Exploiting Cloud Object Storage for High-Performance Analytics

Dominik Durner
Technische Universität München
dominik.durner@tum.de

Viktor Leis
Technische Universität München
viktor.leis@tum.de

Thomas Neumann
Technische Universität München
thomas.neumann@tum.de

ABSTRACT

Elasticity of compute and storage is crucial for analytical cloud database systems. All cloud vendors provide disaggregated object stores, which can be used as storage backend for analytical query engines. Until recently, local storage was unavoidable to process large tables efficiently due to the bandwidth limitations of the network infrastructure in public clouds. However, the gap between remote network and local NVMe bandwidth is closing, making cloud storage more attractive. This paper presents a blueprint for performing efficient analytics directly on cloud object stores. We derive cost- and performance-optimal retrieval configurations for cloud object stores with the first in-depth study of this foundational service in the context of analytical query processing. For achieving high retrieval performance, we present *AnyBlob*, a novel download manager for query engines that optimizes throughput while minimizing CPU usage. We discuss the integration of high-performance data retrieval in query engines and demonstrate it by incorporating *AnyBlob* in our database system *Umbra*. Our experiments show that even without caching, *Umbra* with integrated *AnyBlob* achieves similar performance to state-of-the-art cloud data warehouses that cache data on local SSDs while improving resource elasticity.

PVLDB Reference Format:
Dominik Durner, Viktor Leis, and Thomas Neumann. Exploiting Cloud Object Storage for High-Performance Analytics. PVLDB, 16(11): 2769 - 2782, 2023.
doi:10.14778/3611479.3611486

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/durner/AnyBlob.

1 INTRODUCTION

Data warehousing moves to the cloud. Estimates show that the revenue of cloud database systems has reached that of on-premise systems in 2021 [1] – and by VLDB 2023, the cloud market share will presumably be significantly higher. A major part of this change is the shift of data warehousing and analytical query processing to the cloud. The main drivers behind that are elasticity and the flexibility to provision storage and compute separately and on demand.

Cloud object stores. Cloud object stores such as AWS S3, IBM COS, and GCP Storage enable separating compute from storage in a cost-effective (e.g., ~23$/TiB per month) way [13]. They also provide strong durability guarantees (e.g., 11 9's per year for S3 [4]), practically unlimited capacity, and scalable access bandwidth. These properties make disaggregated cloud object storage a natural fit for analytical database systems. In future data centers, database systems may even run on hardware that separates memory and compute. There, disaggregated storage is crucial to provide durability [83, 91].

High-bandwidth networks. Until recently, the major issue of cloud object storage for analytics was the limited network bandwidth between instances and storage. In 2018, AWS introduced instances with 100 Gbit/s (≈12 GB/s) networking – resulting in a four-fold increase in per-instance bandwidth [22, 26]. In contrast to Infiniband, 100 Gbit/s Ethernet has not only become widely-available but also affordable[1]. This effectively closes the gap between remote network and local NVMe bandwidth[2] and makes relying more on cloud storage more attractive for bandwidth-dominated workloads.

Cloud storage analytics. Most cloud-native data warehouse systems, such as Snowflake [33, 82], Databricks [25], and AWS Redshift [19], use cloud object storage as their ground-truth data source. Although the bandwidth gap between local storage and network is closing, most research focuses on caching to avoid fetching data from remote storage [37, 46, 85, 89]. Early research investigates object storage for transactional database systems but limits its focus on OLTP [27]. Surprisingly, no empirical study for general-purpose analytics (OLAP) on cloud object stores has been conducted.

Challenge 1: Achieving instance bandwidth. Because the latency of each object request is high, saturating high-bandwidth networks requires many concurrently outstanding requests. Therefore, a careful network integration into the DBMS is crucial to achieve the complete bandwidth available on network-optimized instances.

Challenge 2: Network CPU overhead. In contrast to fetching data from local disks, network retrieval has higher CPU overhead. Query engines, however, also contend for computation resources to simultaneously analyze large sets of data. Consequently, reducing the CPU footprint of network retrieval is essential.

Challenge 3: Multi-cloud support. Many cloud database systems are able to run in different clouds – allowing the user to choose the vendor of their liking. In contrast to the desire for multi-cloud systems, each cloud vendor provides its own networking library. Thus, multiple libraries need to be integrated, which increases complexity.

Approach. In this paper, we present a blueprint for performing efficient analytics directly on data residing in disaggregated cloud object stores. We studied the cloud object stores of different vendors to derive cost- and performance-optimal retrieval configurations. To reduce resource utilization for network retrieval, we developed a download manager that is able to fetch data from multiple cloud vendors. We seamlessly integrate high-bandwidth object retrieval with the database engine's scan operator. Our DBMS Umbra, equipped

Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611486

[1] Comparing the on-demand prices of c5n.18xlarge (100 Gbit/s) and c5.18xlarge (25 Gbit/s) while taking c5n's larger DRAM into account (~30% more DRAM), we find that adding 100 Gbit/s networking increases cost by only 22%.
[2] Consider i3en.24xlarge, the AWS instance with the fastest local NVMe bandwidth. Its local read bandwidth is 16 GB/s, while its full-duplex network bandwidth is 12 GB/s.

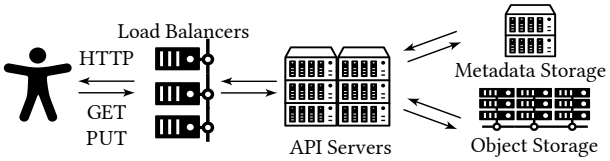

Figure 1: Schematic architecture of AWS S3.

with our download manager and caching disabled, achieves similar performance on a single instance as large configurations of state-of-the-art cloud database systems that cache data on local SSDs. Our fast and low overhead networking integration facilitates switching instances without performance cliffs, improving elasticity. As switching comes without performance cliffs, our approach is able to better utilize spot instances, available at huge discounts.

Contribution 1: Experimental study of cloud object stores. To achieve high-bandwidth data processing, we first study the properties of cloud object stores. In Section 2, we explain the design of disaggregated storage, discuss the cost structure, and then provide detailed experiments on the latency and throughput of different object stores. We define an optimal request size range that minimizes cost while maximizing throughput. Our concurrency analysis helps to schedule enough requests to meet the throughput goal (i.e., instance bandwidth). Our in-depth experimental study of this foundational cloud service helps to exploit disaggregated storage for analytical query processing.

Contribution 2: AnyBlob, a low overhead multi-cloud library. With the insights gained from our characterization of object stores, we developed *AnyBlob*, an open-source, multi-cloud download manager for object stores that is optimized for large data analytics [36]. *AnyBlob*, described in Section 3, achieves the same throughput as the libraries provided by the cloud vendors while reducing CPU resource consumption significantly. CPU resource utilization is vital to process data concurrently. In contrast to existing solutions, our approach does not need to start new threads for parallel requests because it uses `io_uring` [21], which facilitates asynchronous system calls. To saturate the network bandwidth, our analysis shows that hundreds of requests have to be outstanding simultaneously. Our solution helps to reduce thread scheduling overhead and allows seamless integration into database query engines.

Contribution 3: Blueprint for retrieval integration. Tight integration of the download manager into the database engine enables efficient analytics on disaggregated storage. We present a blueprint to incorporate *AnyBlob* into database engines in Section 4. By carefully designing the scan operator and developing an object retrieval scheduler, we can seamlessly interleave the downloading of objects with the analytical processing.

2 CLOUD STORAGE CHARACTERISTICS

Methodology. In order to design an efficient analytics engine based on cloud object storage, we need to understand its basic characteristics. We start with an analysis on the performance characteristics and cost of disaggregated object stores and compute instances. To gain insights into the storage architecture, we perform various micro-experiments on AWS S3 and two other cloud providers to understand latency and throughput limitations. A study on AWS shows that instances are able to achieve high network throughput to S3 [80]. With the best practices in mind [10], we conduct this in-depth experimental study that helps exploiting cloud storage for analytical query processing. Unless otherwise specified, we use our *AnyBlob* library as the retrieval manager, presented in Section 3.

Table 1: Cloud storage cost by cloud vendor for zone-redundant replication (default; multiple AZs within region).

Cloud Provider (cheapest region)	Storage ($ / TiB / month)	GET ($ / 1 M)	PUT ($ / 1 M)
AWS (us-east-2) [13]	23.55	0.40	5.00
GCP (us-east-1) [39]	20.48	0.40	5.00
IBM (us-east) [45]	23.55	0.42	5.20
Azure (East US 2) [60]	23.55	0.40	6.25
OCI (us-ashburn-1) [64]	26.11	0.34	0.34

2.1 Object Storage Architecture

Overview. All major cloud vendors provide disaggregated storage solutions such as AWS S3, Azure Blob, IBM COS, OCI Object Storage, and GCP Storage. Data is stored in immutable blocks called *objects*. These objects are distributed and replicated across several storage servers for availability and durability. After resolving the domain name of the cloud object store, the user requests an object from a storage server which then sends the data. All major cloud providers use a similar API that transfers data via HTTP (TCP).

Architecture of S3. The architecture of S3 is depicted in Figure 1 [32]. AWS S3 defines *prefixes* that are similar to unique file paths in operating systems. Objects are similar to files and all levels above objects are similar to directories. Data is stored in *buckets* that resemble hard drive partitions in our analogy. According to AWS, S3 partitions all prefixes to scale to thousands of requests per second [32]. A prefix can range from covering a bucket down to individual objects. With an update in 2020, S3 is now a strongly consistent system [23]. Other providers were already strongly consistent. S3 replicates the data to at least 3 different availability zones (AZs). A geographic region consists of AZs that are separated data centers for increasing availability and durability [75].

Bandwidth limits. Data access performance is characterized by the network connection of the instance, the network connection of the cloud storage, and the network itself. At AWS, general-purpose instances achieve 100 Gbit/s and more to the object stores [3, 5].

2.2 Object Storage Cost

Cost structure. All major cloud vendors structure their object storage pricing similarly. They categorize expenses as storage cost, data retrieval and data modification cost (API cost), and inter-region network transfer cost. Cloud providers operate object stores on the level of a region (e.g., eu-central-1). When accessing data within one region, only API costs are charged because intra-region traffic is free to the object store. On the other hand, AWS inter-region data transfer, for example, from the US east to Europe costs 0.02$/GB.

Size-independent retrieval cost. Table 1 shows that the pricing of cloud providers is similar for zone-redundant replication (default), which provides high durability and optimal retrieval performance.

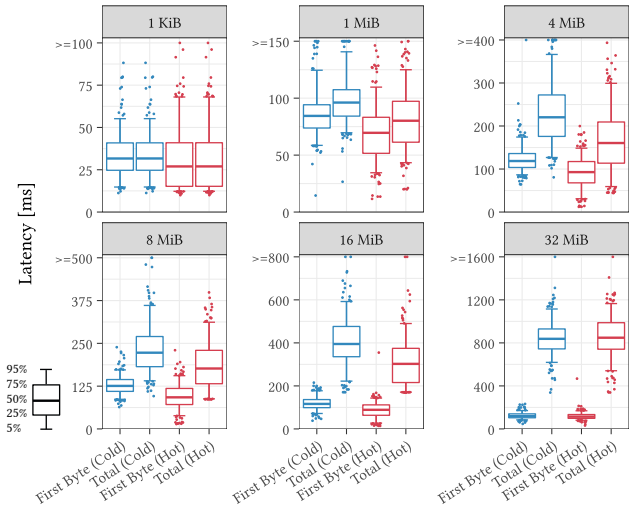

Figure 2: First byte and total latency for different requests sizes on hot and cold objects (AWS, eu-central-1, c5n.large).

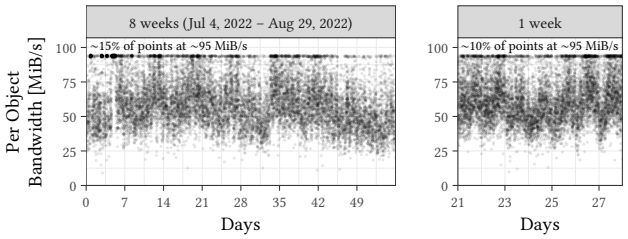

Figure 3: S3 bandwidth over 8 weeks (AWS, eu-central-1).

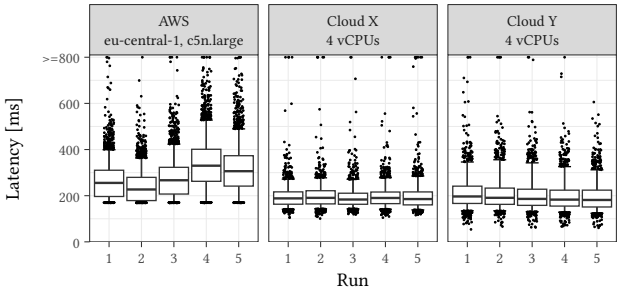

Figure 4: Total latency distribution of different object stores over multiple runs on sparsely accessed data (12h interval).

Surprisingly, retrieval cost in the same region depends only on the number of requests sent to the cloud object store, and does not depend on the object size. Downloading a 1 KiB object costs the same as a 1 TiB object, as long as only one HTTP request is issued.

Cloud storage alternatives. Other storage solutions are not as elastic as disaggregated storage and are often more expensive. For example, AWS Elastic Block Storage (EBS) (gp2 SSD) costs 102.4$/TiB compared to 23.2$/TiB per month. HDD storage pricing is comparable to S3, but bandwidth is very limited. Although EBS is elastic in its size, it can only be attached to a single node. Instance-based SSD storage is also expensive. For example, the price difference between c5.18xlarge and c5d.18xlarge is 0.396$/h and yields in 1.8 TB NVMe SSD. There, instance storage costs 158.4$/TB per month, which is 7× more expensive. Another example for instance-based storage is the largest HDD cluster instance d3en.12xlarge. This instance features 24 HDDs with 14 TB storage each at a price of 13.5$/TB per month. Although this seems cheaper initially, such an instance cannot provide S3's durability guarantees (11 9's). The parallelism of disaggregated storage enables higher throughput than local storage devices, which we will discuss in Section 2.8.

> **Finding 1:** Cloud object storage provides the best durability guarantees while being the cheapest storage option.

2.3 Latency

Different request sizes. Disaggregated storage incurs higher latency than SSD-based storage solutions. We examine the latency distribution for different request sizes to understand storage latency. We distinguish between total duration and latency until the first byte is retrieved. The results of using only a single request at a time are depicted in Figure 2. We differentiate between the first and 20th consecutive iteration to simulate hot accesses. Our experiment shows that first byte latency often dominates the overall runtime for small sizes. First byte and total duration are similar for small request sizes. This highlights that round-trip latency limits the overall throughput. For sufficiently large requests, bandwidth is the limiting factor. From 8 to 16 MiB, we see minor improvements but the duration already rises by ~1.9× while object size doubles. Increasing the size from 16 to 32 MiB results in doubling the retrieval duration. Thus, the bandwidth limit is reached, and further increasing the size does not benefit the retrieval performance. When data is hot, first byte and total latency are generally reduced.

Noisy neighbors. Cloud-based storage solutions are shared between customers, resulting in less predictable latency. We continuously retrieve a single object from a set of objects with one request to analyze trends in access performance. We generate random 16 MiB objects since increasing the size does not lead to a lower latency per byte. Figure 3 shows the bandwidth for accessing an object (bytes divided by duration) over a period of 8 weeks. Object bandwidth has a high variance ranging from ~25 to 95 MiB/s, with a considerable number of data points being at the maximum (15%). The median performance stabilizes at 55-60 MiB/s. Weekly patterns in the data show that the bandwidth is influenced by the day of the week. Especially at the weekends (first day of the week is Monday), the performance is higher – most likely due to lower demand from other customers. When we zoom into one week, clear daily patterns are visible. The performance fluctuations between day and night indicate variations in network utilization during different times of the day. Surprisingly, no outlier lies above the large cluster at ~95 MiB/s even though millions of objects were downloaded. This suggests that the per-request bandwidth is limited within S3 or that server-side caching effects are intentionally not passed on to users.

Latency variations between cloud vendors. In addition to using AWS, we also examine latency characteristics of two other cloud

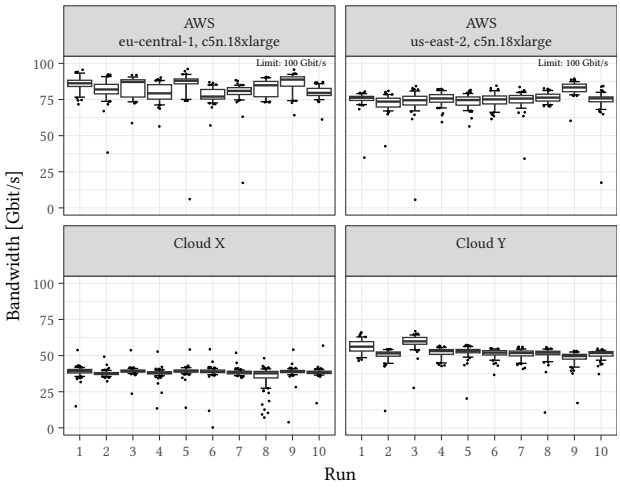

Figure 5: Throughput distribution of different object stores over multiple runs on sparsely accessed data (12h interval).

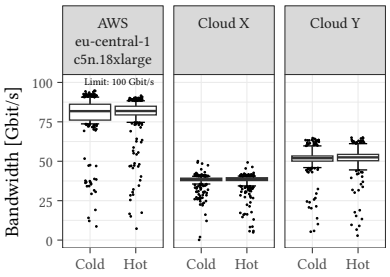

Figure 6: Throughput comparison of cold and hot runs.

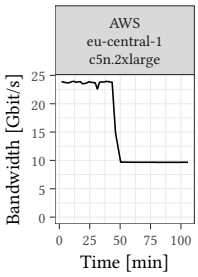

Figure 7: Instance burst bandwidth.

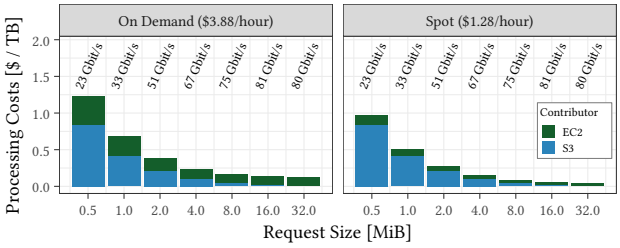

Figure 8: Cost vs. throughput of different request sizes (AWS, eu-central-1, c5n.18xlarge).

providers. The experiment, plotted in Figure 4, accesses randomly generated 16 MiB objects. After each run, the bucket is not accessed until the next run. The interval between executions is (at least) 12 hours to reduce caching effects. AWS S3 has the highest overall latency for individual objects. The other two providers have similar average latencies, but Cloud Y has more variance. Latency between different executions is fairly stable across all cloud providers. As mentioned, S3 has a minimum latency with no outliers below it, which suggests a restricted per-request maximum bandwidth. In contrast to AWS, outliers in the low latency spectrum indicate that the other vendors do not hide caching effects.

2.4 Throughput

Importance of throughput. Aside from latency, we also show insights on the throughput of accessing object stores. For analytics, the most important factor is the combined throughput since OLAP requires large amounts of data to be processed. Thus, the first byte latency is less important for bandwidth-dominated workloads.

Cloud storage throughput similar to instance bandwidth. Similar to our previous latency experiment, we access randomly generated 16 MiB objects. One request retrieves one complete object. In this experiment, we maximize the throughput available on each cloud provider with a single instance. We schedule up to 256 simultaneous requests using many threads to maximize throughput. Further increasing requests did not lead to higher throughput. Section 2.8 discusses the optimal number of requests. We use instances that achieve up to 100 Gbit/s (or the cloud's maximum bandwidth) and have similar on-demand pricing. Figure 5 shows the throughput experiment with (at least) 12 hours between different runs to reduce caching effects. Each throughput data point is calculated as an aggregate of all downloaded objects over a 1-second window. The results show that we achieve a median bandwidth of at least 75 Gbit/s for AWS. Most runs have a median bandwidth between 80 and 90 Gbit/s in eu-central-1, close to the maximum instance bandwidth. At Cloud X, we observe a bandwidth limit of ~40 Gbit/s and almost no fluctuations. Cloud Y achieves a median bandwidth of 50 Gbit/s to its object store, but we notice higher variance.

Different regions have slightly different performance. Throughput is similar for the two tested regions of AWS; however, one region performs slightly better. The difference between the two regions does not vary much between iterations.

High bandwidth is achievable for cold objects. In Figure 6, we investigate the throughput differences between the first and the 20th consecutive execution. The access frequency spike of the same objects does not result in vastly different execution times.

Small instances allow bursting. In the AWS instance specifications, the network bandwidth of smaller instances is often denoted with an up-to bandwidth limit. Instances achieve the baseline bandwidth (relative to the number of CPUs) in the steady state after utilizing all burst credits [14]. Figure 7 shows that the instance falls back to the baseline throughput after bursting for ~45 min.

> **Finding 2:** Object retrieval can reach network bandwidth.

2.5 Optimal Request Size

Request size implications. An important design decision is the size of requests. Requests can either be full objects or byte ranges within objects. The most crucial factors are performance and request cost. Since cloud providers charge by the number of requests, larger requests result in lower cost for the same overall data size. On the other hand, the size should be as small as possible so that small tables also benefit from parallel downloads. Our experiments in Section 2.3 demonstrate that performance does not improve beyond the bandwidth limit for a single request.

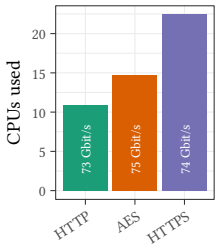

Figure 9: Impact of encryption on CPU usage.

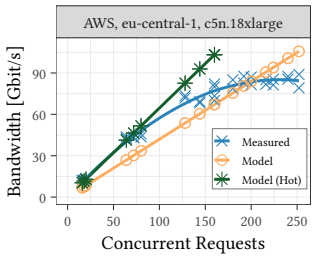

Figure 10: Request modeling for reaching throughput goal.

Cost-throughput optimal requests. In Figure 8, we show the cost of retrieving data from S3 with different request sizes. The achieved throughput with hundreds of simultaneous requests and many threads is denoted above the bars. Each request size class contains randomly generated objects. We distinguish between compute instance cost (c5n.18xlarge) and storage retrieval cost. Storage cost dominates the total cost for small objects. Computational cost is the most significant contributor to requests in the ~10 MiB range. This applies to instances at on-demand prices and spot instances, which come at a huge discount (we calculate with 60%). Because the throughput plateaus in the same range of request sizes, we classify request sizes of 8 - 16 MiB as cost-throughput optimal for OLAP.

> **Finding 3:** Sizes of 8 - 16 MiB are cost-throughput optimal.

2.6 Encryption

CPU consumption of encryption. All experiments so far use an unsecured connection to S3 (HTTP), but S3 also supports encrypted connections through HTTPS. We measure the CPU overhead of different encryption strategies while reaching the same throughput in Figure 9. HTTPS requires more than 2× CPU resources of HTTP, but AES end-to-end encryption only increases CPU usage by ~30%.
Encryption-at-rest superior to HTTPS. At AWS, all traffic between regions and even all traffic between AZs is automatically encrypted by the network infrastructure. Thus, all traffic leaving an AWS physical location is automatically secured [8]. Within a location, no other user is able to intercept traffic between an EC2 instance and the S3 gateway due to the isolation of VPCs, making HTTPS superfluous. However, encryption-at-rest is required to ensure full data encryption outside the instance (e.g., at S3).

2.7 Tail Latency & Request Hedging

Hedging against slow responses. Missing or slow responses from storage servers are a challenge for users of cloud object stores. In our latency experiments, we see requests that have a considerable tail latency. Some requests get lost without any notice. To mitigate these issues, cloud vendors suggest restarting unresponsive requests, known as request hedging [10, 34]. For example, the typical 16 MiB request duration is below 600ms for AWS. However, less than 5% of objects are not downloaded after 600ms. Missing responses can also be found by checking the first byte latency. Similarly to the duration, less than 5% have a first byte latency above 200ms. Hedging these requests does not introduce significant cost overhead.

2.8 Model for Cloud Storage Retrieval

Concurrency analysis. During our analysis, we saw that the bandwidth of individual requests is similar to accessing data on an HDD. To saturate network bandwidth, many simultaneous requests are required. Requests in the range of 8 - 16 MiB are cost-effective for analytical workloads. We design a model to predict the number of requests needed to reach a given throughput goal:

$$\text{requests} = \text{throughput} \cdot \frac{\text{baseLatency} + \text{size} \cdot \text{dataLatency}}{\text{size}}$$

For sufficiently large request sizes at S3, we calculate the median base latency as ~30 ms and the median data latency as ~20 ms/MiB The base latency is computed from the 1 KiB experiment in Figure 2, the average latency of 16 MiB minus the base latency defines the median data latency. Figure 4 shows that the median data latency of Cloud X and Cloud Y is lower (12–15 ms/MiB). For S3, the optimal request concurrency for saturating 100 Gibt/s instances is ~200–250. Figure 10 evaluates the model with the previous data latency and the latency representing the 25^{th} percentile (hot). The measurements are between both models until the bandwidth limit is reached.
Storage medium. An access latency in the tens of ms and a per-object bandwidth of ~50 MiB/s strongly suggest that cloud object stores are based on HDDs. This implies that reading from S3 with ~80 Gbit/s is accessing on the order of 100 HDDs simultaneously.

> **Finding 4:** Saturating high-bandwidth networks requires hundreds of outstanding requests to the cloud object store.

3 ANYBLOB

Unified interface with smaller CPU footprint. Different cloud providers have their own download libraries with different APIs and performance characteristics [7, 40, 44, 61, 65]. To offer a unified interface, we present a general-purpose and open-source object download manager called *AnyBlob* [36]. In addition to transparently supporting multiple clouds, our `io_uring`-based download manager requires fewer CPU resources than the cloud-vendor-provided ones. Resource usage is vital as our download threads run in parallel with the query engine working on the retrieved data. Existing download libraries start new threads for each parallel request. For example, the S3 download manager of the AWS SDK executes one request per thread using the open-source HTTP library `curl`. In contrast to spinning up threads for individual requests, *AnyBlob* uses asynchronous requests, which allows us to schedule fewer threads. Because hundreds of requests must be outstanding simultaneously in high-bandwidth networks, a one-to-one thread mapping would result in thread oversubscription. This results in many context switches, which negatively impacts performance and CPU utilization.

3.1 AnyBlob Design

Multiple requests per thread. *AnyBlob* uses `io_uring` to manage multiple connections per thread asynchronously [31]. With this model, the system does not have to oversubscribe threads which would incur additional scheduling cost. In the following, we discuss the three major components of *AnyBlob*. The components and their relationship are shown in Figure 11.
io_uring - low-overhead system call interface. `io_uring` (available since Linux kernel 5.1) provides a generic kernel interface for

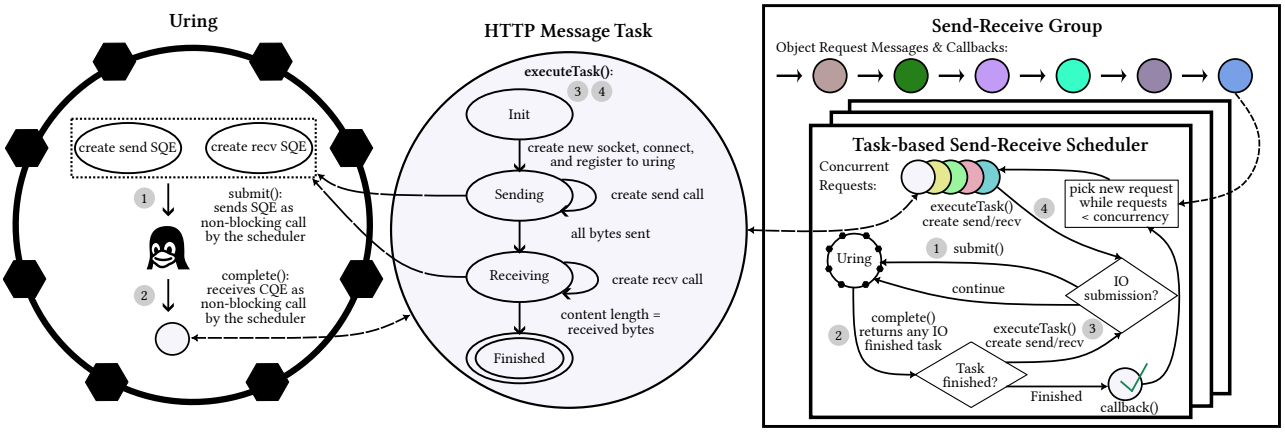

Figure 11: AnyBlob uses state-machine based message tasks that are asynchronously processed with the help of `io_uring`.

storage and network tasks. It builds on two lock-free ring buffers, the submission and completion queues, that are shared between user and kernel space. The user inserts new submission queue entries (SQE), such as receive (recv) and send operations, into the submission queue. Inserting into the queue does not require any syscalls. To notify the kernel of new entries in the submission queue, the `io_uring_enter` system call processes the entries on the kernel side in a non-blocking fashion until the request is transmitted to the network or storage device. This device uses interrupt requests (IRQs) to notify the kernel of finished operations. The request is then processed during the interrupt and placed on the completion queue. To check if a request was successful, the user periodically peeks for available completion queue entries (CQE). `io_uring` was found to be highly efficient for storage applications [35, 41, 52, 55, 68], but is less studied for networking tasks [28]. Didona et al. suggest to study `io_uring` for networking in more depth [35].

State-machine-based messages. *AnyBlob* uses a state machine for each request. The message information (address, port, and raw data) combined with a state machine is denoted as a *Message Task*. Optionally, a receive buffer can be attached that avoids additional data copies since the kernel transfers data directly from the network device to our desired location. Cloud object stores use HTTP messages to transfer data. We implement the different phases of an HTTP request within the state machine. On successfully completing a phase, we transition to the next phase until the object is fully fetched. The state machine enables asynchronous and multiplexed messages with a single thread. Several send and recv system calls are required during transfer until the object is downloaded. After each system call, we suspend the execution of this message until we are notified about the successful syscall. Afterward, we reevaluate the state machine until a final state is reached.

Asynchronous system calls. Our asynchronous handling of send and recv system calls in the *Message Task* is facilitated by `io_uring`. Instead of directly scheduling the system call and waiting for the result, we insert the operation into the submission queue of the uring. Each SQE has a user-defined member that allows passing information to later identify its origin *Message Task*. System calls are processed only when the submission queue is submitted to the kernel (1). This submission process is non-blocking, allowing the executing thread to work on other requests while the transfer is handled by the network device. The uring is periodically checked for available completion queue entries (CQE) (2). When a CQE is available, a system call has been processed. With the retrieved information, we can evaluate the next *Message Task* step.

Task-based send-receive scheduler. With `io_uring`-based sockets and *Message Tasks*, we develop a task-based send-receive scheduler. The task scheduler uses one thread that continuously executes 1 – 3 as an event loop. This event loop coordinates the execution of the steps of *Message Tasks* (3) and processes completion entries (2). Furthermore, new object requests are scheduled as new *Message Tasks* (4). To optimize single-threaded throughput, a task scheduler works concurrently on multiple *Message Tasks*. Multiple *Message Tasks'* send and recv system calls can be batched before submitting the submission queue to reduce system call overhead (1). In multi-threaded environments, it is beneficial to reduce system calls as parts of them are protected by kernel locks. When a *Message Task* is finished, it invokes a callback to notify the requester.

Send-receive groups. Although a single task-based send-receive scheduler has high throughput (multiple Gbit/s), it is not sufficient to satisfy network-optimized instances. Thus, multiple schedulers need to run simultaneously. For ease of use, a lock-free send-receive task group manages requests for multiple send-receive schedulers.

3.2 Authentication & Security

Transparent authentication. Although all cloud providers use a similar API to access objects, some details of signing requests and the authentication are different. *AnyBlob* implements operations to upload and download objects from multiple cloud storage providers. We implement a custom signing process using the library `openssl` to maintain high throughput with as few cores as possible [20]. For users of *AnyBlob*, it is transparent which provider is chosen, as the interaction with the library remains unchanged. For AWS, we support the automatic short-term key metadata service [11].

AnyBlob enables encryption-at-rest. *AnyBlob* supports the user application to use encryption-at-rest by providing easy-to-use, in-place, and fast encryption and decryption functions for AES. Further, *AnyBlob* allows the usage of HTTPS for requests. However,

we discourage this in controlled environments, such as AWS EC2 connected to AWS S3, due to high CPU overhead. HTTPS is useful for authentication if data is sent outside the controlled environment, e.g., from your computer to S3. In contrast to the high overhead for HTTPS, encryption-at-rest can be used with only moderate overhead. As shown in Section 2.6, this client-side encryption provides superior encryption against third parties, e.g., cloud providers.

3.3 Domain Name Resolver Strategies

Resolution overhead. In analytical scenarios, many requests are scheduled to the cloud object storage. Section 2.1 highlights that we can connect to different server endpoints. Resolving a domain name for each request adds considerable latency overhead due to additional round trips. Thus, it is essential to cache endpoint IPs.
Throughput-based resolver. Our default resolver stores statistics about requests to determine whether an endpoint is performing well. We cache multiple endpoint IPs and schedule requests to these cached IPs. If the throughput of an endpoint is worse than the performance of the other endpoints, we replace this endpoint. Thereby, we allow the load to balance across different endpoints.
MTU-based resolver. We found that the path maximum transmission unit (MTU) differs for S3 endpoints. In particular, the default MTU to hosts outside a VPC is typically 1500 bytes. Some S3 nodes, however, support Jumbo frames using an MTU of up to 9001 bytes [9]. Jumbo frames reduce CPU cost significantly because the per-packet kernel CPU overhead is amortized with larger packets.
MTU discovery. The S3 endpoints addressable with a higher path MTU use 8400 bytes as packet size. Our AWS resolver attempts to find hosts that provide good performance and use a higher path MTU. We ping the IP with a payload (> 1500 bytes) and set the DNF (do not fragment) flag to determine if a higher path MTU is available.

3.4 Performance Evaluation

Competitors. To demonstrate *AnyBlob*'s performance and CPU usage utilization, we experiment with different settings on AWS. We compare against two libraries provided by Amazon. They are both part of the official AWS C++ SDK (1.9.140). S3 is the traditional API that uses the library `curl` internally to retrieve objects. Similar concepts are applied by the download managers of other vendors' SDKs. S3Crt is a newer alternative S3 library released by AWS that uses a custom C network implementation (C++ API). With *AnyBlob*'s design, S3 Select can be implemented, but it would only support few types (JSON, CSV, Parquet) and no client-side encryption [15].
Cost-throughput Pareto-optimal retrieval. Figure 12 shows different settings for each tested download manager. Note that we plot performance and CPU utilization such that the optimal settings lie in the top-left corner of the Pareto curve. Within one download strategy, we highlight the points on their respective Pareto curve. *AnyBlob*, with our throughput-based resolver, always dominates the AWS-provided download managers. We achieve the same maximum throughput using only 0.7× the CPU resources of the best competitor. Given a fixed CPU budget, we get up to 1.5× performance. Our specialized AWS resolver achieves the same throughput but reduces CPU usage by an additional 10%. We validated *AnyBlob* on recently deployed Graviton instances (200 Gbit/s) [5] and observed greater CPU reduction while retrieving objects with up to 180 Gbit/s.

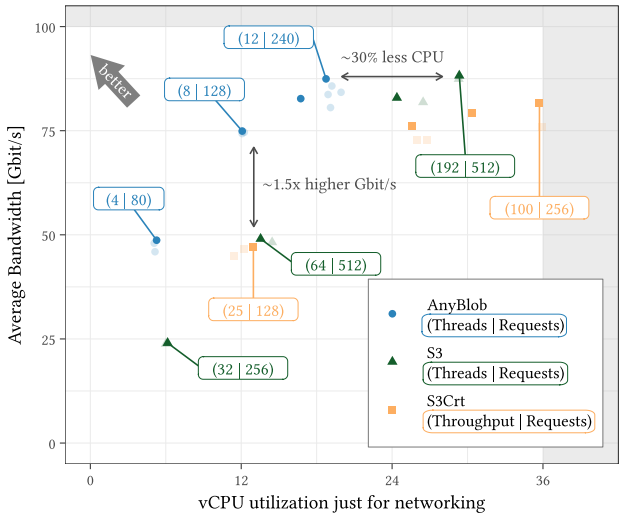

Figure 12: Throughput and CPU usage Pareto curves for AnyBlob, S3, and S3Crt (AWS, eu-central-1, c5n.18xlarge).

4 CLOUD STORAGE INTEGRATION

Query engine integration options. To unleash the full performance potential of disaggregated cloud storage, we have to carefully integrate the analytical query engine with the networking components. A naive approach would let each worker thread download its currently-needed data chunk synchronously. This way, each worker thread would schedule at most one request at a time, but the threads would be blocked most of the time – waiting for network I/O. A more common approach in database systems is the usage of asynchronous I/O. Our cloud storage retrieval approach builds upon this common I/O strategy. Database systems that use the AWS S3 SDK [7] also leverage asynchronous retrieval from cloud object storage. As discussed in Section 3, the AWS S3 SDK often results in oversubscription, which has not only a negative impact on performance but also other undesirable effects on database systems. For example, a huge download task with hundreds of threads could make the DBMS unresponsive to newly arriving queries since the DBMS has no control over the retrieval threads. Furthermore, the mix of downloading and processing threads is hard to balance, especially with this vast number of concurrently active threads.
Approach. In this section, we show how to integrate efficient object store retrieval into high-performance query engines. We rely on *AnyBlob* and the empirical results presented in Section 2 to saturate the available network bandwidth with low CPU resource consumption. A key challenge is how to balance query processing and downloading. Without enough retrieval threads, the network bandwidth limit can not be reached. On the other hand, if we use too few worker threads for computation-intensive queries, we lose the in-memory computation performance of our DBMS. We, therefore, propose a scheduling component to balance object store retrieval and query processing, allowing us to schedule threads effectively in terms of query performance and CPU usage. With this scheduler, we then develop an efficient table scan operator based on a cost-effective columnar storage format.

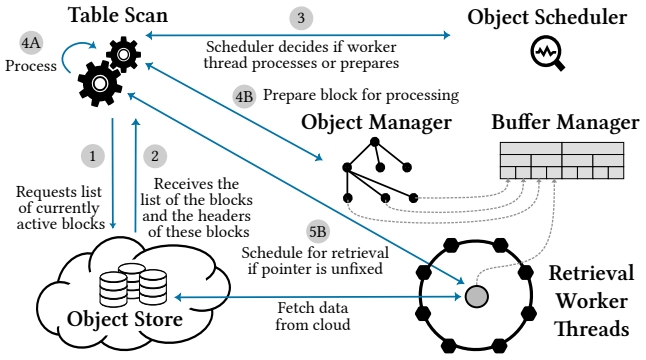

Figure 13: DBMS design overview for efficient analytics with the flow of information between different components.

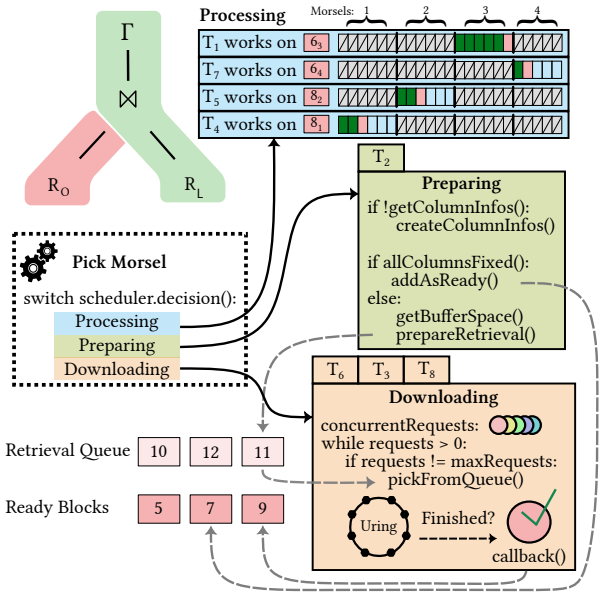

Figure 14: Table scan example with 8 threads.

4.1 Database Engine Design

Tasks and scheduling of worker threads. The overall design of our cloud storage-optimized DBMS centers around the table scan operator. Like most database systems, our system Umbra uses a pool of worker threads to process queries in parallel. In our design, worker threads do not only perform (i) regular query processing, but can also (ii) prepare new object store requests or (iii) serve as network threads. Our object scheduler, which we present in Section 4.3, dynamically determines each worker's job (i-iii) depending on network bandwidth saturation and processing progress.

Task adaptivity. To overcome issues with long-running queries that block resources, many database systems use tasks to process queries. These tasks can either be suspended or run only for a small amount of time. Both concepts lead to a query engine that is able to adapt to changing workloads quickly. Regardless of the specific task system, our asynchronous retrieval integration only requires the mechanism to switch tasks of workers during query runtime.

Columnar format. The raw data is organized in a column-major relation format chunked in immutable blocks of columns. The metadata of a block, e.g., column types and offsets, are stored in the block header. The database schema information is also stored on cloud storage, which requires fetching at start-up.

Table metadata retrieval. In the following, we describe the flow of information during a table scan operation, illustrated in Figure 13. In steps 1 and 2, the scan operator first requests table metadata, i.e., the list of blocks. Afterward, all relevant block metadata is downloaded as a requirement to start the table scan's data retrieval.

Worker thread scheduling. After initializing the table scan, we dedicate multiple worker threads to this operation. Because partitioning worker threads into retrieval and processing threads is difficult and requires adaptations over the duration of the query, we implement an object scheduler to solve this problem. Step 3 shows that each scanning thread asks the scheduler which job to work on. If enough data is retrieved, the worker thread proceeds to process data, as demonstrated in 4A. Otherwise, we dedicate the thread to preparing blocks for retrieval. Since we only execute jobs for a short time, this decision can be quickly adapted.

Download preparation. To saturate the network bandwidth, it is important to continuously download with enough retrieval threads and many outstanding requests. In Step 4B, the preparation worker creates new requests that allow the retrieval threads to execute their event loop without interruption. The object manager holds metadata of tables, blocks, and their column chunk data. The column chunk data is managed by our variable-sized buffer manager. If the data is not in memory, we create a new request and schedule it for retrieval, shown in 5B. Finally, retrieval threads fetch the data.

4.2 Table Scan Operator

Scan design preliminaries. We carefully integrate *AnyBlob* into our RDBMS Umbra, which compiles SQL to machine-code and supports efficient memory and buffer management [38, 48, 63]. Umbra uses worker threads to parallelize operators, such as table scans, and schedules as many worker threads as there are hardware threads available on the instance. If there is only one active query, all workers are used to process that specific query. Umbra's tasks consist of morsels which describe a small chunk of data of the task [53]. Worker threads are assigned to tasks and process morsels until the task is finished or the thread is assigned to a different task.

Morsel picking. After Umbra initializes the table scan, the worker threads start calling the `pickMorsel` method. This function assigns chunks of the task's data to worker threads. This is repeated after each morsel completion as long as the thread continues to work on this table scan task. The only difference in our approach is that our workers do not only need to process data but also prepare new blocks or retrieve blocks from storage servers. Our object scheduler, which we explain in Section 4.3, decides the job of a worker thread based on past processing and retrieval statistics. Note that similar to our `pickMorsel`, every task-based system has a method that determines the next task of a worker thread.

Worker jobs. If a thread is assigned to process data, a morsel is picked from the currently active block in `pickMorsel`. In contrast to the processing job, the other jobs (preparation and retrieval) do not pick a morsel for scanning. Instead, these jobs start routines

Algorithm 1: Scheduler: Adaptivity Computation

```
1 retrieveSpeed = statistics[epoch].retrievedBytes / statistics[epoch].elapsed
2 processSpeed = (workerThreads - currentRetriever) *
    statistics[epoch].processedBytes / statistics[epoch].processedTime
3 ratio = processSpeed / retrieveSpeed
4 requiredBandwidth = min(bandwidth, bandwidth * ratio)
5 requiredRetrieverThreads = min(maxRetrievers * ratio, maxRetrievers)
```

that are required to prepare or retrieve blocks. Regardless of the job, all workers return to `pickMorsel` to get a new job assigned after finishing their current work.

Scan example overview. Figure 14 shows the full table scan operation with multiple (8) active threads working on different jobs. In the example table scan, 4 threads are dedicated to processing data, 3 for data retrieval, and 1 for preparing new blocks.

Processing job. After receiving a morsel for processing, the thread scans and filters the data according to the semantics of the table scan. When all morsels of an active block (global or thread-local with stealing) are taken, the thread picks the morsel from a new, already retrieved block. In the example, each block is divided into 4 non-overlapping morsels. Each thread works on its unique morsel range.

Preparation job. With the already retrieved table metadata, threads prepare new blocks and register unknown blocks in the object manager. If the data of all columns currently resides in physical memory, the preparing thread marks the block as ready. Otherwise, the preparing thread gets free space from the buffer manager for each unfixed column. With the block metadata (column type, offset, and size), HTTP messages for fetching columns from cloud storage are created. After that, the block is queued for retrieval, where the data is downloaded.

Retrieval job. In the example, three threads are scheduled to act as *AnyBlob* retrieval threads. After finishing the download of a block's column chunk, a callback is invoked and marks this column as ready. Only if all columns have been retrieved, we mark the block as ready. Note that different retrieval threads may download column chunks from the same block concurrently. The worker finishes when *AnyBlob*'s request queue gets empty. Because threads always try to keep the queue at its maximum request length, unnecessary retrieval threads will eventually encounter an empty queue and stop downloading. These threads can then be reused to work on different jobs, such as processing or preparing new blocks. As long as enough requests are in the queue, the threads constantly retrieve data.

4.3 Object Scheduler

Balance of retrieval and processing performance. The main goal of the object scheduler is to strike a balance between processing and retrieval performance. It assigns different jobs to the available worker threads to achieve this balance. If the retrieval performance is lower than the scan performance, it increases the amount of retrieval and preparation threads. On the other hand, reducing the number of retrieval threads results in higher processing throughput. Note that the retrieval performance is limited by the network bandwidth, which the object scheduler considers.

Processing and retrieval estimations. The decision process requires performance statistics during retrieval and processing. Each processing thread tracks the execution time and the amount of data processed. The aggregated data allow us to compute the mean processing throughput per thread. For the network throughput, we aggregate the overall retrieved bytes during our current time epoch.

Balancing retrieval threads and requests. Sections 2.8 and 3.4 analyze how many concurrent requests are needed to achieve our throughput goal and the corresponding number of *AnyBlob* retrievers. We track the number of threads used for retrieval and limit it according to the instance bandwidth specification. By counting the number of outstanding requests (e.g., column chunks), we compute an upper bound on the outstanding network bandwidth. An outstanding request is a prepared HTTP request currently downloaded or awaiting retrieval. Because the number of threads and the outstanding requests limit the network bandwidth, our object scheduler always requires that the outstanding bandwidth is at least as high as the maximum bandwidth possible according to the current number of retrieval threads. Hence, it schedules enough preparation jobs to match the number of retrieval threads.

Performance adaptivity. The scheduler computes the global ratio between processing and retrieval to balance the retrieval and processing performance. This ratio is used to adapt the number of retrieval threads and the outstanding bandwidth. If processing is slower, fewer blocks are prepared, and fewer retrieval threads are scheduled. Some of the running retrieval threads will stop due to fewer outstanding requests. These threads are then scheduled as processing workers, increasing the global processing performance. Algorithm 1 shows these adaptivity computations.

Overpreparation. Because it is undesirable to stall retrieval threads due to unprepared columns, overpreparation is encouraged. Our scheduler ensures that up to 2× of the required bandwidth is outstanding and schedules preparation jobs accordingly.

Fast statistics aggregation. Lock-free atomic values for statistics and global counters provide fast object scheduler decisions. For every new scan request, we update the epoch to store representative statistics of the current workload.

4.4 Relation & Storage Format

Columnar format. To leverage the cost-throughput optimal download sizes, we require a column-major format that is chunked into different blocks. The database format is adapted from data blocks [51]. For each column chunk, we store min and max values in the metadata, enabling us to prune unnecessary blocks early. Our blocks use low-overhead byte-level encodings, e.g., frame-of-reference and dictionaries, to reduce storage requirements.

Tuple count in blocks. For cost-effective downloading, each column chunk of a block should have a desired size of 16 MiB. As query processing usually works on a block granularity, all columns within one block need to have the same number of tuples. However, this results in imperfect column chunk sizes due to different datatype sizes and our byte-level encoding scheme. The range per tuple in an encoded column is between 1 and 16 bytes, excluding the variable-sized columns. Because of this wide byte spread, we need to balance the sizes of the individual column chunks by optimizing the tuple count. During block construction, we adaptively compute mean tuple counts such that no encoded column falls below ~2 MiB to limit retrieval cost. Some fixed-sized and variable-sized columns may exceed 16 MiB, which is undesirable for retrieval. To avoid

large differences in download latency between columns, Umbra splits larger column chunks into multiple smaller range requests.
Zero user-space copies. Our implementation is tightly coupled with the buffer manager to reduce copies of data. The blocks of data are aligned to the page sizes of the buffer manager, but we reserve space for the HTTP header and the chunk size of the recv system call. By using the result data offset, we avoid user-space data copies.
Transparent paging. We extend the buffer manager with anonymous pages not backed by files to take advantage of the paging and in-memory buffer management features. If a new retrieval on the same page is necessary, we check if the page is still available. If not, we download the data again. With this unified and transparent buffer manager, we avoid retrieval and buffer space trade-offs.
Structure of metadata. Figure 15 shows the object structure in the cloud object storage. Within the database prefix, we store the schema information that contains all the necessary information to initialize the database. Each table has its own subprefix, which contains a list of headers, headers, and data blocks. Because header objects are also cost-throughput optimized, we store fewer header objects than blocks, as each header object contains multiple block headers. The data is organized for append-only storage, which mimics most analytical engines. Because objects can be replaced atomically in cloud storage, updating the list of headers creates consistent data snapshots. Versioning the metadata is common in cloud DBMSs to provide consistent views of the data. Apache Iceberg [17] and Data Lake [18] use an analogous technique. Iceberg's manifest files are similar to our list of headers and header objects [43].
Scan optimizations. Our implementation checks a header's min/-max values to avoid unnecessary downloads. A block is only scheduled for retrieval if all table scan restrictions match the min/max values within the block metadata. Before scanning the encoded data, the processing thread has to decode the data. We repeatably fill a small chunk with decoded data and process it. Umbra can either decode the data entirely or only decode tuples that satisfy the restrictions. Both approaches leverage vectorized SIMD instructions.

4.5 Encryption & Compression

Size reduction with strong compression. Although the bandwidth to external storage is high, modern engines might still wait on data arrival. With the encoding schemes presented in Section 4.4, the size of the columns is already reduced. Additional stronger compression allows for reducing them further. We use bit-packing for integer-encoded columns and apply LZ4 on the remaining ones.
Security due to encryption-at-rest. As already described in Section 2.6, encryption-at-rest does not only secure the traffic in transit but also stores the data inaccessible to third-parties. Before uploading a column, we use *AnyBlob* to encrypt the individual columns of a block. Although encryption with AES has a slight performance penalty, most real-world users prefer the gained security benefits.

5 EXPERIMENTAL EVALUATION

Setup. We extended our high-performance database system Umbra to support efficient analytics on disaggregated cloud object stores. All experiments are conducted at AWS in region *eu-central-1*. Unless otherwise noted, we use a single *c5n.18xlarge* (72 vCPUs / 36 cores, 192 GiB main memory, 100 Gbit/s network) instance with Ubuntu.

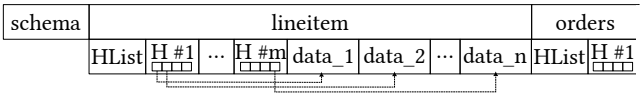

Figure 15: Object structure overview on S3 for TPC-H.

5.1 Data Retrieval Performance

Comparison with in-memory cached data. In order to analyze the retrieval capabilities of Umbra, we perform self-tests against a fully in-memory version of Umbra on the popular TPC-H benchmark. Although storing only the current query data is sufficient, we are restricted to scale factor 500 to fit all query data into the memory of our in-memory version. Table 2 shows the performance of the remote-only (no caching of buffer pages) and the in-memory version of our database, the end-to-end bandwidth, and the cost of the remote-only version. As mentioned, our remote-only version ignores buffered pages and retrieves all required data from remote storage. The bandwidth is computed by a sum of the retrieved data divided by the total query runtime, which serves as a lower bound.
Processing at instance bandwidth. Queries can be separated into retrieval-heavy and computation-heavy ones. The bandwidth is a good indicator for categorizing the queries. For example, Queries 1, 6, and 19 are the strongest representatives of the retrieval-heavy group. Umbra achieves an end-to-end bandwidth of up to 78 Gbit/s which is close to the limit. However, the factor between the in-memory and the remote execution time is large because Umbra could process more tuples than the network can provide.
No overhead for computationally-intensive queries. On the other hand, we observe only minor differences between the in-memory and remote-only versions for computationally intensive queries. For example, Queries 9 and 18 have a factor of $\leq 1.3\times$. Because the DBMS is at its processing limit due to intensive joins and aggregations, fetching of blocks is not very noticeable.
Effective scheduling. This shows the effectiveness of our scheduling algorithm. If the query is retrieval intensive, we saturate network bandwidth while continuing to process data. On the other hand, if Umbra is limited by computation, our scheduler does not waste CPU resources on idle downloading processes.
Spot instances. In the remote Umbra scenario, spot instances can be leveraged without any performance cliffs. However, additional safeguards need to be in place due to early instance termination. Queries affected by termination might require restarts, and commit persistence must be guaranteed.

5.2 Retrieval Manager Study

Different retrieval managers on chokepoint queries. To demonstrate the properties of our design and validate our *AnyBlob* results, we test different retrieval options within Umbra. We test our DBMS on EBS (gp3, no page cache) and on cloud object storage (no object cache). For retrieving data from S3, we implemented three different strategies. First, we use the worker threads to download their currently required object from remote storage with the AWS S3 library. The second strategy uses our asynchronous retrieval integration design, shown in Section 4, and combines it asynchronously with the AWS library. The last configuration leverages our integration and *AnyBlob* (Sections 3 and 4). To demonstrate

Table 2: In-memory and remote-only Umbra comparison demonstrates small cloud retrieval overhead (SF 500, c5n.18xlarge).

Query	GM	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
In-Memory [s]	2.03	1.14	0.38	2.93	2.08	3.35	0.52	2.73	3.38	10.61	4.27	0.25	1.99	9.50	1.35	0.99	1.81	1.36	18.91	0.74	1.45	6.04	1.75
Remote [s]	4.94	3.52	1.97	5.87	4.18	5.77	2.47	6.41	6.86	13.34	7.68	1.14	4.74	12.47	4.15	3.97	2.42	4.63	22.20	3.82	5.06	12.24	2.54
Factor	2.42	3.08	5.16	2.01	2.01	1.72	4.78	2.35	2.03	1.26	1.80	4.58	2.39	1.31	3.07	4.01	1.34	3.41	1.17	5.15	3.50	2.03	1.45
Gbit/s	49.80	75.00	46.00	55.76	55.95	65.20	77.73	64.43	69.40	40.67	52.42	40.73	62.01	30.86	64.63	67.35	14.13	73.65	15.41	76.87	66.34	65.35	23.20
Cost S3 [¢]	0.15	0.29	0.04	0.21	0.15	0.20	0.17	0.23	0.24	0.31	0.27	0.02	0.23	0.28	0.17	0.17	0.02	0.21	0.22	0.25	0.21	0.43	0.03
Cost EC2 [¢]	0.53	0.38	0.21	0.63	0.45	0.62	0.27	0.69	0.74	1.44	0.83	0.12	0.51	1.34	0.45	0.43	0.26	0.50	2.39	0.41	0.55	1.32	0.27

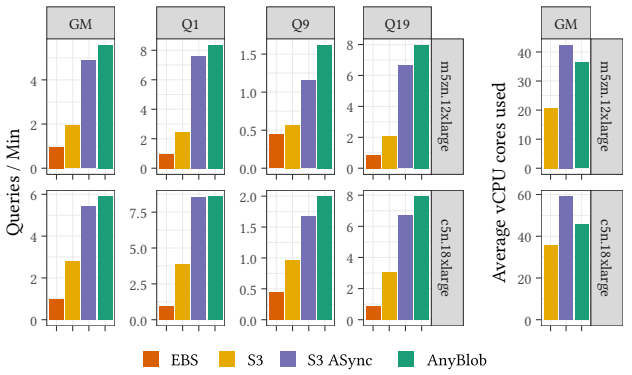

Figure 16: Internal comparison of Umbra on EBS, and on S3, + ASync (Sec. 4), + AnyBlob (Sec. 3) (SF 1000, 2 instance types).

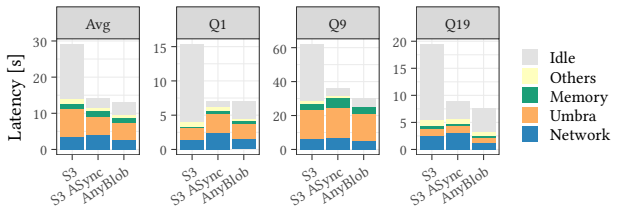

Figure 17: CPU usage traces for different networking implementations collected with Linux perf (SF 1000, c5n.18xlarge).

our cloud storage performance, all remaining experiments force Umbra to ignore columns already available in the buffer manager. Umbra always fetches these columns from remote storage.

Higher throughput while reducing CPU usage. In Figure 16, we test all TPC-H queries on two different machine types, both supporting 100 Gbit/s networking. EBS has the worst throughput due to the bandwidth limit of 1 GB/s. Asynchronous retrieval of more requests than cores is crucial for performance. By simply swapping the retrieval library from the asynchronous AWS SDK to *AnyBlob*, Umbra achieves up to a factor of 1.2× better geometric mean performance and an improvement of up to 40% on computationally expensive queries. Additionally, *AnyBlob* reduces the mean CPU usage by up to 25%. Recent trends indicate that the networking bandwidth increases faster than the number of CPU cores, making the resource usage of networking essential [5].

Retrieval requires significant CPU resources. Figure 17 breaks the query resource CPU utilization down into fine-grained tasks, such as network I/O, memory and buffer management, and processing (similar to [66]). We used perf to trace the resource utilization of different functions and aggregate the results. Umbra achieves an average CPU utilization of ~75% with asynchronous networking. Networking uses a large share of CPU time that accounts for up to 25% of the total utilization, significantly reduced by *AnyBlob*.

5.3 Scaling Properties

Thread scaling on chokepoint queries. Since our approach is highly elastic, it is very interesting to see how Umbra scales on a varying number of cores and different instances. Figure 18 shows two chokepoint queries, which we already identified in Section 5.1. The results are measured on the same instance, but we artificially reduce the amount of parallelism within our DBMS (number of worker threads). We contrast to the aforementioned in-memory version of our system. For retrieval-heavy queries (e.g., Query 1), we can see a plateau if enough cores are available to utilize the network completely. For the in-memory version, we measure a linear increase in performance until the hyper-threading boundary is reached. The performance of the computation-heavy queries (Query 9) increases as we add more cores. The remote-only Umbra version has almost the same throughput as the in-memory version.

Instance scaling. To demonstrate our scalability on different instances, we use smaller versions of the c5n.18xlarge. The c5n.9xlarge has a maximum bandwidth of 50 Gbit/s and 36 vCPUs; the c5n.4xlarge has 16 vCPUs and 25 Gbit/s bandwidth. The additional resources of larger instances improve the query runtime. Because our approach retains performance without warm caches, we can switch to larger instances as the workload increases.

5.4 End-To-End Study with Compression & AES

Workload & competitors. In this experiment, we compare the end-to-end performance on the TPC-H benchmark. To mimic a realistic OLAP scenario analyzing large amounts of data, we test scale factors (SF) of 100 (~100 GiB) and 1,000 (~1 TiB of data). Since we optimize the retrieval properties, Umbra does not cache any data to showcase our retrieval integration. We compare against Spark on a single c5n.18xlarge instance and a large warehouse of Snowflake. In 2019, Snowflake used c5d.2xlarge instances for xsmall warehouses, which was reported by a Snowflake error log [70]. Assuming this instance type for xsmall, a large warehouse would use an instance or cluster similar to our instance but with local SSDs (e.g., c5d.18xlarge or 8 × c5d.2xlarge). For Snowflake, we measure the throughput with warm cache (multiple TPC-H runs) and on another large configuration that is shut down after each query execution to enforce remote retrieval.

Fast processing from cloud storage. Figure 20 shows the performance results of different systems. As discussed in Section 4.5,

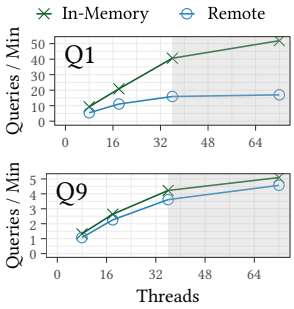

Figure 18: Scalability of queries (c5n.18xlarge).

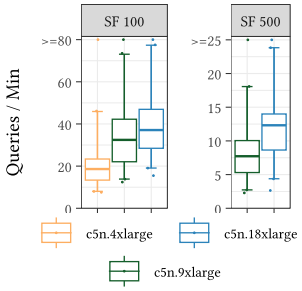

Figure 19: Scalability on different instances.

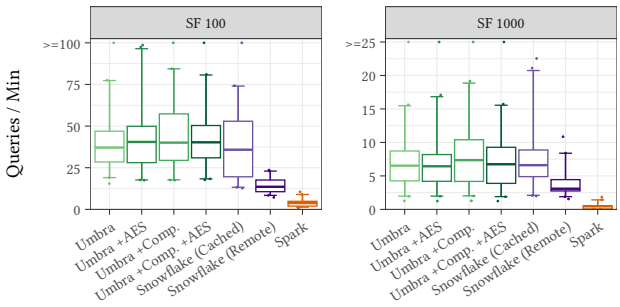

Figure 20: End-to-end system comparison on SF 100 and 1000.

Umbra is able to encrypt data automatically and implements strong compression. Compression improves performance, but encryption has a slight overhead. In a real-world scenario, we recommend using both settings for higher security without performance degradation. Although Umbra always retrieves the data from cloud storage, the performance is similar to Snowflake, which uses data caching (e.g., local SSDs). As mentioned earlier, the actual hardware configuration of Snowflake is unknown. For example, the runtime of Query 6 suggests that the instance has higher disk bandwidth than both mentioned instance settings. Clearly, these end-to-end results are influenced by the database, its execution model, and the hardware.

6 RELATED WORK

Cloud DBMS. With the dominance of the cloud for scalable solutions, many software-as-a-service database management systems emerged. Often specialized systems for either OLTP [16, 29, 30, 81] or OLAP [2, 25, 33, 58, 59, 67, 79, 87] were developed to cope with the new challenges in the cloud era [56]. Redshift [19] leverages Aqua, a computational caching layer, unaffected by resizing nodes [6]. Until recently, caching was unavoidable even for analytics dominated by the bandwidth. However, the gap between network and NVMe bandwidth is closing, making cloud storage more attractive. AWS Athena, based on Presto [73], works directly on remote data. An experimental study contrasts the architecture of these systems [77].
Processing in the cloud. Brantner et al. [27] discuss challenges and opportunities of S3 for OLTP workloads. In 2010, an experimental study provided insights into the computation power of EC2 instances; in particular, it studies the CPU resources, memory, and disk operations [72]. Our experimental study on cloud storage provides an in-depth analysis that provides all details for fast analytics on cloud storage. Leis and Kuschewski present a model for cost-optimal instance selection [54]. Although systems such as Hive and Spark can be self-hosted [78, 86], managed Hadoop is common [71].
Spot instances. Because spot instances come with huge discounts, mitigating the termination risk and hopping between instances was researched [74, 76]. Our approach is a perfect fit for spot hopping since caching is not needed for good performance. Although our experiments run faster than the termination delay of AWS (2 min) [12], a migration to another instance can retain query state [84].
Serverless computing. Serverless functions are another short-term service, which allow users to deploy resources only for the duration of a request. Since a serverless function has little memory, compute resources, and a time limit [42], many parallel function invocations are required to execute a single query. Starling [69] and Lambada [62] propose to run analytics on serverless functions. Although Lambada and Starling provide a small study on S3 in serverless environments, the characteristics are very different as these functions only have limited threads and networking (300 MiB/s), which does not require a careful retrieval design such as *AnyBlob*.
Cloud storage for DBMS. Cloud object stores attract attention as data warehouses due to their low costs. Two prominent storage solutions are Apache Iceberg and Data Lake [17, 18]. Both systems use metadata stored on the cloud object stores to provide consistent snapshots. As our storage structure is similar, our fast processing on remote data can be adapted to these storage backends. Ephemeral storage systems, such as Pocket [49], and caching for cloud storage [37, 46, 85, 89] sparked a wide variety of research. Caching solutions extend from using semantic caching on a local node [37] to leveraging spot instances as caching and offloading layer [89].
Memory disaggregation. Similar to disaggregating storage, future data centers may separate CPU from memory to improve resource flexibility. Most research finds that disaggregated memory is orthogonal to the current storage-separated design [50, 83, 90, 91].
Networking and kernel APIs. Following recent trends, future data centers will be equipped with fast Ethernet connections reaching Tbit/s [28]. OS and kernel research presents approaches to integrate these high-bandwidth network devices with low latency [28, 88]. RDMA is already explored in DBMS for fast networks with low latency [24, 47, 57, 92]. A kernel storage API study found `io_uring`, used in *AnyBlob*, to be promising [35]. Especially for fast NVMe SSDs, it is already used widespread [35, 41, 52, 55, 68].

7 CONCLUSION

This paper discusses the efficient and cost-effective usage of cloud object storage for analytics. Our first contribution is a detailed analysis on the characteristics of cloud object stores. With these insights, we developed *AnyBlob*, a modern object storage download manager based on `io_uring`. *AnyBlob* requires fewer CPU resources to achieve the same or higher throughput compared to libraries provided by cloud vendors. Finally, we demonstrated a blueprint to utilize efficient analytics on disaggregated object stores in DBMSs. Our results show that even with disabled caching, Umbra with *AnyBlob* achieves performance similar to large configurations of state-of-the-art cloud database systems that cache data locally.

REFERENCES

[1] Merv Adrian. 2022. DBMS Market Transformation 2021: The Big Picture. https://blogs.gartner.com/merv-adrian/2022/04/16/dbms-market-transformation-2021-the-big-picture/. accessed: 2022-09-30.

[2] Josep Aguilar-Saborit and Raghu Ramakrishnan. 2020. POLARIS: The Distributed SQL Engine in Azure Synapse. *Proc. VLDB Endow.* 13, 12 (2020), 3204–3216.

[3] Amazon. 2021. What's the maximum transfer speed between Amazon EC2 and Amazon S3? https://aws.amazon.com/premiumsupport/knowledge-center/s3-maximum-transfer-speed-ec2/. accessed: 2022-09-15.

[4] Amazon. 2022. Amazon S3 Storage Classes. https://aws.amazon.com/s3/storage-classes/. accessed: 2022-10-05.

[5] Amazon. 2022. Announcing Amazon EC2 C7gn instances (Preview). https://aws.amazon.com/about-aws/whats-new/2022/11/announcing-amazon-ec2-c7gn-instances-preview/. accessed: 2023-06-17.

[6] Amazon. 2022. AQUA (Advanced Query Accelerator) for Amazon Redshift. https://aws.amazon.com/redshift/features/aqua/. accessed: 2022-10-12.

[7] Amazon. 2022. AWS SDK for C++. https://github.com/aws/aws-sdk-cpp. accessed: 2022-10-05.

[8] Amazon. 2022. Encryption in transit. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/data-protection.html#encryption-transit. accessed: 2022-09-30.

[9] Amazon. 2022. Network maximum transmission unit (MTU) for your EC2 instance. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/network_mtu.html. accessed: 2022-10-11.

[10] Amazon. 2022. Performance Guidelines for Amazon S3. https://docs.aws.amazon.com/AmazonS3/latest/userguide/optimizing-performance-guidelines.html. accessed: 2022-10-11.

[11] Amazon. 2022. Retrieve security credentials from instance metadata. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/iam-roles-for-amazon-ec2.html#instance-metadata-security-credentials. accessed: 2022-10-15.

[12] Amazon. 2022. Spot Instance interruption notices. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-instance-termination-notices.html. accessed: 2022-10-08.

[13] Amazon. 2023. Amazon S3 pricing. https://aws.amazon.com/s3/pricing. accessed: 2023-06-17.

[14] Amazon. 2023. Compute optimizes instances: Network performance. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/compute-optimized-instances.html. accessed: 2023-05-02.

[15] Amazon. 2023. Filtering and retrieving data using Amazon S3 Select. https://docs.aws.amazon.com/AmazonS3/latest/userguide/selecting-content-from-objects.html. accessed: 2023-05-02.

[16] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In *SIGMOD Conference*. ACM, 1743–1756.

[17] Apache. 2022. Apache Iceberg. https://iceberg.apache.org/. accessed: 2022-09-10.

[18] Michael Armbrust, Tathagata Das, Sameer Paranjpye, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Liwen Sun, Peter A. Boncz, Mostafa Mokhtar, Herman Van Hovell, Adrian Ionescu, Alicja Luszczak, Michal Switakowski, Takuya Ueshin, Xiao Li, Michal Szafranski, Pieter Senster, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12 (2020), 3411–3424.

[19] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *SIGMOD Conference*. ACM, 2205–2217.

[20] OpenSSL Project Authors. 2022. OpenSSL - Cryptography and SSL/TLS Toolkit. https://www.openssl.org/. accessed: 2022-10-15.

[21] Jens Axboe. 2019. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf. accessed: 2022-10-12.

[22] Jeff Barr. 2019. New C5n Instances with 100 Gbps Networking. https://aws.amazon.com/blogs/aws/new-c5n-instances-with-100-gbps-networking/. accessed: 2022-09-10.

[23] Jeff Barr. 2020. Amazon S3 Update Strong Read-After-Write Consistency. https://aws.amazon.com/blogs/aws/amazon-s3-update-strong-read-after-write-consistency/. accessed: 2022-10-05.

[24] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-Scale In-Memory Join Processing using RDMA. In *SIGMOD Conference*. ACM, 1463–1475.

[25] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman Van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *SIGMOD Conference*. ACM, 2326–2339.

[26] Brendan Bouffler and Chris Liu. 2019. Deep-Dive Into 100G networking & Elastic Fabric Adapter on Amazon EC2. AWS re:Invent, https://d1.awsstatic.com/events/reinvent/2019/REPEAT_2_Deep-dive_into_100G_networking_&_Elastic_Fabric_Adapter_on_Amazon_EC2_CMP334-R2.pdf. accessed: 2022-09-10.

[27] Matthias Brantner, Daniela Florescu, David A. Graf, Donald Kossmann, and Tim Kraska. 2008. Building a database on S3. In *SIGMOD Conference*. ACM, 251–264.

[28] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. 2022. Towards μs tail latency and terabit ethernet: disaggregating the host network stack. In *SIGCOMM*. ACM, 767–779.

[29] Wei Cao, Feifei Li, Gui Huang, Jianghang Lou, Jianwei Zhao, Dengcheng He, Mengshi Sun, Yingqiang Zhang, Sheng Wang, Xueqiang Wu, Han Liao, Zilin Chen, Xiaojian Fang, Mo Chen, Chenghui Liang, Yanxin Luo, Huanming Wang, Songlei Wang, Zhanfeng Ma, Xinjun Yang, Xiang Peng, Yubin Ruan, Yuhui Wang, Jie Zhou, Jianying Wang, Qingda Hu, and Junbin Kang. 2022. PolarDB-X: An Elastic Distributed Relational Database for Cloud-Native Applications. In *ICDE*. IEEE, 2859–2872.

[30] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *SIGMOD Conference*. ACM, 2477–2489.

[31] Jonathan Corbet. 2020. The rapid growth of io_uring. https://lwn.net/Articles/810414/. accessed: 2022-09-20.

[32] Craig Cotton, Henry Zhang, and Jamal Mazhar. 2019. New C5n Instances with 100 Gbps Networking. AWS re:Invent, https://www.youtube.com/watch?v=FJJxcwSfWYg. accessed: 2022-09-10.

[33] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD Conference*. ACM, 215–226.

[34] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[35] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring. In *SYSTOR*. ACM, 120–127.

[36] Dominik Durner. 2022. AnyBlob. https://github.com/durner/AnyBlob/.

[37] Dominik Durner, Badrish Chandramouli, and Yinan Li. 2021. Crystal: A Unified Cache Storage System for Analytical Databases. *Proc. VLDB Endow.* 14, 11 (2021), 2432–2444.

[38] Dominik Durner, Viktor Leis, and Thomas Neumann. 2019. On the Impact of Memory Allocation on High-Performance Query Processing. In *DaMoN*. ACM, 21:1–21:3.

[39] Google. 2023. Cloud Storage pricing. https://cloud.google.com/storage/pricing. accessed: 2023-06-17.

[40] Google. 2023. Google Cloud Platform C++ Client Libraries. https://github.com/googleapis/google-cloud-cpp. accessed: 2023-06-17.

[41] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR*. www.cidrdb.org.

[42] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *CIDR*. www.cidrdb.org.

[43] Jason Hughes. 2021. Apache Iceberg: An Architectural Look Under the Covers. https://www.dremio.com/resources/guides/apache-iceberg-an-architectural-look-under-the-covers/. accessed: 2022-09-10.

[44] IBM. 2023. About IBM COS SDKs. https://cloud.ibm.com/docs/cloud-object-storage?topic=cloud-object-storage-sdk-about. accessed: 2023-06-17.

[45] IBM. 2023. Cloud Object Storage. https://cloud.ibm.com/objectstorage/create#pricing. accessed: 2023-06-17.

[46] Virajith Jalaparti, Chris Douglas, Mainak Ghosh, Ashvin Agrawal, Avrilia Floratou, Srikanth Kandula, Ishai Menache, Joseph (Seffi) Naor, and Sriram Rao. 2018. Netco: Cache and I/O Management for Analytics over Disaggregated Stores. In *SoCC*. ACM, 186–198.

[47] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *OSDI*. USENIX Association, 185–201.

[48] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. *VLDB J.* 30, 5 (2021), 883–905.

[49] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *OSDI*. USENIX Association, 427–444.

[50] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan S. Milojicic, and Gustavo Alonso. 2022. Farview: Disaggregated Memory

with Operator Off-loading for Database Engines. In *CIDR*. www.cidrdb.org.
[51] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD Conference*. ACM, 311–326.
[52] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management. In *SIGMOD Conference*. ACM.
[53] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD Conference*. ACM, 743–754.
[54] Viktor Leis and Maximilian Kuschewski. 2021. Towards Cost-Optimal Query Processing in the Cloud. *Proc. VLDB Endow.* 14, 9 (2021), 1606–1612.
[55] Alberto Lerner and Philippe Bonnet. 2021. Not your Grandpa's SSD: The Era of Co-Designed Storage Devices. In *SIGMOD Conference*. ACM, 2852–2858.
[56] Feifei Li. 2019. Cloud native database systems at Alibaba: Opportunities and Challenges. *Proc. VLDB Endow.* 12, 12 (2019), 2263–2272.
[57] Feilong Liu, Lingyan Yin, and Spyros Blanas. 2017. Design and Evaluation of an RDMA-aware Data Shuffling Operator for Parallel Database Systems. In *EuroSys*. ACM, 48–63.
[58] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. VLDB Endow.* 3, 1 (2010), 330–339.
[59] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proc. VLDB Endow.* 13, 12 (2020), 3461–3472.
[60] Microsoft. 2023. Azure Blob Storage pricing. https://azure.microsoft.com/en-us/pricing/details/storage/blobs/. accessed: 2023-06-17.
[61] Microsoft. 2023. Azure SDK for C++. https://github.com/Azure/azure-sdk-for-cpp/. accessed: 2023-06-17.
[62] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *SIGMOD Conference*. ACM, 115–130.
[63] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. www.cidrdb.org.
[64] Oracle. 2023. Cloud Storage Pricing. https://www.oracle.com/cloud/storage/pricing/. accessed: 2023-06-17.
[65] Oracle. 2023. Software Development Kits. https://docs.oracle.com/en-us/iaas/Content/API/Concepts/sdks.htm. accessed: 2023-06-17.
[66] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *NSDI*. USENIX Association, 293–307.
[67] Ippokratis Pandis. 2021. The evolution of Amazon Redshift. *Proc. VLDB Endow.* 14, 12 (2021), 3162–3163.
[68] Jong-Hyeok Park, Soyee Choi, Gihwan Oh, and Sang Won Lee. 2021. SaS: SSD as SQL Database System. *Proc. VLDB Endow.* 14, 9 (2021), 1481–1488.
[69] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *SIGMOD Conference*. ACM, 131–141.
[70] Simeon Pilgrim. 2019. What are the specifications of a Snowflake server? https://stackoverflow.com/questions/58973007/what-are-the-specifications-of-a-snowflake-server/58982398. accessed: 2023-05-02.
[71] Nicolás Poggi, Josep Lluis Berral, Thomas Fenech, David Carrera, José A. Blakeley, Umar Farooq Minhas, and Nikola Vujic. 2016. The state of SQL-on-Hadoop in the cloud. In *IEEE BigData*. IEEE Computer Society, 1432–1443.
[72] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. 2010. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *Proc. VLDB Endow.* 3, 1 (2010), 460–471.
[73] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *ICDE*. IEEE, 1802–1813.
[74] Supreeth Shastri and David E. Irwin. 2017. HotSpot: automated server hopping in cloud spot markets. In *SoCC*. ACM, 493–505.
[75] Matt Sidley and Sally Guo. 2021. Deep dive on Amazon S3. AWS re:Invent, https://www.slideshare.net/AmazonWebServices/stg301deep-dive-on-amazon-s3-and-glacier-architecture, https://www.youtube.com/watch?v=9_vScxbIQLY. accessed: 2022-09-10.
[76] Supreeth Subramanya, Tian Guo, Prateek Sharma, David E. Irwin, and Prashant J. Shenoy. 2015. SpotOn: a batch computing service for the spot market. In *SoCC*. ACM, 329–341.
[77] Junjay Tan, Thanaa M. Ghanem, Matthew Perron, Xiangyao Yu, Michael Stonebraker, David J. DeWitt, Marco Serafini, Ashraf Aboulnaga, and Tim Kraska. 2019. Choosing A Cloud DBMS: Architectures and Tradeoffs. *Proc. VLDB Endow.* 12, 12 (2019), 2170–2182.
[78] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. 2010. Hive - a petabyte scale data warehouse using Hadoop. In *ICDE*. IEEE Computer Society, 996–1005.
[79] Ben Vandiver, Shreya Prasad, Pratibha Rana, Eden Zik, Amin Saeidi, Pratyush Parimal, Styliani Pantela, and Jaimin Dave. 2018. Eon Mode: Bringing the Vertica Columnar Database to the Cloud. In *SIGMOD Conference*. ACM, 797–809.
[80] Daniel Vassallo. 2023. Measure Amazon S3's performance from any location. https://github.com/dvassallo/s3-benchmark. accessed: 2023-05-02.
[81] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD Conference*. ACM, 1041–1052.
[82] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *NSDI*. USENIX Association, 449–462.
[83] Ruihong Wang, Jianguo Wang, Stratos Idreos, M. Tamer Özsu, and Walid G. Aref. 2022. The Case for Distributed Shared-Memory Databases with RDMA-Enabled Memory Disaggregation. *Proc. VLDB Endow.* 16, 1 (2022), 15–22.
[84] Christian Winter, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2022. On-Demand State Separation for Cloud Data Warehousing. *Proc. VLDB Endow.* 15, 11 (2022), 2966–2979.
[85] Yifei Yang, Matt Youill, Matthew E. Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2021. FlexPushdownDB: Hybrid Pushdown and Caching in a Cloud DBMS. *Proc. VLDB Endow.* 14, 11 (2021), 2101–2113.
[86] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.
[87] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, and Chengliang Chai. 2019. AnalyticDB: Real-time OLAP Database System at Alibaba Cloud. *Proc. VLDB Endow.* 12, 12 (2019), 2059–2070.
[88] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. 2021. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems. In *SOSP*. ACM, 195–211.
[89] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, Badrish Chandramouli, Vincent Liu, and Boon Thau Loo. 2022. CompuCache: Remote Computable Caching using Spot VMs. In *CIDR*. www.cidrdb.org.
[90] Qizhen Zhang, Yifan Cai, Sebastian Angel, Vincent Liu, Ang Chen, and Boon Thau Loo. 2020. Rethinking Data Management Systems for Disaggregated Data Centers. In *CIDR*. www.cidrdb.org.
[91] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Jimmy Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. 2021. Towards Cost-Effective and Elastic Cloud Database Deployment via Memory Disaggregation. *Proc. VLDB Endow.* 14, 10 (2021), 1900–1912.
[92] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA. In *SIGMOD Conference*. ACM, 685–699.