Noname manuscript No.
(will be inserted by the editor)

OLTP in the Cloud: Architectures, Tradeoffs, and Cost

Preprint; Accepted for publication in the VLDB Journal

Michael Haubenschild · Viktor Leis

the date of receipt and acceptance should be inserted later

Abstract What is the best architecture for cloud OLTP systems? How costly is it to run a specific workload? Which and how many hardware instances should be provisioned? To answer such questions systematically, we develop an analytical model framework for cloud OLTP. It enables the analysis of a wide variety of workloads and determines the cost-optimal architecture and hardware configuration for each. Workloads are specified in terms of dataset size, performance, latency, availability and durability requirements. System designs are evaluated based on the CPU, memory, storage, and network resources they require. We study a concrete model instance that is calibrated with the LeanStore storage engine and real-world hardware/service options and prices from AWS, one of the major cloud providers. Our analysis yields several observations on how to achieve fast, durable and cost-efficient OLTP in the cloud.

1 Introduction

Cloud OLTP. OLTP is increasingly moving into public clouds, and mission-critical workloads often have strict requirements in terms of durability, availability, and performance. Any cloud database system must ensure that these workload requirements are met.

Lift-And-Shift Architectures. The most straightforward way to run a database system in the cloud is to take a *Classic* (i.e., disk-based) or *In-Memory* system and deploy it on a cloud instance with built-in storage. The downside of this approach is that any hardware infrastructure fault will likely result in data loss. When a cloud instance has a hardware defect, there is no way to recover the data even when instance storage is unaffected. To increase durability, cloud vendors therefore offer network-attached Remote Block Devices (*RBD*) such as AWS Elastic Block Store or Azure Managed Disk. If the database instance fails, its remote block device can simply be attached to another instance that can resume operation after running database recovery. Another conventional way to increase durability is the High Availability Disaster Recovery (*HADR*) architecture. With HADR, the primary database instance sends its Write Ahead Log (WAL) continuously to one or more secondary instances with identical hardware.

Cloud-Native Architectures. Arguably the most influential cloud-native OLTP architecture has been Amazon *Aurora* [1]. Instead of instance storage or RBD, Aurora relies on disaggregated, multi-tenant, and fault-tolerant storage for both the database pages and the WAL. This allows scaling compute and storage independently. The architecture of Microsoft *Socrates* [2] is even more disaggregated by having not just a page service but also a separate log service for the WAL.

Options and Tradeoffs. Each of the six architectures just described (*Classic*, *In-Memory*, *RBD*, *HADR*, *Aurora*, *Socrates*) offers a different tradeoff in terms of durability, availability, throughput, latency, and monetary cost. Public clouds also offer multiple variants of RBD and hundreds of heterogeneous hardware instances, creating even more options. OLTP workloads themselves can have different requirements in terms of dataset size, transaction rate, and durability. Together, these options and constraints form a large multi-dimensional design space.

M. Haubenschild
Salesforce, Inc.
E-mail: mhaubenschild@salesforce.com

V. Leis
Technical University of Munich, Garching, Germany
E-mail: leis@in.tum.de

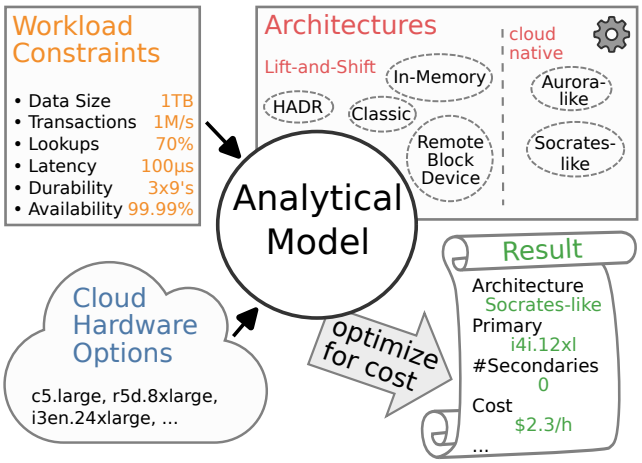

Fig. 1 Quantitative model-based architecture comparison.

Current Practice: Uninformed Decisions. In practice, system designers and users cannot fully explore this large architectural and configuration space. Instead, they must rely on intuition, vendor claims, or limited benchmarking, which can only cover a few points of the design space [3]. To the best of our knowledge, no systematic comparison of cloud OLTP systems exists that would enable rigorous decision-making.

Our Approach: Cost Optimization. Given specific workload requirements and the cloud hardware landscape, how can one determine the best architecture and hardware configuration? For example, if a workload requires a high lookup performance, is it better to

- choose an instance with a large main memory,
- use secondary instances to offload the lookups, or
- rely on a multi-tenant page service?

We argue that the only sensible way to answer such questions is to select the cheapest option that satisfies the workload constraints. In other words, the fastest architecture (e.g., In-Memory) is not necessarily the best, as it may be much more expensive than the alternatives.

Contribution 1: Model Framework. Given the large size of the design space, a purely experimental approach that exhaustively measures all configurations is not feasible. Prior experimental work measuring existing cloud DBMS can only cover a small subset of the design and configuration space [4–6]. We therefore argue that the only realistic option to explore the full design space is through an analytical model. This paper presents a general framework for such a model. It incorporates six OLTP architectures and derives their CPU, memory, network, and storage requirements within the characteristics of public cloud environments. As Figure 1 illustrates, for a given workload, the framework enumerates all feasible hardware/architecture combinations and computes the cheapest setup. Using actual AWS hardware options, we calibrate two concrete models and validate each of them. One is for our open-source storage engine LeanStore and the other one is for the commercial AWS Aurora PostgreSQL.

Contribution 2: Analysis. With the LeanStore model, we analyze OLTP workloads of varying dataset sizes, transaction rates, and durability requirements. Based on the results, the paper presents 12 observations that are valuable for researchers and database system architects. The analysis shows that cloud-native architectures perform well and scale to very large dataset sizes, while usually being cheaper than traditional designs.

Model Applications. Our model-based approach can serve multiple use cases. It enables database administrators to determine which system and hardware configuration to use to minimize cost while satisfying workload requirements. It informs database engineers which architecture and building blocks are worth implementing and optimizing, e.g., whether to build a cache on instance SSDs. Finally, it can help database researchers identify areas for further research, e.g., how to build a low-latency log service on cloud hardware.

Towards Cost-Optimal Systems. This study deviates from most research papers, as it does not provide incremental advancements for a specific problem. Instead, it seeks to consolidate existing knowledge and expand the conceptual understanding of an extensive design space and a broad cloud landscape. The paper is the result of a multi-year effort to determine the best way to move our OLTP system LeanStore [7] to the cloud. We believe that the findings are of interest to designers of durable cloud systems.

Limitations of this Study. Our framework models systems using page-based clustered B-tree indexes, which is a widely-used design choice for OLTP. Using other data structures, such as LSMs, could lead to different results. For determining the optimal data structure for a particular workload, our work could be combined with Cosine [8]. Our findings are based on the premise that systems are able to exploit available hardware resources. Prior work [7, 9] has shown that our LeanStore storage engine achieves this at least for NVMe drives and CPU. However, this limits the applicability of our results to other systems. Furthermore, while the six OLTP architectures we model span a variety of lift-and-shift and cloud-native ones, other (novel) architectures may be even more cost efficient. The workloads we consider are static, i.e., they have a fixed data set size and lookup/update rate. Modeling dynamic workloads, such as those with growing datasets typical in real-world applications, requires setting the workload parameters to the peak values that should be supported.

Adding such a buffer is a good idea in practice anyway to reduce the frequency of instance and system migrations. Lastly, we focus our analysis on AWS as it is the largest cloud provider. Given that most competitors have a similar pricing structure and service offering, they can be added to our framework with low effort, which would be interesting future work.

Outline. Section 2 provides the necessary background information by presenting six widely-used OLTP architectures and discussing related work on OLTP and cost optimization in the cloud. Section 3 introduces our methodology and describes the model framework. We calibrate and validate two concrete models in Section 4. With the LeanStore-calibrated model, we analyze the different architectures across a multitude of workload dimensions and hardware options in Section 5, yielding a number of key observations both for researchers and system designers. We discuss the implications of those findings for cloud OLTP systems in Section 6 and conclude with potential future work in Section 7.

2 Background and Related Work

Single-Writer Designs in The Cloud. We focus on general-purpose, ACID compliant transaction systems that employ a single-writer design [10]. Most common commercial cloud OLTP systems, including AWS Aurora [1], Microsoft Socrates [2], and the recently-launched Google AlloyDB [11] and Neon [12,13], follow this paradigm. Ziegler et al. [10] qualitatively compared single-writer systems to other designs and found that the alternatives — partitioned-writer and shared-writer — have higher complexity, which may explain why the former are dominant.

2.1 Classic

Everything on a Single Node. Historically, OLTP systems were deployed on a single dedicated server, using a scale-up approach to increase performance. We call this architecture *Classic*. In the cloud, it can run on a single compute instance, e.g., using AWS EC2. The whole database is stored on instance-local storage and organized in fixed-size pages. We only consider current generation instances with NVMe SSDs, as high access latency makes HDDs unsuitable for modern OLTP. Tuple accesses might either hit the in-memory buffer cache, or require reading the data from SSD. Modifications to database pages are first accumulated in the buffer cache before the buffer manager asynchronously writes them out. Durability of committed transactions is guaranteed by the WAL protocol. Figure 2 shows the flow of data in this design: WAL records are written to SSD alongside dirty pages from the cache, while pages are read back into the cache during data accesses. The SSD provides durability in this design, and is in fact how people have looked at the concept of durability in the past. In-memory changes are at risk of being lost (when the system fails), but once they have reached the non-volatile SSD, they are safe against power failures and software crashes. This architecture generates no (internal) network traffic, all CPU cores are used for transaction processing, and storage is modeled in terms of capacity and I/O operations per second (IOPS).

Discussion. Having everything on a single node avoids the intricacies of a distributed system, e.g., handling high tail latency in the network. Another benefit is that instance-local storage is fast (e.g., up to 3.3M random 4 KB reads per second in AWS [14]) and has no additional per-request cost. On the downside, if an instance is shut down or has a hardware failure, all the data on the ephemeral local SSD is lost. This differs from the on-premise world, where data on an SSD can still be recovered if, for example, the CPU or power supply breaks down. The fixed coupling between storage and compute, resulting from the limited number of cloud instance types, is not ideal for low-intensity workloads on large datasets, or for high-intensity workloads on small datasets, and limits the maximum database size.

2.2 High Availability Disaster Recovery (HADR)

Multiple Replicas. The *High Availability Disaster Recovery* (HADR) architecture builds on the *Classic* architecture, but adds one or more additional instances that continuously replay the WAL on their local copy of the database [15–18]. The primary node still handles all write transactions, while the secondary nodes can be used to handle read-only transactions. Other names for the latter are standbys, hot-failovers, or read replicas. The flow of data inside each node is the same as in *Classic*, with additional network traffic between the instances for shipping the log. This architecture has no further costs besides the instances themselves as long as everything is deployed in the same data center, but when instances are put into different data centers, cloud providers typically charge for network traffic.

Discussion. Having multiple replicas of the entire database increases availability, durability, and adds processing capacity for read-only transactions. In case of a failover, a replacement instance can be spawned automatically in the cloud, which is more convenient than in on-premise deployments where machines need to be physically replaced. In order for data loss to occur, all nodes would need to fail before replacement instances

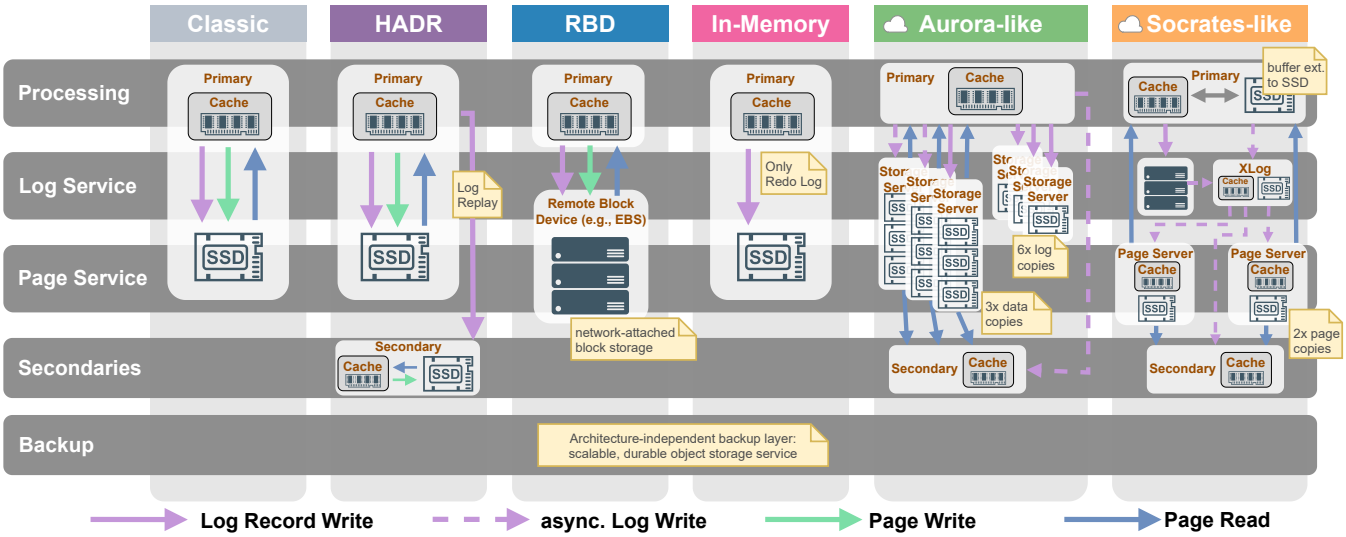

Fig. 2 Architecture overview with data flows and technologies used.

can be provisioned and restored from the remaining instances, which becomes increasingly unlikely with a larger number of secondaries. Secondaries need to replay all changes on their own local copy of the data, which reduces their capacity for processing read-only transactions. The maximum dataset size of *HADR* is the same as for *Classic*, as each replica has to keep a copy of the entire dataset.

2.3 Remote Block Device (RBD)

Remote Disks. As mentioned above, instance-local storage is ephemeral, so data is lost when an instance is stopped or fails. To solve this problem, cloud providers started to offer remote block storage early on, e.g., AWS Elastic Block Store (EBS) or Azure Managed Disks. This storage is exposed to compute instances as a regular block device, but internally data is sent over the network and replicated like in a storage area network (SAN). Amazon, for example, uses dedicated hardware (called Nitro) to connect an instance to the storage devices (likely via NVMe over Fabrics) so that bandwidth is not shared with regular network traffic [19]. Due to the different pricing, durability, and provisioning model, we treat systems that store the data on remote block devices as a separate architecture.

Discussion. As remote disks can be re-attached to other (smaller or larger) instances, this architecture offers a crude solution for scaling storage independent of compute. Since the storage device is internally replicated and survives instance failures, it offers higher durability than instance storage. Cloud providers even offer different durability options [20, 21]. While in theory there is a feature for attaching a remote disk to multiple instances at the same time, the raw unsynchronized access over the block device interface makes this only feasible for special applications, i.e., distributed file systems. *RBD* can be seen as an optimization of *HADR*, as instead of the whole instance, only the storage device is (internally) replicated to achieve higher durability. This saves cost, but takes away the option of having replicas handle read-only transactions. As we show in Section 4.1, remote disks have worse latency and throughput than instance-local NVMe, placing *RBD* at a potential performance disadvantage compared to *Classic* and *HADR*.

2.4 In-Memory

Simple And Fast. Before discussing cloud-native architectures, we examine the *In-Memory* architecture, which is somewhat orthogonal to other architectures. It keeps the entire dataset in RAM, which enables major simplifications in the DBMS design. There is no need for a buffer manager nor for storing the data on fixed-size pages. Transactions directly access and modify in-memory data structures. As Figure 2 shows, redo log records are still written to SSD to guarantee transaction durability. The undo log is kept in memory, as no uncommitted changes are written out (*no-steal*). During normal processing, no data other than the log is written to instance storage, and no data is read from SSD at all. On startup, the entire database is loaded from some persistent snapshot (not shown in the Figure 2). Cloud providers today offer instances with dozens of terabytes of main memory, but they are expensive and their avail-

ability is limited [22]. But even regular instance types offer up to 4 TB of main memory, e.g., the `x2iedn.32xl` in EC2, allowing large workloads to be run.

Discussion. *In-Memory* achieves high performance, but keeps the entire dataset in expensive DRAM. Intuitively, it should best fit workloads with high transaction rates on small datasets. *In-Memory* has the same durability as *Classic*, as both store the log on instance storage.

2.5 Aurora: Disaggregated, Redundant Storage

Separate Storage Servers. Cloud providers offer fully managed OLTP services such as Azure SQL Database, GCP Cloud SQL, and the multi-engine AWS RDS. One of the offered OLTP engines in RDS is Aurora, an in-house development by Amazon [1]. To differentiate Aurora's underlying architecture from the commercial product, we will refer to it as *Aurora-like*. In this architecture, the database is outsourced from the primary to a fleet of multi-tenant storage servers. The data is sharded into chunks of 10 GB (which the authors call *protection groups*), and each chunk is managed by a different group of six instances that store the data on their local SSDs. As Figure 2 shows, the primary does not write dirty pages back to the storage layer. Instead, it sends WAL records for a particular chunk to all of its six servers, and considers them durable when at least four of them have acknowledged it. Storage servers continuously apply WAL records to page images and garbage collect old log records. Materialized page images are retained only on three of the storage nodes to reduce the storage footprint [23]. The consistency model of *Aurora-like* is that of a distributed, quorum based system. However, the primary holds bookkeeping information, so during normal processing it can pick a single node when it needs to fetch a page image from the storage servers. Only during recovery, the latest state is reconstructed via quorum reads. *Aurora-like* supports read-only replicas, which replay the log from the primary as in the *HADR* architecture.

Discussion. This architecture decouples storage and compute, which allows scaling them independently. Since both the log and the data are replicated multiple times, the durability of the system is very high. At the same time, keeping at least three full copies of the database on instance SSDs incurs significant cost for some workloads, as detailed in Section 5.2. *Aurora-like*'s storage servers make most sense in a multi-tenant environment, as the substantial fleet of nodes required to adequately distribute data chunks can be shared among many tenants. Thus, this architecture is inherently cloud-native.

2.6 Socrates: Separate Log and Page Services

Separate Logging And Page Services. Microsoft also offers a commercial cloud-native OLTP system called SQL Database Hyperscale. We call its architecture, consistent with its academic name [2], *Socrates-like*. It partitions the database across multiple page servers, each of which manages a shard for which it applies log records, serves pages to the processing nodes, and creates snapshots/backups. In contrast to *Aurora-like*, a dedicated log service receives log records from the primary, persists them, and disseminates them further to secondaries and the respective page servers. As shown in Figure 2, the log service itself consists of two components: a compute instance with local storage, and a remote block device (which the authors call *landing zone*). The primary writes log records to the block device for synchronous transaction commits, while the log instance asynchronously consolidates, distributes, and eventually archives the log. Both the primary and the page servers employ a technique called *resilient buffer pool extension* (RBPEX), i.e., each instance combines its memory and local storage into one large uniform buffer pool. Page servers use a covering cache for their particular shard, i.e., pages move between memory and SSD but are never evicted.

Discussion. *Socrates-like* divides the DBMS into five components: primary, storage service, log service, a number of secondaries, and the backup. The remote disk together with the backup provide durability. Page servers are conceptually only a cache on top of the backup, enabling low-latency access. Each page is cached on two nodes [24], which is different from the *Aurora-like* model which keeps three copies. Both architectures offload the application of log records to pages from the primary, which does not have to write back dirty pages. *Socrates-like* can increase availability by adding secondaries and/or page servers.

2.7 Backup

OLTP Needs Backups. So far we have excluded the backup component from our discussion, but it is an important part of every OLTP system. A dedicated separate backup facility provides high durability for the log archive and database checkpoints. Some architectures integrate it actively into their design, e.g., *In-Memory* has to write the entire dataset to SSD on shutdown, and periodically for snapshots, which essentially are also backups. *Socrates-like* relies on the backup for durability and utilizes it for fast point-in-time recovery. In contrast, *Aurora-like* and *HADR* already store redundant copies of the data in their main system, so they do not

strictly need it for durability. However, even these two still require a backup, as in practice all production-level OLTP systems need to guard against data loss or corruption due to end user mistakes or software bugs [25].

Backups in the Cloud. The target service for the backup should be scalable, highly durable, and cheap in terms of storage cost per month. It does not, however, have to provide cheap point accesses nor low latency. All cloud providers offer a foundational storage service that fits those requirements well, e.g., AWS S3, Azure Blob Storage, or Google Cloud Storage. Since these services are deployed and operated at a very large scale, it is hard to match their price point and reliability with other solutions. Thus, all sensible cloud systems should just back up their data regularly to such a service.

2.8 Related Work

While there is a large body of work on running analytical workloads in the cloud [4,26–36], with some focusing on serverless query processing [37–39], there is less research on OLTP in cloud environments [40–45]. Li et al. [46] give a qualitative comparison of cloud database systems. To the best of our knowledge, our paper is the first extensive and quantitative analysis of different OLTP designs in the cloud.

Commercial Cloud OLTP Systems. In the recent past, there has been one industry paper each year about commercial cloud-native OLTP systems. Amazon presented Aurora, which is available in AWS RDS, in 2017 [1] and 2018 [23]. Microsoft described Socrates [2], which is commercially marketed as SQL Database Hyperscale, in 2019. We presented the architectures of both systems in detail in Section 2.5 and Section 2.6. In 2020, Huawei proposed Taurus [47], which refines the Socrates approach and presents implementation details for page servers. Finally, in 2021 Alibaba presented PolarDB Serverless [48], which exploits RDMA to enable efficient transaction processing in disaggregated data centers. We do not model PolarDB, as RDMA is not available in AWS EC2. So while the industry is clearly interested in cloud OLTP systems, we are only aware of one academic work in this area which analyzed different design decisions of disaggregated storage architectures by implementing them in PostgreSQL [3].

Model-Based Approaches. Chatterjee et al. [8, 49] propose a self-optimizing key-value store as part of their *Data Calculator* project [50]. Their goal is to find an instance optimized data system [51] using LSM, B-tree, and hash-table templates as building blocks, and taking cloud hardware and costs into account. Leis and Kuschewski [52] employ a cost-optimizing approach to analytical query processing in the cloud. They consider data size and CPU time for analytical queries that use full table scans and model a Snowflake-like distributed query processing architecture where data is stored on S3 and cached on compute nodes. In contrast, we model index operations, compare six different architectures, and consider additional workload parameters that are important in the context of OLTP, e.g., latency, durability and availability. Ziegler et al. [10] analyze different architecture types for distributed OLTP in a high-level qualitative fashion. This paper, in contrast, focuses on the market-dominating single-writer design and performs a detailed quantitative cost-based comparison. Such cost-optimizing approaches are becoming more popular in the context of cloud query processing, as the many recent papers on this topic show [53–57]. Mozafari et al. [42] developed a framework called *DBSeer* which combines both analytical and regression models for OLTP workloads. We model log writes, dirty page writes, and cache misses similar to them, and share some of their assumptions, e.g., transactions use index operations instead of full table scans. Their paper predates cloud-native OLTP systems and the widespread adoption of NVMe SSDs. It focuses on modeling the CPU, memory, and IO characteristics of MySQL. In contrast, our work is based on recent storage engines which can exploit the massive IO parallelism of modern NVMe drives, models multiple architectures including disaggregated cloud-native ones, and considers a large variety of cloud compute instances to find the cost-optimal setup.

3 A Model Framework For Cloud OLTP

3.1 Methodology

Downsides of Comparing Existing Systems. Which of the architectures described in Section 2 is the best? They all utilize different building blocks for data storage, access, and log writing. With hundreds of instance types available in the cloud and diverse application requirements for performance, availability, and durability, this question becomes a multi-dimensional problem with no obvious answers. One could imagine selecting existing systems that implement each of the presented architectures and running all possible workload combinations on them. However, the vast number of workloads and deployment options makes such an approach impractical. For instance, the analysis in Section 5 required evaluating ∼380M model configurations. Furthermore, measuring existing (commercial) systems would not conceptually compare the architectures themselves but rather specific implementations of

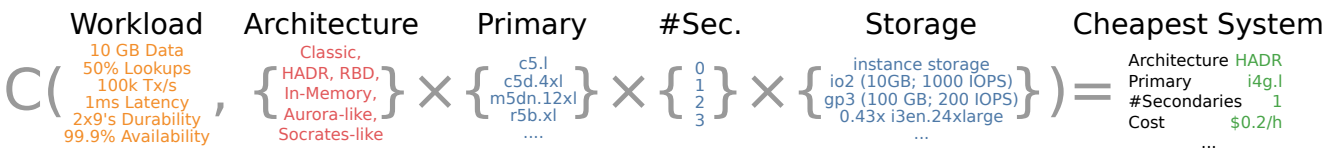

Fig. 3 Model inputs: workload constraints and possible hardware options; Outputs: concrete system with performance and cost.

them. Implementations vary significantly in code efficiency, pricing models, and hardware options.

Modeling Ideal Archetypes. For the reasons mentioned above, we instead design an analytical model that captures the essence of each architecture and predicts their behavior based on real-world cloud pricing and hardware capabilities. The balance to strike here is — as with every model that abstracts away from the real world — to identify the right parameters to produce meaningful results, and leave out the unimportant ones to keep to model understandable. Our model is based on the expected values of CPU work, I/O operations, and network traffic that each architecture requires for a given workload assuming an underlying engine that is both efficient and can utilize the available hardware. Elastic resources, e.g., a remote block device or a multi-tenant storage service, are sized appropriately for those requirements. For fixed resources like the primary, only instance types with sufficient CPU, storage, and network specifications are considered. For our analysis in Section 5, we calibrated the model with data from AWS (Section 4.1), though it could also be populated with data from other cloud providers such as Google Cloud or Microsoft Azure, which are structurally similar to AWS.

3.2 Model Overview

Exhaustively Enumerating All Configurations. As Figure 3 illustrates, our model acts as a cost function that takes a workload specification and a list of cloud hardware options, and outputs the system setup with the lowest cost. To compute this function, the model framework iterates through all architectures and cloud compute instances, assembling a separate system configuration for each. For the three architectures from Figure 2 with additional components, e.g., a remote block device or a log service, it further evaluates each combination of device type and viable instance type (for that service). Then, it computes the achievable performance for each configuration. Configurations that do not meet the workload requirements are removed. Finally, the remaining systems are sorted by ascending total cost. For example, one of the resulting configurations could be the *RBD* architecture on an AWS `c5.4xl` instance, with an attached EBS `io2` device provisioned with 100 GB and 8000 I/O operations per second (IOPS). To calculate its performance capabilities, the model considers the amount of all resources one update/lookup consumes, and checks how many the system can handle before the first resource is exhausted. Not all transactions take the same amount of resources though; for example, if 10% of the data is in an in-memory cache, then accessing it only requires CPU and memory resources, while for the remaining 90%, additional storage or network requests are considered.

Illustrating Example. Imagine the `c5.4xl` instance mentioned above manages a dataset of 96 GB, from which 32 GB fit into the in-memory cache while the remaining 64 GB need to be accessed from the block device. The specified target throughput is 9K lookups and 1K updates per second, easily handled by the 16 CPU cores of the instance (see Section 4.1). Ignoring group commit for now, each of those 1K updates flushes one log entry to SSD. Assuming a random access pattern, 2/3 of the operations result in a cache miss and read data from SSD, which translates to 6.66k IOPS. Thus, the EBS device needs to be provisioned with at least 7.66k IOPS in total. Note that our actual model would employ group commit, account for dirty page writes, and provision the EBS device with the exact amount of required IOPS for that.

Model Assumptions For All Architectures. To model the behavior of an OLTP system, we make a number of general assumptions. All architectures store data in page-based clustered B-tree indexes. Key accesses are assumed to be randomly uniform, or use a Zipf distribution for skewed workloads. The systems are able to do effective caching using the available storage hierarchy, i.e., DRAM and/or local NVMe caches. Transactions are hardened via write-ahead logging using group commit. Based on these assumptions we calculate the expected hardware resources a lookup or update requires. If, for example, the dataset is ten times the size of the cache, then a random lookup will have to load the data from storage with 90% likelihood, thus requiring on average 0.9 read ops. The details on how we derive these expected values are presented in the rest of this section. We also make the implementation of the model available for full reference [58].

3.3 Workload Definition

Parameters. How well the six architectures presented in Section 2 perform and how expensive they are depends on the workload requirements, which we specify through the following parameters:

Parameter	Unit	Range	Abbr.
Dataset size	Gigabyte	10 GB - 100 TB	DB
Transactions	#/s	1k/s - 100M/s	TXS
Lookup Ratio	%	0% - 100%	READ
Durability	9's/year	1×9 - 11×9's	DUR
Latency	µs	1µs - 1ms	LAT
Availability	Failure types	Node, AZ, Region	AVAI

Dataset Size. The most basic property of an OLTP workload is the dataset size, which we assume to be static. We analyze datasets that fit into the memory of a moderate node, e.g., 10 GB, as well as ones that exceed the total storage capacity of even the largest cloud SSD instances, e.g., 100 TB. Only systems that decouple their storage layer and partition the data over a cluster of machines can handle the latter.

Performance. Most OLTP workloads can be decomposed into single-tuple index operations, which is why we chose to focus on point lookups and updates. We usually model the most challenging situation where the access pattern is randomly uniform. When we look at skewed workloads, we use a Zipf distribution and only consider lookups. We specify system performance in terms of transactions per second the system can sustain in steady state and which percentage of those transactions are (read-only) lookups. An update (write transaction) is modeled as a lookup which additionally marks the affected data page as dirty and writes a log entry. In our experiments, transaction rates range from a light load of 1000 tx/s up to a very demanding 100M tx/s to cover the whole spectrum of possible OLTP workloads.

Latency. We consider the average latency of transactions in the database system itself, excluding the round-trip time to the client. Sensible latency bounds vary in the range of 1 µs up to 1 ms, as an in-memory B-tree lookup takes about one microsecond. Even if an operation has a cache miss and needs to retrieve the data from remote storage in the same data center, this operation takes less than one millisecond (see Section 4.1). In order to improve average operation latency, systems have to cache a larger share of the data.

Availability. Availability is commonly measured as the percentage of successful requests or the time the system is able to answer requests [59]. Cloud instances use commodity hardware, so they are susceptible to occasional hardware failures [60]. In addition, there are more components and systems that could lead to an instance not being reachable, e.g., top-of-rack switches, power supply, DNS, public internet connectivity, etc. [61, 62]. Cloud providers typically offer a service level agreement (SLA) that contractually guarantees a certain "uptime" for an instance on a monthly basis. Typical promised uptimes are 99.5% per month, but range down to as low as 95% for certain instances with HDDs [63, 64]. If the cloud provider misses their SLA, i.e., a machine is not reachable for the guaranteed time, the customer is eligible to a refund. AWS, for example, gives back 10% (30%; 100%) of service credits if the machine is reachable less than 99.5% (99%; 95%) of the time. But even an availability of 99.5% means an instance could be unreachable for more than 3 hours per month, which is unacceptable for mission-critical OLTP systems. A refund on the raw instance cost will likely not compensate the losses of a company that cannot access their data. Since there is no option to increase the availability of a single instance, one must design cloud systems for failover, meaning that when (not if) the primary becomes unavailable, another instance needs to take over. Correlated failures are also a risk. When an instance becomes unavailable, other instances in the same rack, or even the same data center, are disproportionately affected as well. Thus, cloud providers recommend deploying additional instances in other, geographically-close data centers often called "availability zones" (AZs). Our model allows placing secondary instances and multi-tenant services, e.g., Aurora storage servers, across AZs to reduce the impact of failures affecting a whole data center. However, this can increase operation latency and have a non-negligible impact on cost, as most cloud providers charge for inter-AZ network traffic by volume [65, 66]. This policy might change, though, as Azure recently switched to no longer charging for inter-AZ traffic [67].

Durability. A central aspect of any OLTP system is durability. In contrast to availability, it describes the ability of a system to access data *at some point* in the future, possibly after recovery has run. Cloud providers usually state it as the number of leading nines of the probability that data is intact after one year, e.g., 0.9997 would translate to 3×9's. We assume that OLTP systems do regular backups to a cheap object storage service (see Section 2.7) with such high durability that the loss of the backup is practically irrelevant. To show that this assumption is justified, let us assume the backup is stored in chunks of 100 MB on AWS S3. With the 11×9's durability that S3 provides per object per year [68], one can back up 1 PB of data and expect only one chunk to be lost in 10,000 years, which is unlikely enough to be ignored. We further model that all architectures keep the last hour of changes in their system, and that log records older than that are incorporated into the log archive/backup. Thus, the durability of the whole

OLTP system translates to the durability of the tail of the WAL. How different architectures store it determines their durability.

3.4 General Model Traits

Assumptions in Detail. We assume that all architectures store their data in page-based clustered B-trees indexes. By default, we configure a page size of 4 KB to match the block size of NVMe drives. Larger page sizes would increase read/write amplification for random access workloads. Each CPU core can process a fixed number of B-tree index ops/s, and in-memory transactions scale with the number of CPU cores (see Section 4.1), i.e., the system is not limited by unscalable synchronization. Thus, we ignore the overhead of transaction isolation and synchronization of data structures. In practice, there will be an architecture-independent CPU overhead which does not affect how architectures fare relative to each other. Secondary indexes can be modeled with additional lookups and a larger data size in the workload specification. We assert that 90% of the memory of a machine can be used as cache, or for *In-Memory* to store the entire dataset. The cache hit rate for random accesses is calculated as the memory size divided by the total data size. We model the probability of a page being dirty as the ratio of updates to all transactions for architectures that write back dirty pages. Furthermore, we assume that the DBMS is able to keep inner B-tree nodes always in the buffer cache. Thus, an operation can have either zero pages misses (if the leaf node is in the cache) or exactly one page miss (if the leaf node is on secondary storage). The model considers WAL records and page images that are sent over the network, but ignores client traffic as it is the same across all architectures. For systems that can employ secondaries, we limit their number to a maximum of three. The secondaries use the same instance type as the primary, as they must be able to handle the primary's workload when a failover occurs. For the same reason, we exclude the first secondary from read-only processing, as otherwise the system capacity could be exceeded. By default, all resources of the system are placed into the same data center. All architectures employ group commit by default, as this feature is widely used in real systems and often reduces the required IOPS significantly. We assume each log record has a 48 byte header, and the total size for log storage is set to keep one hour of updates, after which log records are staged to a remote log archive (e.g., a cloud object store such as S3) which we do not model as it is common across architectures.

Algorithm 1 Computing the properties of *Classic*

computeClassicProperties(W, N)
Input: workload parameter W,
cloud compute node N
Output: system properties S for *Classic*

1: **if**$(W_{dbSize} + ariesLogStorage > N_{storageCapacity})$ **return** $\perp$
2: $readsPerUpdate = p(cacheMiss)$
3: $logWritesPerUpd = \begin{cases} \frac{W_{ariesRecordSize}}{N_{maxIOSize}} & \textbf{if } W_{groupCommit} \\ 1.0 & \textbf{otherwise} \end{cases}$
4: $writesPerUpdate = p(dirtyPageEvict) + logWritesPerUpd$
// Limit updates by the most restrictive resource
5: $S_{updates} = min \left\{ \begin{array}{l} \frac{N_{readOps}}{readsPerUpdate}, \frac{N_{writeOps}}{writesPerUpdate}, \\ \frac{N_{cpuCycles}}{cpuCyclesPerTx}, W_{reqUpdates} \end{array} \right\}$
// Calculate the resources left for lookups
6: $readsPerLookup = p(cacheMiss)$
7: $writesPerLookup = p(dirtyPageEvict)$
8: $cpuLeft = N_{cpuCycles} - S_{updates} * cpuCyclesPerTx$
9: $readOpsLeft = N_{readOps} - (S_{updates} * readsPerUpdate)$
10: $writeOpsLeft = N_{writeOps} - (S_{updates} * writesPerUpdate)$
// Limit lookups using the remaining resources
11: $S_{lookups} = min \left\{ \begin{array}{l} \frac{readOpsLeft}{readsPerLookup}, \frac{writeOpsLeft}{writesPerLookup}, \\ \frac{cpuLeft}{cpuCyclesPerTx}, W_{reqLookups} \end{array} \right\}$
// The data is durable if the SSD does not fail
12: $S_{durability} = 1 - AFR_{SSD}$
// The average latency of transactions
13: $S_{txLatency} = \begin{array}{l} p(cacheHit) * memoryOpLatency + \\ p(cacheMiss) * nvmeReadLatency \end{array}$
14: **return** $(S_{updates}, S_{lookups}, S_{durability}, S_{opLatency})$

Model Implementation. Our model is implemented as a 4,000 line C++ program. The hardware configurations are compiled with data from three sources: Basic instance information is retrieved programmatically, enhanced with manually crawled data from the vendor documentation, and calibrated through extensive experiments (see Section 4.1). In order to process the 1.5M combinations that are possible for each set of workload parameters, the program eagerly removes provably suboptimal combinations. This is done by pruning configurations that do not meet workload requirements and by employing heap select instead of exhaustively establishing a total order over qualifying options. With these optimizations, the invocation of the model for one specific workload takes around one second, which even allows running the model in online scenarios.

3.5 Architecture-Specific Model Traits

Classic. The *Classic* architecture runs on a single instance with local storage which has to fit the entire dataset. Its properties are calculated as shown in Al-

gorithm 1. First, in line 1, we check if the instance has sufficient storage capacity for the database and the WAL. If not, the model immediately returns $\perp$, indicating that the workload cannot run on this instance. Then in line 2-4 the model calculates how many storage reads and writes are required on average per update transaction. A page read occurs for every cache miss, and a page write occurs when a dirty page is evicted from the cache. When group commit is enabled (the default), the log records of multiple transactions can be hardened with a single write. Without group commit, each transaction requires a separate write for its WAL records. In line 5, the achievable update rate is determined by the most constrained resource (reads, writes, or CPU). It is capped by the workload's requirements, so the model can calculate how many resources are remaining for lookups (line 8-10), which are then calculated in a similar fashion (line 11). The durability (line 12) of the *Classic* architecture is determined by the annualized failure rate (AFR) of SSDs, as they are the component with the highest chance to fail. Also, storage device failures are most often not transient and lead to data loss [69]. A study by Microsoft [70] showed an SSD AFR ranging between 0.1% for enterprise grade devices up to 1.0% for consumer drives used in data centers. The average transaction latency (line 13) is modeled as the weighted average of in-memory and SSD access times, which we measure in Section 4.1. For brevity, the general model parameters as well as the remaining architecture specifications are listed in Appendix A.

HADR. In addition to the constraints that apply to the primary instance for *Classic*, we put a resource limit on log records that are sent to secondary instances based on the network bandwidth of the instance. In case those secondaries are placed in other availability zones, we account for inter-AZ traffic costs as well. Since secondaries need to individually apply log records they receive from the primary, we model their remaining capacity for lookups to be equal to that of the primary. Also, we do not make any assumptions about the possibility of partitioning lookups to increase cache efficiency on the read-only replicas, and employ the same cache hit rate, and thus also latency, on all instances. For our analysis, we vary the number of secondary nodes between one and three. *HADR* has a much higher durability than *Classic*, because for data loss to occur, all instances would need to fail simultaneously before the dataset could be replicated to newly spawned nodes. We model the probability that k instances fail in the same repair interval with a Poisson distribution $Pr(X = k) = \lambda^k * e^{-\lambda}/k!$ where λ is the number of failures we expect during one repair interval. We calculate λ as $\#\text{nodes} * p_{\text{node failure}}$ and estimate $p_{\text{node failure}} = \frac{1-\text{monthly node availability}}{\#\text{repair intervals per month}}$. The number of repair intervals per month is calculated via the *mean time to repair* (MTTR), for which we conservatively assert that we can copy the dataset to a new instance with 50 MB/s even if the remaining nodes are under high load. Finally, when we have n instances in total, we calculate the durability as the probability that at most $n - 1$ instances fail in every single repair interval of a year. As an example, with a monthly availability of 99.5%, an MTTR=10h and two secondaries, *HADR* would have 8×9's durability.

Remote Block Device. The storage volume is sized so that it exactly fits the dataset plus the log tail, i.e., 1 hour of WAL records. It is provisioned with enough bandwidth and IOPS to handle the workload. If one device does not suffice, multiple devices are combined in a RAID-0-like fashion. There are additional instance limits that apply, e.g., an AWS `c5.large` instance can only handle at most 27 EBS devices with an aggregate bandwidth of 594 MB/s and 20k IOPS [71, 72]. Thus, larger instances are required for higher IOPS. Latency of remote block devices is higher than, for instance, local storage. When the system fails, the block device is attached to a new instance, which needs to start up and run database recovery. The durability of remote storage devices is documented by cloud providers, e.g., in AWS "99.8-99.9%" for an `io1` volume [73], while Azure replicates data "across three availability zones in the region" for zone-redundant managed disks [21]. The primary does not use local storage in this design.

In-Memory. All data is kept in memory, while the log is written to local storage. We estimate an index operation to cost the same CPU resources as in the other architectures, and all lookups and updates have in-memory latency, as all B-tree nodes are kept in memory. Durability is equivalent to the *Classic* architecture.

Aurora-like. The primary uses an in-memory buffer cache but does not require local storage. Both the primary and the secondaries need sufficient network bandwidth to receive page images from the storage servers when they have a cache miss. Additionally, the primary requires enough network bandwidth to send each log record to six storage servers and all secondaries. The storage servers are modeled as a multi-tenant scale-out service. Thus, a particular *Aurora-like* database can employ one or more NVMe-backed instances, not necessarily matching the primary instance type. In fact, the multi-tenant nature of this component makes it reasonable to provision arbitrary fractions or multiples of an instance for one specific database. The factor is set so that the log tail can be stored six times and the dataset three times. It is further increased if more IOPS, network bandwidth (for more log writes and page reads),

or memory (for lower latency page reads) are required. Unlike in the commercial Aurora system, we place all storage servers in the same AZ as the primary for a meaningful comparison with the other architectures, and vary this only for the availability analysis.

Socrates-like. The primary uses instance storage not for storing the dataset, but for extending the buffer pool. In case there is a cache hit for a page that resides on SSD, both a storage read and a write are required to swap it with a page from memory. The same applies to the buffer pool of page servers, which are modeled much like the *Aurora-like* storage servers as a fraction/-multiple of an instance with local NVMe storage. Their combined memory and SSD capacity is sized to fit two copies of the entire database. A remote block device is provisioned and conceptually attached to the primary for storing log records, and another fraction of an instance with local storage is allocated for the log service.

4 Model Calibration and Validation

4.1 Calibration in AWS

EC2 Instance Types. We instantiate our model with the specifications of the hardware available in AWS, currently the largest cloud services provider. All data we use is as of March 2024. A basic list of EC2 instances with their CPU cores and frequencies, memory capacity, network bandwidth, and storage devices is retrieved from Vantage [74], a cloud cost management provider. We manually assembled data not available from Vantage, such as instance storage read and write operations, EBS instance limits for IOPS and bandwidth, and provisioning constraints for EBS, from the AWS documentation. We exclude GPU and machine learning instances. We also filter out instances that lack clearly quantified performance metrics, such as those with network throughput described as "moderate" or similar, as well as CPU burstable and legacy instance types. For instances with burstable network or EBS performance, we only consider the baseline performance. These decisions minimize exposure to noisy neighbor effects, as for the remaining instances AWS statically assigns CPU cores, memory bandwidth, and IOPS to each instance slice. All in all, we end up with 555 different EC2 instance types and five different types of EBS volumes. For the multi-tenant storage and page servers, we consider the largest NVMe-backed instance sizes of these types: *is4gen*, *im4gn*, *i4i*, *i4g*, *c6gd*, *r7gd*, *i3en*, *i3*, *r6gd*, *m5d*, *m6gd*, *r6id*, *r5dn*, *m7gd*, *m5ad*, *m6id*, *c5ad*, *c5d*, *m6idn*, *c6id*, *c7gd*, *r6idn*, *m5dn*.

AWS Prices. All prices we report are for the `us-east-1` region. For EBS, S3, and network traffic, we use the list price. For EC2 instances, there are different pricing models to consider. One can choose between on-demand, provisioned for one or three years (paid up-front or monthly), savings plans, and spot instances. A cost-optimizing company will likely use a mixture of 1-year and 3-year reserved instances to best fit changing workload requirements. For example, the hourly price for the different reserved categories for a *c7g.16xl* instance varies between \$0.88 and \$1.53, which is on average 52% of the on-demand price of \$2.32. Across all EC2 instance types, a mix of reserved instances on average costs 50.8% of the on demand price. For simplicity, we assume a uniform discount of 50% on the on-demand instance prices in this paper. We do not consider spot instances, although they provide even higher discounts, since we model steady-state production OLTP workloads and spot instances could be reclaimed by AWS at any time due to high demand. Note that the discount on EC2 prices is conceptually different from any blanket discount that big companies may receive, as the latter applies to all AWS resources, while the former changes the relative pricing model of resources inside AWS, and thus has an impact on the outcome of our analysis. Specifically, it makes EC2 instances relatively more attractive compared to storage services such as EBS, which do not offer discount models. In conclusion, calculating with 50% of the EC2 on-demand prices and regular prices for all other resources creates a more realistic AWS pricing structure.

Network. Network is modeled in terms of bandwidth limits and latency. Experiments have shown that the limit can be reached with page-sized packets. Even with smaller packets, high packet rates can be achieved, e.g., 2.2M/s on an `m5.24xl` instance [75]. We assume a latency of 90µs (2ms) between two instances in the same data center (different AZs in the same region) [76–78]. Although network latency in cloud data centers is not stable due to multi-tenancy and quality-of-service mechanisms, we do not model complex latency distributions. Extending our model or exploring the impact of techniques like ENA Express [79] on latency variability would be interesting future work.

Storage. We performed extensive experiments to determine the limits of both instance storage and EBS using the `fio` benchmarking tool (v3.32) configured with direct I/O and the libaio engine on an `i3en.24xl` instance. Its local storage comprises 8×7.5 TB NVMe SSDs, which we combine into a RAID-0 using `mdadm`. The EBS device is an `io2` volume provisioned with the maximum possible 64K IOPS. Based on the IOPS and latency results in Table 1, instance storage in our model is configured with a write latency of 44µs, a read latency of 132µs, and 80% of the documented read ops.

Table 1 Model calibration using microbenchmarks.
EBS: `io2`(64k iops); NVMe: `i3en.24xl` instance store.

Device	Thr.	Q-Dep.	IOPS	Limit	Op. (Fsync) Lat.
4 KB Random Reads					
NVMe	32	64	1.66M	2M	1.2ms
NVMe	1	1	7.5K	2M	132µs
NVMe	1	64	193K	2M	331µs
EBS	1	1	2.7K	64K	374µs
EBS	1	32	64.0K	64K	500µs
Sequential Writes with fdatasync every 100 operations					
NVMe	1	1	22.5K	1.6M	44us(31µs)
NVMe	16	64	935K	1.6M	1.1ms(1.1ms)
EBS	1	1	3.4k	64K	292µs(7.4µs)
EBS	16	64	3k		336ms(336ms)
Sequential Writes with fdatasync every operation					
NVMe	1	1	20.9k	1.6M	42µs(45µs)
NVMe	16	64	283K	1.6M	3.6ms(3.6ms)
EBS	1	1	3.4k	64k	292µs(6.5µs)
EBS	16	64	2.9k	64K	343ms(344ms)

For EBS, we assume 100% of the provisioned IOPS, and use a write (read) latency of 292µs (374µs).

B-tree Operations. To approximate how many transactions an OLTP system can achieve, we benchmark B-tree operations using our high-performance storage engine, LeanStore, which is available as open source [7,80]. Again, we use an `i3en.24xl` instance, which has 96 virtual CPU cores running at 3.1 Ghz and 768 GB of main memory. We create a B-tree with 147M entries (8 byte key and a 60 byte value), totaling 10 GB of data. The buffer cache is configured so that the entire tree fits into memory. With a single thread, LeanStore achieves 1.8M lookups/s, or alternatively, 1.1M updates/s. With 10 threads, it achieves 18M lookups/s. These results illustrate that in-memory operation performance is rarely the bottleneck with modern storage engines. However, we conservatively cap the number of transactions to 1M/s per CPU core ($\sim$ 4000 cycles) to ensure that, even on small instances, enough CPU capacity is left for non-transaction processing related work.

Concrete Example. To illustrate how the model uses all the numbers obtained above, we apply them to Algorithm 1 for an example workload of DB=100GB, TXS=200k tx/s, READ=50% on a *c7gd.4* instance. With a log retention time of 1 hour and 68 byte tuples, the required storage is calculated as $100\,\text{GB} + 3600s * 200k\,tx/s * 0.5 * (2 * 68 + 48)\,\text{byte} = 161\,\text{GB}$ which fits within the 950 GB instance storage. The 16 CPU cores can sustain $16 * 2.5\,\text{GHz}/4000\,\text{cycles} = 10\text{M}$ tx/s. With 32 GB RAM, the probability of a cache miss for a page access is 68%. For the instance store's 268k read operations, this translates to at most $268k/0.68 = 316\text{k}$ tx/s from a storage read perspective. Using group commit, each update requires $(2{*}68{+}48)\,\text{byte}/4096\,\text{byte} = 0.045$ log writes. Dirty page writes per transaction are estimated as $\#updates/(\#updates + \#lookups) * 0.68 = 0.34$, so together with log writes, the storage can handle $134k/(0.34 + 0.045) = 349\text{k}$ updates/s. After the updates have been accounted for, the remaining CPU cycles would suffice for 9.9M, storage reads for 216k, and storage writes for $(134k - 100k * 0.39)/0.34 = 281k$ lookups/s. Finally, the average transaction latency is computed as $0.32 * 2\mu s + 0.68 * 132\mu s = 90\mu s$.

Table 2 Validating model accuracy for the different architectures. Each workload is executed on the predicted cost-optimal instance. READ=70%, DUR=1x9, LAT=1ms, AVAI=Node.

Workload		Model Prediction		Measured
Dataset	Target Perf.	Architecture	Instance	Perf.
1 TB	100k tx/s	Classic	*is4gen.xl*	104k tx/s
1 TB	100k tx/s	In-Memory	*x2gd.16*	3,150k tx/s
1 TB	100k tx/s	HADR	*is4gen.xl*	105k tx/s
1 TB	100k tx/s	RBD	*c6gn.12*	99.9k tx/s
1 TB	100k tx/s	Aurora-like	*c7gn.l*	106k tx/s
1 TB	100k tx/s	Socrates-like	*c6gd.2*	171k tx/s
10 GB	1M tx/s	Classic	*i4g.l*	1.21M tx/s
10 GB	1M tx/s	In-Memory	*r6gd.l*	1.20M tx/s
10 GB	1M tx/s	HADR	*i4g.l*	1.23M tx/s
10 GB	1M tx/s	RBD	*c6g.2*	3.12M tx/s
10 GB	1M tx/s	Aurora-like	*r5n.l*	1.13M tx/s
10 GB	1M tx/s	Socrates-like	*r6gd.l*	1.05M tx/s
100 GB	2M tx/s	Classic	*r6gd.4*	9.71M tx/s
100 GB	2M tx/s	In-Memory	*x2gd.2*	4.80M tx/s
100 GB	2M tx/s	HADR	*r6gd.4*	9.18M tx/s
100 GB	2M tx/s	RBD	*r5b.4*	2.43M tx/s
100 GB	2M tx/s	Aurora-like	*r6g.4*	3.83M tx/s
100 GB	2M tx/s	Socrates-like	*x2gd.2*	4.81M tx/s

4.2 Model Validation using LeanStore

Implementing Architectures. We discussed in Section 3.1 that there are too many configurations to exhaustively validate them all. Nevertheless, we still want to confirm the accuracy of our model. For that, we choose LeanStore as the storage engine, as it can fully utilize modern NVMe devices, which traditional engines often fail to achieve [9]. It implements a buffer manager and logging, which means the I/O is the same as in a full-fledged OLTP system. Out of the box, we can use it for the *Classic*, *In-Memory*, and *RBD* architectures. For validating the architectures that send the WAL to a different node (*HADR*, *Socrates-like*, and *Aurora-like*), we extended LeanStore to send the WAL over the network using TCP and *io_uring* with a configurable write amplification, e.g., six for *Aurora-like*. We simulate page server accesses for *Aurora-like* and *Socrates-like* by requesting dummy pages from another node over the network with the cache miss probability our model predicts for a certain architecture, e.g., 99.6%

Table 3 Validating model accuracy for different workloads. Each workload dimension is varied individually.

Workload						Model Prediction		Measured Performance	
DB	TXS	READ	DUR	LAT	AVAI	Architecture	Instance	Throughput	Latency
100GB	**10k**	**70%**	**1x9**	**1000µs**	**Node**	classic	c6gd.l	25.1k tx/s	318µs
10GB	10k	70%	1x9	1000µs	Node	classic	c6gd.m	15.4k tx/s	519µs
500GB	10k	70%	1x9	1000µs	Node	classic	is4gen.m	20.3k tx/s	394µs
1TB	10k	70%	1x9	1000µs	Node	classic	i3en.l	29.9k tx/s	535µs
5TB	10k	70%	1x9	1000µs	Node	classic	is4gen.2	75.9k tx/s	422µs
100GB	**1k**	70%	1x9	1000µs	Node	rbd	c6g.m	1.1k tx/s	(*) 7257µs
100GB	**100k**	70%	1x9	1000µs	Node	classic	c6gd.2	129k tx/s	496µs
100GB	**1M**	70%	1x9	1000µs	Node	in-memory	x2gd.2	4,850k tx/s	12µs
100GB	**10M**	70%	1x9	1000µs	Node	in-memory	i4g.4	9,900 tx/s	1µs
100GB	10k	**0%**	1x9	1000µs	Node	classic	c6gd.l	10.5k tx/s	762µs
100GB	10k	**30%**	1x9	1000µs	Node	classic	c6gd.l	14.3k tx/s	558µs
100GB	10k	**100%**	1x9	1000µs	Node	classic	c6gd.l	25.4k tx/s	314µs
100GB	10k	70%	**{3,4,11}x9**	1000µs	Node	aurora	c6g.m	19.8k tx/s	404µs
100GB	10k	70%	1x9	**100µs**	Node	classic	x2gd.l	(**) 10k tx/s	83µs
100GB	10k	70%	1x9	**90µs**	Node	classic	x2gd.xl	(**) 10k tx/s	50µs
100GB	10k	70%	1x9	**20µs**	Node	in-memory	x2gd.2	4,850k tx/s	12µs
100GB	10k	70%	1x9	1000µs	**AZ**	aurora	c6g.m	19.8k tx/s	404µs
100GB	10k	70%	1x9	1000µs	**Region**	hadr	c6gd.l	25.1k tx/s	318µs

(*) When the EBS devices is under load, the read latency increases drastically. Our current model does not capture this property.
(**) The transaction rate is fixed to the requested rate as the focus is on the achievable latency.

for the 1 TB dataset using *Aurora-like* on an *c7gn.l* instance. Note that this way, the CPU work, disk I/O, and network I/O on the primary are realistic, while the log and page servers only execute the network stack. Since they are not doing CPU heavy work, they should not become the bottleneck in a full implementation.

Experiment 1: Validating Architectures. First, we want to confirm that our model produces accurate predictions across all six architectures. Table 2 shows the evaluated workloads. They vary in dataset size and transaction rate, with the lookup ratio set to 70%, durability to 1x9, latency to 1ms, and no guard against node failures. We configure LeanStore for each of the architectures and execute each workload on the cheapest EC2 instance that the model predicts for it. Every workload runs for 150s to ensure a steady state, from which we report the average of the last 20s. Throughput shows little variance, with deviations in any one-second interval remaining within 5% of the average. Table 2 shows that our model predictions are accurate, i.e., all configurations meet the required performance bar. It is not surprising that *RBD* achieves performance closest to the required level in the 1 TB dataset workload, as its IOPS are provisioned precisely for the high page miss rate which is the bottleneck in this scenario. For the workload with 2M tx/s, the achieved performance is up to 4.9× higher than required, but on smaller instances that cannot cache the entire dataset performance would immediately drop far below 2M tx/s. Similarly, *In-Memory* on the 1 TB dataset is much faster than required, but there is no cheaper instance that would provide enough DRAM for it. Thus, the results also indicate that our model does not recommend overly expensive instances, but indeed the cheapest ones that can still execute the workload.

Experiment 2: Validating Workload Dimensions. Now that we have established the model captures each architecture well, we validate it across a greater variety of workloads. As a baseline we use a workload with 10k tx/s on a 100 GB dataset, 70% lookups, 1x9 durability, 1ms latency, and no availability guarantees. Our model recommends using the *Classic* architecture for that on an `c6gd.l` instance, which has 4 GB RAM and 2 vCPUs. With that setup LeanStore achieves 25.1k tx/s with an average latency of 318µs. Next, we vary each workload dimension individually and use the architecture that results in the cheapest setup for that workload. As the results in Table 3 show, the required performance and latency can be achieved on the recommended instance in all cases except for two. With TXS=1k, *RBD* is not able to stay within the predicted latency because our model fails to capture that latency of the remote block device increases significantly under load. Also, with TXS=10M, *In-Memory* is actually missing the required throughput by 1%. However, the results generally show that our model can predict cloud OLTP performance reasonably well for the LeanStore engine across a variety of workloads. Cases where the model prediction deviates from the observed performance indicate that either the model has to be extended or that the tested system fails to exploit the given hardware, and both of these outcomes are instructive.

Table 4 Model validation with AWS Aurora. For 12 out of 18 workloads, Aurora achieves ≥ 90% of the predicted throughput.

Workload		Pred.	Measured Perf.	
Dataset	Target Perf.	Instance	READ=100%	READ=70%
10 GB	10k tx/s	db.r6g.l	10.5k tx/s	10.4k tx/s
10 GB	100k tx/s	db.r6g.xl	147k tx/s	90.7k tx/s
10 GB	1000k tx/s	db.r6g.8	1,180k tx/s	509k tx/s
100 GB	10k tx/s	db.r6g.xl	10.8k tx/s	9.85 tx/s
100 GB	100k tx/s	db.r6g.4	115k tx/s	97.8k tx/s
100 GB	1000k tx/s	db.r6g.8	1,160k tx/s	478k tx/s
1 TB	10k tx/s	db.r6g.xl	11.5k tx/s	9.98k tx/s
1 TB	100k tx/s	db.r6g.8	49.3k tx/s	46.4k tx/s
1 TB	1000k tx/s	db.r6i.32	424k tx/s	385k tx/s

4.3 Model Validation with AWS Aurora

Recalibrating the Model. To demonstrate that our model framework is not tailor-made for LeanStore, but is adaptable to production-grade cloud OLTP systems, we configure it for AWS Aurora PostgreSQL. This requires several small changes to the model because Aurora's frontend is based on a traditional disk-based system, which is not optimized for efficient CPU usage. We increase the in-memory cost per transaction to 60.000 cycles (∼24µs) and set the page size to 8 KB, a characteristic inherited from PostgreSQL. Additionally, since Aurora can only be configured on a subset of EC2 instance types, we restrict our model to use only those instances. Finally, we add support for non-clustered indexes to the model, as Aurora PostgreSQL does not have the capability to create index-only tables. For the measurements, we use a stored procedure which executes 1000 transactions with just a single client interaction, minimizing network roundtrips. Each transaction retrieves or updates a random tuple based on the specified lookup ratio. We use `pgbench` as the benchmark driver, configure it with multiple clients, and run it until the system achieves steady-state performance. The reported `tps` is the average from a 20s run.

Aurora Benchmark Results. Table 4 shows Aurora's performance for different dataset sizes and requested transaction rates. Interestingly, the model predicts the same instance type for both 100% and 70% lookup ratios in all scenarios. Aurora meets the required throughput in 8 out of the 18 experiments. In 4 experiments, the performance is within 10% of the predicted value, while in the remaining 6 cases, the actual performance ranges from 51% to 38% of the prediction. In particular, we were not able to achieve more than half a million tx/s in any of the workloads containing updates, which indicates some scalability limit in Aurora. For the workload with 100k tx/s on 1 TB data, our model predicts the primary reading pages at 1.1 GB/s over the network, but the actual rate is only 630 MB/s, despite the 9.5 Gbit/s network interface and CPU load at 43%. We suspect the reason for this is contention and overhead in the IO stack, but as Aurora is a hosted service we cannot obtain detailed performance profiles, so the exact reason remains unknown. Having an engine that we can fully study, profile, and tweak is one of the reasons why we use LeanStore for our study in Section 5.

5 Multi-Dimensional Analysis of OLTP

Choice of Model Configuration. In the previous section we have shown that our model can be configured for a flash-optimized research system like LeanStore as well as for a full commercial system like AWS Aurora. Which one of the two should be used for gaining insights across the cloud OLTP landscape? For this analysis, we choose the LeanStore configuration. It is closer to what database engines are theoretically capable of on current cloud hardware, and thus gives more generic insights than any particular production-grade system would.

5.1 Choice of Workloads

Reasonable Parameter Ranges. For the initial workloads, we assume a moderate update rate of 30%, allow transaction latencies up to 1ms, and require at least 3×9's durability. The latter corresponds to a 0.1% chance of log tail loss per year, which seems reasonable for many OLTP applications. In later experiments we vary durability between 1×9 (10% chance of data loss in a year) and 11×9's (0.000000001%). We choose 100 TB as the upper limit for the dataset size, as it is just above what a single EC2 instance can store (`i3en.24xl`), and also the maximum database size that Azure SQL Hyperscale supports [2]. We also look at workloads with up to 100M tx/s, which is pushing the limits for an OLTP database and only needed in extremely large operational systems [81]. However, our analysis shows that some system architectures conceptually allow such high throughput[1]. The vast majority of customer workloads will fall more towards the center of the parameter ranges, not the extremes.

Example Workload. One such workload could for example be: DB=1TB, OPS=100k/s, READ=70%, DUR=3x9, LAT=1ms, AVAI=Node. Our model determines these system configurations are the cheapest:

[1] At these rates, network communication with clients would likely be a bottleneck. This issue could be solved by using stored procedures or co-locating the application with the database.

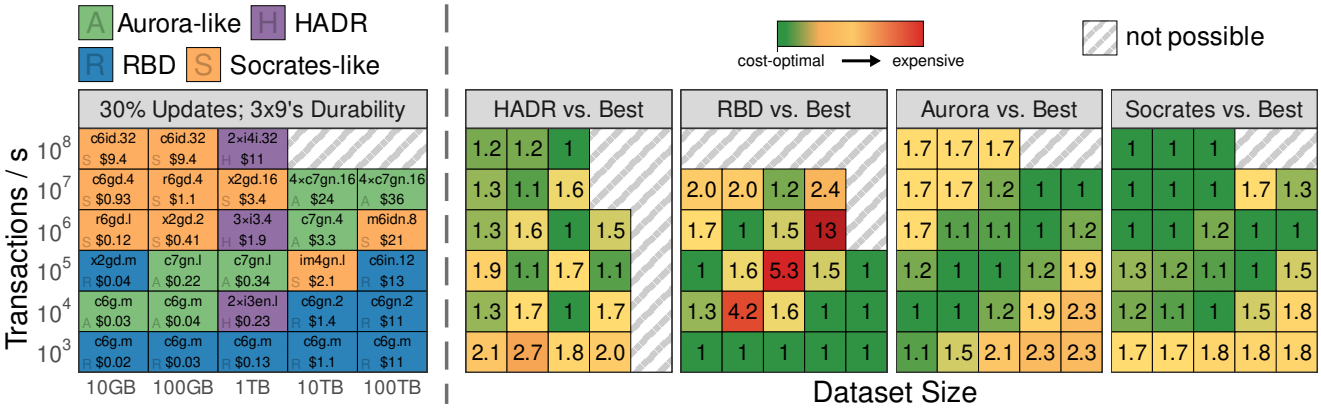

Fig. 4 Left: The most cost-efficient configuration for different dataset sizes and transaction rates. Cost is per hour. Right: The cost of each architecture compared to the cheapest architecture for that workload. READ=70%, DUR=3×9's, LAT=1ms, AVAI=Node.

Architecture	Primary	Total Hourly Cost	Factor
Classic	*cannot satisfy workload constraints*		
In-Memory	*cannot satisfy workload constraints*		
HADR	is4gen.xl	$0.58	1.7x
RBD	c6gn.12	$1.81	5.3x
Aurora-like	c7gn.l	$0.34	1.0x
Socrates-like	c6gd.2	$0.37	1.1x

The cost numbers are per hour, and cover all resources that the DBMS internally uses, i.e., instances, network, and EBS (if used). For *Classic* and *In-Memory*, there is no result because instance storage cannot guarantee the 3×9's durability that the workload requires. *Aurora-like* is the architecture with the overall cheapest configuration in this case, while the cheapest one using *RBD* costs 5.3× as much. Note that the results for *Aurora-like* and *Socrates-like* should not be confused with the corresponding commercial systems. In the next experiments, we will only show the overall cheapest architecture configuration (not the one with the highest performance) for each workload using the following notation:

Next, starting with the example workload above, we vary the dataset size and transaction rate. Here, the full potential our of model comes into effect, as it allows us to look at many different combinations of the two parameters and compare them with each other in a single plot.

5.2 Dataset Size and Transaction Rate

Overview. The plot on the left-hand side of Figure 4 shows the cost-optimal architecture and instance type combination for datasets ranging from 10 GB to 100 TB and transaction rates between 1k and 100M per second. As expected, system cost increases for larger datasets and higher transaction rates. The plots on the right of Figure 4 show for each architecture the cost ratio compared to the optimal choice for each particular workload. Note that the workload discussed in the previous subsection appears in the fourth row and third column from the top-left in each of the plots.

Low Transaction Rates. For low transaction rates, the *RBD* architecture offers the lowest cost, because storage capacity on EBS gp3 devices is inexpensive (see Table 5). However, at higher transaction rates, the remote block device must be provisioned with more (expensive) IOPS, and an instance capable of supporting these has to be selected. These requirements make this architecture less attractive. In the worst case, for a 10 TB dataset and 1m tx/s, it is 13 times more expensive than *Aurora-like* and *Socrates-like*, as it must use a very large (and expensive) primary instance to keep over 90% of the dataset in cache. No instance is available that can handle the EBS IOPS that would be required for smaller cache hit rates. Similarly, *RBD* cannot support workloads with 100M tx/s at all. Note that all configurations shown in Figure 4 use EBS gp3 devices, as they provide a strictly better price point than io2 devices, and the 3×9's durability of the workloads can be provided by the gp3 devices.

> **Observation 1:** *RBD* becomes expensive with higher transaction rates due to the cost of provisioning IOPS and instances than can handle them.

Higher Transaction Rates. For higher transaction rates, the picture is more diverse. For very high transaction rates on small datasets, the *Socrates-like* architecture is the cost-optimal choice. As the right-hand side of Figure 4 shows, *Aurora-like* is 70% more expensive

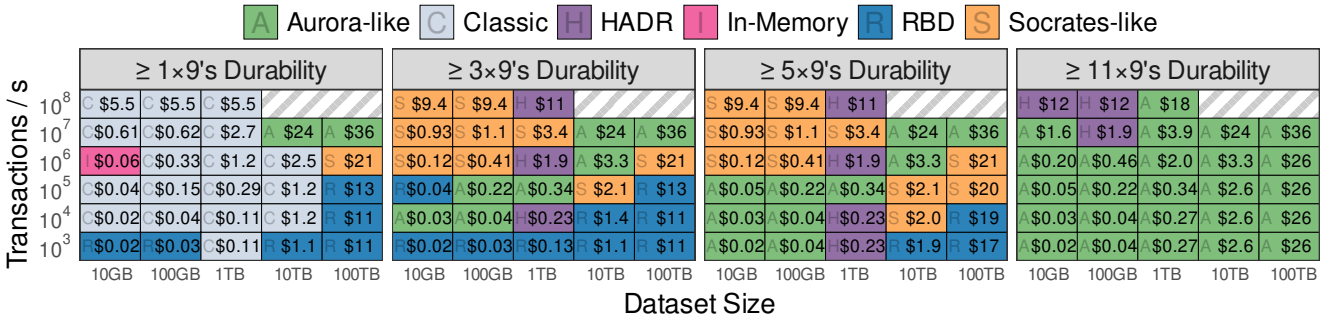

Fig. 5 The cost of having at least {1, 3, 5, 11}×9's of durability. READ=70%, LAT=1ms, AVAI=Node.

here, because it cannot scale the page and log service independently, even though a small page service would be sufficient for these workloads. *HADR* wins for three particular workloads, but there is no clear pattern when it is a good choice, indicating that the workload has to closely fit the available instance sizes for *HADR* to be cost effective. Furthermore, *HADR* is not able to handle the 100 TB dataset, as there are currently no cloud instances with that much NVMe storage available.

> **Observation 2:** The *HADR* architecture has limited and inflexible scaling behavior, which makes it cost-efficient for very few use cases.

Classic and *In-Memory* do not appear in the plot at all, as they do not meet the (only moderate) durability constraint. We reconsider them for suitable workloads in Section 5.3.

Good Default Choice. As the rightmost plot in Figure 4 shows, *Socrates-like* is a reasonable default choice, being cost-optimal or nearly cost-optimal for most workloads except those with very few transactions. Furthermore, along with *Aurora-like*, it can handle the widest range of workloads, with only two extreme workloads in the upper right not being conceptually possible.

> **Observation 3:** *Socrates-like* is the most cost-efficient architecture for a wide variety of workloads under moderate durability requirements.

For the workloads on the lower right, *Aurora-like* has higher cost because it replicates the database on three storage servers, while *Socrates-like* keeps only two copies. This effect would have been even more pronounced according to the original papers, which proposed that Aurora stores six copies and Socrates only one.

> **Observation 4:** The 3× replication of the database dominates overall system cost of *Aurora-like* for large cold datasets.

Finally, for 21 out of the 28 workloads, using a Graviton instance is the most cost effective choice, underlining the cost effectiveness of ARM instances in AWS [82]. For instance, for the workload with 100k tx/s on a 1 TB dataset, the cheapest alternative with an Intel CPU would have been the *c5n.xl*, which is 13% pricier.

> **Observation 5:** ARM support in modern cloud OLTP systems offers access to more economical instances.

5.3 Durability

As we argue in Section 2.7, in the cloud it always makes sense to store backups, including the log archive, on cheap blob storage with very high durability, e.g., S3. Thus, data loss can only occur for the tail of the WAL which is not backed up yet. Accordingly, our model calculates durability based on how the log component is implemented. In line with common practice, we specify durability as the leading nines of the probability that no data loss occurs over a period of one year. To analyze the cost of varying durability requirements, we consider the workloads from Figure 4 again, which require at least 3×9's (99.9%), and compare them against ones requiring at least 1, 5, and 11×9's. Figure 5 shows the resulting architecture and system cost for each combination. *RBD* cannot obtain 11×9's durability in AWS, as there are no EBS devices with more than 5×9's durability. As *Socrates-like* stores the WAL on an EBS device, it is subject to the same limitation. Only *HADR* and *Aurora-like* achieve 11×9's durability. In fact, due to the six copies of the log, we calculate *Aurora-like*'s durability at around 20×9's [83]. At the same time, for most workloads it is cheaper than *HADR*.

> **Observation 6:** *Aurora-like*'s log replication is suitable for workloads with extremely high durability requirements.

Although there are mission-critical OLTP uses cases where durability cannot be too high, staging and test systems are also common. For these systems, data loss is not an issue and they are regularly reset anyway. As

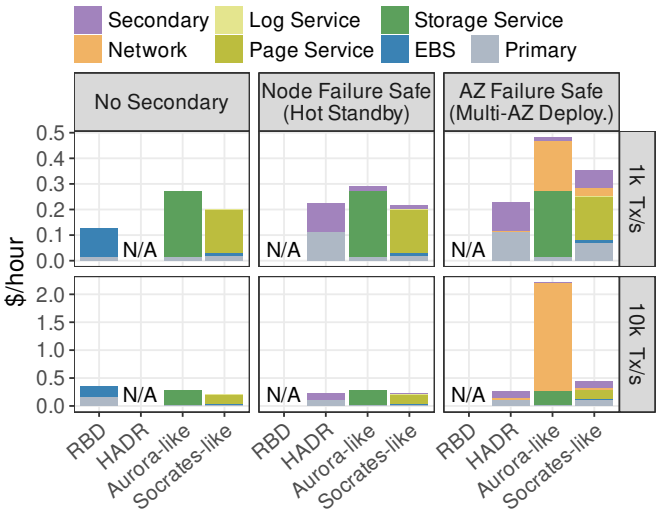

Fig. 6 The cost of guarding against different failure cases. DB=1 TB, READ=70%, DUR=3×9's, LAT=1ms.

the left plot in Figure 5 shows, our model automatically exploits this property, and now most often selects *Classic*, as it proves to be the most cost-effective architecture choice. *In-Memory* only wins for one combination of parameters we examined, which highlights that while this architecture is fast, it is not very economical.

> **Observation 7:** For workloads with low durability requirements, the *Classic* architecture is often still the most cost effective choice in the cloud.

Comparing the leftmost and rightmost plots in Figure 5, one can see that the cost savings between requiring a durability of at least 1×9 and 11×9's are up to 3.5× (1M tx/s on a 10 GB dataset). For many dataset sizes and throughput rates, in particular the common ones on the diagonal, the cost difference is surprisingly low, often below 60%. However, when looking at the lower right corners of all four plots, which represent workloads on large but mostly cold datasets, *Aurora-like* is more than twice as expensive as the alternatives. This comes from the threefold replication of the entire dataset, which we already highlighted in Observation 4. That is unfortunate, as only the replication of the (small) log tail would be required to achieve high durability. We discuss this aspect further in Section 6.

5.4 Availability

For mission-critical operational systems, availability can be as important as durability. In the cloud, failure cases include single instance failures up to whole data center outages. We analyze the cost composition for a basic system configuration with no secondaries and workloads with 1k and 10k tx/s. The two plots in the left of Figure 6 show *Socrates-like* is the cheapest, as its page service costs less than either the EBS device for *RBD* or *Aurora-like*'s storage servers. *HADR* is not shown here as it always has a secondary node. The two plots in the middle of Figure 6 show the additional cost of adding a standby node which takes over in case the primary fails. For *HADR*, the total cost is twice that of the primary. *Aurora-like* and *Socrates-like*, in contrast, can execute the workload with a smaller (cheaper) primary, as storage is separated. Thus, adding a secondary node does not double the system cost for these architectures. So far, all resources (primary, secondaries, shared services) were placed into a single data center (availability zone or AZ). For a system to become robust against whole data center outages, which are not that uncommon [61, 62], cloud vendors recommend placing instances across AZs. This has an impact on cost, as inter-AZ traffic is priced expensively in public clouds, e.g., $20 / TB in AWS. The two plots on the right of Figure 6 show that placing the secondary in another AZ has minimal cost impact on *HADR*, as only log records are shipped over the network. But *Aurora-like* and *Socrates-like* get significantly more expensive, as log records are shipped multiple times, and some page accesses go to other AZs. In particular for the workload with 10k tx/s, *Aurora-like* costs increase by 13× compared to the single AZ deployment, just because it reads ∼17 MB/s of database pages from storage servers in remote AZs. Note, however, that as mentioned in Section 3.3, Azure recently decided to stop charging for inter-AZ traffic, so other cloud providers might follow, which would invalidate this observation.

> **Observation 8:** Inter-AZ traffic cost can be significant in distributed OLTP workloads.

For *Socrates-like*, failures of the page servers are also relevant. Imagine a page server managing 10 TB of data becoming unavailable once a month. Even if only 1 TB of that data is accessed frequently, (bulk) recovery from S3 takes at least 100 s, during which data accesses must wait. The problem worsens with larger databases or more granular partitioning. With 100 page servers and assuming 99.5% instance availability, recovery would run for some part of the data almost 40% of the time. Microsoft apparently realized this issue, as their commercial version caches each page on two different page servers [24], which we do in our model as well.

> **Observation 9:** *Socrates-like* page server replication is important not for durability, but for availability of the system.

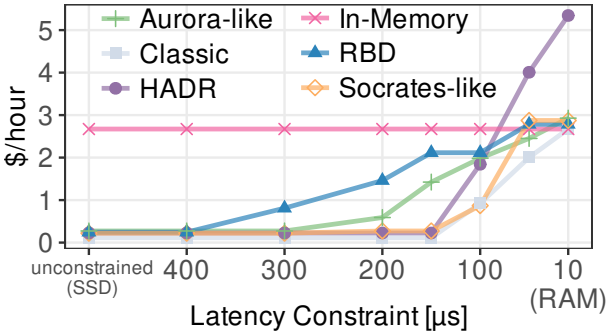

Fig. 7 The effect of latency constraints on workload cost. DB=1 TB, TXS=10k/s, READ=100%, DUR=1×9, AVAI=Node.

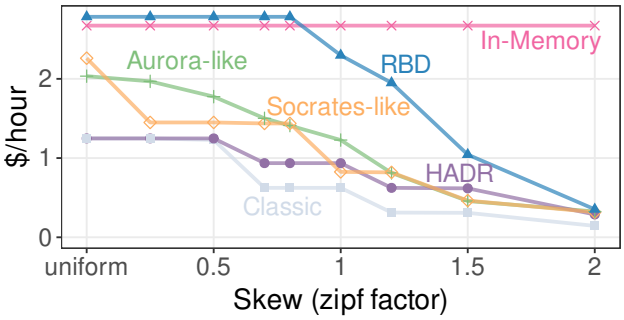

Fig. 8 The effect of access skew on workload cost. DB=1 TB, TXS=1M/s, READ=100%, DUR=1×9, AVAI=Node.

5.5 High Update Ratios

Intuitively, an update transaction involves more work than a lookup. Thus, a workload which is more update-heavy should be more expensive to execute. To examine the impact, we compare different update/lookup ratios against the 30% update rate that we used in our previous experiments. The largest cost difference is for high transaction rates on smaller datasets:

100% Lookups					10% Updates					30% Updates					100% Updates				
0.3	0.3	0.3			0.6	0.6	0.6			1	1	1			6.2	6.2	6.0		
0.3	0.4	0.8	0.9	0.9	0.6	0.6	0.9	0.9	0.9	1	1	1	1	1	2.7	2.2	1.4	1.2	1.1
0.4	0.8	0.7	1	1	0.7	0.9	1	1	1	1	1	1	1	1	2.4	1.3	1.2	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1.5	1.1	1	1	1.1
0.8	0.9	1	0.9	1	0.9	1	1	0.9	1	1	1	1	1	1	1	1	1	1.1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

This has two reasons: 1) For low transaction rates, storage dominates workload cost (lower half) and 2) for a larger database, the scaled-up storage can sustain the required writes for the workload anyway (right area).

> **Observation 10:** Update are more costly than lookups mostly for workloads with high transaction rates on small datasets.

5.6 Operation Latency

Our analysis focused on throughput so far. While important for many OLTP applications, latency can also be crucial for the end-user experience. Figure 7 shows how the system cost is affected when different latency constraints are applied on a workload consisting of 10K lookups per second on a 1 TB dataset[2]. Without a latency constraint, most accesses are served from flash-based storage, i.e., NVMe instance SSDs, EBS, or NVMe-based page servers. The one exception is *In-Memory*, which always has the same cost as data is always in DRAM. For the other architectures, the cost varies between \$0.11/h (*Classic*) and \$0.27/h (*Aurora-like*). In the range between 400µs and 50µs, an increasing portion of the data has to be kept in DRAM to satisfy the latency constraint. To reach 10µs latency, all systems effectively have to keep the entire dataset in DRAM using EC2's `x2` instances. As DRAM dominates cost here, all systems converge to a similar cost between \$2.7/h (*Classic*) and \$2.9/h (*Aurora-like*). The only exception is *HADR*, which needs an additional stand-by node, doubling the cost.

> **Observation 11:** Sub-millisecond latency can be achieved in the cloud at low cost using flash storage. Reducing the latency below 100 µs requires caching the data in DRAM, which increases cost by an order of magnitude.

Another way to look at this is that for applications without very strict latency constraints, caching the entire data in DRAM is unreasonably expensive given current cloud prices.

5.7 Data Access Skew

Uniform accesses over the entire dataset exhibit no locality which the DBMS could exploit. We assumed this access pattern in our experiments so far, as it constitutes the most challenging, and thereby the most generic case. However, in reality, most workloads have some degree of locality. Thus, in this section we look at the effect skew has on the cost of different architectures. We focus on lookups and use a Zipf distribution to model that some parts of the dataset are accessed more often than others. Also, we assume that the systems implement a replacement strategy that can successfully identify the hot part of the data. Figure 8 shows for a workload of 1M lookups/s on a 1 TB dataset that, in general, costs decrease for higher Zipf factors. *RBD* can benefit the most from higher skew, as the EBS device can be provisioned with much fewer IOPS,

[2] Note that we do not account for the round-trip latency to the client, which – similar to the packet rate limitation – could be removed by co-locating the application with the database server.

Table 5 Cloud storage options in AWS, their cost, durability, and latency; ordered by storage cost. Read/write granularity is 4 KB. Durability is taken from documentation or estimated where not available. EC2 instances use 50% of the on-demand price, see Section 4.1.

Storage Primitives	$/month per TB↓	$/month for 1K reads/s	10K reads/s	100K reads/s	10K writes/s	Durability	Latency
S3	**$23**	$1,037	$10,368	$103,680	$129,600	**11×9's**	>10ms
Instance NVMe (i3en) (*)	$66	**$2**	$20	$198	$25	∼1×9	<1ms
EBS gp3	$80	**$0**	$35	$395 (**)	$35	2×9's	<1ms
Instance NVMe (i3) (*)	$120	**$0.6**	**$5.5**	$55	$13	∼1×9	<1ms
EBS io2	$125	$65	$650	$4,704	$650	5×9's	<1ms
S3 Express One Zone	$160	$518	$5,184	$51,840	$64,800	∼5×9's (‡)	∼1ms
Instance DRAM (x2iedn) (*)	$2,434	**$0.2**	**$2**	**$19**	**$2**	N/A	**<1µs**
Services							
Aurora Standard [84]	$100	$518	$5,184	$51,840	$5,184	**>11×9's** (†)	<1ms
Aurora I/O-Optimized [84]	$225	**$0**	**$0**	**$0**	**$0**	**>11×9's** (†)	<1ms
DynamoDB (provisioned) [85]	$244	$94	$936	$9,360	$18,720	N/A	∼1ms
DynamoDB (on-demand) [85]	$244	$648	$6,480	$64,800	$129,600	N/A	∼1ms

(*) Total EC2 cost is $max(storageCost, accessCost)$, e.g., on an *i3* instance 4 TB and 500k reads/s cost $max(\$480, \$220) = \$480$
(**) Requires 7 EBS volumes and at least 194GiB storage to provision the required IOPS
(†) At least the durability of S3 standard due to its sixfold log replication into three AZs
(‡) Similar durability as EBS *io2* due to intra-datacenter replication

which reduces cost from $2.8 for uniform accesses down to $0.4 for a high Zipf factor of 2.0. Looking at this the other way around, it once more validates Observation 1, which shows that IOPS are what makes EBS devices, and thus *RBD*, expensive. The *In-Memory* architecture, in contrast, cannot benefit from access skew at all, as the entire dataset (including the cold part) has to fit into DRAM. All other architectures exploit higher skew by scaling down the primary and employing a smaller buffer cache. *Aurora-like* and *Socrates-like* can also scale down their storage service.

> **Observation 12:** In all architectures except *In-Memory*, exploiting access locality can drastically reduce system cost.

6 Discussion: Implications for Cloud OLTP

Summary. Our analysis confirms the intuition that cloud-native engines generally outperform traditional designs deployed in a cloud environment. They handle a wider variety of workloads while providing high availability and durability. In particular, the Aurora architecture achieves extremely high durability with six copies of the log, while Socrates is cost-efficient by caching data on NVMe. Moreover, for many (though not all) workloads, these two cloud-native architectures are more cost-efficient than lift-and-shift architectures (*Classic*, *In-Memory*, or *HADR*). For workloads with low transaction throughput up to 1k tx/s and a durability requirement of up to 3×9's, *RBD* is a viable alternative.

Storage Options for Cloud OLTP. One of the main reasons for the cost-effectiveness of the cloud-native architectures is their choice of storage technology, as storage is a significant cost factor in OLTP workloads. To better understand the results from Section 5, we examine storage options in the cloud. Table 5 shows different storage primitives available in AWS with their cost, durability, and latency properties. As one can see, S3 has the lowest storage cost and high durability, but its high access latency and access cost are prohibitive for primary OLTP storage. However, its properties make it the ideal archival storage for both the log and the database backup. At the other end of the spectrum, main memory offers the best performance and lowest latency, but capacity costs at least one order of magnitude more than for the other options, rendering it uneconomical in many situations.

NVMe Instances as a Building Block. Table 5 also shows that NVMe instance storage, such as on an *i3en* instance, has low access cost and latency at a reasonable storage cost, making it highly suitable for OLTP storage. However, storing only one copy of the data on NVMe does not provide sufficient durability for many workloads. To address this issue, using NVMe instances as a building block, Aurora stores three redundant copies of the database. In contrast, Socrates stores one copy of the database on object storage for durability and maintains two copies on NVMe for cost-effective access. Note that the second copy on NVMe is required for availability in this case, not durability (see Obs. 9).

NVMe vs. EBS. An alternative to NVMe that looks appealing from a cost perspective is EBS. However, EBS *gp3* only guarantees low durability. While EBS *io2* offers better durability, IO is an order of magnitude more expensive. Besides, a dedicated storage

service provides even better durability, can implement some DBMS functions itself as it includes compute and memory, can be deployed across multiple availability zones, and NVMe flash itself provides almost 3× lower access latency than EBS (see Table 1). For all these reasons, we believe a storage service based on NVMe instances is the most promising approach for storing OLTP databases in the cloud.

Higher-Level Services. For comparison, the lower part of Table 5 shows a selection of available OLTP services in AWS. Their price structure aligns with the storage primitives with some exceptions, such as the on-demand variant of DynamoDB which is relatively expensive. Interestingly, the pricing structure would also allow building services on top of EC2 that are competitive with existing in-house AWS services.

Further Benefits of a Storage Service. A large-scale storage service provides further benefits that we do not even capture in our model: The separation of storage and compute leads to better elasticity, and operating it as a multi-tenant service allows fine-grained and almost unlimited allocation of storage to individual databases while saving costs. Users can profit from consumption-based billing, where they do not have to reserve capacity upfront and only pay for what they actually use. Finally, distributing a single database over many storage nodes enables absorbing load spikes, transparent migration off faulty machines, and shorter recovery intervals. Further research should investigate how exactly such a page service should be implemented.

Towards Workload-Adaptive Systems. Another observation from the analysis in Section 5 is the cost-saving potential of exploiting workloads with relaxed requirements, such as latency, durability, or availability. For example, a system that does not need require high availability does not need to store database pages redundantly to guard against failures. Similarly, for most testing and staging systems, a single-node *Classic* architecture would be sufficient. Thus, one could imagine a DBMS service which allows users to specify their actual workload requirements and proposes an adequate hardware and storage engine configuration for them, for example, by storing a variable number of copies of each page depending on the required durability. This would enable users to make informed decisions and allow them to choose, e.g., if they are willing to pay extra for higher durability or lower transaction latency. Additionally, such an adaptive system would enable substantial cost savings compared to the current situation where users can typically only choose between instances with different numbers of CPU cores and memory capacity, but the architecture configuration is fixed.

Potential Architecture Improvements. Our main goal in this work is to compare existing OLTP designs for the cloud. Therefore, we deliberately modeled Aurora and Socrates as closely as possible to how they are described in literature. However, our analysis indicates that there are potential improvements to both of them. While the Socrates architecture nicely divides responsibility for the database and the log into separate services, choosing a remote block device for persisting log records limits its durability (see Section 2.6). Aurora's approach of streaming the log to multiple nodes seems to be the better alternative, as it provides very high durability and performance. Unfortunately, Aurora also ties the log and the database storage together in one service, which makes independent development and scaling more complicated. An architecture that combines the two into a new design would provide the best of both worlds: 1) Cache the database pages on cheap NVMe storage backed by an object storage service and 2) send the WAL redundantly to multiple logging nodes that form a separate service. In future work, we plan to evaluate such an alternative design by incorporating it into our model and validating it with the same methodology we have applied in this paper, comparing it against existing designs. However, even in that new architecture each page would still be cached twice for availability reasons. As we have found that storage capacity dominates overall system cost for larger datasets, the reduction of this space amplification is yet another promising avenue for further research.

7 Future Work

Model Extensions. We believe cost optimization is a sensible design principle in a cloud-dominated world. In the future, our model framework could be extended with additional architectures and include other index and data structures. One could even integrate it with Cosine [8] to find workload-optimal data structures. Furthermore, the accuracy and predictive power of the model could be improved by accounting for complex latency distributions, modeling more complex transactions, and incorporating network variability.

Other Cloud Providers and Pricing Models. A study that compares cost across different cloud vendors would be interesting, as subtle differences in pricing structures could lead to vastly different total costs for certain workloads, such as those dominated by cross-data-center traffic cost. Even within a single cloud provider, one could model various pricing strategies (e.g., reserved, on-demand, and spot instances) to explore their potential for cost savings.

References

1. A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao, "Amazon aurora: Design considerations for high throughput cloud-native relational databases," in *SIGMOD*, 2017, pp. 1041–1052.
2. 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*, 2019, pp. 1743–1756.
3. X. Pang and J. Wang, "Understanding the performance implications of the design principles in storage-disaggregated databases," in *SIGMOD*, 2024.
4. J. Tan, T. M. Ghanem, M. Perron, X. Yu, M. Stonebraker, D. J. DeWitt, M. Serafini, A. Aboulnaga, and T. Kraska, "Choosing A cloud DBMS: architectures and tradeoffs," *PVLDB*, vol. 12, no. 12, pp. 2170–2182, 2019.
5. C. Chen and B. He, "A framework for analyzing monetary cost of database systems in the cloud," in *WAIM*, ser. Lecture Notes in Computer Science, vol. 7923, 2013, pp. 118–129.
6. A. van Renen and V. Leis, "Cloud analytics benchmark," *PVLDB*, vol. 16, no. 6, pp. 1413–1425, 2023.
7. V. Leis, M. Haubenschild, A. Kemper, and T. Neumann, "Leanstore: In-memory data management beyond main memory," in *ICDE*, 2018, pp. 185–196.
8. S. Chatterjee, M. Jagadeesan, W. Qin, and S. Idreos, "Cosine: A cloud-cost optimized self-designing key-value storage engine," *PVLDB*, vol. 15, no. 1, pp. 112–126, 2021.
9. G. Haas and V. Leis, "What modern nvme storage can do, and how to exploit it: High-performance I/O for high-performance storage engines," *Proc. VLDB Endow.*, vol. 16, no. 9, pp. 2090–2102, 2023.
10. T. Ziegler, P. A. Bernstein, V. Leis, and C. Binnig, "Is scalable oltp in the cloud a solved problem?" in *CIDR*, 2023.
11. Google, "AlloyDB for postgreSQL," https://cloud.google.com/alloydb, 2023.
12. H. Linnakangas, "Architecture decisions in neon," https://neon.tech/blog/architecture-decisions-in-neon, 2022.
13. Neon, "Manage computes," https://neon.tech/docs/manage/endpoints, 2022.
14. Amazon, "Instance store volume I/O performance," https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/storage-optimized-instances.html#storage-instances-diskperf, 2022.
15. IBM, "High availability disaster recovery (HADR)," https://www.ibm.com/docs/en/db2/11.5?topic=server-high-availability-disaster-recovery-hadr, 2022.
16. Microsoft, "About log shipping (SQL server)," https://learn.microsoft.com/en-us/sql/database-engine/log-shipping/about-log-shipping-sql-server?view=sql-server-ver16, 2022.
17. ——, "Business continuity and HADR for SQL server on azure virtual machines," https://learn.microsoft.com/en-us/azure/azure-sql/virtual-machines/windows/business-continuity-high-availability-disaster-recovery-hadr-overview?view=azuresql, 2022.
18. J. Gray, P. Helland, P. E. O'Neil, and D. E. Shasha, "The dangers of replication and a solution," in *SIGMOD*, 1996, pp. 173–182.
19. J. Hamilton, "Aws Nitro system," https://perspectives.mvdirona.com/2019/02/aws-nitro-system/, 2019.
20. Amazon, "EBS volume types," https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-volume-types.html, 2022.
21. Microsoft, "Redundancy options for managed disks," https://learn.microsoft.com/en-us/azure/virtual-machines/disks-redundancy, 2022.
22. Amazon, "EC2 high memory update – new 18 tb and 24 tb instances," https://aws.amazon.com/blogs/aws/ec2-high-memory-update-new-18-tb-and-24-tb-instances/, 2019.
23. A. Verbitski, A. Gupta, D. Saha, J. Corey, K. Gupta, M. Brahmadesam, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao, "Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes," in *SIGMOD*, 2018, pp. 789–796.
24. Microsoft, "Hyperscale distributed functions architecture - page server," https://learn.microsoft.com/en-us/azure/azure-sql/database/hyperscale-architecture?view=azuresql-db#page-server, 2023.
25. Google, "How cloud storage delivers 11 nines of durability—and how you can help," https://cloud.google.com/blog/products/storage-data-transfer/understanding-cloud-storage-11-9s-durability-target, 2021.
26. D. Kaulakiene, C. Thomsen, T. B. Pedersen, U. Çetintemel, and T. Kraska, "Spotadapt: Spot-aware (re-)deployment of analytical processing tasks on amazon EC2," in *DOLAP*, 2015, pp. 59–68.
27. 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*, 2016, pp. 215–226.
28. A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan, "Amazon redshift and the case for simpler data warehouses," in *SIGMOD*, 2015, pp. 1917–1923.
29. T. Kraska, E. Dadashov, and C. Binnig, "Spotlytics: How to use cloud market places for analytics?" in *BTW*, ser. LNI, vol. P-265, 2017, pp. 361–380.
30. A. Mahgoub, A. Medoff, R. Kumar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi, "OPTIMUSCLOUD: heterogeneous configuration optimization for distributed databases in the cloud," in *USENIX ATC*, 2020, pp. 189–203.
31. O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *NSDI*, 2017, pp. 469–482.
32. S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *NSDI*, 2016, pp. 363–378.
33. R. Marcus and O. Papaemmanouil, "Releasing cloud databases from the chains of performance prediction models," in *CIDR*, 2017.
34. C. Zhan, M. Su, C. Wei, X. Peng, L. Lin, S. Wang, Z. Chen, F. Li, Y. Pan, F. Zheng, and C. Chai, "Analyticdb: Real-time OLAP database system at alibaba cloud," *PVLDB*, vol. 12, no. 12, pp. 2059–2070, 2019.
35. Y. Yang, M. Youill, M. E. Woicik, Y. Liu, X. Yu, M. Serafini, A. Aboulnaga, and M. Stonebraker, "Flexpushdowndb: Hybrid pushdown and caching in a cloud DBMS," *PVLDB*, vol. 14, no. 11, pp. 2101–2113, 2021.
36. C. Winter, J. Giceva, T. Neumann, and A. Kemper, "On-demand state separation for cloud data warehousing," *PVLDB*, vol. 15, no. 11, pp. 2966–2979, 2022.
37. R. Marroquín, I. Müller, D. Makreshanski, and G. Alonso, "Pay one, get hundreds for free: Reducing cloud costs through shared query execution," *CoRR*, vol. abs/1809.00159, 2018.
38. M. Perron, R. C. Fernandez, D. J. DeWitt, and S. Madden, "Starling: A scalable query engine on cloud functions," in *SIGMOD*, 2020, pp. 131–141.

39. I. Müller, R. Marroquín, and G. Alonso, "Lambada: Interactive data analytics on cold data using serverless cloud infrastructure," in *SIGMOD*, 2020, pp. 115–130.
40. A. Floratou, J. M. Patel, W. Lang, and A. Halverson, "When free is not really free: What does it cost to run a database workload in the cloud?" in *TPCTC*, ser. Lecture Notes in Computer Science, vol. 7144, 2011, pp. 163–179.
41. C. Curino, D. E. Difallah, A. Pavlo, and P. Cudré-Mauroux, "Benchmarking oltp/web databases in the cloud: the oltpbench framework," in *CloudDB@CIKM*, 2012, pp. 17–20.
42. B. Mozafari, C. Curino, A. Jindal, and S. Madden, "Performance and resource modeling in highly-concurrent OLTP workloads," in *SIGMOD Conference*, 2013, pp. 301–312.
43. Y. Cao, C. Chen, F. Guo, D. Jiang, Y. Lin, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu, "Es^2: A cloud data storage system for supporting both OLTP and OLAP," in *ICDE*, 2011, pp. 291–302.
44. X. Qiu, M. Hedwig, and D. Neumann, "SLA based dynamic provisioning of cloud resource in OLTP systems," in *WEB*, ser. Lecture Notes in Business Information Processing, vol. 108, 2011, pp. 302–310.
45. S. Das, D. Agrawal, and A. E. Abbadi, "Elastras: An elastic, scalable, and self-managing transactional database for the cloud," *ACM Trans. Database Syst.*, vol. 38, no. 1, p. 5, 2013.
46. G. Li, H. Dong, and C. Zhang, "Cloud databases: New techniques, challenges, and opportunities," *PVLDB*, vol. 15, no. 12, pp. 3758–3761, 2022.
47. A. Depoutovitch, C. Chen, J. Chen, P. Larson, S. Lin, J. Ng, W. Cui, Q. Liu, W. Huang, Y. Xiao, and Y. He, "Taurus database: How to be fast, available, and frugal in the cloud," in *SIGMOD*, 2020, pp. 1463–1478.
48. W. Cao, Y. Zhang, X. Yang, F. Li, S. Wang, Q. Hu, X. Cheng, Z. Chen, Z. Liu, J. Fang, B. Wang, Y. Wang, H. Sun, Z. Yang, Z. Cheng, S. Chen, J. Wu, W. Hu, J. Zhao, Y. Gao, S. Cai, Y. Zhang, and J. Tong, "Polardb serverless: A cloud native database for disaggregated data centers," in *SIGMOD*, 2021, pp. 2477–2489.
49. S. Chatterjee, M. Jagadeesan, W. Qin, and S. Idreos, "Cosine technical report pages," http://daslab.seas.harvard.edu/cosine/appendix.pdf, 2019.
50. S. Idreos, K. Zoumpatianos, B. Hentschel, M. S. Kester, and D. Guo, "The data calculator: Data structure design and cost synthesis from first principles and learned cost models," in *SIGMOD*, 2018, pp. 535–550.
51. T. Kraska, "Towards instance-optimized data systems," *PVLDB*, vol. 14, no. 12, pp. 3222–3232, 2021.
52. V. Leis and M. Kuschewski, "Towards cost-optimal query processing in the cloud," *PVLDB*, vol. 14, no. 9, pp. 1606–1612, 2021.
53. H. Zhang, Y. Liu, and J. Yan, "Cost-intelligent data analytics in the cloud," in *CIDR*, 2024.
54. Z. Wang, E. Adamiak, and A. Aiken, "A model for query execution over heterogeneous instances," in *CIDR*, 2024.
55. P. A. Boncz, Y. Chronis, J. Finis, S. Halfpap, V. Leis, T. Neumann, A. Nica, C. Sauer, K. Stolze, and M. Zukowski, "SPA: economical and workload-driven indexing for data analytics in the cloud," in *ICDE*, 2023, pp. 3740–3746.
56. H. Park, G. R. Ganger, and G. Amvrosiadis, "Mimir: Finding cost-efficient storage configurations in the public cloud," in *SYSTOR*, 2023, pp. 22–34.
57. R. Lasch, T. Legler, N. May, B. Scheirle, and K. Sattler, "Cost modelling for optimal data placement in heterogeneous main memory," *Proc. VLDB Endow.*, vol. 15, no. 11, pp. 2867–2880, 2022.
58. M. Haubenschild, "Source code for cloud OLTP model," https://github.com/haubenmi/navigating-cloud-oltp, 2024.
59. Google, "Available ... or not? that is the question—CRE life lessons," https://cloud.google.com/blog/products/gcp/available-or-not-that-is-the-question-cre-life-lessons, 2017.
60. G. Wang, L. Zhang, and W. Xu, "What can we learn from four years of data center hardware failures?" in *DSN*, 2017, pp. 25–36.
61. W. Gawroński, "The complete history of AWS outages," https://awsmaniac.com/aws-outages/, 2022.
62. Amazon, "AWS post-event summaries," https://aws.amazon.com/premiumsupport/technology/pes/, 2022.
63. Microsoft, "SLA for virtual machines," https://azure.microsoft.com/en-us/support/legal/sla/virtual-machines/v1_9/, 2022.
64. Amazon, "Amazon compute service level agreement," https://aws.amazon.com/compute/sla/, 2022.
65. ——, "Amazon EC2 on-demand pricing," https://aws.amazon.com/ec2/pricing/on-demand/, 2024.
66. Google, "VM-VM egress pricing within Google Cloud," https://cloud.google.com/vpc/pricing#egress-within-gcp, 2024.
67. Azure, "Update on inter-availability zone data transfer pricing," https://azure.microsoft.com/en-us/updates/update-on-interavailability-zone-data-transfer-pricing/, 2024.
68. Amazon, "S3 FAQs," https://aws.amazon.com/s3/faqs/, 2022.
69. K. V. Vishwanath and N. Nagappan, "Characterizing cloud computing hardware reliability," in *SoCC*, 2010, pp. 193–204.
70. I. Narayanan, D. Wang, M. Jeon, B. Sharma, L. Caulfield, A. Sivasubramaniam, B. Cutler, J. Liu, B. M. Khessib, and K. Vaid, "SSD failures in datacenters: What? when? and why?" in *SYSTOR*, 2016, pp. 7:1–7:11.
71. Amazon, "Nitro system volume limits," https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/volume_limits.html#instance-type-volume-limits, 2022.
72. ——, "EBS–optimized instances," https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-optimized.html#ebs-optimization-support, 2022.
73. ——, "Provisioned IOPS SSD volumes," https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/provisioned-iops.html, 2022.
74. Vantage, "Easy Amazon EC2 instance comparison," https://instances.vantage.sh/, 2024.
75. Stressgrid, "Packets-per-second limits in EC2," https://stressgrid.com/blog/pps_limits_in_ec2/, 2019.
76. R. Imaoka, "Using ping to test AWS VPC network latency within a single region," https://blog-en.richardimaoka.net/network-latency-analysis-with-ping-aws, 2019.
77. Amazon, "Architecture guidelines and decisions - networking - latency across availability zones," https://docs.aws.amazon.com/sap/latest/general/arch-guide-architecture-guidelines-and-decisions.html#arch-guide-networking, 2022.
78. S. Overflow, "AWS latency between zones within a same region," https://stackoverflow.com/questions/54190445/aws-latency-between-zones-within-a-same-region, 2019.
79. Amazon, "Using ENA express to improve workload performance on AWS," https://aws.amazon.com/blogs/networking-and-content-delivery/using-ena-express-to-improve-workload-performance-on-aws/, 2023.
80. V. Leis, A. Alhomssi, and G. Haas, "LeanStore source code," https://github.com/leanstore/leanstore, 2024.
81. M. Elhemali, N. Gallagher, N. Gordon, J. Idziorek, R. Krog, C. Lazier, E. Mo, A. Mritunjai, S. Perianayagam, T. Rath, S. Sivasubramanian, J. C. S. III, S. Sosothikul, D. Terry, and A. Vig, "Amazon dynamodb: A scalable, predictably performant, and fully managed nosql database service," in *USENIX ATC*. USENIX Association, 2022, pp. 1037–1048.
82. D. Loghin, "Are arm cloud servers ready for database workloads? an experimental study," *IEEE Transactions on Cloud Computing*, no. 01, 2024.

83. Backblaze, "Backblaze durability calculates at 99.999999999% — and why it doesn't matter," https://www.backblaze.com/blog/cloud-storage-durability/, 2018.
84. Amazon, "Amazon Aurora pricing," https://aws.amazon.com/rds/aurora/pricing/, 2022.
85. ——, "Amazon DynamoDB - pricing for provisioned capacity," https://aws.amazon.com/dynamodb/pricing/provisioned/, 2022.

A Appendix

A.1 General Model Parameters

Input: workload parameter W, cloud compute node N

// Cache and in-memory performance

$cpuCyclesPerTx = 4000$

$dataInCache = min(N_{memorySize}, W_{dbSize})$

$p(cacheHit) = dataInCache / W_{dbSize}$

$p(cacheMiss) = 1.0 - p(cacheHit)$

$pageSize = 4096\,\text{byte}$

// Log size

$recordHeaderSize = 48\,\text{byte}$

$ariesRecordSize = 2 * W_{tupleSize} + recordHeaderSize$

$ariesLogStorage = ariesRecordSize * W_{reqUpdates} * W_{logRetention}$

$redoRecordSize = W_{tupleSize} + recordHeaderSize$

$redoLogStorage = redoRecordSize * W_{reqUpdates} * W_{logRetention}$

// Latencies

$memoryOpLatency = cpuCyclesPerTx / N_{cpuFrequency} \approx 2\,\mu s$

$nvmeReadLatency = 132\,\mu s$

$ec2Latency = 90\,\mu s$

// Durability

$AFR_{SSD} = 0.995$

A.2 Model for HADR Architecture

$computeHADRProperties(W, N, numSecondaries)$

Input: workload parameter W,
cloud compute node N,
number of secondaries $numSec$

Output: system properties S for $HADR$

// Return "incompatible" ($\perp$) if instance is too small

if $(W_{dbSize} + ariesLogStorage > N_{storageCapacity})$ **return** $\perp$

// Updates like in Classic, but also send WAL to secondaries

$readsPerUpdate = p(cacheMiss)$

$maxIOSize = 4096\,\text{byte}$

$$logWritesPerUpdate = \begin{cases} \frac{ariesRecordSize}{maxIOSize} & \textbf{if } W_{groupCommit} \\ \lceil \frac{ariesRecordSize}{maxIOSize} \rceil & \textbf{otherwise} \end{cases}$$

$writesPerUpdate = p(dirtyPageEvict) + logWritesPerUpdate$

$networkOutPerUpdate = W_{logRecordSize} * numSecondaries$

// Limit updates by the most restrictive resource

$$S_{updates} = min \left\{ \begin{array}{l} \frac{N_{readOps}}{readsPerUpdate}, \frac{N_{writeOps}}{writesPerUpdate}, W_{reqUpdates}, \\ \frac{N_{cpuCycles}}{cpuCyclesPerTx}, \frac{N_{networkOutLimit}}{networkOutPerUpdate} \end{array} \right\}$$

// Calculate the resource requirements for lookups

$readsPerLookup = p(cacheMiss)$

$writesPerLookup = p(dirtyPageEvict)$

// Calculate the resources left for lookups

$readOpsLeft = N_{readOps} - (S_{updates} * readsPerUpdate)$

$writeOpsLeft = N_{writeOps} - (S_{updates} * writesPerUpdate)$

$cpuLeft = N_{cpuCycles} - S_{updates} * cpuCyclesPerTx$

// Calculate the lookups the primary can do

$$lookupsPrimary = min \left\{ \begin{array}{l} \frac{readOpsLeft}{readsPerLookup}, \frac{writeOpsLeft}{writesPerLookup}, \\ \frac{cpuLeft}{cpuCyclesPerTx}, W_{reqLookups} \end{array} \right\}$$

// Assume each secondary can do just as many lookups as the
// primary, as they need capacity to replay updates as well.
// Exclude one secondary from processing lookups for failover.

$S_{lookups} = min(lookupsPrimary * numSecondaries, W_{reqLookups})$

// The average latency of transactions

$$S_{opLatency} = p(cacheHit) * memoryOpLatency + p(cacheMiss) * nvmeReadLatency$$

// MTTR = mean time to repair, i.e., how long does it take
// to replicate the data to a new instance.

$$t_{MTTR} = \frac{W_{dbSize}}{50\,\text{MB/s}}$$

// λ is the expected number of failures in one MTTR

$numNodes = numSecondaries + 1$

$monthlyFailureRate = 1.0 - N_{monthlyAvailability}$

$$\lambda = \frac{numNodes * monthlyFailureRate * t_{MTTR}}{30 * 24 * 3600s}$$

// Durable in one MTTR interval if ≥ 1 instance survives.
// Calculate probability that 0..($numNodes$-1) fail using
// Poisson distributions.

$$Durability_{MTTR} = \sum_{i=0}^{numNodes-1} \frac{\lambda^i * e^{-\lambda}}{i!}$$

// Durable for an entire year if durable in all MTTR intervals

$S_{durability} = Durability_{MTTR}^{(365*24*3600s)/t_{MTTR}}$

return $(S_{updates}, S_{lookups}, S_{durability}, S_{opLatency})$

A.3 Model for Remote Block Device Architecture

$computeRBDProperties(W, N, R_{type})$

Input: workload parameter W,
cloud compute node N,
device type R_{type}

Output: system properties S for RBD

// Provision the block device(s) as necessary

$maxIOSize = 256\,\text{kbyte}$

$readOps = p(cacheMiss) * W_{totalTx}$

$writeOps = p(dirtyPageEvict) * W_{totalTx}$

$$logWriteOps = W_{updates} * \begin{cases} \frac{ariesRecordSize}{maxIOSize} & \textbf{if } W_{groupCommit} \\ \lceil \frac{ariesRecordSize}{maxIOSize} \rceil & \textbf{otherwise} \end{cases}$$

$capacity = W_{dbSize} + ariesLogStorage$

$iops = readOps + writeOps + logWriteOps$

$$bandwidth = \begin{array}{l} pageSize * (pageWrites + pageReads) + \\ ariesRecordSize * logWriteOps \end{array}$$

$R = configureBlockDevice(capacity, iops, bandwidth, R_{type})$

// Check if R exceeds documented instance limits

if($(R_{\#devices} > N_{maxDevices})$ || $(R_{iops} > N_{maxIops})$ ||
$(R_{bandwidth} > N_{maxBandwidth})$) **return** $\perp$
// Still cap if CPU is the bottleneck
$S_{updates} = min\left\{ \frac{N_{cpuCycles}}{cpuCyclesPerTx}, W_{reqUpdates} \right\}$
$cpuLeft = N_{cpuCycles} - S_{updates} * cpuCyclesPerTx$
$S_{lookups} = min\left\{ \frac{cpuLeft}{cpuCyclesPerTx}, W_{reqLookups} \right\}$
$S_{opLatency} = p(cacheHit) * memoryOpLatency$
$+ p(N_{cacheMiss}) * R_{readLatency}$
$S_{durability} = R_{durability}$
return $(S_{updates}, S_{lookups}, S_{durability}, S_{opLatency})$

A.4 Model for In-Memory Architecture

$computeInMemoryProperties(W, N)$
Input: workload parameter W,
cloud compute node N
Output: system properties S for *In-Memory*
// Return "incompatible" ($\perp$) if instance is too small
if$(redoLogStorage > N_{storageCapacity})$ **return** $\perp$
if$(W_{dbSize} > N_{memorySize})$ **return** $\perp$
// In-Memory only writes log entries
$maxIOSize = 4096$ byte
$logWritesPerUpdate = \begin{cases} \frac{redoRecordSize}{maxIOSize} & \textbf{if } W_{groupCommit} \\ \lceil \frac{redoRecordSize}{maxIOSize} \rceil & \textbf{otherwise} \end{cases}$
// Limit updates by the most restrictive resource
$S_{updates} = min\left\{ \frac{N_{writeOps}}{writesPerUpdate}, \frac{N_{cpuCycles}}{cpuCyclesPerTx}, W_{reqUpdates} \right\}$
// Calculate the CPU left for lookups
$cpuLeft = N_{cpuCycles} - S_{updates} * cpuCyclesPerTx$
// Limit lookups using the remaining resources
$S_{lookups} = min\left\{ \frac{cpuLeft}{cpuCyclesPerTx}, W_{reqLookups} \right\}$
// All data is retrieved from memory
$S_{opLatency} = memoryOpLatency$
// Same durability as Classic
$S_{durability} = 1 - AFR_{SSD}$
return $(S_{updates}, S_{lookups}, S_{durability}, S_{opLatency})$

A.5 Model for Aurora-like Architecture

$computeAuroraProperties(W, N, D_{type}, numSec)$
Input: workload parameters W,
cloud compute node N
storage node type D_{type},
number of secondaries $numSec$
Output: system properties S for *Aurora-like*
// Storage servers can scale and are never a bottleneck
$D = configureAuroraStorageService(W, D_{type})$
// Send log to storage servers and secondaries
$networkOutPerTx = W_{redoRecordSize} * (D_{logReplicas} + numSec)$
// Read pages from storage servers for cache missses
$networkInPerTx = p(cacheMiss) * pageSize$
// Calculate the resource requirements for updates
$S_{updates} = min\left\{ \begin{matrix} \frac{N_{networkOutLimit}}{networkOutPerTx}, \frac{N_{networkInLimit}}{networkInPerTx}, \\ \frac{N_{cpuCycles}}{cpuCyclesPerTx}, W_{reqUpdates} \end{matrix} \right\}$
// Calculate the resources left for lookups
$cpuLeft = N_{cpuCycles} - S_{updates} * cpuCyclesPerTx$
$networkInForUpdates = S_{updates} * networkInPerTx$
$networkInLeft = N_{networkInLimit} - networkInForUpdates$
$lookupsPrimary = min\left\{ \begin{matrix} \frac{networkInLeft}{networkInPerTx}, \frac{N_{cpuLeft}}{cpuCyclesPerTx}, \\ W_{reqLookups} \end{matrix} \right\}$
// Lookups on secondaries as in HADR
$S_{lookups} = min(lookupsPrimary * numSec, W_{reqLookups})$
// The average latency of transactions on the primary
$S_{opLatency} = \begin{matrix} p(cacheHit) * memoryOpLatency + \\ p(cacheMiss) * (ec2Latency + D_{latency}) \end{matrix}$
// The durability is determined by the storage service
$S_{durability} = D_{durability}$
return $(S_{updates}, S_{lookups}, S_{opLatency}, S_{durability})$

A.6 Model for Socrates-like Architecture

$computeSocratesProperties(W, N, P_{type}, L_{type}, numSec)$
Input: workload parameters W,
cloud compute node N
page node type P_{type},
log node type L_{type},
number of secondaries $numSec$
Output: system properties S for *Socrates-like*
$L = configureSocratesLogService(W, L_{type}, numSec)$
$P = configureSocratesPageService(W, P_{type})$
// Send log to log node and block device
$networkOutPerTx = W_{redoRecordSize}$
// Read pages from page servers for cache missses
// Use local NVMe drive to extend buffer pool
$p(rbpexHit) = \frac{min(N_{memorySize} + N_{storageCapacity}, W_{dbSize})}{W_{dbSize}}$
$p(rbpexMiss) = 1.0 - p(rbpexHit)$
$networkInPerTx = p(rbpexMiss) * pageSize$
// Calculate the resource requirements for updates
$S_{updates} = min\left\{ \begin{matrix} \frac{N_{networkOutLimit}}{networkOutPerTx}, \frac{N_{networkInLimit}}{networkInPerTx}, \\ \frac{N_{cpuCycles}}{cpuCyclesPerTx}, W_{reqUpdates} \end{matrix} \right\}$
// Calculate the resources left for lookups
$cpuLeft = N_{cpuCycles} - S_{updates} * cpuCyclesPerTx$
$networkInForUpdates = S_{updates} * networkInPerTx$
$networkInLeft = N_{networkInLimit} - networkInForUpdates$
$lookupsPrimary = min\left\{ \begin{matrix} \frac{networkInLeft}{networkInPerTx}, \frac{N_{cpuLeft}}{cpuCyclesPerTx}, \\ W_{reqLookups} \end{matrix} \right\}$
// Lookups on secondaries as in Aurora and HADR
$S_{lookups} = min(lookupsPrimary * numSec, W_{reqLookups})$
// The average latency of transactions on the primary
$S_{opLatency} = \begin{matrix} p(cacheHit) * memoryOpLatency + \\ (p(rbpexHit) - p(cacheHit)) * nvmeReadLatency + \\ p(cacheMiss) * (ec2Latency + P_{latency}) \end{matrix}$
// Durability determined by log service
$S_{durability} = L_{Durability}$

return $(S_{updates}, S_{lookups}, S_{opLatency}, S_{durability})$

A.7 Models for Supplemental Building Blocks

configureBlockDevice$(CAP, IOPS, BW, type)$
Input: required capacity of the block device CAP,
required IO operations per second $IOPS$
required bandwidth (reads+writes) BW
device type $type$
Output: block device configuration

$$R_{durability} = \begin{cases} 0.999 & \textbf{if } R_{type} \in \{gp2, gp3, io1\} \\ 0.99999 & \textbf{if } R_{type} \in \{io2, io2x\} \end{cases}$$

$R_{IOPS} = IOPS$

// The current implementation is limited to EBS, but can easily
// be extended to the specifics of other cloud providers

$R_{readLatency} = 374\,\mu s$

$R_{bandwidth} = BW$

$$maxIopsPerGB = \begin{cases} 3 & \textbf{if } R_{type} \in \{gp2\} \\ 50 & \textbf{if } R_{type} \in \{io1\} \\ 500 & \textbf{if } R_{type} \in \{io2, gp3\} \\ 1,000 & \textbf{if } R_{type} \in \{io2x\} \end{cases}$$

$$maxIopsPerDevice = \begin{cases} 16,000 & \textbf{if } R_{type} \in \{gp2, gp3\} \\ 64,000 & \textbf{if } R_{type} \in \{io1, io2\} \\ 256,000 & \textbf{if } R_{type} \in \{io2x\} \end{cases}$$

$$maxBWPerDevice = \begin{cases} 250\,\text{MB/s} & \textbf{if } R_{type} \in \{gp2\} \\ 1\,\text{GB/s} & \textbf{if } R_{type} \in \{gp3, io1, io2\} \\ 4\,\text{GB/s} & \textbf{if } R_{type} \in \{io2x\} \end{cases}$$

$$maxCapPerDevice = \begin{cases} 16\,\text{TB} & \textbf{if } R_{type} \in \{gp2, gp3, io1, io2\} \\ 64\,\text{TB} & \textbf{if } R_{type} \in \{io2x\} \end{cases}$$

$R_{capacity} = max\{CAP, \frac{IOPS}{maxIopsPerGB}\}$

// Provision as many devices as necessary

$$R_{\#devices} = max\left\{ \frac{R_{capacity}}{maxCapPerDevice}, \frac{IOPS}{maxIopsPerDevice}, \frac{B}{maxBWPerDevice} \right\}$$

return $(R_{\#devices}, R_{capacity}, R_{IOPS}, R_{bandwidth}, R_{readLatency})$

---

configureAuroraStorageService(W, D_{type})
Input: workload parameters W,
storage node type D_{type}
Output: properties of storage service D

// Total storage required for copies of db and log

$D_{logReplicas} = 6,\ D_{dbReplicas} = 3$

$$requiredCapacity = \begin{array}{l} D_{logReplicas} * redoLogStorage + \\ D_{dbReplicas} * W_{dbSize} \end{array}$$

// Receive log records for every replica

$networkRecv = W_{updates} * redoRecordSize * D_{logReplicas}$

// Send a page for every cache miss on the primary

$pageRequests = W_{totalTx} * p(cacheMiss)$

$networkSend = pageRequests * pageSize$

// The page servers have a cache as well

$p(cacheMissPageServer) = \frac{D_{storageCapacity} - D_{memorySize}}{D_{storageCapacity}}$

$storageReads = pageRequests * p(cacheMissPageServer)$

$maxIOSize = 4096\,\text{byte}$

$storageWrites = W_{updates} * \frac{redoRecordSize}{maxIOSize}$

// Allocate a share (can be smaller or greater 1.0)
// of storage nodes so all requirements are satisfied

$$D_{nodeAllocation} = max\left\{ \frac{requiredCapacity}{D_{storageCapacity}}, \frac{networkSend}{D_{networkOutLimit}}, \frac{networkRecv}{D_{networkInLimit}}, \frac{storageReads}{D_{readOps}}, \frac{storageWrites}{D_{writeOps}} \right\}$$

$$D_{latency} = \begin{array}{l} p(storageCacheHit) * memoryOpLatency + \\ p(storageCacheMiss) * nvmeReadLatency \end{array}$$

// When a storage node from the fleet fails,
// one 10 GB protection group needs to be restored.
// Assume this takes 10 sec with a 10 Gbit/sec NIC.

$t_{MTTR} = 10\,s$

$monthlyFailureRate = 1.0 - D_{monthlyAvailability}$

$$\lambda = \frac{D_{logReplication} * AFR * t_{MTTR}}{30 * 24 * 3600s}$$

// Durable in one MTTR interval if a read chorum
// of three instances survives. Otherwise like HADR.

$$durability_{MTTR} = \sum_{i=0}^{3} \frac{\lambda^i * e^{-\lambda}}{i!}$$

$D_{durability} = durability_{MTTR}{}^{(365*24*3600s)/t_{MTTR}}$

return $(D_{nodeAllocation}, D_{durability}, D_{latency}, D_{logReplicas})$

---

configureSocratesLogService$(W, L_{type}, numSec)$
Input: workload parameters W,
storage node type L_{type}
number of secondaries $numSec$
Output: properties of log service L

$networkRecv = W_{updates} * W_{redoRecordSize}$

// Ship log to secondaries and two page servers

$networkSend = (2 + numSec) * W_{updates} * W_{redoRecordSize}$

$$logIops = \begin{cases} \frac{redoRecordSize}{maxIOSize} & \textbf{if } W_{groupCommit} \\ \lceil \frac{redoRecordSize}{maxIOSize} \rceil & \textbf{otherwise} \end{cases}$$

$logBW = W_{updates} * redoRecordSize$

$R = configureBlockDevice(redoLogStorage, logIops, logBW, io2)$

if$(!R)$ **return** $\perp$

$$L_{nodeAllocation} = max\left\{ \frac{redoLogStorage}{L_{storageCapacity}}, \frac{networkRecv}{L_{networkInLimit}}, \frac{networkSend}{L_{networkOutLimit}}, \frac{logIops}{L_{writeOps}} \right\}$$

$L_{durability} = R_{durability}$

return $(L_{nodeAllocation}, L_{durability}, R)$

---

configureSocratesPageService(W, P_{type})
Input: workload parameters W,
page node type P_{type}
Output: properties of page service P

// Keep two copies of DB for availability

$requiredCapacity = 2 * W_{dbSize}$

$networkRecv = 2 * W_{updates} * W_{redoRecordSize}$

$networkSend = W_{totalTx} * p(rbpexMiss) * pageSize$

$$P_{nodeAllocation} = max\left\{ \frac{requiredCapacity}{P_{storageCapacity}}, \frac{networkRecv}{P_{networkInLimit}}, \frac{networkSend}{P_{networkOutLimit}} \right\}$$

return $(P_{nodeAllocation})$