Building An Elastic Query Engine on Disaggregated Storage

Midhul Vuppalapati
Cornell University

Justin Miron
Cornell University

Rachit Agarwal
Cornell University

Dan Truong
Snowflake Computing

Ashish Motivala
Snowflake Computing

Thierry Cruanes
Snowflake Computing

Abstract

We present operational experience running *Snowflake*, a cloud-based data warehousing system with SQL support similar to state-of-the-art databases. Snowflake design is motivated by three goals: (1) compute and storage elasticity; (2) support for multi-tenancy; and, (3) high performance. Over the last few years, Snowflake has grown to serve thousands of customers executing millions of queries on petabytes of data every day.

We discuss Snowflake design with a particular focus on ephemeral storage system design, query scheduling, elasticity and efficiently supporting multi-tenancy. Using statistics collected during execution of 70 million queries over a 14 day period, our study highlights how recent changes in cloud infrastructure have altered the many assumptions that guided the design and optimization of Snowflake, and outlines several interesting avenues of future research.

1 Introduction

Shared-nothing architectures have been the foundation of traditional query execution engines and data warehousing systems. In such architectures, *persistent* data (*e.g.*, customer data stored as tables) is partitioned across a set of compute nodes, each of which is responsible only for its local data. Such shared-nothing architectures have enabled query execution engines that scale well, provide cross-job isolation and good data locality resulting in high performance for a variety of workloads. However, these benefits come at the cost of several major disadvantages:

- **Hardware-workload mismatch:** Shared-nothing architectures make it hard to strike a perfect balance between CPU, memory, storage and bandwidth resources provided by compute nodes, and those required by workloads. For instance, a node configuration that is ideal for bandwidth-intensive compute-light bulk loading may be a poor fit for compute-extensive bandwidth-light complex queries. Many customers, however, want to run a mix of queries without setting up a separate cluster for each query type. Thus, to meet performance goals, resources usually have to be over-provisioned; this results in resource underutilization on an average and in higher operational costs.
- **Lack of Elasticity:** Even if one could match the hardware resources at compute nodes with workload demands, static parallelism and data partitioning inherent to (inelastic) shared-nothing architectures constrain adaptation to data skew and time-varying workloads. For instance, queries run by our customers have extremely skewed intermediate data sizes that vary over five orders of magnitude (§4), and have CPU requirements that change by as much as an order of magnitude within the same hour (§7). Moreover, shared-nothing architectures do not admit efficient elasticity; the usual approach of adding/removing nodes to elastically scale resources requires large amounts of data to be reshuffled. This not only increases network bandwidth requirements but also results in significant performance degradation since the set of nodes participating in data reshuffling are also responsible for query processing.

Traditional data warehousing systems were designed to operate on recurring queries on data with predictable volume and rate, *e.g.*, data coming from within the organization: transactional systems, enterprise resource planning application, customer relationship management applications, etc. The situation has changed significantly. Today, an increasingly large fraction of data comes from less controllable, external sources (*e.g.*, application logs, social media, web applications, mobile systems, etc.) resulting in ad-hoc, time-varying, and unpredictable query workloads. For such workloads, shared-nothing architectures beget high cost, inflexibility, poor performance and inefficiency, which hurts production applications and cluster deployments.

To overcome these limitations, we designed Snowflake — an elastic, transactional query execution engine with SQL support comparable to state-of-the-art databases. The key insight in Snowflake design is that the aforementioned limitations of shared-nothing architectures are rooted in tight coupling of compute and storage, and the solution is to decouple the

two! Snowflake thus disaggregates compute from persistent storage; customer data is stored in a persistent data store (*e.g.*, Amazon S3 [5], Azure Blob Storage [8], etc.) that provides high availability and on-demand elasticity. Compute elasticity is achieved using a pool of pre-warmed nodes, that can be assigned to customers on an on-demand basis.

Snowflake system design uses two key ideas (§2). First, it uses a custom-designed storage system for management and exchange of ephemeral/intermediate data that is exchanged between compute nodes during query execution (*e.g.*, tables exchanged during joins). Such an ephemeral storage system was necessary because existing persistent data stores [5, 8] have two main limitations: (1) they fall short of providing the necessary latency and throughput performance to avoid compute tasks being blocks on exchange of intermediate data; and (2) they provide stronger availability and durability semantics than what is needed for intermediate data. Second, Snowflake uses its ephemeral storage system not only for intermediate data, but also as a write-through "cache" for persistent data. Combined with a custom-designed query scheduling mechanism for disaggregated storage, Snowflake is able to reduce the additional network load caused by compute-storage disaggregation as well as alleviate the performance overheads of reduced data locality.

Snowflake system has now been active for several years and today, serves thousands of customers executing millions of queries over petabytes of data, on a daily basis. This paper describes Snowflake system design, with a particular focus on ephemeral storage system design, query scheduling, elasticity and efficiently supporting multi-tenancy. We also use statistics collected during execution of ∼70 million queries over a period of 14 contiguous days in February 2018 to present a detailed study of network, compute and storage characteristics in Snowflake. Our key findings are:

- Customers submit a wide variety of query types; for example, read-only queries, write-only queries and read-write queries, each of which contribute to ∼28%, ∼13% and ∼59%, respectively, of all customer queries.
- Intermediate data sizes can vary over multiple orders of magnitude across queries, with some queries exchanging hundreds of gigabytes or even terabytes of intermediate data. The amount of intermediate data generated by a query has little or no correlation with the amount of persistent data read by the query or the execution time of the query.
- Even with a small amount of local storage capacity, skewed access distributions and temporal access patterns common in data warehouses enable reasonably high average cache hit rates (60-80% depending on the type of query) for persistent data accesses.
- Several of our customers exploit our support for elasticity (for ∼20% of the clusters). For cases where customers do request elastic scaling of resources, the number of compute nodes in their cluster can change by as much as two orders of magnitude during the lifetime of the cluster.
- While the peak resource utilization can be high, the average resource utilization is usually low. We observe average CPU, Memory, Network Tx and Network Rx utilizations of ∼51%, ∼19%, ∼11%, ∼32%, respectively.

Our study both corroborates exciting ongoing research directions in the community, as well as highlights several interesting venues for future research:

- **Decoupling of compute and ephemeral storage:** Snowflake decouples compute from persistent storage to achieve elasticity. However, currently, compute and ephemeral storage are still tightly coupled. As we show in §4, the ratio of compute capacity and ephemeral storage capacity in our production clusters can vary by several orders of magnitude, leading to either under utilization of CPU or thrashing of ephemeral storage, for ad-hoc query processing workloads. To that end, recent academic work on decoupling compute from ephemeral storage [22, 27] is of extreme interest. However, more work is needed in ephemeral storage system design, especially in terms of providing fine-grained elasticity, multi-tenancy, and cross-query isolation (§4, §7).
- **Deep storage hierarchy:** Snowflake ephemeral storage system, similar to recent work on compute-storage disaggregation [14, 15], uses caching of frequently read persistent data to both reduce the network traffic and to improve data locality. However, existing mechanisms for improving caching and data locality were designed for two-tier storage systems (memory as the main tier and HDD/SSD as the second tier). As we discuss in §4, the storage hierarchy in our production clusters is getting increasingly deeper, and new mechanisms are needed that can efficiently exploit the emerging deep storage hierarchy.
- **Pricing at sub-second timescales:** Snowflake achieves compute elasticity at fine-grained timescales by serving customers using a pool of pre-warmed nodes. This was cost-efficient with cloud pricing at hourly granularity. However, most cloud providers have recently transitioned to sub-second pricing [6], leading to new technical challenges in efficiently achieving resource elasticity and resource sharing across multiple tenants. Resolving these challenges may require design decisions and tradeoffs that may be different from those in Snowflake's current design (§7).

This paper focuses on Snowflake system architecture along with compute, storage and network characteristics observed in our production clusters. Accordingly, we focus on details that are necessary to make the paper self-contained (§2). For details on Snowflake query planning, optimization, concurrency control mechanisms, etc., please refer to [12]. To aid future research and studies, we are releasing an anonymized

version of the dataset used in this paper; this dataset comprising statistics collected per-query for ∼70 million queries. The dataset is available publicly along with documentation and scripts to reproduce all results in this paper at https://github.com/resource-disaggregation/snowset.

Our study has an important caveat. It focuses on a specific system (Snowflake), a specific workload (SQL queries), and a specific cloud infrastructure (S3). While our system is large-scale, has thousands of customers executing millions of queries, and runs on top of one of the most prominent infrastructures, it is nevertheless limited. We leave it to future work an evaluation of whether our study and observations generalize to other systems, workloads and infrastructures. However, we are hopeful that just like prior workload studies on network traffic characteristics [9] and cloud workloads [28] (each of which also focused on a specific system implementation running a specific workload on a specific infrastructure) fueled and aided research in the past, our study and publicly released data will be useful for the community.

2 Design Overview

We provide an overview of Snowflake design. Snowflake treats persistent and intermediate data differently; we describe these in §2.1, followed by a high-level overview of Snowflake architecture (§2.2) and query execution process (§2.3).

2.1 Persistent and Intermediate data

Like most query execution engines and data warehousing systems, Snowflake has three forms of application state:

- *Persistent data* is customer data stored as tables in the database. Each table may be read by many queries, over time or even concurrently. These tables are thus long-lived and require strong durability and availability guarantees.
- *Intermediate data* is generated by query operators (*e.g.*, joins) and is usually consumed by nodes participating in executing that query. Intermediate data is thus short-lived. Moreover, to avoid nodes being blocked on intermediate data access, low-latency high-throughput access to intermediate data is preferred over strong durability guarantees. Indeed, in case of failures happening during the (short) lifetime of intermediate data, one can simply rerun the part of the query that produced it.
- *Metadata* such as object catalogs, mapping from database tables to corresponding files in persistent storage, statistics, transaction logs, locks, etc.

This paper primarily focuses on persistent and intermediate data, as the volume of metadata is typically relatively small and does not introduce interesting systems challenges.

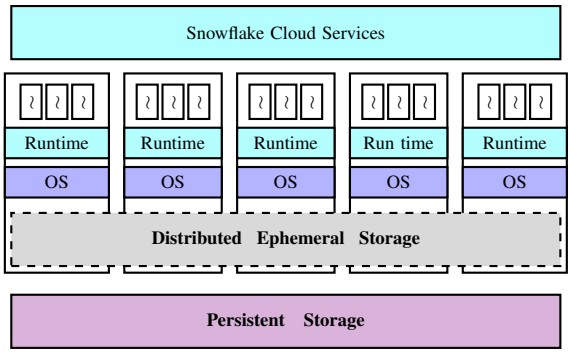

Figure 1: **Snowflake (Virtual) Warehouse Architecture (§2.2).**

2.2 End-to-end System Architecture

Figure 1 shows the high-level architecture for Snowflake. It has four main components — a centralized service for orchestrating end-to-end query execution, a compute layer, a distributed ephemeral storage system and a persistent data store. We describe each of these below[1].

Centralized Control via Cloud Services. All Snowflake customers interact with and submit queries to a centralized layer called Cloud Services (CS) [12]. This layer is responsible for access control, query optimization and planning, scheduling, transaction management, concurrency control, etc. CS is designed and implemented as a multi-tenant and long-lived service with sufficient replication for high availability and scalability. Thus, failure of individual service nodes does not cause loss of state or availability, though some of the queries may fail and be re-executed transparently.

Elastic Compute via Virtual Warehouse abstraction. Customers are given access to computational resources in Snowflake through the abstraction of a *Virtual Warehouse* (VW). Each VW is essentially a set of AWS EC2 instances on top which customer queries execute in a distributed fashion. Customers pay for compute-time based on the VW size. Each VW can be elastically scaled on an on-demand basis upon customer request. To support elasticity at fine-grained timescales (*e.g.*, tens of seconds), Snowflake maintains a pool of pre-warmed EC2 instances; upon receiving a request, we simply add/remove EC2 instances to/from that VW (in case of addition, we are able to support most requests directly from our pool of pre-warmed instances thus avoiding instance startup time). Each VW may run multiple concurrent queries. In fact, many of our customers run multiple VWs (*e.g.*, one for data ingestion, and one for executing OLAP queries).

Elastic Local Ephemeral Storage. Intermediate data has different performance requirements compared to persistent data (§2.1). Unfortunately, existing persistent data stores do not meet these requirements (*e.g.*, S3 does not provide the desired low-latency and high-throughput properties needed

[1]This paper describes design and implementation of Snowflake using Amazon Web Services as an example infrastructure; however, Snowflake runs on Microsoft Azure and Google Cloud Platform as well.

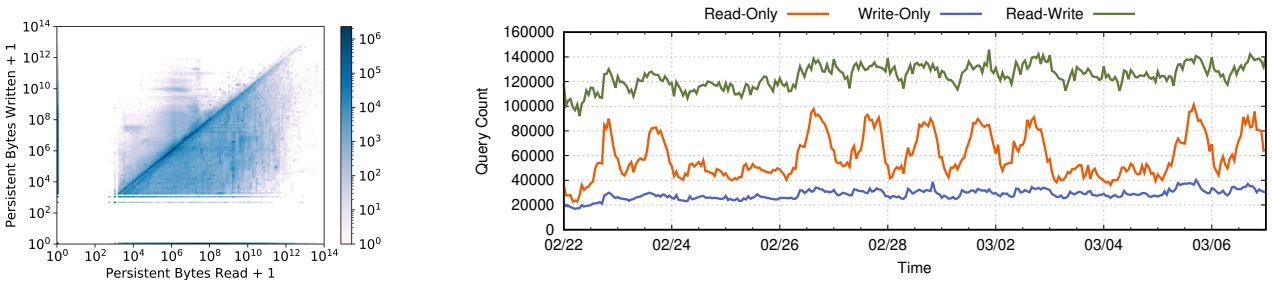

Figure 2: **Persistent data read/write, and submission time characteristics of queries in our dataset. (left)** Scatter plot with each point representing a query based on the total number of persistent data bytes read and written by the query. The density of points is concentrated along three regions: (1) read-only queries along x-axis; (2)write-only queries along y-axis; and, (3) read-write queries along the middle region. **(right)** for each query class, the number of queries submitted at different times of day over the 14 day period, binned on an hourly basis. Read-only queries have significantly higher variation in load compared to the other query classes, with spikes during daytime hours on weekdays.

for intermediate data to ensure minimal blocking of compute nodes); hence, we built a distributed ephemeral storage system custom-designed to meet the requirements of intermediate data in our system. The system is co-located with compute nodes in VWs, and is explicitly designed to automatically scale as nodes are added or removed. We provide more details in §4 and §6, but note here that as nodes are added and removed, our ephemeral storage system does not require data repartitioning or reshuffling (thus alleviating one of the core limitations of shared-nothing architectures). Each VW runs its own independent distributed ephemeral storage system which is used only by queries running on that particular VW.

Elastic Remote Persistent Storage. Snowflake stores all its persistent data in a remote, disaggregated, persistent data store. We store persistent data in S3 despite the relatively modest latency and throughput performance because of S3's elasticity, high availability and durability properties. S3 supports storing immutable files — files can only be overwritten in full and do not even allow append operations. However, S3 supports read requests for parts of a file. To store tables in S3, Snowflake partitions them horizontally into large, immutable files that are equivalent to blocks in traditional database systems [12]. Within each file, the values of each individual attribute or column are grouped together and compressed, as in PAX [2]. Each file has a header that stores offset of each column within the file, enabling us to use the partial read functionality of S3 to only read columns that are needed for query execution.

All VWs belonging to the same customer have access to the same shared tables via remote persistent store, and hence do not need to physically copy data from one VW to another.

2.3 End-to-end query execution

Query execution begins with customers submitting their query text to CS for execution on a specific customer VW. At this stage, CS performs query parsing, query planning and optimization, producing a set of tasks that need to be executed. It then schedules these tasks on compute nodes of the VW; each task may perform read/write operations on both ephemeral storage system and remote persistent data store. We describe the scheduling and query execution mechanisms in Snowflake in §5. CS continually tracks the progress of each query, collects performance counters, and upon detecting a node failure, reschedules the query on compute nodes within the VW. Once the query is executed, the corresponding result is returned back to the CS and eventually to the customer.

3 Dataset

Snowflake collects statistics at each layer of the system — CS collects and stores information for each individual VW (size over time, instance types, failure statistics, etc.), performance counters for individual queries, time spent in different phases of query execution, etc. Each node collects statistics for ephemeral and persistent store accesses, resource (CPU, memory and bandwidth) utilization characteristics, compression properties, etc. To aid future research and studies, we are publicly releasing a dataset containing most of these statistics for ∼70 million queries over a period of 14 days, aggregated per-query. The dataset is publicly available at `https://github.com/resource-disaggregation/snowset`. For privacy reasons, the dataset does not contain information on query plans, table schemas and per-file access frequencies. To ensure reproducibility, this paper uses only those statistics that are contained in publicly released dataset.

Query Classification. We classify queries in the dataset based on number of persistent data bytes read and written (Figure 2 (left)). Figure 2 (right) shows number of queries submitted at different times of day for each query class.

- **Read-only queries:** Queries along the x-axis are the ones that do not write any persistent data; however, the amount of data read by these queries can vary over nine orders of magnitude. These queries contribute to ∼28% of all customer queries, and represent ad-hoc and interactive OLAP

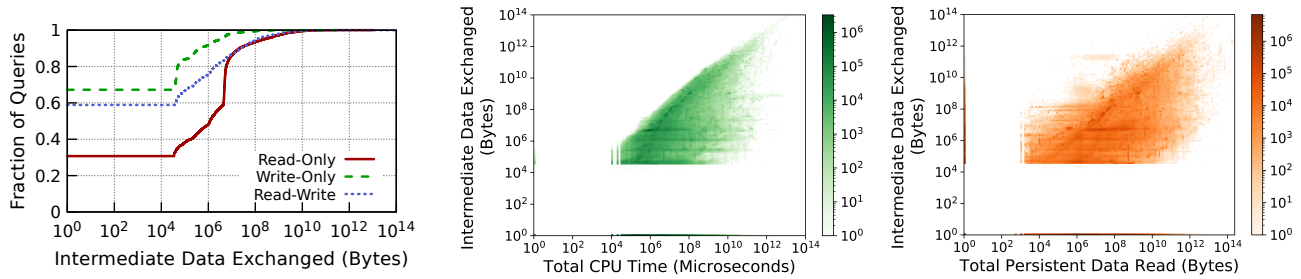

Figure 3: **Intermediate data characteristics. (left)** Intermediate data sizes vary over multiple orders of magnitude across queries; a non-trivial fraction of queries in each query class exchange zero intermediate data, while some read-only and read-write queries exchange 10 – 100TB of intermediate data. **(center)** Scatter plot with each point representing a query based on its total CPU time and amount of intermediate data exchanged by the corresponding query; queries with the same total CPU time exchange vastly different amounts of intermediate data. **(right)** Scatter plot with each point representing a query based on its total persistent data read and amount of intermediate data exchanged by the corresponding query; queries that read the same amount of persistent data exchange vastly different amounts of intermediate data.

workloads [10] typical in data warehouses, where the result is usually small in size and is directly returned to the client. The right figure demonstrates an interesting trend for read-only queries — the number of such queries submitted by customers spike during daytime hours on weekdays.

- **Write-only queries:** Queries along the y-axis are the ones that do not read any persistent data; however, the amount of data written by these queries can vary over eight orders of magnitude. These queries contribute to ∼13% of all customer queries, and essentially represent data ingestion queries that bring data into the system. Unlike read-only queries, we observe that the rate of submission for write-only queries is fairly consistent across time.
- **Read-Write (RW) queries:** The region in the middle of the plot contains ∼59% of customer queries, and represents queries that both read and write persistent data. Here we see a wide spectrum of queries. Many queries have read-write ratio close to 1, in terms of number of bytes, that represent Extract Transform Load (ETL) pipelines [29, 31, 32] typical in data warehouses. For other queries, the read-write ratio can vary over multiple orders of magnitude.

The above classification is based on number of persistent data bytes read and written by the queries, a measure that is not an artifact of Snowflake's architecture; rather, it is a property of the queries themselves. Indeed, even if these queries were to run on, say, any other data analytics framework (*e.g.*, Hadoop) or even a single node database (*e.g.*, MySQL), the persistent read/write characteristics would remain the same. We do not classify queries based on semantics as the focus of this paper is on systems characteristics, and our dataset does not contain detailed information about individual query plans. We will use the above query classification throughout the paper.

4 Ephemeral Storage System

Snowflake uses a custom-designed distributed storage system for management and exchange of intermediate data, due to two limitations in existing persistent data stores [5, 8]. First, they fall short of providing the necessary latency and throughput performance to avoid compute tasks being blocks on intermediate data exchange. Second, they provide much stronger availability and durability semantics than what is needed for intermediate data. Our ephemeral storage system allows us to overcome both these limitations. Tasks executing query operations (*e.g.*, joins) on a given compute node write intermediate data locally; and, tasks consuming the intermediate data read it either locally or remotely over the network (depending on the node where the task is scheduled, §5).

4.1 Storage Architecture, and Provisioning

We made two important design decisions in our ephemeral storage system. First, rather than designing a pure in-memory storage system, we decided to use both memory and local SSDs — tasks write as much intermediate data as possible to their local memory; when memory is full, intermediate data is spilled to local SSDs. Our rationale is that while purely in-memory systems can achieve superior performance when entire data fits in memory, they are too restrictive to handle the variety of our target workloads. Figure 3 (left) shows that there are queries that exchange hundreds of gigabytes or even terabytes of intermediate data; for such queries, it is hard to fit all intermediate data in main memory.

The second design decision was to allow intermediate data to spill into remote persistent data store in case the local SSD capacity is exhausted. Spilling intermediate data to S3, instead of other compute nodes, is preferable for a number of reasons — it does not require keeping track of intermediate data location, it alleviates the need for explicitly handling out-of-memory or out-of-disk errors for large queries, and overall, allows to keep our ephemeral storage system thin and highly performant.

Future Directions. For performance-critical queries, we want intermediate data to entirely fit in memory, or at least in SSDs,

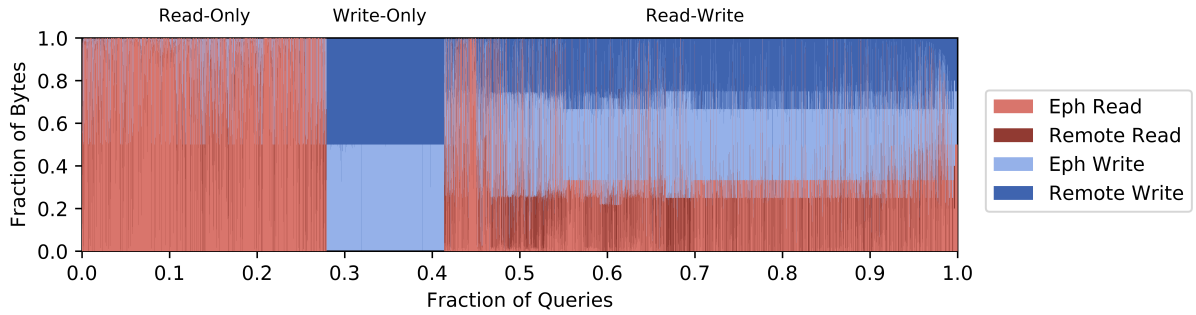

Figure 4: **Persistent data I/O traffic distribution between ephemeral storage system and remote persistent data store.** Each vertical bar corresponds to a query, the four colors correspond to read/write from ephemeral/remote systems, and the y-axis represents the fraction of total persistent data that was served by the corresponding storage system. Queries are sorted, within each class, in decreasing order of number of persistent data bytes read/written. Discussion in §4.2.

and not spill to S3. This requires accurate resource provisioning. However, provisioning CPU, memory and storage resources while achieving high utilization turns out to be challenging due to two reasons. The first reason is limited number of available node instances (each providing a fixed amount of CPU, memory and storage resources), and significantly more diverse resource demands across queries. For instance, Figure 3 (center) shows that, across queries, the ratio of compute requirements and intermediate data sizes can vary by as much as six orders of magnitude. The available node instances simply do not provide enough options to accurately match node hardware resources with such diverse query demands.

Second, even if we could match node hardware resources with query demands, accurately provisioning memory and storage resources requires a priori knowledge of intermediate data size generated by the query. However, our experience is that predicting the volume of intermediate data generated by a query is hard, or even impossible, for most queries. As shown in Figure 3, intermediate data sizes not only vary over multiple orders of magnitude across queries, but also have little or no correlation with amount of persistent data read or the expected execution time of the query.

To resolve the first challenge, we could decouple compute from ephemeral storage. This would allow us to match available node resources with query resource demands by independently provisioning individual resources. However, the challenge of unpredictable intermediate data sizes is harder to resolve. For such queries, simultaneously achieving high performance and high resource utilization would require both decoupling of compute and ephemeral storage, as well as efficient techniques for fine-grained elasticity of ephemeral storage system. We discuss the latter in more detail in §6.

4.2 Persistent Data Caching

One of the key observations we made during early phases of ephemeral storage system design is that intermediate data is short-lived. Thus, while storing intermediate data requires large memory and storage capacity *at peak*, the demand is low *on an average*. This allows statistical multiplexing of our ephemeral storage system capacity between intermediate data and frequently accessed persistent data. This improves performance since (1) queries in data warehouse systems exhibit highly skewed access patterns over persistent data [10]; and (2) ephemeral storage system performance is significantly better than that of (existing) remote persistent data stores.

Snowflake enables statistical multiplexing of ephemeral storage system capacity between intermediate data and persistent data by "opportunistically" caching frequently accessed persistent data files, where opportunistically refers to the fact that intermediate data storage is always prioritized over caching persistent data files. However, a persistent data file cannot be cached on any node — Snowflake assigns input file sets for the customer to nodes using consistent hashing over persistent data file names. A file can only be cached at the node to which it consistently hashes to; each node uses a simple LRU policy to decide caching and eviction of persistent data files. Given the performance gap between our ephemeral storage system and remote persistent data store, such opportunistic caching of persistent data files improves the execution time for many queries in Snowflake. Furthermore, since storage of intermediate data is always prioritized over caching of persistent data files, such an opportunistic performance improvement in query execution time can be achieved without impacting performance for intermediate data access.

Maintaining the right system semantics during opportunistic caching of persistent data files requires a careful design. First, to ensure data consistency, the "view" of persistent files in ephemeral storage system must be consistent with those in remote persistent data store. We achieve this by forcing the ephemeral storage system to act as a write-through cache for persistent data files. Second, consistent hashing of persistent data files on nodes in a naïve way requires reshuffling of cached data when VWs are elastically scaled. We implement a lazy consistent hashing optimization in our ephemeral storage system that avoids such data reshuffling *altogether*; we describe this when we discuss Snowflake elasticity in §6.

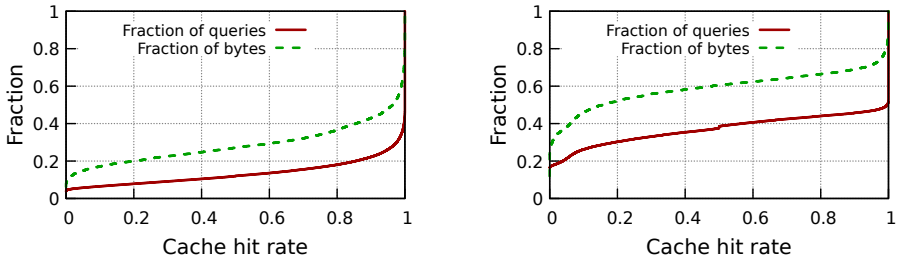

Figure 5: **Cache hit rate distribution of Read-only (left) and Read-Write queries (right).** We plot the CDFs for both, the queries (independent of their persistent data read sizes) and for bytes (queries weighed by the amount of persistent data read). For example, for Read-only queries, 80% of the queries have hit rate greater than 75%, but these queries account for only a little more than 60% of the total number of persistent bytes read by all Read-only queries.

Persistent data being opportunistically cached in the ephemeral storage system means that some subset of persistent data access requests could be served by the ephemeral storage system (depending on whether or not there is a cache hit). Figure 4 shows the persistent data I/O traffic distribution, in terms of fraction of bytes, between the ephemeral storage system and remote persistent data store. The write-through nature of our ephemeral storage system results in amount of data written to ephemeral storage being roughly of the same magnitude as the amount of data written to remote persistent data store (they are not always equal because of prioritizing storage of intermediate data over caching of persistent data).

Even though our ephemeral storage capacity is significantly lower than that of a customer's persistent data (around 0.1% on an average), skewed file access distributions and temporal file access patterns common in data warehouses [7] enable reasonably high cache hit rates (avg. hit rate is close to 80% for read-only queries and around 60% for read-write queries). Figure 5 shows the hit rate distributions across queries. The median hit rates are even higher.

Future Directions. Figure 4 and Figure 5 suggest that more work is needed on caching. In addition to locality of reference in access patterns, cache hit rate also depends on effective cache size available to the query relative to the amount of persistent data accessed by the query. The effective cache size, in turn, depends on both the VW size and the volume of intermediate data generated by concurrently executing queries. Our preliminary analysis has not led to any conclusive observations on the impact of the above two factors on the observed cache hit rates, and a more fine-grained analysis is needed to understand factors that impact cache hit rates.

We highlight two additional technical problems. First, since end-to-end query performance depends on both, cache hit rate for persistent data files and I/O throughput for intermediate data, it is important to optimize how the ephemeral storage system splits capacity between the two. Although we currently use the simple policy of always prioritizing intermediate data, it may not be the optimal policy with respect to end-to-end performance objectives (*e.g.*, average query completion time across all queries from the same customer). For example, it may be better to prioritize caching a persistent data file that is going to be accessed by many queries over intermediate data that is accessed by only one. It would be interesting to explore extensions to known caching mechanisms that optimize for end-to-end query performance objectives [7] to take intermediate data into account.

Second, existing caching mechanisms were designed for two-tier storage systems (memory as the main tier and HDD/SSD as the second tier). In Snowflake, we already have three tiers of hierarchy with compute-local memory, ephemeral storage system and remote persistent data store; as emerging non-volatile memory devices are deployed in the cloud and as recent designs on remote ephemeral storage systems mature [22], the storage hierarchy in the cloud will get increasingly deeper. Snowflake uses traditional two-tier mechanisms — each node implements a local LRU policy for evictions from local memory to local SSD, and an independent LRU policy for evictions from local SSD to remote persistent data store. However, to efficiently exploit the deepening storage hierarchy, we need new caching mechanisms that can efficiently coordinate caching across multiple tiers.

We believe many of the above technical challenges are not specific to Snowflake, and would apply more broadly to any distributed application built on top of disaggregated storage.

5 Query (Task) Scheduling

We now describe the query execution process in Snowflake. Customers submit their queries to the Cloud Services (CS) for execution on a specific VW. CS performs query parsing, query planning and optimization, and creates a set of tasks to be scheduled on compute nodes of the VW.

Locality-aware task scheduling. To fully exploit the ephemeral storage system, Snowflake colocates each task with persistent data files that it operates on using a locality-aware scheduling mechanism (recall, these files may be cached in ephemeral storage system). Specifically, recall that Snowflake assigns persistent data files to compute nodes using consistent

(a) Read-only (b) Write-only (c) Read-Write

Figure 6: **Persistent data read / write and intermediate data exchange characteristics of queries sorted by the number of nodes used.** Each plot uses the same axis for bytes (left axis) and nodes used (right axis). Persistent Read and write bytes vary by three orders of magnitude across each node count.

hashing over table file names. Thus, for a fixed VW size, each persistent data file is cached on a specific node. Snowflake schedules the task that operates on a persistent data file to the node on which its file consistently hashes to.

As a result of this scheduling scheme, query parallelism is tightly coupled with consistent hashing of files on nodes — a query is scheduled for cache locality and may be distributed across all the nodes in the VW. For instance, consider a customer that has 1million files worth of persistent data, and is running a VW with 10 nodes. Consider two queries, where the first query operates on 100 files, and the second query operates on 100,000 files; then, with high likelihood, both queries will run on all the 10 nodes because of files being consistently hashed on to all the 10 nodes.

Figure 6 illustrates this— the number of persistent bytes read and written vary over orders of magnitude, almost independent of the number of nodes in the VW. As expected, the intermediate data exchanged over the network increases with the number of nodes used.

Work stealing. It is known that consistent hashing can lead to imbalanced partitions [19]. In order to avoid overloading of nodes and improve load balance, Snowflake uses *work stealing*, a simple optimization that allows a node to steal a task from another node if the expected completion time of the task (sum of execution time and waiting time) is lower at the new node. When such work stealing occurs, the persistent data files needed to execute the task are read from remote persistent data store rather than the node at which the task was originally scheduled on. This avoids increasing load on an already overloaded node where the task was originally scheduled (note that work stealing happens only when a node is overloaded).

Future Directions. Schedulers can place tasks onto nodes using two extreme options: one is to colocate tasks with their cached persistent data, as in our current implementation. As discussed in the example above, this may end up scheduling all queries on all nodes in the VW; while such a scheduling policy minimizes network traffic for reading persistent data, it may lead to increased network traffic for intermediate data exchange. The other extreme is to place all tasks on a single node. This would obviate the need of network transfers for intermediate data exchange but would increase network traffic for persistent data reads. Neither of these extremes may be the right choice for all queries. It would be interesting to codesign query schedulers that would pick just the right set of nodes to obtain a sweet spot between the two extremes, and then schedule individual tasks onto these nodes.

6 Resource Elasticity

In this section, we discuss how Snowflake design achieves one of its core goals: resource elasticity, that is, scaling of compute and storage resources on an on-demand basis.

Disaggregating compute from persistent storage enables Snowflake to independently scale compute and persistent storage resources. Storage elasticity is offloaded to persistent data stores [5]; compute elasticity, on the other hand, is achieved using a pre-warmed pool of nodes that can be added/removed to/from customer VWs on an on-demand basis. By keeping a pre-warmed pool of nodes, Snowflake is able to provide compute elasticity at the granularity of tens of seconds.

6.1 Lazy Consistent Hashing

One of the challenges that Snowflake had to resolve in order to achieve elasticity efficiently is related to data management in ephemeral storage system. Recall that our ephemeral storage system opportunistically caches persistent data files; each file can be cached only on the node to which it consistently hashes to within the VW. The problem is similar to shared-nothing architectures: any fixed partitioning mechanism (in our case, consistent hashing) requires large amounts of data to be reshuffled upon scaling of nodes; moreover, since the very same set of nodes are also responsible for query processing, the system observes a significant performance impact during the scaling process.

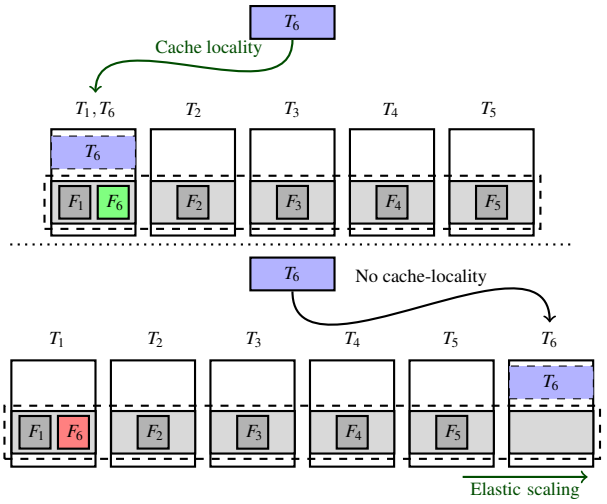

Figure 7: **Snowflake uses lazy consistent hashing to avoid data reshuffling during elastic scaling of VWs. (Top)** VW in steady state with all task inputs cached. **(Bottom)** VW immediately after adding one node. See discussion in §6.1.

Snowflake resolves this challenge using a lazy consistent hashing mechanism, that completely avoids any reshuffling of data upon elastic scaling of nodes by exploiting the fact that a copy of cached data is stored at remote persistent data store. Specifically, Snowflake relies on the caching mechanism to eventually "converge" to the right state. For instance, consider the example in Figure 7 that shows a VW with 6 tasks $T_1, T_2, \ldots, T_6$, with task T_i operating on a single file F_i. Suppose at time t_0, we have 5 nodes in the VW and that node N_1 stores files F_1 and F_6, and nodes $N_2 - N_5$ store file $F_2 - F_5$, respectively. Suppose at time $t > t_0$, a node N_6 is added to the warehouse. Then, rather than immediately reshuffling the files (which would result in F_6 being moved from node N_1 to N_6), Snowflake will wait until task T_6 is executed again. When the next time T_6 is scheduled (*e.g.*, due to work stealing or the same query being executed again), Snowflake will schedule it on N_6 since consistent hashing will now place file F_6 on that node. At this time, file F_6 will be read by N_6 from remote persistent store and cached locally. File F_6 on node N_1 will no longer be accessed and will eventually be evicted from the cache. Such lazy caching allows Snowflake to achieve locality without reshuffling data and without interfering with ongoing queries on nodes already in the VW.

6.2 Elasticity Characteristics

Our customer warehouses exhibit several interesting elasticity characteristics. Figure 8 shows that many of our customers already exploit our support for elasticity (for ∼20% of the VW). For such cases where customers do request VW resizing, the number of nodes in their VW can change by as much as two orders of magnitude during the lifetime of the VW! Figure 9 (top) shows two cases where customers leverage elasticity rather aggressively (even at hourly granularity).

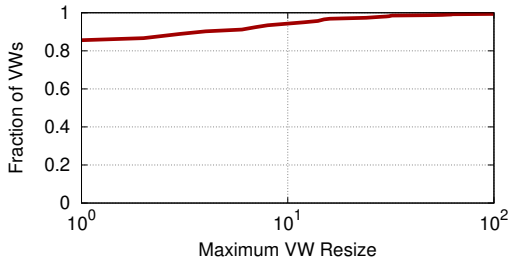

Figure 8: **VW elasticity usage.** 82% of all VW never exploit elasticity; however, some of the customers elastically scale their VW size by as much as two orders of magnitude!

Future Directions. Figure 8 also shows that our customers do not exploit our support for elasticity for more than 80% of the VW. Even for customers that do request VW resizing, there are opportunities for further optimizations — Figure 9 shows that query inter-arrival times in these VW are much finer-grained than the granularity of VW scaling requested by our customers. We believe the main reason for these characteristics is that customers lack the visibility (and the right demand estimation) into the system to accurately request scaling their VW to meet the demand. VW1 in Figure 9, for instance, is one of the heavily utilized VW from a large customer; the characteristics for this VW demonstrate how elastic scaling can mismatch the demand (approximated by query inter-arrival time). Since the time the dataset in the paper was recorded, a lot of work has been done to improve support for auto-scaling VW at the granularity of inter-query arrival times. However, much more work needs to be done.

First, we would like to achieve elasticity at intra-query granularity. Specifically, resource consumption can vary significantly even within the lifetime of individual queries. This is particularly prevalent in long running queries with many internal stages. Hence, in addition to auto-scaling VW at the granularity of query inter-arrivals, we would ideally like to support some level of task-level elasticity even during the execution of a query.

Second, we would like to explore serverless-like platforms. Serverless infrastructures such as AWS Lambda, Azure Functions and Google Cloud Functions which provide auto-scaling, high elasticity and fine-grained billing, are seeing increasing adoption across many application types. However, the key barrier for Snowflake to transition to existing serverless infrastructures is their lack of support for isolation, both in terms of security and performance [34]. Snowflake serves several customers who store and query sensitive and confidential data, and thus require strong isolation guarantees. One possibility for Snowflake is to build its own custom serverless-like compute platform. We believe this is an intriguing direction to explore, but will require resolving several challenges in efficient remote ephemeral storage access (§4.1), and in multi-tenant resource sharing (which we will discuss in §7).

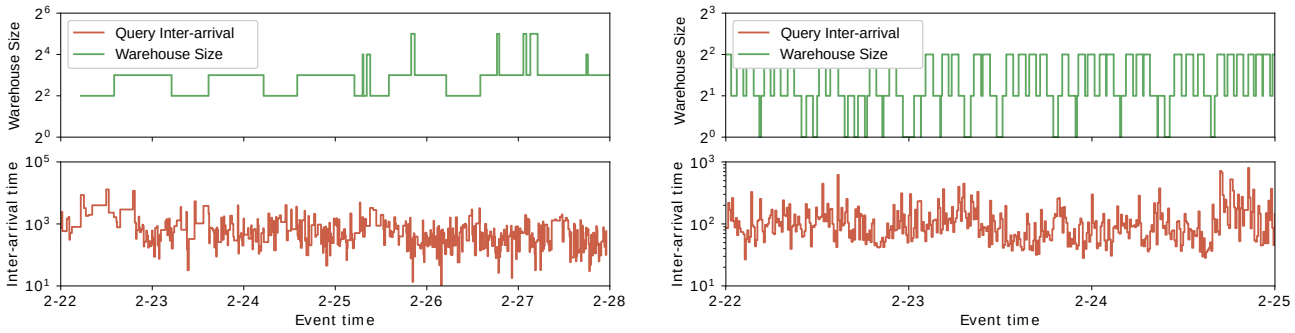

Figure 9: **Comparison of VW resizing (elastic scaling) and query inter-arrival times, binned per minute, for two heavily utilized VW1 (left) and VW2 (right).** Customers request VW resizing at much coarser grained timescales than query inter-arrival times for both VW. Note the difference in x-axis: 7 days and 4 days for VW1 and VW2, respectively.

7 Multi-tenancy

Snowflake currently supports multi-tenancy through the VW abstraction. Each VW operates on an isolated set of nodes, with its own ephemeral storage system. This allows Snowflake to provide performance isolation to its customers. In this section, we present a few system-wide characteristics for our VWs and use these to motivate an alternate sharing based architecture for Snowflake.

The VW architecture in Snowflake leads to the traditional performance isolation versus utilization tradeoff. Figure 10 (top four) show that our VWs achieve fairly good, but not ideal, average CPU utilization; however, other resources are usually underutilized on an average. Figure 11 provides some reasons for the low average resource utilization in Figure 10 (top four): the figure shows the variability of resource usage across VW; specifically, we observe that for up to 30% of VW, standard deviation of CPU usage over time is as large as the mean itself. This results in underutilization as customers tend to provision VWs to meet peak demand. In terms of peak utilization, several of our VWs experience periods of heavy utilization, but such high-utilization periods are not necessarily synchronized across VWs. An example of this is shown in Figure 10 (bottom two), where we see that over a period of two hours, there are several points when one VW's utilization is high while the other VW's utilization is simultaneously low.

While we were aware of this performance isolation versus utilization tradeoff when we designed Snowflake, recent trends are pushing us to revisit this design choice. Specifically, maintaining a pool of pre-warmed instances was cost-efficient when infrastructure providers used to charge at an hourly granularity; however, recent move to per-second pricing [6] by all major cloud infrastructure providers has raised interesting challenges. From our (provider's) perspective, we would like to exploit this finer-grained pricing model to cut down operational costs. However doing so is not straightforward, as this trend has also led to an increase in customer-demand for finer-grained pricing. As a result, maintaining a pre-warmed pool of nodes for elasticity is no longer cost-effective: previously in the hourly billing model, as long as at least one customer VW used a particular node during a one hour duration, we could charge that customer for the entire duration. However, with per-second billing, we cannot charge unused cycles on pre-warmed nodes to any particular customer. This cost-inefficiency makes a strong case for moving to a sharing based model, where compute and ephemeral storage resources are shared across customers: in such a model we can provide elasticity by statistically multiplexing customer demands across a shared set of resources, avoiding the need to maintain a large pool of pre-warmed nodes. In the next subsection, we highlight several technical challenges that need to be resolved to realize such a shared architecture.

7.1 Resource Sharing

The variability in resource usage over time across VW, as shown in Figure 11, indicates that several of our customer workloads are bursty in nature. Hence, moving to a shared architecture would enable Snowflake to achieve better resource utilization via fine-grained statistical multiplexing. Snowflake today exposes VW sizes to customers in abstract "T-shirt" sizes (small, large, XL etc.), each representing different resource capacities. Customers are not aware of how these VWs are implemented (no. of nodes used, instance types, etc.). Ideally we would like to maintain the same abstract VW interface to customers and change the underlying implementation to use shared resources instead of isolated nodes.

The challenge, however, is to achieve isolation properties close to our current architecture. The key metric of interest from customers' point of view is query performance, that is, end-to-end query completion times. While a purely shared architecture is likely to provide good average-case performance, maintaining good performance at tail is challenging. The two key resources that need to be isolated in VWs are compute and ephemeral storage. There has been a lot of work [18, 35, 36] on compute isolation in the data center context,

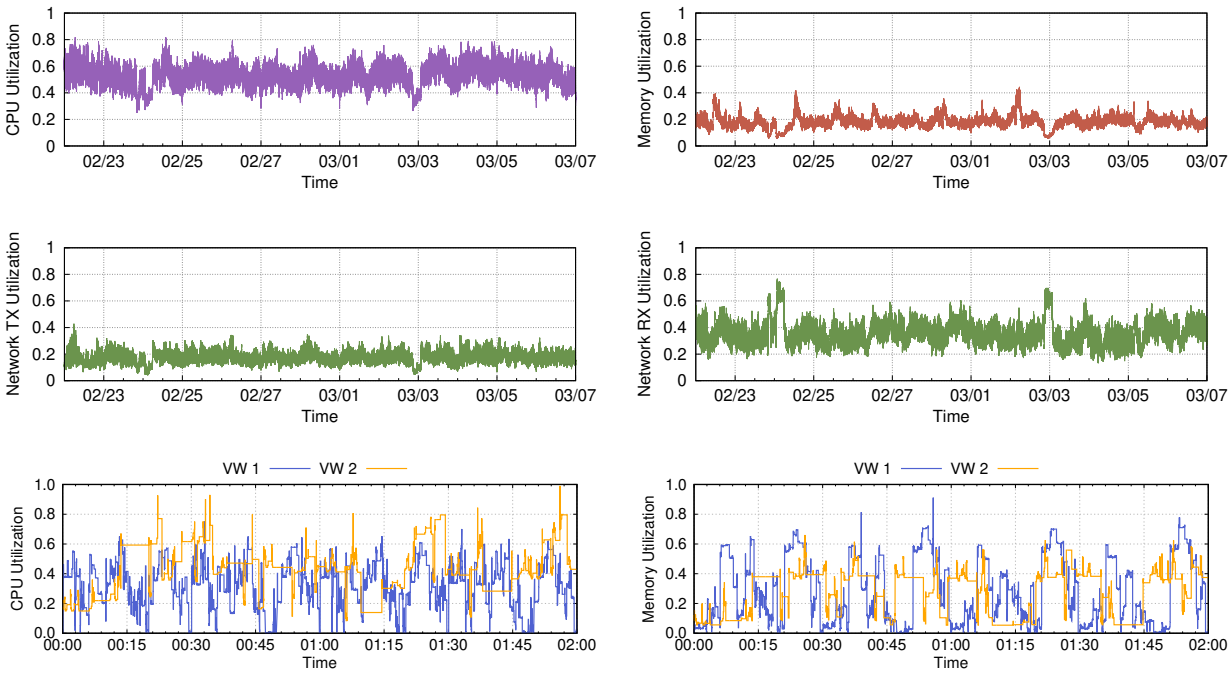

Figure 10: **System-wide CPU, Memory, and Network TX/RX utilization over time, averaged across all VWs. (top four)**. Average CPU, Memory, Network TX and Network RX utilizations are roughly 51%, 19%, 11%, 32%, respectively, indicating that there is significant room for improvement. **(bottom two)** zoomed-in CPU and Memory utilization for two highly active VW over a 2 hour duration. At several points, we see that one of the warehouses experiences high utilization while the other sees low utilization.

that Snowflake could leverage. Moreover, the centralized task scheduler and uniform execution runtime in Snowflake make the problem easier than that of isolating compute in general purpose clusters. Here, we instead focus on the problem of isolating memory and storage, which has only recently started to receive attention in the research community [25].

The goal here is to design a shared ephemeral storage system (using both memory and SSDs) that supports fine-grained elasticity without sacrificing isolation properties across tenants. With respect to sharing and isolation of ephemeral storage, we outline two key challenges. First, since our ephemeral storage system multiplexes both cached persistent data and intermediate data, both of these entities need to be jointly shared while ensuring cross-tenant isolation. While Snowflake could leverage techniques from existing literature [11, 26] for sharing cache, we need a mechanism that is additionally aware of the co-existence of intermediate data. Unfortunately, predicting the effective lifetime of cache entries is difficult. Evicting idle cache entries from tenants and providing them to other tenants while ensuring hard isolation is not possible, as we cannot predict when a tenant will next access the cache entry. Some past works [11, 33] have used techniques like idle-memory taxation to deal with this issue. We believe there is more work to be done, both in defining more reasonable isolation guarantees and designing lifetime-aware cache sharing mechanisms that can provide such guarantees.

The second challenge is that of achieving elasticity without cross-tenant interference: scaling up the shared ephemeral storage system capacity in order to meet the demands of a particular customer should not impact other tenants sharing the system. For example, if we were to naïvely use Snowflake's current ephemeral storage system, isolation properties will be trivially violated. Since all cache entries in Snowflake are consistently hashed onto the same global address space, scaling up the ephemeral storage system capacity would end up triggering the lazy consistent hashing mechanism for all tenants. This may result in multiple tenants seeing increased cache misses, resulting in degraded performance. Resolving this challenge would require the ephemeral storage system to provide private address spaces to each individual tenant, and upon scaling of resources, to reorganize data only for those tenants that have been allocated additional resources.

Memory Disaggregation. Average memory utilization in our VWs is low (Figure 10); this is particularly concerning since DRAM is expensive. Although sharing resource sharing would improve CPU and memory utilization, it is unlikely to lead to optimal utilization across both dimensions. Further, variability characteristics of CPU and memory are significantly different (Figure 11), indicating the need for independent scaling of these resources. Memory disaggregation [1, 14, 15] provides a fundamental solution to this problem. However, as discussed in §4.2, accurately provisioning re-

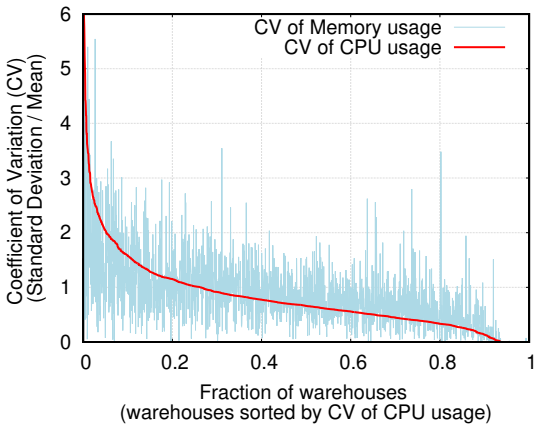

Figure 11: **Coefficient of Variation (CV) of CPU and memory usage over time, across customer VWs.** We see significant variability with respect to both resources.

sources is hard; since over-provisioning memory is expensive, we need efficient mechanisms to share disaggregated memory across multiple tenants while providing isolation guarantees.

8 Related Work

In this section we discuss related work and other systems similar to Snowflake. Our previous work [12] discusses SQL-related aspects of Snowflake and presents related literature on those aspects. This paper focuses on the disaggregation, ephemeral storage, caching, task scheduling, elasticity and multi-tenancy aspects of Snowflake; in the related work discussion below, we primarily focus on these aspects.

SQL-as-a-Service systems. There are several other systems that offer SQL functionality as a service in the cloud. These include Amazon Redshift [16], Aurora [4], Athena [3], Google BigQuery [30] and Microsoft Azure Synapse Analytics [24]. While there are papers that describe the design and operational experience of some of these systems, we are not aware of any prior work that undertakes a data-driven analysis of workload and system characteristics similar to ours.

Redshift [16] stores primary replicas of persistent data within compute VM clusters (S3 is only used for backup); thus, it may not be able to achieve the benefits that Snowflake achieves by decoupling compute from persistent storage. Aurora [4] and BigQuery [30] (based on the architecture of Dremel [23]) decouple compute and persistent storage similar to Snowflake. Aurora, however, relies on a custom-designed persistent storage service that is capable of offloading database log processing, instead of a traditional blob store. We are not aware of any published work that describes how BigQuery handles elasticity and multi-tenancy.

Decoupling compute and ephemeral storage systems. Previous work [20] makes the case for flash storage disaggregation by studying a key-value store workload from Facebook. Our observations corroborate this argument and further extend it in the context of data warehousing workloads. Pocket [22] and Locus [27] are ephemeral storage systems designed for serverless analytics applications. If we were to disaggregate compute and ephemeral storage in Snowflake, such systems would be good candidates. However, these systems do not provide fine-grained resource elasticity during the lifetime of a query. Thus, they either have to assume a priori knowledge of intermediate data sizes (for provisioning resources at the time of submitting queries), or suffer from performance degradation if such knowledge is not available in advance. As discussed in §4.1, predicting intermediate data sizes is extremely hard. It would be nice to extend these systems to provide fine-grained elasticity and cross-query isolation. Technologies for high performance access to remote flash storage [13, 17, 21] would also be integral to efficiently realize decoupling of compute and ephemeral storage system.

Multi-tenant resource sharing. ESX server [33] pioneered techniques for multi-tenant memory sharing in the virtual machine context, including ballooning and idle-memory taxation. Memshare [11] considers multi-tenant sharing of cache capacity in DRAM caches in the single machine context, sharing un-reserved capacity among applications in a way that maximizes hit rate. FairRide [26] similarly considers multi-tenant cache sharing in the distributed setting while taking into account sharing of data between tenants. Mechanisms for sharing and isolation of cache resources similar to the ones used in these works would be important in enabling Snowflake to adopt a resource shared architecture. As discussed previously, it would be interesting to extend these mechanisms to make them aware of the different characteristics and requirements of intermediate and persistent data.

9 Conclusion

We have presented operational experience running *Snowflake*, a data warehousing system with state-of-the-art SQL support. The key design and implementation aspects that we have covered in the paper relate to how Snowflake achieves compute and storage elasticity, as well as high-performance in a multi-tenancy setting. As Snowflake has grown to serve thousands of customers executing millions of queries on petabytes of data every day, we consider ourselves at least partially successful. However, using data collected from various components of our system during execution of ∼70 million queries over a 14 day period, our study highlights some of the shortcomings of our current design and implementation and highlights new research challenges that may be of interest to the broader systems and networking communities.

Acknowledgments

We would like to thank our shepherd, Asaf Cidon, and the anonymous NSDI reviewers for their insightful feedback. This work was supported in part by NSF 1704742 and a Google Faculty Research Award.

References

[1] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote memory in the age of fast networks. In *SOCC*, 2017.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, 2001.

[3] Amazon. Amazon Athena. "https://aws.amazon.com/athena/".

[4] Amazon. Amazon Aurora Serverless. "https://aws.amazon.com/rds/aurora/serverless/".

[5] Amazon. Amazon simple storage service (S3). "http://aws.amazon.com/s3/".

[6] Amazon. Per-second billing for EC2 instances. "https://aws.amazon.com/blogs/aws/new-per-second-billing-for-ec2-instances-and-ebs-volumes/".

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

[8] M. Azure. Azure blob storage. "https://azure.microsoft.com/en-us/services/storage/blobs/".

[9] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.

[10] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD*, 1997.

[11] A. Cidon, D. Rushton, S. M. Rumble, and R. Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *ATC*, 2017.

[12] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, et al. The snowflake elastic data warehouse. In *SIGMOD*, 2016.

[13] N. Express. Nvme over fabrics overview. "https://nvmexpress.org/wp-content/uploads/NVMe_Over_Fabrics.pdf".

[14] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *OSDI*, 2016.

[15] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with Infiniswap. In *NSDI*, 2017.

[16] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. Amazon Redshift and the case for simpler data warehouses. In *SIGMOD*, 2015.

[17] J. Hwang, Q. Cai, A. Tang, and R. Agarwal. TCP ≈ RDMA: Cpu-efficient remote storage access with I10. In *NSDI*, 2020.

[18] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, et al. PerfIso: Performance isolation for commercial latency-sensitive services. In *ATC*, 2018.

[19] D. Karger and M. Ruhl. New algorithms for load balancing in peer-to-peer systems. 2003.

[20] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar. Flash storage disaggregation. In *EuroSys*, 2016.

[21] A. Klimovic, H. Litz, and C. Kozyrakis. ReFlex: Remote flash ≈ local flash. In *ASPLOS*, 2017.

[22] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *OSDI*, 2018.

[23] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *VLDB*, 2010.

[24] Microsoft. Azure synapse analytics. "https://azure.microsoft.com/en-us/services/synapse-analytics/".

[25] M. Nanavati, J. Wires, and A. Warfield. Decibel: Isolation and sharing in disaggregated rack-scale storage. In *NSDI*, 2017.

[26] Q. Pu and H. Li. FairRide: Near-optimal, fair cache sharing. In *NSDI*, 2016.

[27] Q. Pu, S. Venkataraman, and I. Stoica. Shuffling, fast and slow: scalable analytics on serverless infrastructure. In *NSDI*, 2019.

[28] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC*, 2012.

[29] R. J. Santos and J. Bernardino. Real-time data warehouse loading methodology. In *IDEAS*, 2009.

[30] K. Sato. An inside look at google bigquery. "https://cloud.google.com/files/BigQueryTechnicalWP.pdf".

[31] A. Simitsis, P. Vassiliadis, and T. Sellis. Optimizing ETL processes in data warehouses. In *ICDE*, 2005.

[32] P. Vassiliadis. A survey of extract–transform–load technology. *IJDWM*, 2009.

[33] C. A. Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 2002.

[34] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *ATC*, 2018.

[35] X. Yang, S. M. Blackburn, and K. S. McKinley. Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading. In *ATC*, 2016.

[36] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI2: CPU performance isolation for shared compute clusters. In *EuroSys*, 2013.