The Five-Minute Rule for the Cloud: Caching in Analytics Systems

Kira Duwe
EPFL
kira.duwe@epfl.ch

Angelos Anadiotis
Oracle
angelos.anadiotis@oracle.com

Andrew Lamb
InfluxData
alamb@apache.org

Lucas Lersch*
Amazon Web Services
llersch@amazon.com

Boaz Leskes
MotherDuck
boaz@motherduck.com

Daniel Ritter
SAP
daniel.ritter@sap.com

Pınar Tözün
ITU
pito@itu.dk

ABSTRACT

For almost 40 years, Gray and Putzolu's five-minute rule has helped quickly guide system architects to the break-even point between memory caching and direct local storage access. We believe similar rules of thumb are needed for object caches and storage in disaggregated cloud database system designs. However, it is not straightforward to adapt the established rules to the cloud as they presume fixed hardware, while, in the cloud, resources are dynamic and costs are determined by usage.

This paper reviews requirements driving object caches, analyzes the design space, defines a cost model, and proposes new rules of thumb to help system designers determine when caches become cost-effective for analytical workloads in the cloud. While perhaps unsurprising, our analysis on AWS shows that caches are beneficial when a system makes (1) two requests *per hour* for latency-sensitive workloads, or (2) seven requests *per second* for non-latency-sensitive workloads. These results are consistent with and help explain the near ubiquity of object store caches in cloud analytics systems.

1 INTRODUCTION

Traditionally, deploying database systems required acquiring and managing large, expensive servers on-premise. These servers feature a memory and storage hierarchy, including CPU caches, DRAM memory, HDD and/or SSDs, and sometimes tape. Database systems apply multiple techniques to efficiently process data using available hardware. Caches are commonly used to take advantage of the different characteristics of memory and storage, such as cost, latency, and bandwidth. In this context, previous work on the five-minute rule [4, 9, 11, 12] reviewed the storage hierarchy every ten years to determine when data should be cached in DRAM rather than directly read from slower devices (SSDs/HDDs).

The rise of cloud computing has significantly changed the way systems are deployed. Cloud services offer on-demand computing and storage, removing the need for companies to acquire and manage their own hardware. Furthermore, the cloud has introduced a new layer of the storage hierarchy for database systems: *object stores*. Every cloud provider offers some flavor of an object store, providing inexpensive, highly durable networked storage. This durability and low cost have made object stores ubiquitous in modern cloud-native databases and data infrastructures [5, 6]. Data lakes and "lakehouse" architectures often rely on data initially stored in object stores and offer query services on top of this data.

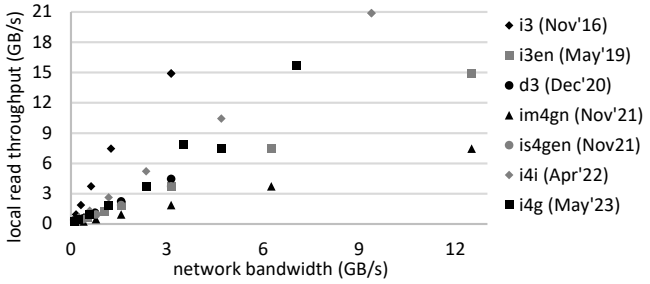


Figure 1: Network bandwidth has increased more rapidly than local read throughput for all instance types. Greater network bandwidth coupled with high durability and low cost of object storage has driven the adoption of disaggregated database architectures and, thus, the need for caching.

One can argue that accessing object stores over the network incurs higher latency and lower bandwidth than directly attached storage devices, placing them at the bottom of the storage hierarchy. In this scenario, attached storage would cache hot data from object stores. However, advances in cloud storage and networking have significantly impacted cost-effectiveness, latency variability, and dynamic workload optimization, challenging these traditional assumptions. Figure 1 compares the read bandwidth from local storage (Y-axis) and the read bandwidth from S3 (X-axis) for "storage optimized" EC2 instances available on Amazon Web Services (AWS). The S3 read bandwidth for *im4gn* instances is higher than the local read bandwidth. Other instance types, such as *i3en* and *is4gen*, offer slightly higher local read bandwidth than S3 bandwidth but the gap is small and closing over time. In non-storage optimized instances, network bandwidth often *exceeds* local storage read bandwidth, and writing to local storage devices tends to be half as fast as reading, while writing to S3 can match network bandwidth. These observations might suggest that it is not advisable to cache on locally-attached storage, but these devices offer interesting characteristics:

- **Lower Latency**: Locally-attached storage avoids network round-trips and typically has more predictable tail latency than object stores. Latency-sensitive workloads can often achieve more robust performance with locally-attached storage.
- **Cost**: Object stores charge based on the number of requests, while compute instances with locally-attached storage cost a flat hourly rate, regardless of how much data is read or written.

*Work not related to position at Amazon.

 15th Annual Conference on Innovative Data Systems Research (CIDR '25). January 19-22, Amsterdam, The Netherlands

Metric	DRAM					HDD					NVMe SSD	
	1987	1997	2007	2018	2024	1987	1997	2007	2018	2024	2018	2024
Unit price ($)	5k	15k	48	80	42	30k	2k	80	49	343	589	180
Unit capacity	1MB	1GB	1GB	16GB	32GB	180MB	9GB	250GB	2TB	20TB	800GB	2TB
$/MB	5k	14.6	0.05	0.005	0.0014	83.33	0.22	0.0003	0.00002	0.00001	0.0007	0.00009
Random IOPS (r/w)	–	–	–	–	-	5	64	83	200	168 / 550	460 k	1,400k/1,550k
Seq bandwidth (MB/s) (r/w)	–	–	–	–	–	1	10	300	200	285	2500	7,450/6,900

Table 1: Evolution of storage device properties for DRAM, NVMe SSDs, HDD. Extended table from [4]

Given these non-trivial design constraints, it is unclear how to use locally-attached storage to cache data from object stores while balancing performance and cost. In the rest of this paper, we revise the five-minute rule for when it makes economic sense to use a cache in cloud-native database systems for analytical workloads. Specifically, we make the following contributions.

- Section 2: Reviews past iterations of the five-minute rule .
- Section 3: Surveys common cloud caching architectures.
- Section 4: Proposes a five-minute rule for the object store caches.
- Section 5: Analyzes the proposal for one cloud provider (AWS).

2 EVOLUTION OF THE FIVE-MINUTE RULE

The five-minute rule was originally conceived in 1987 by Gray and Putzolu [12] as a practical guideline for sizing buffer pools based on the frequency of page accesses and the relative cost of memory. In its simplest form, the 1987 rule suggests caching disk pages that are re-used at least every 5 minutes. Several subsequent papers propose refinements as system designs, available hardware, and relative costs evolved. Gray and Graefe's [11] 1997 adaptation, for instance, introduced distinctions between randomly and sequentially accessed pages, providing nuanced recommendations for different access patterns. Subsequent contributions by Graefe [9] in 2009 extended the rule's applicability to diverse storage media, considering factors such as disk and flash storage characteristics. Another proposed update to the five-minute rule by Appuswamy et al. [4] in 2019 reflected the evolving landscape of storage technologies at that time, including time thresholds for data transfer between various storage mediums, such as DRAM to HDD, DRAM to SSDs, highlighting the increasing performance differences.

Continued Evolution of Storage Devices. Table 1 shows the evolution of storage hardware characteristics spanning the last four decades. The results for 1987, 1997, 2007 and 2018 are taken from the respective revisions of the five-minute rule [4, 9, 11, 12], and the results for 2024 are found online[1].

Break-even Intervals. The five-minute rule is expressed as *break-even intervals*, defined in Equation (1) across various hardware configurations. This formula relies on two key factors: 1) the technology ratio (first term), which reflects the relative speeds of different storage media, and 2) the economic ratio (second term), which accounts for the cost differences between cache (DRAM) and disk (HDD, SSD) access. Using this formula, system designers could quickly identify the expected time between page accesses where caching the page was economically beneficial.

$$\frac{1000\text{ kB} \times (\text{page size in kB})^{-1}}{\text{read IOPS}} \times \frac{\$\text{ per Disk}}{\$\text{ per MB of DRAM}} \quad (1)$$

Equation (1) only includes the *$ per per disk*, but not its capacity. This means that calculating the results using the 2 TB HDD (2018) instead of the 20 TB HDD (2024) will yield very different results. For example, the HDDs from 2018 and 2024 achieve very similar IOPS and have a *$/MB* in the same order of magnitude. However, the break-even interval for a 2024 HDD is 108 hours, but only 12 hours for the 2018 HDD. This order of magnitude difference is due to the much higher absolute price of the 2024 HDD.

Although the evolution of storage technologies requires regular updates to calculate break-even times, the evolution of storage and network technologies in the cloud requires more significant changes to the rule due to the different factors affecting that environment, such as the pay-as-you-go model, access latency variability, and in-cluster caching.

3 CACHING IN THE CLOUD

Before adapting the five-minute rule for the cloud, we first motivate the metrics of interest by describing several cloud caching architectures. We avoid overwhelmingly complex analysis by focusing on the design patterns that come with unique trade-offs.

3.1 Cache Architectures

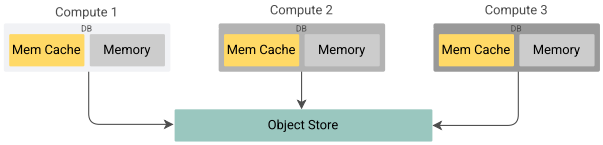
Analytic services deployed on clouds and backed by object storage use the following strategies: Figure 2:

No cache: The system retrieves data directly from object storage for every data access. While this is simple both architecturally and operationally, it exposes end users directly to the latency variability of the underlying object store, and incurs costs that scale linearly with the number of accesses. We do not know of any major production system that uses this approach.

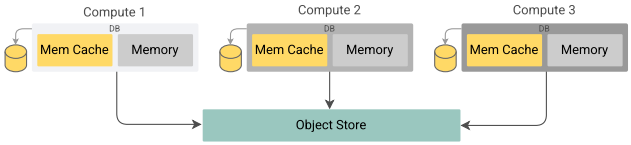
Compute local (mem): Figure 2a: Each compute node stores previously accessed objects in memory. Compute nodes operate independently and may contain duplicate copies of the same object. This design provides fast cache access (no network access) and is operationally simple (no additional configuration or communication). However, it is expensive due to the cost of memory, and often has a low cache hit rate when cache contents are rebuilt as new nodes are added.

Compute local: Figure 2b: Each compute node utilizes a hierarchy of both memory and locally-attached storage such as SSD to cache prior object store requests. As in the compute local (mem) cache, the

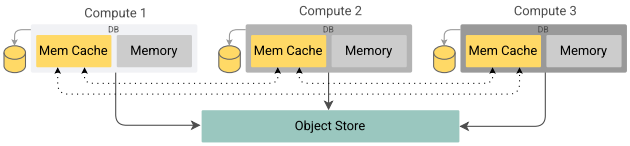
[1]Accessed 18.07.2024:
DRAM: https://jcmit.net/memoryprice.htm
HDD: https://tinyurl.com/SeagateExosX20
NVMe SSD: https://www.newegg.com/samsung-2tb-990-pro/p/N82E16820147861



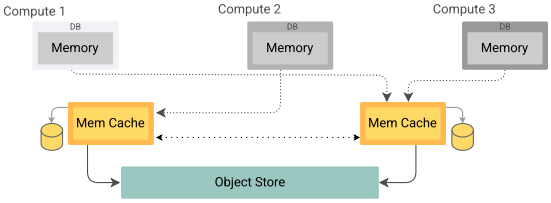
(a) Compute local (mem): Separate in-memory cache on each compute node.



(b) Compute local: Separate local storage and memory-based cache on each compute node.



(c) Shared nothing: Shared cache collaboratively managed and accessed by compute nodes.



(d) Cache Service: Cache is managed by a separate set of nodes.

Figure 2: Common cloud cache architectures.

nodes manage caches independently, potentially storing duplicate objects. This design is less expensive than compute local (mem) as disk / SDD costs less than RAM, but slightly more operationally complex due to managing persistent storage. This design is found in systems such as Sokrates / Microsoft SQL Server [3], Amazon Redshift [13], Snowflake's virtual warehouse [6], Databricks' delta cache [5], as well as in cloud caching vendors like Alluxio [1] and recent research prototype Crystal [7].

Shared nothing: Figure 2c: Resources on local compute nodes, such as memory and locally attached SSD, store previously accessed data from the object store. Unlike the previous compute local caches, in this design compute nodes collaborate to store non-overlapping subsets of the objects, e. g., via hash partitioning, and retrieve objects from other compute nodes. This system is less expensive than compute local variants as the cache capacity is shared, but operationally more complex as the nodes must work together to share and manage the cache state. In addition, this architecture ties the lifecycle of compute and caches together, making it harder to elastically scale the cluster in response to changed workloads. Probably due to complexity, we do not know of any industrial implementations that use this design.

Cache Service: Figure 2d: Separate nodes with dedicated resources run a specialized cache service. The cache service offers a unified interface to object storage, managing individual node contents in a unified manner. This system is expensive as it requires dedicated nodes and operationally complex as it requires running a separate distributed stateful service. This architecture is used for result set caches as in cross-virtual warehouse caches in Snowflake [6] and Alluxio [1], as well as metadata caches through Snowflake's metadata service that is based on FoundationDB [14], Google NAPA's distributed shared cache [2], Alluxio's leveled metadata cache [1], and Crystal's ring buffer based metadata cache [7].

3.2 Design and Performance Trade-offs

Table 2 summarizes some trade-offs between cache designs:

Latency Variability between object store request and response is known to be unpredictable [8]. *No Cache* setups directly expose systems to object store latency variability. Since cache contents do not survive node restarts, *Compute local (mem)* and *Compute local* configurations have a significant cache miss rate, and each cache miss results in variable latency. *Shared-Nothing* architectures maintain some cache state as nodes restart, but face variability from hotspots due to unbalanced cluster loads and the need to re-balance cache contents. *Cache Service* architectures provide the lowest latency variability by decoupling the cache from changes in compute nodes.

Implementation Complexity includes the cost of implementing, debugging, and maintaining the software system. Obviously, *No Cache* systems have no cache implementation cost. *Compute local (mem)* and *Compute local* designs must implement cache eviction policies and resource allocation. In addition, they require data movement between cache levels and timing policies to execute operations. *Shared Nothing* and *Cache Service* configurations add more complexities, such as replication policies, distributed cache consistency, and optimizing network communications.

Operational Complexity describes the ongoing burden of operating distributed systems, such as configuration management, monitoring, node lifecycle management, and failure analysis. *No Cache* and *Compute local (mem)* configurations require no additional interactions with the cloud service provider or other nodes, thus not increasing the operational burden significantly. *Compute local* setups have a small additional complexity of provisioning local storage. *Shared Nothing* setups require additional cross-node communication and monitoring of that communication. *Cache Service* designs typically require full-service management, configuration, deployment, and monitoring.

Object Store Request Count is directly proportional to cost. *No Cache* requires a request for each object access. *Compute local (mem)* can reuse objects after the first request. With their larger capacity, *Compute local* decreases requests, but each node must still request each object at least once. *Shared Nothing* and *Cache Service* architectures decrease requests even more by sharing access to objects requested by one node within the cluster.

Cache Capacity Elasticity is the ability of a cache to increase and reduce resource usage based on demand. *Compute local (mem)* and *Compute local* configurations offer limited elasticity by changing

Design	Latency Variability	Implementation Complexity	Operational Complexity	Object Store Request Count	Cache Capacity Elasticity
No cache	↑↑↑↑↑	None	None	↑↑↑↑↑	None
Compute local (memory only)	↑↑↑	↑	None	↑↑↑↑	↑
Compute local	↑↑↑	↑↑	↑	↑↑↑	↑
Shared nothing	↑↑	↑↑↑	↑↑	↑↑	↑↑↑
Cache Service	↑	↑↑↑	↑↑↑↑	↑	↑↑↑↑↑

Table 2: Trade-off of different cache architectures illustrated in Figure 2.

resources on each node (scale up). *Shared Nothing* and *Cache Service* architectures can more easily add incremental capacity by adding new nodes. However, *Shared Nothing* designs are more constrained due to sharing resources with computation.

4 FIVE-MINUTE RULE FOR THE CLOUD

Developing a five-minute rule for cloud environments requires a question in terms of relevant metrics. We propose: ***How often must an application access an object to justify caching instead of directly fetching from object storage, given a latency target?*** To answer this question, we determine when the cost of caching, Equation (2), and the cost without a cache, Equation (3), are equal.

Note that we assume the costs of hosting the application, storing data in the object store, and the first access is the same in all circumstances, and thus do not include these costs in our calculations.

$$CostOfCaching = CostOfCacheHardware \ . \tag{2}$$

$$\begin{aligned} CostOfNoCache = {} & CostPerObjectStoreRequest \\ & \times (NumberOfRequests - 1) \ . \end{aligned} \tag{3}$$

Cost of caching is determined by how long the nodes are used, given the cloud providers' consumption-based billing model. Costs also vary by the resources required for cache storage: memory caches cost more than SSD-based caches for the same capacity, which in turn costs more than HDD-based caches.

In cloud services, compute node costs typically increase linearly with storage capacity: If an instance with XGB SSD capacity costs Y\$, an instance with $2X$GB SSD capacity costs $2Y$\$. Besides the instance cost, specific storage options such as network-attached storage also incur additional fees. For example, in AWS, using EBS (Elastic Block Storage) for the cache adds an additional monthly storage cost per GB. Finally, we must account for cache misses. Each miss requires paying for a new object request from object storage. Taking these factors into account results in a per instance cache cost:

$$\begin{aligned} CostCache_{SepInst} = {} & (HourlyStorageCostPerGB + InstCostPerGB) \\ & \times SizeCacheInGB \times LifetimeOfCacheInHours \\ & + CostPerObjectStoreReqs \times NumReqs \\ & \times CacheMissRate \ . \end{aligned} \tag{4}$$

Equation (4) assumes that the cache is hosted on a separate compute node from the application. If the cache and the application share the same node, we must subtract the application hosting cost:

$$\begin{aligned} CostCache_{SameInst} = {} & CostCache_{SepInst} \\ & - InstCostWithoutCache \ . \end{aligned} \tag{5}$$

Cost without a cache. Equation (3) computes costs linearly from the number of requests but must be extended to account for latency requirements as well. The *Racing Reads* technique, explained in Section 5.2, uses *RepeatsToGuaranteeLatency* requests for the same object to reduce expected latency. This number depends both on the target latency and the requested object's size. The cost for cacheless systems that use Racing Reads is in Equation (6).

$$\begin{aligned} CostOfNoCache = {} & CostPerObjectStoreReq \\ & \times (NumberOfApplicationReqs - 1) \\ & \times RepeatsToGuaranteeLatency \ . \end{aligned} \tag{6}$$

Calculating *RepeatsToGuaranteeLatency* requires knowing the latency distribution of requests to read an object of a certain size, and then choosing the acceptable probability that at least one request completes within the target latency. Using the latency distribution, one can calculate the probability, P, of reading an object within the target latency budget from the object store. If there are n independent requests for this object, the probability that all of them take longer than the target is $(1-P)^n$. The probability, P', that at least one of the n requests reaches the target is therefore $1-(1-P)^n$. Solving for n yields:

$$n = RepeatsToGuaranteeLatency \quad = log_{(1-P)}(1-P') \ . \tag{7}$$

Latency-sensitive applications typically use values of P' such as 99%, and report P99 latency (99 requests out of 100 are within the latency target). When there are no latency requirements, Equation (6) simplifies to Equation (3).

How to obtain the term values? `RepeatsToGuaranteeLatency` is determined by the target latency and request latency distribution for reading a specific size object. This means that one must measure the read latency values for specific object sizes on the target object storage and generate a latency distribution. This is the only term in the cost calculations that requires prior measurements. `CacheMissRate` is a function of the workload and cache capacity, and can be adjusted to reflect how cache performance affects costs. `NumberOfRequests` is the term we solve for to answer the question posed at the beginning of this section. The other terms in equations 2-6 are obtained from the costs of compute instances and storage published online by the cloud vendors.

Assumptions. Equation (2) - Equation (6) determine the break-even infrastructure cost for designs with and without a cache. However, due to wide variability across organizations, our model does not account for software engineering or operational costs associated with caches. We also assume similar cost structures across cloud providers and that all computation costs for the cache are fixed and small compared to object store access.

S3 cost per request	0.0000004$
EBS monthly storage cost per GB	0.08 $
Hourly on-demand compute instance costs	
m7g instance with EBS	0.0408 $
m7gd instance with 59GiB NVMe SSD	0.0534 $

Table 3: AWS instance costs (Graviton processors).

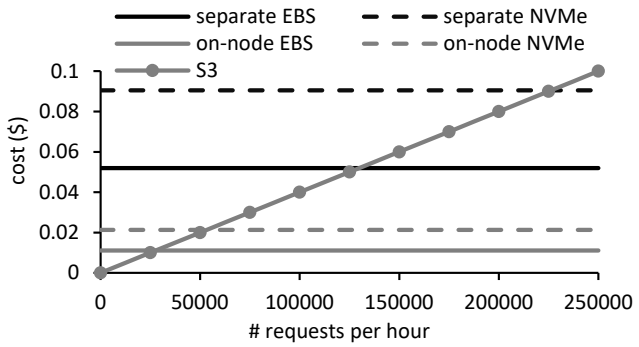


Figure 3: Hourly costs for non-latency-sensitive workloads, assuming a 100GiB cache with 100% hit rate. No cache requires reading from object store, which increases linearly with number of requests. Caches require additional resources, whose cost is fixed per hour irrespective of the number of requests.

5 EVALUATION

We analyze several scenarios using the cost model from Section 4 to determine the number of requests per hour when caches become less expensive than direct object storage access. The results are based on AWS costs, listed in Table 3, and all measurements of object store access latency are derived from [8].

5.1 Non-Latency-Sensitive Workloads

Batched analytic workloads, such as data preparation pipelines or billing, have relatively relaxed latency requirements and are typically optimized for cost and overall throughput. Although queries must eventually complete, other stages of processing, like complex joins or string manipulation, often dominate processing, and these workloads more easily tolerate object store access latency.

Caching Costs. When latency and query robustness are not primary concerns, designers can compare the costs of accessing S3 directly, Equation (3), with those of using caches, Equation (4) and Equation (5), and choose the least expensive option. Figure 3 plots the cost vs requests per hour for different cache configurations and reading directly from S3, assuming a 100GiB cache size and a perfect hit rate. Directly accessing S3 is cost equivalent at:

- **On-Node Cache (EBS)**: 25,000 requests per hour, roughly equivalent to 420 requests per minute or *7 requests per second.*
- **On-Node Cache (NVMe)**: 50,000 requests per hour, 840 requests per minute or *14 requests per second.*
- **Dedicated Cache (EBS)**: 125,000 requests per hour, 2,000 requests per minute, *35 requests per second.*
- **Dedicated Cache (NVMe)** 225,000 requests per hour, 3,700 requests per minute, *62 requests per second.*

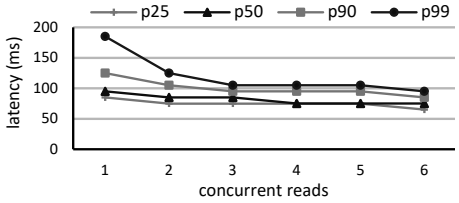


Figure 4: Latency distribution for racing 1MiB reads. pN represents the Nth percentile: N% of the requests completed within this time. With a single request, p99 latency is around 180ms, decreases to around 125ms with two concurrent requests and to less than 100ms with 6 concurrent requests.

request size	# requests
1MiB	20480
4MiB	94720
8MiB	57600

Table 4: Requests needed to download 10GiB from S3, where each request finishes in 150ms with 99% probability.

Discussion. Our analysis shows that caching is more expensive than direct access until a system does at least 7 (repeated) requests per second. Batch workloads typically read and transform data *once* each stage, so unsurprisingly they often simply read from (and write to) object storage directly, without a cache.

5.2 Latency-Sensitive Workloads

In systems such as streaming databases, time-series database alerting, and interactive analytics platforms, the latency between query start and result is often critical. We model this importance as query latency budget and object store access latency usually consume a significant portion of the total latency budget.

Racing reads. In systems without a cache, one commonly used technique to reduce latency variability is *Racing Reads*. Systems using this technique issue multiple requests for the same object, and proceed with whichever response arrives first, ignoring any other responses. As shown in Figure 4, the 99th percentile latency (99 percent of requests complete within the given time), significantly improves from one to two concurrent requests, from around 180 ms to 145 ms. Additional concurrent requests improve the expected latency further, but the returns decrease.

Request size. Another factor that affects latency is request size. Since object stores charge per request rather than per byte, fewer larger objects are preferable. However, larger objects take longer to retrieve, reducing the chance of meeting a target latency. Table 4 shows the total number of requests needed to download 10GiB, where per request latency is 150 ms with a 99% probability, using various request sizes, based on Equation (7). Surprisingly, given this latency target, Table 4 indicates that making more smaller requests is cheaper than making fewer larger ones. Only a small percent of requests for larger objects complete in the required latency, necessitating many concurrent reads. Using 4 MiB objects costs almost 5x more than 1 MiB objects. Interestingly 8 MiB objects only costs around 3x more than 1 MiB objects: while many concurrent requests are still needed, fewer objects are required overall.

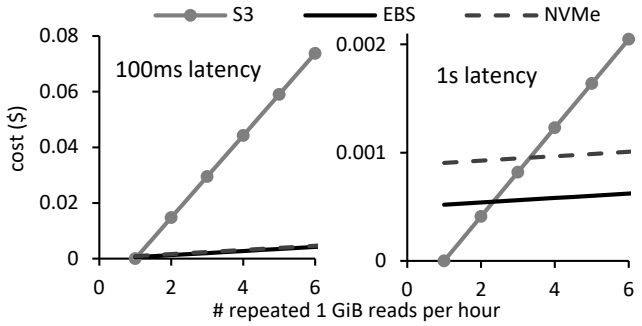


Figure 5: Cost of repeatedly reading 1 GiB of data using 1MiB requests, where per request latency is 100 ms (left) or 1s (right) with 99% probability. No cache (S3) vs. a 1 GiB block storage (EBS) or SSD (NVMe) cache with a 95% hit rate.

Caching costs. We use Equation (6) to evaluate cache costs for latency-sensitive workloads, using 1 MiB requests, the best option in Table 4. Figure 5 plots the costs for repeatedly reading 1 GiB data using 1MiB requests that must be completed within either 100 ms or 1 s with 99% probability. We compare costs between S3 (no caches) and EBS or local NVMe SSD 1 GiB caches with 95% hit rate.
Discussion. For latency-sensitive workloads, racing reads on S3 are very costly. Once a dataset is accessed twice an hour, an EBS-based cache is more economical. Even the more costly NVMe-based cache is cost-effective once the data is accessed four times per hour. Since data for interactive workloads is commonly accessed more than twice an hour, this result likely explains the prevalence of caches in interactive systems in the cloud, despite their additional development, maintenance, and operational costs.

6 OPEN QUESTIONS / FUTURE RESEARCH

We distilled a complex design space into a sentence with many simplifying assumptions. We suggest revisiting:
Other vendors. Our analysis could be applied to vendors other than AWS. Since the infrastructure costs for VMs and cloud object storage are similar between vendors, we do not expect dramatically different results than what Section 5 presents.
Alternative cloud instances and storage offerings. Including a broader range of compute instances or storage offerings (such as different classes of EBS, provisioned IOps, io2 vs gp2, etc.) or different object store performance tiers, such as S3 Express, would more fully evaluate the design space.
CPU resource needs. In practice, larger compute instances with more CPU resources are often required to ensure high utilization of NVMe SSDs and network bandwidth. It would be interesting to evaluate the impact such additional constraints have on our model.
Access patterns on popular storage formats. We also assume uniform object access patterns, but systems using open columnar storage formats such as Apache Parquet may have two distinct sets of latency requirements for the metadata and the data pages.
Advanced caching. It would be interesting to model the cost of more advanced caching techniques, such as data transformations, materialized views, partial results caching, decompression, on-the-fly indexes, and shuffling.
Other workloads. In this work, we focused on analytical workloads. It would also be interesting to extend our model and analysis to include other workloads, such as transaction processing, by incorporating the impact of write requests in addition to reads.

7 CONCLUSION

To offer guidance for cloud-native databases to optimize storage and caching strategies in modern cloud infrastructure, we revisit the 5-minute rule and adapt it to the cloud storage hierarchy. Our evaluation shows that caches are economical in the cloud after 7 requests per second in non-latency-sensitive workloads and 2 requests per hour in latency-sensitive cases.

ACKNOWLEDGMENTS

We thank Kai-Uwe Sattler, Anastasia Ailamaki, and Goetz Graefe for their invaluable insights, the rest of the participants of Dagstuhl Seminar 24101 [10] where we began this work, the authors of [8] for sharing their data, and the anonymous reviewers whose comments strengthen this paper.

REFERENCES

[1] Alluxio: Cloud Analytics Caching. https://www.alluxio.io/use-cases/cloud-analytics-caching/. Accessed: 2024-08-02.
[2] A. Agiwal, K. Lai, G. N. B. Manoharan, I. Roy, J. Sankaranarayanan, H. Zhang, T. Zou, J. Chen, M. Chen, M. Dai, T. Do, H. Gao, H. Geng, R. Grover, B. Huang, Y. Huang, A. Li, J. Liang, T. Lin, L. Liu, Y. Liu, X. Mao, M. Meng, P. Mishra, J. Patel, R. Sr, V. Raman, S. Roy, M. S. Shishodia, T. Sun, J. Tang, J. Tatemura, S. Trehan, R. Vadali, P. Venkatasubramanian, J. Zhang, K. Zhang, Y. Zhang, Z. Zhuang, G. Graefe, D. Agrawal, J. F. Naughton, S. Kosalge, and H. Hacigümüs. Napa: Powering Scalable Data Warehousing with Robust Query Performance at Google. *Proc. VLDB Endow.*, 14(12):2986–2998, 2021.
[3] P. Antonopoulos, A. Budovski, C. Diaconu, A. H. Saenz, J. Hu, H. Kodavalla, D. Kossmann, S. Lingam, U. F. Minhas, N. Prakash, V. Purohit, H. Qu, C. S. Ravella, K. Reisteter, S. Shrotri, D. Tang, and V. Wakade. Socrates: The New SQL Server in the Cloud. In *SIGMOD*, pages 1743–1756. ACM, 2019.
[4] R. Appuswamy, G. Graefe, R. Borovica-Gajic, and A. Ailamaki. The five-minute rule 30 years later and its impact on the storage hierarchy. *Commun. ACM*, 62(11):114–120, 2019.
[5] M. Armbrust, T. Das, S. Paranjpye, R. Xin, S. Zhu, A. Ghodsi, B. Yavuz, M. Murthy, J. Torres, L. Sun, P. A. Boncz, M. Mokhtar, H. V. Hovell, A. Ionescu, A. Luszczak, M. Switakowski, T. Ueshin, X. Li, M. Szafranski, P. Senster, and M. Zaharia. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. VLDB Endow.*, 13(12):3411–3424, 2020.
[6] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner. The Snowflake Elastic Data Warehouse. In *SIGMOD*, pages 215–226. ACM, 2016.
[7] D. Durner, B. Chandramouli, and Y. Li. Crystal: A Unified Cache Storage System for Analytical Databases. *Proc. VLDB Endow.*, 14(11):2432–2444, 2021.
[8] D. Durner, V. Leis, and T. Neumann. Exploiting Cloud Object Storage for High-Performance Analytics. *Proc. VLDB Endow.*, 16(11):2769–2782, 2023.
[9] G. Graefe. The five-minute rule 20 years later (and how flash memory changes the rules). *Commun. ACM*, 52(7):48–59, 2009.
[10] G. Graefe, A. Lee, and C. Sauer. Robust Query Processing in the Cloud (Dagstuhl Seminar 24101). *Dagstuhl Reports*, 14(3):1–8, 2024.
[11] J. Gray and G. Graefe. The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb. *SIGMOD Rec.*, 26(4):63–68, 1997.
[12] J. Gray and G. R. Putzolu. The 5 Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference*, pages 395–398. ACM 'Press, 1987.
[13] T. Schmidt, A. Kipf, D. Horn, G. Saxena, and T. Kraska. Predicate Caching: Query-Driven Secondary Indexing for Cloud Data Warehouses. In *SIGMOD*, pages 347–359. ACM, 2024.
[14] J. Zhou, M. Xu, A. Shraer, B. Namasivayam, A. Miller, E. Tschannen, S. Atherton, A. J. Beamon, R. Sears, J. Leach, D. Rosenthal, X. Dong, W. Wilson, B. Collins, D. Scherer, A. Grieser, Y. Liu, A. Moore, B. Muppana, X. Su, and V. Yadav. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *SIGMOD*, pages 2653–2666. ACM, 2021.