoscaling tiered cloud storage in anna. *Proceedings of the VLDB Endowment*, 12(6):624–638, 2019.

[59] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: Enabling Queries on Compressed Data. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*.

[60] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Blowfish: Dynamic storage-performance tradeoff in data stores. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*, pages 485–500.

[61] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of munin. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 152–164, 1991.

[62] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[63] Bill Nitzberg and Virginia Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, 24(8):52–60, 1991.

[64] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *USENIX Winter Technical Conference (WTEC)*, 1994.

[65] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with Infiniswap. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, pages 649–667.

[66] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *USENIX Annual Technical Conference (USENIX ATC'18)*.

[67] Postgres: User defined Functions. `https://www.postgresql.org/docs/8.0/xfunc.html`.

[68] Oracle: User defined Functions. `https://docs.oracle.com/cd/B19306_01/server.102/b14200/functions231.htm`.

[69] SQL Server: User defined Functions. `https://docs.microsoft.com/en-us/sql/relational-databases/user-defined-functions/user-defined-functions`.

[70] Postgres: Stored Procedures. `https://www.postgresql.org/docs/11/sql-createprocedure.html`.

[71] Oracle: Stored Procedures. `https://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm`.

[72] SQL Server: Stored Procedures. `https://docs.microsoft.com/en-us/sql/relational-databases/stored-procedures/create-a-stored-procedure?view=sql-server-2017`.

[73] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Symposium on Operating Systems Design and Implementation (OSDI'04)*.

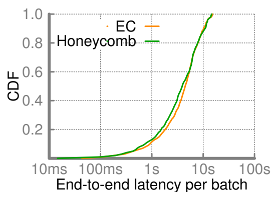
[74] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *ACM Symposium on Operating Systems Principles (SOSP '07)*.

[75] Wikipedia Dataset. `https://en.wikipedia.org/wiki/Wikipedia:Database_download`.

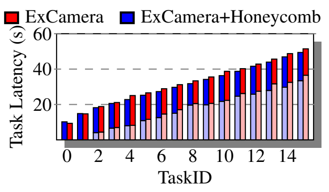
[76] Ton Roosendaal. Sintel. In *ACM SIGGRAPH Computer Animation Festival*, 2011.

A Appendix: Additional Evaluation

We now present additional results for Honeycomb, including: evaluation for analytics applications other than those that employ the MapReduce model (Appendix A.1), an evaluation of its control plane (Appendix A.2), and sensitivity analysis for various system parameters (Appendix A.3).



(a) Streaming word-count



(b) Encoding videos on ExCamera

Figure 12. Honeycomb evaluation for additional applications. See Appendix A.1 for details.

A.1 Additional Applications on Honeycomb

The Snowflake workload evaluated in §6 shows Honeycomb performance a SQL application (MapReduce model, §5.1). We now evaluate Honeycomb for two additional serverless applications, which make use of other models.

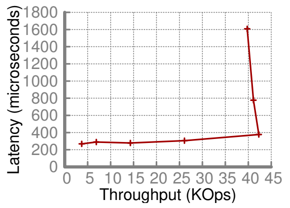
Streaming word-count (DataFlow+Piccolo models, §5.2, §5.3). Our stateful streaming word-count workload comprises 50 partition tasks that split input sentences randomly sampled from the Wikipedia dataset [75] into words and partition them based on their string hashes, and 50 count tasks that collect words within a partition and compute their counts. The job employs queues as data channels (Dataflow model) and stores word counts in a KV-store (Piccolo model). We compare Honeycomb performance with Elasticache (both hosted on 5 m4.16xlarge EC2 instances), since both support queue and KV-store data models. Figure 12(a) shows the CDF of end-to-end batch latency (batch size = 64 sentences). Despite providing fine-grained elastic resource scaling, isolation and lifetime management, Honeycomb can match the performance of an over-provisioned Elasticache cluster.

Encoding videos with ExCamera (DataFlow model, §5.2). ExCamera [12] is a video processing framework that facilitates fine-grained parallelism for video encoding on AWS Lambda. ExCamera performs encoding using serverless tasks, while state exchange between them proceeds via a dedicated rendezvous server that forwards messages from one task to another. We compare the rendezvous server approach with state exchange via Honeycomb queues in ExCamera, both being hosted on a single m4.16xlarge instance. Figure 12(b) shows ExCamera task latencies for uncompressed 4k raw frames from [76]. Despite its benefits (§6.3), Honeycomb not only matches ExCamera performance, but reduces task wait times (lighter shade) by 10-20% via queue notifications.

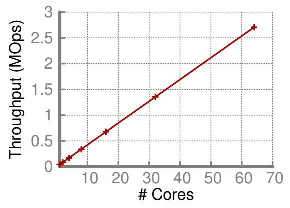
A.2 Controller Performance

Honeycomb adds several components at the controller compared to Pocket, including all of metadata management, lease management and handling requests for data repartitioning. As such, we expect its performance to be lower than Pocket's metadata server. We deem this to be acceptable as long as it can still handle control plane request rates typically seen for real world workload, *e.g.*, a peak of a few hundred requests per second, including lease renewal requests, for all of our evaluated workloads and those evaluated in [8].

Figure 13(a) shows the throughput-vs-latency curve for Honeycomb controller operations on a single CPU core of an m4.16xlarge EC2 instance. The controller throughput saturates at roughly 42 KOps, with a latency of 370us. While this throughput is lower than Pocket (~ 90KOps per core), it is more than sufficient to handle control plane load for real-world workloads. In addition, the throughput scales almost linearly with the number of cores, since each core can handle requests independent of other cores for a distinct subset of virtual address hierarchies (Figure 13(b)). Moreover, the Honeycomb control plane readily scales to multiple servers by partitioning the set of virtual address hierarchies across them.



(a) Controller throughput vs. latency on a single CPU core.



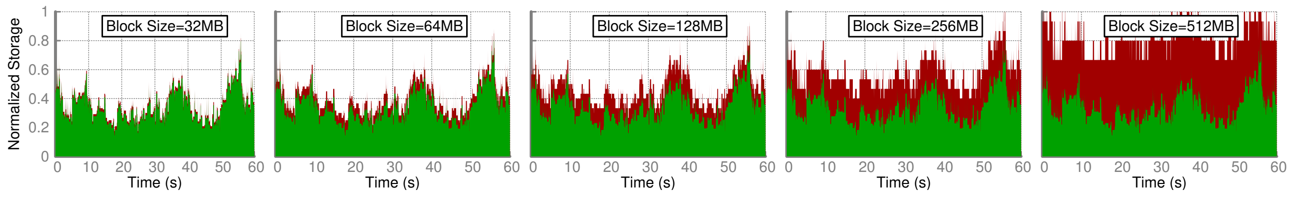
(b) Controller throughput scaling with multiple cores.

Figure 13. Honeycomb controller performance. Details in Appendix A.2.

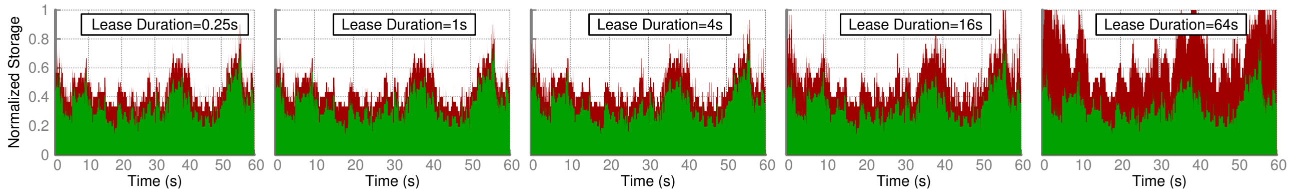
A.3 Sensitivity Analysis

We now perform sensitivity analysis for various system parameters in Honeycomb, including block size (§3.1), lease duration (§3.2) and thresholds for data repartitioning (§3.3). We use files as our underlying data structure, and use the Snowflake workload from Figure 1. These results can be contrasted directly with Figure 11(a) (center), which corresponds to our default system parameters (128MB blocks, 1s lease duration and 95% of block occupancy as repartition threshold). For each parameter that we vary, the other to remain fixed at their default values.

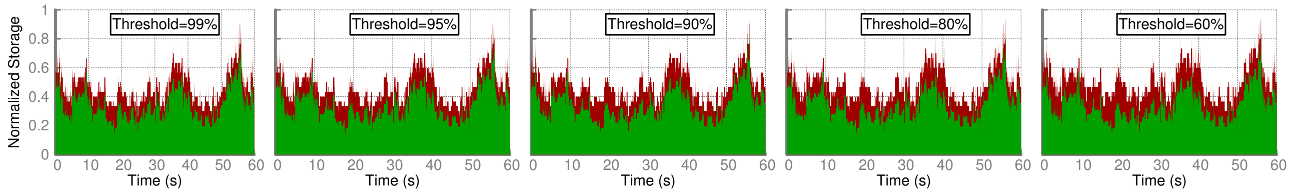
Block size (Figure 14(a)). As discussed in §3.1, the block size in Honeycomb exposes a tradeoff between the amount of metadata that needs to be stored at the control plane, and resource utilization. This is confirmed in Figure 14(a), where increasing the block size from 32MB to 512MB increases the disparity between allocated and used capacity, and therefore



(a) Sensitivity analysis for block size



(b) Sensitivity analysis for lease duration



(c) Sensitivity analysis for (high) repartition threshold

Figure 14. Honeycomb sensitivity analysis for (a) block size (b) lease duration and (c) repartition threshold for the file data structure. Green area corresponds to used capacity, while red area corresponds to allocated capacity under Honeycomb. See Appendix A.3 for details.

decreases the resource utilization. The default block size in Honeycomb is set to 128MB for two main reasons: (1) it allows high enough utilization with low enough metadata overhead (a few megabytes for even thousands of gigabytes of application data), and (2) it is the default block size used in existing data analytics platforms; as such, 128MB blocks ensure seamless compatibility with such frameworks.

Lease duration (Figure 14(b)). As shown in Figure 14(b), lease duration in Honeycomb controls resource utilization over time. As we increase lease durations from 0.25 seconds to 64 seconds, resource utilization increases since Honeycomb does not reclaim (potentially unused) resource resources from jobs until their leases expire. At the same time, if we keep lease duration too low, applications would renew leases too often, resulting in higher traffic to the Honeycomb controller. We find a lease duration of 1s to be a sweet spot, ensuring high enough resource utilization, while ensuring the number of lease requests for even thousands of concurrent applications is only a few thousand requests per second — well within Honeycomb controller's limits on a single CPU core.

Repartition threshold (Figure 14(c)). Finally, Figure 14(c) shows the impact of (high) repartition threshold on resource utilization. As expected, lowering the repartition threshold leads to poor utilization, since it triggers pre-mature allocation of new blocks to most files in our evaluated workload. However, since the size of the block (128MB) is much smaller than the amount of data written to each file in the workload (often several gigabytes), this overhead is relatively small when compared to effect of other parameters. However, a larger value of high repartitioning threshold results in more frequent block allocation requests to the controller; we find that our default value of 95% provides a reasonable compromise between resource utilization and number of control plane requests.