Socrates: The New SQL Server in the Cloud

Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, Vikram Wakade
Microsoft Azure & Microsoft Research

ABSTRACT

The database-as-a-service paradigm in the cloud (DBaaS) is becoming increasingly popular. Organizations adopt this paradigm because they expect higher security, higher availability, and lower and more flexible cost with high performance. It has become clear, however, that these expectations cannot be met in the cloud with the traditional, monolithic database architecture. This paper presents a novel DBaaS architecture, called Socrates. Socrates has been implemented in Microsoft SQL Server and is available in Azure as SQL DB Hyperscale. This paper describes the key ideas and features of Socrates, and it compares the performance of Socrates with the previous SQL DB offering in Azure.

CCS CONCEPTS

• **Information systems → DBMS engine architectures**;

KEYWORDS

Database as a Service, Cloud Database Architecture, High Availability

ACM Reference Format:
Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam,, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3299869.3314047

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3314047

1 INTRODUCTION

The cloud is here to stay. Most start-ups are cloud-native. Furthermore, many large enterprises are moving their data and workloads into the cloud. The main reasons to move into the cloud are security, time-to-market, and a more flexible "pay-as-you-go" cost model which avoids overpaying for under-utilized machines. While all these reasons are compelling, the expectation is that a database runs in the cloud at least as well as (if not better) than on premise. Specifically, customers expect a "database-as-a-service" to be highly available (e.g., 99.999% availability), support large databases (e.g., a 100TB OLTP database), and be highly performant. Furthermore, the service must be elastic and grow and shrink with the workload so that customers can take advantage of the pay-as-you-go model.

It turns out that meeting all these requirements is not possible in the cloud using the traditional monolithic database architecture. One issue is cost elasticity which never seemed to have been a consideration for on-premise database deployments: It can be prohibitively expensive to move a large database from one machine to another machine to support a higher or lower throughput and make the best use of the computing resources in a cluster. Another, more subtle issue is that there is a conflict between the goal to support large transactional databases and high availability: High availability requires a small mean-time-to-recovery which traditionally could only be achieved with a small database. This issue does not arise in on-premise database deployments because these deployments typically make use of special, expensive hardware for high availability (such as storage area networks or SANs); hardware which is not available in the cloud. Furthermore, on-premise deployments control the software update cycles and carefully plan downtimes; this planning is typically not possible in the cloud.

To address these challenges, there has been research on new OLTP database system architectures for the cloud over the last ten years; e.g., [5, 8, 16, 17]. One idea is to decompose the functionality of a database management system and deploy the compute services (e.g., transaction processing) and storage services (e.g., checkpointing and recovery) independently. The first commercial system that adopted this idea is Amazon Aurora [20].

	Today	Socrates
Max DB Size	4TB	100TB
Availability	99.99	99.999
Upsize/downsize	O(data)	O(1)
Storage impact	4x copies (+backup)	2x copies (+backup)
CPU impact	4x single images	25% reduction
Recovery	O(1)	O(1)
Commit Latency	3 ms	< 0.5ms
Log Throughput	50MB/s	100+ MB/s

Table 1: Socrates Goals: Scalability, Availability, Cost

This paper presents Socrates, a new architecture for OLTP database systems born out of Microsoft's experience of managing millions of databases in Azure. Socrates is currently available in Azure under the brand SQL DB Hyperscale [2]. The Socrates design adopts the separation of compute from storage. In addition, Socrates separates database log from storage and treats the log as a first-class citizen. As we will see, separating the log and storage tiers separates *durability* (implemented by the log) and *availability* (implemented by the storage tier). Durability is a fundamental property of any database system to avoid data loss. Availability is needed to provide good quality of service in the presence of failures. Traditionally, database systems have coupled the implementation of durability and availability by dedicating compute resources to the task of maintaining multiple copies of the data. However, there is significant, untapped potential by separating the two concepts: (a) In contrast to availability, durability does not require copies in fast storage; (b) in contrast to durability, availability does not require a fixed number of replicas. Separating the two concepts allows Socrates to use the best fit mechanism for the task at hand. Concretely, Socrates requires less expensive copies of data in fast local storage, fewer copies of data overall, less network bandwidth, and less compute resources to keep copies up-to-date than other database architectures currently on the market.

Table 1 shows the impact of Socrates on Azure's DBaaS offerings in terms of database scalability, availability, elasticity, cost (CPU and storage), and performance (time to recovery, commit latency and log throughput). How Socrates achieves these improvements concretely is the topic of this paper.

The remainder of this paper is organized as follows: Section 2 discusses the state-of-the-art. Section 3 summarizes existing SQL Server features that we exploited to build Socrates. Section 4 explains the Socrates architecture. Section 5 gives an overview of important distributed workflows in Socrates. Section 6 demonstrates the flexibility of Socrates to address cost/availability/performance tradeoffs. Section 7 presents the results of performance experiments. Section 8 concludes this paper with possible avenues for future work.

2 STATE OF THE ART

This section revisits four prominent DBaaS systems which are currently used in the marketplace.

SQL DB is Microsoft's DBaaS in Azure. Before Socrates, SQL DB was based on an architecture called HADR that is shown in Figure 1. HADR is a classic example of a log-replicated state machine. There is a Primary node which processes all update transactions and ships the update logs to all Secondary nodes. Log shipping is the de facto standard to keep replicas consistent in distributed database systems [13]. Furthermore, the Primary periodically backups data to Azure's Standard Storage Service (called XStore): log is backed up every five minutes, a delta of the whole database once a day, and a full backup every week. Secondary nodes may process read-only transactions. If the Primary fails, one of the Secondaries becomes the new Primary. With HADR, SQL DB needs four nodes (one Primary and three Secondaries) to guarantee high availability and durability: If all four nodes fail, there is data loss because the log is backed up only every five minutes.

To date, the HADR architecture has been used successfully for millions of databases deployed in Azure. The service is stable and mature. Furthermore, HADR has high performance because every compute node has a full, local copy of the database. On the negative side, the size of a database cannot grow beyond the storage capacity of a single machine. A special case occurs with long-running transactions when the log grows beyond the storage capacity of the machine and cannot be truncated until the long-running transaction commits. *O(size-of-data)* operations also create issues. For instance, the cost of seeding a new node is linear with the size of the database. Backup / restore, scale-up and down are further examples of operations whose cost grows linearly with the size of the database. This is why SQL DB today limits the size of databases to 4TB (Table 1).

Another prominent example of a cloud database system that is based on log-replicated state machines is Google Spanner [11]. To address the *O(size-of-data)* issues, Spanner automatically shards data logically into partitions called *splits*. Multiple copies of a split are kept consistent using the Paxos protocol [9]. Only one of the partitions, called *leader*, can modify the data; the other partitions are read-only. Spanner supports geo-replication and keeps all copies consistent with the help of a *TrueTime* facility, a datacenter-based time source which limits time drift between disparate replicas. Splits are divided and merged dynamically for load balancing and capacity management.

In the last decade, an alternative architecture called *shared disk* has been studied for databases in the cloud [5, 8, 16, 17].

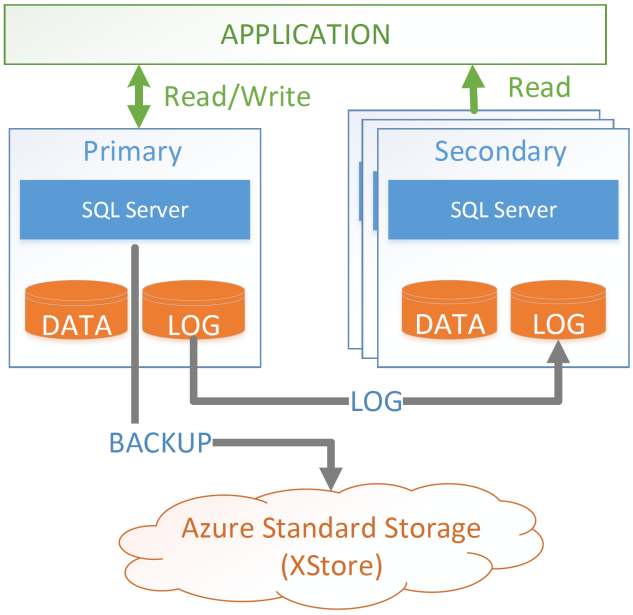

Figure 1: HADR Arch. (Replicated State Machines)

This architecture separates Compute and Storage. AWS Aurora is the first commercial DBaaS that adopted this architecture. An Aurora Primary Compute node processes update transactions (as in HADR) and each log record is shipped to six Storage Servers which persist the data. These six Storage Servers are distributed across three availability zones and a transaction commits safely if its log is persisted successfully in four of the six Storage nodes. Within the Storage tier, the data (and log) is partitioned for scalability. Aurora supports a variable number of Secondary Compute nodes which are attached to the Storage nodes to source data pages.

Oracle pioneered yet a different DBaaS architecture based on Exadata and Oracle RAC. In this architecture, all the nodes of a cluster are tightly coupled on a fast interconnect and a Shared Cache Fusion layer over a distributed storage tier with Storage Cells that use locally attached flash storage [3, 7].

3 IMPORTANT SQL SERVER FEATURES

Socrates builds on foundations already present in SQL Server. This section introduces several important (and not obvious) SQL Server features that were developed independently of Socrates and are critical for Socrates.

3.1 Page Version Store

SQL Server maintains versions of database records for the purpose of providing read snapshots in the presence of concurrent writers (e.g., to implement Snapshot Isolation [4]). In the HADR architecture, all this versioning is done locally in temporary storage. As will become clear in Section 4, Socrates adopts an extended *shared-disk* architecture which requires that row versions can no longer be kept locally in temporary storage: Compute nodes must also share row versions in the shared storage tier.

3.2 Accelerated Database Recovery

SQL Server capitalizes on the persistent version store with a new feature called Accelerated Database Recovery (ADR). Prior to ADR, SQL Server used an ARIES-style recovery scheme [18] that first analyzes the log, then a redo of all transactions that have not committed before the last checkpoint, and finally an undo of all uncommitted (failed) transactions. In this scheme, the *undo* phase can become unboundedly long in the presence of long-running transactions. In production with millions of hosted databases, this unbounded *undo* phase can indeed become a problem. It turns out that the version store can be used to improve this situation significantly: With a shared, persistent version store, the system has access to the committed versions of a row even after a failure which allows to eliminate the *undo* phase in many cases and the database becomes available immediately after the *analysis* and *redo* phases, a constant-time operation bounded by the checkpointing interval.

3.3 Resilient Buffer Pool Extension

In 2012, SQL Server released a feature called buffer pool extension (BPE) which spills the content of the in-memory database buffer pool to a local SSD file (using the same lifetime and eviction policies across both main memory and SSD). In Socrates, we extended this concept and made the buffer pool resilient; i.e., recoverable after a failure. We call this component RBPEX and it serves as the caching mechanism for pages both in the compute and the storage tiers (Section 4). Having a recoverable cache like RBPEX significantly reduces the mean-time-to-recovery until a node reaches peak performance (with warm buffers): If the failure is short (e.g., a reboot of a machine after a software upgrade), it is much cheaper to read and apply the log records of the (few) updated pages than to refetch all (cached) pages from a remote server which is needed in a traditional, non-recoverable cache. A shorter mean-time-to-recovery, increases availability [14].

Architecturally, RBPEX is a simple, straightforward concept. However, a careful implementation, integration, and management of RBPEX is critical for performance. If not done right, performance can even degrade. We built RBPEX as a table in our in-memory storage engine, Hekaton [15], which ensures that read I/O to RBPEX is as fast as direct I/O to the local SSD. Furthermore, Hekaton recovers RBPEX after a failure - just like any other Hekaton table. Write I/O to RBPEX needs to be carefully orchestrated because RPBEX

metadata I/O cannot be allowed to stall data I/O and RBPEX failures cannot be allowed to corrupt the RBPEX state. In order to achieve this we intercepted the buffer pool page lifetime tracking mechanism, which is a highly performance sensitive component.

3.4 RBIO protocol

As we will see, Socrates distributes the components of a database engine across multiple tiers. To support a richer distribution of computation, we extended the traditional SQL Server networking layer (called Unified Communication Stack) with a new protocol called Remote Block I/O, or RBIO for short. RBIO is a stateless protocol, strongly typed, has support for automatic versioning, is resilient to transient failures, and has QoS support for best replica selection.

3.5 Snapshot Backup/Restore

SQL Server 2016 introduced the ability to take an almost instantaneous backup when the database files were stored in Azure. This feature relied on the blob snapshots feature implemented by Azure Storage (XStore) [10] which is organized as a log-structured storage system [19]. In a log-structured file system, a backup is a constant-time operation as it merely needs to keep a pointer (timestamp) to the current head of the log. Socrates extends this feature by making backup/restore work entirely on XStore snapshots. As a result, Socrates can do backups (and restores) in constant time without incurring any CPU or I/O cost in the Compute tier. With XStore's Snapshot mechanism, the database files of even a very large database of hundreds of TBs can be restored in minutes. Of course, it takes longer to apply the log to recover to the right moment in time (using ADR) and spin up the servers and refresh the caches for the restored database, but none of those operations depend on the size of the database. Backup/restore is one prominent example where Socrates eliminated size-of-data operations from a critical operational workflow.

3.6 I/O Stack Virtualization

At the lowest level of the I/O stack, SQL Server uses an abstraction called FCB for "File Control Block". The FCB layer provides I/O capabilities while abstracting the details of the underlying device. Using this abstraction layer, SQL Server can support multiple file systems and a diverse set of storage platforms and I/O patterns. Socrates exploits this I/O virtualization tier extensively by implementing new FCB instances which hide the Socrates storage hierarchy from the compute process. This approach helped us to implement Socrates without changing most core components of SQL Server: Most components "believe" they are components of a monolithic, standalone database system, and no component above the FCB layer needs to deal with the complexity of a distributed, heterogeneous system that Socrates indeed is.

4 SOCRATES ARCHITECTURE

4.1 Design Goals and Principles

Today, Azure successfully hosts many databases using the HADR architecture described in Section 2. Operating all these databases in production has taught us important lessons that guided the design of Socrates. Before explaining the Socrates architecture, we describe these lessons and the corresponding Socrates design goals and principles.

4.1.1 Local Fast Storage vs. Cheap, Scalable, Durable Storage. The first lesson pertains to the storage hierarchy: Direct attached storage (SSD) is required for high performance, whereas cheap storage (hard disks) is needed for durability and scalability of large databases. On premise, these requirements are met with storage systems like SANs that transparently optimize different kinds of storage devices in a single storage stack. In the cloud, such storage systems do not exist; instead, there is local storage (SSD) attached to each machine which is fast, limited, and non-durable as it is lost when the machine fails permanently. Furthermore, clouds like Azure feature a separate, remote storage service for cheap, unlimited, durable storage. To achieve good performance, scalability, and durability in the cloud, Socrates has a layered, scale-out storage architecture that explicitly manages the different storage devices and services available in Azure. One specific feature of this architecture is that it avoids fragmentation and expensive data movement for dynamic storage allocation of fast-growing databases.

4.1.2 Bounded-time Operations. As shown in Table 1, one important design goal of Socrates is to support large databases in the order of 100 TB. Unfortunately, the current HADR architecture involves many operations whose performance depends on the size of the database as described in Section 2. Fast creation of new replicas is particularly important to achieve high availability at low cost because this operation determines the mean-time-to-recovery which directly impacts availability for a given number of replicas [14]. The requirement to avoid any "size-of-data operations" has led us to develop new mechanisms for backup/restore (based on snapshots), management of the database log (staging), tiered caching with asynchronous seeding of replicas, and exploitation of the scale-out storage service.

4.1.3 From Shared-nothing to Shared-disk. One of the fundamental principles of the HADR architecture (and any other replicated state-machine DBMS architecture) is that each replica maintains a local copy of the database. This principle conflicts with our design goal to support large databases of

several hundred TBs because no machine has that amount of storage. Even if it were possible, storage becomes the limiting factor and main criterion when placing databases on machines in HADR; as a result, CPU cycles go to waste if a large, say, 100TB database is deployed with a fairly light workload.

These observations motivated us to move away from the shared-nothing model of HADR (and replicated state machines) and towards a shared-disk design. In this design, all database compute nodes which execute transactions and queries have access to the same (remote) storage service. Sharing data across database nodes requires support for data versioning at different levels. To this end, Socrates relies on the shared version store described in Section 3.1. The combination of a shared version store and accelerated recovery (ADR, Section 3.2) makes it possible for new compute nodes to spin up quickly and to push the boundaries of read scale-out in Socrates well beyond what is possible in HADR.

4.1.4 Low Log Latency, Separation of Log. The log is a potential bottleneck of any OLTP database system. Every update must be logged before a transaction can commit and the log must be shipped to all replicas of the database to keep them consistent. The question is how to provide a highly performant logging solution at cloud scale?

The Socrates answer to this question is to provide a separate logging service. This way, we can tune and tailor the log specifically to its specific access pattern. First, Socrates makes the log durable and fault-tolerant by replicating the log: A transaction can commit as soon as its log records have been made durable. It turns out that our implementation of quorum to harden log records is faster than achieving quorum in a replicated state machine (e.g., HADR). As a result, Socrates can achieve better commit performance as shown in Table 1.

Second, reading and shipping log records is more flexible and scalable if the log is decoupled from other database components. Socrates exploits the asymmetry of log access: Recently created log records are in high demand whereas old log records are only needed in exceptional cases (e.g., to abort and undo a long-running transaction). Therefore, Socrates keeps recent log records in main memory and distributes them in a scalable way (potentially to hundreds of machines) whereas old log records are destaged and made available only upon demand.

Third, separating log makes it possible to stand on giant's shoulders and plug in any external storage device to implement the log. As shown in Appendix A, this feature is already paying off as Socrates can leverage recent innovations in Azure storage without changing any of its architectural tenets. In particular, this feature allows Socrates to achieve low commit latencies without the need to implement its own log shipping, gossip quorum protocol, or log storage system.

4.1.5 Pushdown Storage Functions. One advantage of the shared-disk architecture is that it makes it possible to offload functions from the compute tier onto the storage tier, thereby moving the functions to the data. This way, Socrates can achieve significant performance improvements. Most importantly, every database function that can be offloaded to storage (whether backup, checkpoint, IO filtering, etc.) relieves the Primary Compute node and the log, the two bottlenecks of the system.

4.1.6 Reuse Components, Tuning, Optimization. SQL Server has a rich eco-system with many tools, libraries, and existing applications. Applications on the existing, millions of SQL DB databases in Azure must migrate to Socrates in a seamless way. Moreover, full backward compatibility with the millions of SQL Server on-premise databases that might one day be migrated to Azure is also a top priority. Thus, Socrates needs to support the same T-SQL programming language and basic APIs for managing databases. Furthermore, SQL Server is an enterprise-scale database system with decades of investment into robustness (testing) and high performance. We did not want to and cannot afford to reinvent the wheel and degrade customer experience under any circumstances. So, critical components of SQL Server such as the query optimizer, the query runtime, security, transaction management and recovery, etc. are unchanged. Furthermore, Socrates databases are tuned in the same way as HADR databases and Socrates behaves like HADR for specific workloads (e.g., repeated updates to hot rows). Socrates (like HADR) also embraces a scale-up architecture for high throughput as this is the state-of-the-art and sufficient for most OLTP workloads.

4.2 Socrates Architecture Overview

Figure 2 shows the Socrates architecture. As will become clear, it follows all the design principles and goals outlined in the previous section: (a) separation of Compute and Storage, (b) tiered and scaled-out storage, (c) bounded operations, (d) separation of Log from Compute and Storage, (e) opportunities to move functionality into the storage tier, and (f) reuse of existing components.

The Socrates architecture has four tiers. Applications connect to Compute nodes. As in HADR, there is one Primary Compute node which handles all read/write transactions. There can be any number of Secondaries which handle read-only transactions or serve as failover targets. The Compute nodes implement query optimization, concurrency control, and security in the same way as today and support T-SQL and the same APIs (Section 4.1.6). If the Primary fails, one of the Secondaries becomes the new Primary. All Compute

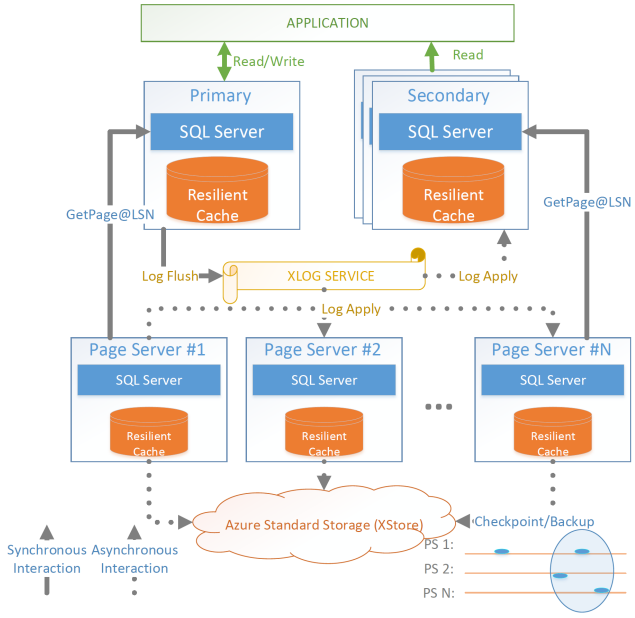

Figure 2: Socrates Architecture

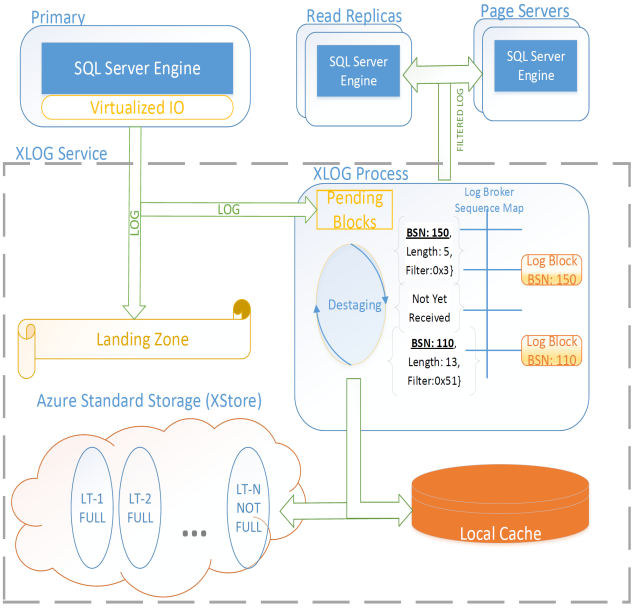

Figure 3: XLOG Service

nodes cache data pages in main memory and on SSD in a resilient buffer pool extension (Section 3.3).

The second tier of the Socrates architecture is the XLOG service. This tier implements the "separation of log" principle, motivated in Section 4.1.4. While this separation has been proposed in the literature before [6, 8], this separation of log differentiates Socrates from other cloud database systems such Amazon Aurora [20]. The XLOG service achieves low commit latencies and good scalability at the storage tier (scale-out). Since the Primary processes all updates (including DML operations), only the Primary writes to the log. This single writer approach guarantees low latency and high throughput when writing to the log. All other nodes (e.g., Secondaries) consume the log in an asynchronous way to keep their copies of data up to date.

The third tier is the storage tier. It is implemented by Page Servers. Each Page Server has a copy of a partition of the database, thereby deploying a scale-out storage architecture which, as we will see, helps to bound all operations as postulated in Section 4.1.2. Page Servers play two important roles: First, they serve pages to Compute nodes. Every Compute node can request pages from a Page Server, following a shared-disk architecture (Section 4.1.3). We are currently working on implementing bulk operations such as bulk loading, index creation, DB reorgs, deep page repair, and table scans in Page Servers to further offload Compute nodes as described in Section 4.1.5. In their second role, Page Servers checkpoint data pages and create backups in XStore (the fourth tier). Like Compute nodes, Page Servers keep all their data in main memory or locally attached SSDs for fast access.

The fourth tier is the Azure Storage Service (called XStore), the existing storage service provided by Azure independently of Socrates and SQL DB. XStore is a highly scalable, durable, and cheap storage service based on hard disks. Data access is remote and there are throughput and latency limits imposed by storage at that scale and price point. Separating Page Servers with locally attached, fast storage from durable, scalable, cheap storage implements the design principles outlined in Section 4.1.1.

Compute nodes and Page Servers are stateless. They can fail at any time without data loss. The "truth" of the database is stored in XStore and XLOG. XStore is highly reliable and has been used by virtually all Azure customers for many years without data loss. Socrates leverages this robustness. XLOG is a new service that we built specifically for Socrates. It has high performance requirements, must be scalable, affordable, and must never lose any data. We describe our implementation of XLOG, Compute nodes, and Page Servers in more detail next.

4.3 XLOG Service

Figure 3 shows the internals of the XLOG Service. Starting in the upper left corner of Figure 3, the Primary Compute node writes log blocks directly to a landing zone (LZ) which is a fast and durable storage service that provides strong guarantees on data integrity, resilience and consistency; in other words, a storage service that has SAN-like capabilities.

The current version of SQL DB Hyperscale uses the Azure Premium Storage service (XIO) to implement the landing zone. For durability, XIO keeps three replicas of all data. As with every storage service, there is a performance, cost, availability, and durability tradeoff. Furthermore, there are many innovations in this space. Socrates naturally benefits from these innovations. Appendix A studies this effect by showing the performance impact of using an alternative storage service instead of XIO.

The Primary writes log blocks synchronously and directly to the LZ for lowest possible commit latency. The LZ is meant to be fast (possibly expensive) but small. The LZ is organized as a circular buffer, and the format of the log is a backward-compatible extension of the traditional SQL Server log format used in all of Microsoft's SQL services and products. This approach obeys the design principle of not reinventing the wheel (Section 4.1.6) and maintaining compatibility between Socrates and all other SQL Server products and services. One key property of this log extension is that it allows concurrent log readers to read consistent information in the presence of log writers without any synchronization (beyond wraparound protection). Minimizing synchronization between tiers leads to a system that is more scalable and more resilient.

The Primary also writes all log blocks to a special XLOG process which disseminates the log blocks to Page Servers and Secondaries. These writes are asynchronous and possibly unreliable (in fire-and-forget style) using a lossy protocol. One way to think about this scheme is that Socrates writes synchronously and reliably into the LZ for *durability* and asynchronously to the XLOG process for *availability*.

The Primary writes log blocks into the LZ and to the XLOG process in parallel. Without synchronization, it is possible that a log block arrives at, say, a Secondary *before* it is made durable in the LZ. We call such an unsynchronized approach *speculative logging* and it can lead to inconsistencies and data loss in the presence of failures. To avoid these situations, XLOG only disseminates *hardened* log blocks. Hardened blocks are blocks which have already been made durable (with write quorum) in the LZ. To this end, the Primary writes all log blocks first into the "pending area" of the XLOG process. Furthermore, the Primary informs XLOG of all hardened log blocks. Once a block is hardened, XLOG moves it from the "pending area" to the LogBroker for dissemination, thereby also filling in gaps and reordering out of order blocks from the lossy protocol to write into the "pending area".

To disseminate and archive log blocks, the XLOG process implements a storage hierarchy. Once a block is moved into the LogBroker, an internal XLOG process called *destaging* moves the log to a fixed size local SSD cache for fast access and to XStore for long term retention. Again, XStore is cheap, abundant, durable, yet slow. We refer to this long-term archive for log blocks as LT. If not specified otherwise, SQL DB keeps log records for 30 days for point-in-time recovery and disaster recovery with fuzzy backups. It would be prohibitively expensive to keep 30 days-worth of log records in the LZ which is a low-latency, expensive service. This destaging pipeline must be carefully tuned: Socrates cannot process any update transactions once the LZ is full with log records that have not been destaged yet. While this tiered architecture is complex, no other log backup process is needed; between the LZ and LT (XStore), all log information is durably stored. Furthermore, this hierarchy meets all our latency (fast commit in LZ) and cost requirements (mass storage in XStore).

Consumers (Secondaries, Page Servers) *pull* log blocks from the XLOG service. This way, the architecture is more scalable as the LogBroker need not keep track of log consumers, possibly hundreds of Page Servers. At the top level, the LogBroker has a main-memory hash map of log blocks (called Sequence Map in Figure 3). In an ideal system, all log blocks are served from this Sequence Map. If the data is not found in the Sequence Map, the local SSD cache of the XLOG process is the next tier. This local SSD cache is another circular buffer of the tail of the log. If a consumer requests a block that has aged out of the local SSD cache, the log block is fetched from the LZ and, if that fails, from the LT as a last resort where the log block is guaranteed to be found.

The XLOG process implements a few other generic functions that are required by a distributed DBaaS system: leases for log lifetime, LT blob cleanup, backup/restore bookkeeping, progress reporting for log consumers, block filtering, etc. All of these functions are chosen carefully to preserve the stateless nature of the XLOG process, to allow for easy horizontal scaling, and to avoid affecting the main XLOG functions of serving and destaging log.

4.4 Primary Compute Node and GetPage@LSN

A Socrates Primary Compute node behaves almost identically to a standalone process in an on-premise SQL Server installation. The database instance itself is unaware of the presence of other replicas. It does not know that its storage is remote, or that the log is not managed in local files. In contrast, a HADR Primary node is well aware that it participates in a replicated state machine and achieves quorum to harden log and commit transactions. In particular, a HADR Primary knows all the Secondaries in a tightly coupled way. The Socrates Primary is, thus, simpler.

The core function of the Primary, however, is the same: Process read/write transactions and produce log. There are several notable differences from an on-premise SQL Server:

- Storage level operations such as checkpoint, backup/restore, page repair, etc. are delegated to the Page Servers and lower storage tiers.
- The Socrates Primary writes log to the LZ using the virtualized filesystem mechanism of Section 3.6. This mechanism produces an I/O pattern that is compatible with the LZ concept described in Section 4.3.
- The Socrates Primary makes use of the RBPEX cache (Section 3.3). RBPEX is integrated transparently as a layer just above the I/O virtualization layer.
- Arguably, the biggest difference is that a Socrates Primary does not keep a full copy of the database. It merely caches a *hot* portion of the database that fits into its main memory buffers and SSD (RBPEX).

This last difference requires a mechanism for the Primary to retrieve pages which are not cached in the local node. We call this mechanism GetPage@LSN. The GetPage@LSN mechanism is a remote procedure call which is initiated by the Primary from the FCB I/O virtualization layer using the RBIO protocol (Section 3.4). The prototype for this call has the following signature:

getPage(pageId, LSN)

Here, *pageId* identifies uniquely the page that the Primary needs to read, and *LSN* identifies a page log sequence number with a value at least as high as the last PageLSN of the page. The Page Server (Section 4.6) returns a version of the page that has applied all updates up to this LSN or higher.

To understand the need for this mechanism, consider the following sequence of events:

(1) The Primary updates Page X in its local buffers.
(2) The Primary evicts Page X from its local buffers (both buffer pool and RBPEX) because of memory pressure or accumulated activity. Prior to the page eviction, the Primary follows the write-ahead logging protocol [18] and flushes all the log records that describe the changes to Page X to XLOG.
(3) The Primary reads Page X again.

In this scenario, it is important that the Primary sees the latest version of Page X in Step 3 and, thus, issues a *getPage(X, X-LSN)* request with a specific *X-LSN* that guarantees that the Page Server returns the latest version of the page.

To guarantee freshness, the Page Server handles a *getPage(X, X-LSN)* request in the following way:

(1) Wait until it has applied all log records from XLOG up to X-LSN.
(2) Return Page X.

This simple protocol is all that is needed to make sure that that the Page Server does not return a stale version of Page X to the Primary. (Section 4.6 contains more details on Page Servers.)

We have not described yet how the Primary knows which X-LSN to use when issuing the *getPage(X, X-LSN)* call. Ideally, X-LSN would be the most recent page LSN for Page X. However, the Primary cannot remember the LSNs of all pages it has evicted (essentially the whole database). Instead, the Primary builds a hash map (on pageId) which stores in each bucket the highest LSN for every page evicted from the Primary keyed by pageId. Given that Page X was evicted at some point from the Primary, this mechanism will guarantee to give an X-LSN value that is at least as large as the largest LSN for Page X and is, thus, safe.

4.5 Secondary Compute Node

Following the reuse design principle (Section 4.1.6), a Socrates Secondary shares the same apply log functionality as in HADR. A (simplifying) difference is that the Socrates Secondary need not save and persist log blocks because that is the responsibility of the XLOG service. Furthermore, Socrates is a loosely coupled architecture so that the Socrates Secondary does not need to know who produces the log (i.e., which node is the Primary). As in HADR, the Socrates Secondary processes read-only transactions (using Snapshot Isolation [4]). The most important components such as the query processor, the security manager, and the transaction manager are virtually unchanged from standalone SQL Server and HADR.

As with the Primary, the most significant changes between Socrates and HADR come from the fact that Socrates Secondaries do not have a full copy of the database. This fact is fundamental to achieving our goal to support large databases and making Compute nodes stateless (with a cache). As a result, it is possible that the Secondary processes a log record that relates to a page that is not in its buffers (neither main memory nor SSD). There are different policies conceivable for this situation. One possible policy is to fetch the page and apply the log record. This way, the Secondary's cache has roughly the same state as the Primary's cache (at least for updated pages) and performance is more stable after a fail-over to the Secondary.

The policy that SQL DB Hyperscale currently implements is that log records that involve pages that are not cached are simply ignored. This policy results in an interesting race condition because the check for existence in the cache can conflict with a concurrent, pending GetPage@LSN request from a read-only transaction processed by the Secondary. To resolve this conflict, Secondaries must *register* GetPage@LSN requests *before* making the actual call and the *apply-log* thread of the Secondary queues log records of pending GetPage@LSN requests until the page is loaded.

Another interesting race condition arises when the Secondary does B-tree traversals to process read-only transactions. The Secondary applies the same GetPage@LSN protocol to fetch pages from Page Servers as the Primary. Again, the Secondary has a PageLSN cache to conservatively determine the right LSN. This LSN is typically lower than the last LSN written by the Primary at the same point in time. This can result in inconsistencies. Consider the following situation as part of a B-tree traversal:

- The Secondary reads B-tree Node P with LSN 23. (The Secondary has applied log until LSN 23.)
- The next Node C, a child of P, is not cached. The Secondary issues a getPage(C, 23) request.
- The Page Server returns Page C with LSN 25, following the protocol described in Section 4.4.

According to the GetPage@LSN protocol, the Page Server may return a page from the *future*. This can create inconsistencies. For instance, what if Page C was split at Time 24? In this case, the Secondary has looked at Page P from its present (before the split) and looks at Page C from the future (after the split) which may result in wrong results. Fortunately, it is easy to detect such inconsistencies. If the Secondary detects such an inconsistency during an index traversal, it will pause to give the log apply thread some time to consume more log to refresh the stale index pages (i.e., Page P). After that pause, it will restart the B-tree traversal, hoping that the index structure is now consistent.

The key insight here is that persistent data structures in SQL Server have been designed such that logically coherent traversals can be achieved even in the presence of pages that are physically obtained from different points in time (from an LSN perspective). Logical traversals rely on the version store described in Section 3.1 to extract the right version of a record from a page given the transaction timestamp (using Snapshot Isolation).

4.6 Page Servers

A Page Server is responsible for (i) maintaining a partition of the database by applying log to it, (ii) responding to GetPage@LSN requests from Compute nodes and (iii) performing distributed checkpoints and taking backups.

Following the reuse principle of Section 4.1.6, Page Servers carry out the log apply task in a manner that is similar to the procedure for Secondaries described in Section 4.5. In contrast to Secondaries which need to be made aware of all changes to the database and, thus, need to consume all log blocks, Page Servers only ever care about log blocks that contain log records that involve pages of the database partition handled by that particular Page Server. To this end, the Primary includes sufficient out-of-band annotations for each log block that indicates which partitions are affected by log records in a log block. XLOG uses this filtering information to disseminate only relevant log blocks to each Page Server.

Serving GetPage@LSN requests is also straightforward. The Page Server simply follows the protocol of Section 4.4. To this end, Page Servers also use RBPEX, the SSD-extended, recoverable cache. The mechanisms to use RBPEX are the same for Page Servers as for Compute nodes, but the policy is different. Compute nodes cache the hottest pages for best performance; their caches are sparse. In contrast, Page Servers cache less hot pages, those pages that are not hot enough to make it in the Compute node's cache. That is why SQL DB Hyperscale currently implements Page Servers using a *covering cache*; i.e., all pages of the partition are stored in the Page Server's RBPEX. Furthermore, Socrates organizes a Page Server's RBPEX in a stride-preserving layout such that a single I/O request from a compute node that covers a multi-page range translates into a single I/O request at the Page Server. Since the Page Server's cache is dense, the Pager Server does not suffer from read amplification while the sparse RBPEX caches at Compute nodes do. This characteristic is important for the performance of scan operations that commonly read up to 128 pages. Another important characteristic of the Page Server cache is that it provides insulation from transient XStore failures. On an XStore outage, the Page Server continues operating in a mode where pages that were written in RBPEX but not in XStore are remembered and checkpointing is resumed (and XStore is caught up) when XStore is back online. The same mechanism allows Socrates to aggregate multiple I/Os being sent to XStore in a single large write operation in order to get the best possible throughput out of the underlying storage service. Finally, when a new Page Server is started, its RBPEX is seeded asynchronously while the Page Server is already available and able to serve requests and apply log. Decoupling long running operations such as seeding new Page Servers from other operational tasks (Section 4.1.3) is one of the key principles we tried to enforce at all layers in the Socrates stack.

Finally, Page Servers take checkpoints and do backups by interacting with XStore. How Page Servers and XStore operate together for these operations is the subject of the next subsection.

4.7 XStore for Durability and Backup/Restore

As shown in the previous sections, the *truth* of the database is stored in XStore. The details of XStore are described in [10]. In a nutshell, XStore is cheap (based on hard disks), durable (virtually no data loss due to a high degree of replication across availability zones), and provides efficient backup and restore by using a log-structured design (taking a backup

merely requires keeping a pointer into the log) [19]. But XStore is potentially slow. The goal of all other components of the Socrates architecture is to add performance and availability. In other words, XStore plays in Socrates the same role as hard disks and tape in a traditional database system. The main-memory and SSD caches (RBPEX) of Compute nodes and Page Servers play in Socrates the same role as main memory in a traditional system.

Once these analogies are understood, it is straightforward to see how Socrates does checkpointing, recovery, backup, and restore. For checkpointing, a Page Server regularly ships modified pages to XStore. Backups are implemented using XStore's snapshot feature which allows to create a backup in constant time by simply recording a timestamp [10]. When a user requests a Point-In-Time-Restore operation (PITR), the restore workflow identifies (i) the complete set of snapshots that have been taken just before the time of the PITR, and (ii) the log range required to bring this set of snapshots from their time to the requested time. These snapshots are then copied to new blobs and a restore operation starts in which each blob is attached to a new Page Server instance, a new XLOG process is bootstrapped on the copied log blobs and the log applied to bring the database all the way to the PITR requested time.

This Socrates PITR process is much more efficient than in HADR today. Like taking a snapshot, restoring a snapshot in XStore is a short, meta-data operation that is executed in constant time (independent of the size of the data). Furthermore, the database becomes available almost instantaneously because the new Page Servers can start serving pages of the restored database instantaneously. However, performance will degrade until the caches of the Page Servers are restored because pages need to be fetched and recovered from XStore first. This example demonstrates again the importance to leverage existing cloud services (Section 4.1.4).

5 SOCRATES AT WORK

The GetPage@LSN protocol described in Sections 4.4 to 4.6 is a nice example of how Socrates implements distributed protocols. The design principles are always the same: The Socrates mini-services like Primary, Secondaries, XLOG, and Page Servers are autonomous and decoupled and communication is asynchronous whenever possible. For scalability, mini-services do not need to know about other mini-services; for instance, the Primary need not know how many Page Servers exist, possibly hundreds (Section 6). Synchronization is done by time travel whenever possible and by waiting if a mini-service is behind.

Socrates implements all distributed algorithms following these principles. Other examples, omitted for brevity, are distributed checkpointing (across all Page Servers), fail-over of the Primary, scaling up and down (i.e., serverless), creating a new Secondary, management of leases for log consumption, and creating new Page Servers.

6 DISCUSSION & SOCRATES DEPLOYMENTS

The Socrates architecture has many advantages. Separating Compute and Storage (Page Servers in Socrates) has been studied extensively in the literature [5, 8, 16, 17, 20]: It helps to build scalable database systems that grow beyond the storage capabilities of a single machine. Furthermore, this principle helps to establish a more fine-grained pay-as-you-go model in the cloud in which customers pay only for the storage that they need and independently for the compute that they consume. Socrates inherits all the advantages of this separating compute and storage principle. It also inherits the disadvantages in that reads can become more expensive as they may need to access remote servers; Socrates remedies this disadvantage by aggressively caching data in Compute nodes in main memory and disk, and Section 4 described the many innovations we did to get caching right.

A novel feature of the Socrates architecture is that it separates *availability* from *durability*. In Socrates, XLOG and XStore are the tiers that are responsible for *durability* whereas the *Compute* and *Page Server* tiers are only there for *availability*: If they fail, no data is lost, but the service becomes unavailable until they are restored. The big advantage of this separation is that it provides flexibility and fine-grained control to navigate the availability / performance / cost tradeoff. This way a Socrates deployment can be tailored to the specific needs of an application.

XLOG and XStore are needed in all Socrates deployments for durability. So, the simplest Socrates deployment consists of a single Compute node (the Primary Compute node, no Secondaries) and one Page Server that handles page requests for the entire database. This deployment is the most cost-effective deployment; its performance depends on the hardware used to run the Primary and the Page Server. The downside of this minimum deployment is *availability*: If, say, the Primary Compute node fails, a new Compute node needs to be spun up to become the new Primary, resulting in downtime. Once the new Primary node is up, the system is available, but peak performance is only achieved after the cache of the new Primary is hot.

To achieve higher availability and avoid performance jitter after failures, any number of Secondaries can be added, at higher cost and better performance because the Secondaries can be used for read-only transactions. Increasing the number of Page Servers also increases cost, performance, and availability. Interestingly, there are two ways in Socrates

to add Page Servers with subtly different availability impact. One way to add a Page Server is to make the sharding of the database more fine-grained. This way, availability is improved because the partitions are smaller and, thus, the *mean-time-to-recovery* is smaller as it is faster to spin up a new Page Server for a partition. According to [14], a lower mean-time-to-recovery implies higher availability. Furthermore, this approach improves performance by increasing the degree of parallelism when bulk operations (e.g., large table scans or bulkloads) are pushed down to Page Servers [12]. With today's network and hardware parameters, we calculated that a good partition size for a Page Server is 128GB. So, a Socrates database with hundreds of TB will result in a deployment with thousands of Page Servers.

A second way to add a Page Server is to create a replica of an existing Page Server. This approach increases availability as the replica is ready (and hot) when the Page Server fails.

Geo-replication is an important way to increase both availability and performance. Socrates allows to deploy Secondaries and Page Servers in different data centers and availability zones. This approach improves performance, for instance, because the database can be queried by local Secondaries around the world. But, of course, geo-replication also comes at a cost of shipping the log across data centers.

Finally, an important advantage of the Socrates architecture is that it makes best use of other, existing cloud services. The Azure XStore service is used to do efficient backups (for point in time recovery) and to implement durability in a scalable and cheap way. The XLOG tier depends on Azure Premium Storage. As we will see in Section 7, Socrates naturally benefits from innovations in these existing services. The Compute and Page Server tiers implement native database functionality only and do not replicate any functionality provided by other, more general cloud services.

7 PERFORMANCE EXPERIMENTS AND RESULTS

This section presents experimental results that assess the effectiveness of the Socrates design. As a baseline, we use the HADR architecture (Section 2).

7.1 Software and Services Used

We implemented Socrates as part of the SQL DB Hyperscale service in Azure. At the time of writing this paper, this service was in preview and we used this preview version for all experiments reported in this paper. We experimented with two different deployments:

- *Production:* This is the same version of SQL DB Hyperscale that Azure customers get. It allowed us to do an apples-to-apples comparison of Socrates with the HADR-based SQL DB service. It uses Azure Premium Storage (XIO) to implement the Socrates LZ.
- *Test:* We also deployed Socrates in a test cluster to study the impact of a new, improved premium storage service to implement the LZ.

The experiments with the Test system are presented in Appendix A.

We used Azure VMs of different T-shirt sizes. For the experiments in the *Test* cluster, we used VMs with 64 cores, 432 GB of main memory, and 32 disks. In the *Production* deployment, we used VMs with 8 and 16 cores for both Socrates and HADR.

We used the CDB benchmark [1] which is Microsoft's *Cloud Database Benchmark* (also known as the DTU benchmark) and has been used to test the performance of all Microsoft DBaaS offerings in Azure. CDB is based on a synthetic database with six tables and a scaling factor to generate databases of different sizes. If not stated otherwise, we used a 1TB CDB database which is a size that is comfortably supported by HADR. We also did experiments that demonstrate the scalability of Socrates to much larger sizes (not supported by HADR).

CDB defines a set of transaction types covering a wide range of operations from simple point lookups to complex bulk updates. Furthermore, CDB specifies *workload mixes* that test system characteristics for specific workloads; e.g., read-only.

	CPU %	Write TPS	Read TPS	Total TPS
HADR	99.1	347	1055	1402
Socrates	96.4	330	1005	1335

Table 2: CDB Throughput: HADR vs. Socrates (1TB)

7.2 Experiment 1: CDB Default Mix, Throughput, Production Cluster

Table 2 shows the throughput of Socrates and HADR in production on an eight core VM with 64 concurrent client threads to generate the workload. For these experiments we used the *default workload mix* of CDB which executes all transaction types of the benchmark. The size of the database was 1TB.

Table 2 shows that Socrates throughput is about 5% lower than the throughput of HADR. This result is not unexpected for a service in preview, especially considering that HADR has been tuned over the years for this benchmark. It is interesting to understand how Socrates loses this 5%. HADR achieves 99.1% CPU utilization which is almost perfect. Socrates has a lower CPU utilization as it needs to wait longer for remote I/Os. This lower CPU utilization explains half of the deficit. Recall that remote I/O is fundamental to

Data Size	Scale Factor	Memory Size	RBPEX Size	Local cache hit %
1TB	20000	56GB	168GB	52

Table 3: Socrates Cache Hit Rate (CDB)

Data Size	Customers	Memory Size	RBPEX Size	Local cache hit %
30TB	3.1M	88GB	320GB	32

Table 4: Socrates Cache Hit Rate (TPC-E)

any DBaaS architecture that scales database sizes beyond what can fit into a single machine. From performance profiles, it seems that the other 2.5% performance gap come from a higher CPU cost of writing the log to a remote service (XLOG). We believe that Socrates can catch up and become even better than HADR with larger caches and further tuning of the I/O and network protocols.

7.3 Experiment 2: Caching Behavior

Table 3 shows the cache hit rate of Socrates for a 1TB CDB database and an RBPEX with 56GB of main memory buffers, and 168 GB of SSD storage space. (Obviously, the hit rate is 100% in HADR because HADR stores a full copy of the database in every Compute node.)

For this experiment, we again used the *default* workload mix of CDB. This workload randomly touches pages scattered across the entire database. Thus, this workload is a bad case for caching. Nevertheless, we achieve a 50% hit rate for a cache that is only 15% of the size of the database. To study a more realistic scenario, we ran Socrates on a 30TB TPC-E database using the TPC-E benchmark. For this experiment, we increased the buffer pool of the Socrates Primary (88 GB of main memory and 320 GB of SSD). Table 4 shows that even though the cache is only about 1% of the size of the database, Socrates had a 32% hit rate.

These results indicate that a smart local SSD cache can be extremely effective. We believe that similar improvements can be made in other operational areas, too, thereby exploiting the flexible, decomposed Socrates architecture; e.g., to reduce the cost of recovery, impact on database engine upgrade, and peak-to-peak performance after a machine restart.

7.4 Experiment 3: Update-heavy CDB, Log Throughput

The goal of this experiment was to evaluate the ability to sustain high update throughputs. Specifically, this experiment studied the logging throughput of Socrates and HADR using a special workload mix of CDB that produces the maximum amount of log data. In this experiment, the log is the bottleneck for both HADR and Socrates and the performance of the system is determined by the logging bandwidth. For these experiments, we used VMs with 16 cores and 256 client threads to generate the transactional workload to make sure that indeed the logging component is saturated.

	SF	Log MB/s	CPU %
HADR	30000	56.9	46.2
Socrates	30000	89.8	73.2

Table 5: CDB Log Throughput: HADR vs. Socrates

Table 5 shows that Socrates beats HADR in this experiment. The low CPU utilization indicates that in both systems indeed the logging component is the bottleneck.

Why is Socrates better in this experiment? What determines the logging bandwidth? To answer these questions, we need to look at the entire logging pipeline. HADR needs to drive log and database backup from the Compute nodes in parallel with the user workload. Log production is restricted to the level at which the log backup egress can be safely handled by the Azure Storage tier (XStore) underneath. In contrast, Socrates can leverage XStore's snapshot feature for backup which results in a much higher log production rate upstream at the Socrates Primary.

This result is a good example of how pushing storage functions (such as backup in this case) down into a dedicated storage tier is an extremely powerful concept. Such optimizations in lower tiers of the system can significantly impact the performance of customer-facing Compute nodes as there are complex performance dependencies even in a loosely coupled system like Socrates.

8 CONCLUSION

This paper presented Socrates, a new architecture for DBaaS offerings in the cloud. Socrates powers Microsoft's new SQL database service in Azure, called SQL DB Hyperscale. Socrates relies on the well established principle of separating *Compute* and *Storage* to achieve better availability and elasticity. Furthermore, Socrates separates *durability* and *availability*. This approach is novel and has not been studied in the literature before. The big advantage of this separation is that it allows to flexibly meet customer requirements regarding the cost / performance / availability tradeoff.

Socrates is a novel architecture, and we are still in the early days of understanding and exploiting its full potential. We are currently implementing bulk operations in parallel in Socrates Page Servers. Further avenues for future work include exploring multi-master variants for Socrates, better HTAP support in Socrates, and making use of the log for other services such as audit and security.

Acknowledgments. Socrates and SQL DB Hyperscale is the result of a large collaborative effort across the Microsoft SQL organization and Microsoft Research. We would like to thank Elias Yousefi Amin Abadi, Ruokun An, Wayne Chen, Christian Damianidis, Paul Larson, Justin Lewandoski, Alexandru Nedelcu, Shweta Raje, Brendan Rowan, Swati Roy, Sridharan Sakthivelu (Intel) and Weidan Yan for their invaluable contributions to the design and implementation of Socrates.

REFERENCES

[1] 2018. CDB/DTU benchmark. https://docs.microsoft.com/en-us/azure/sql-database/sql-database-service-tiers-dtu.
[2] 2018. Microsoft SQL Hyperscale. https://docs.microsoft.com/en-us/azure/sql-database/sql-database-service-tier-hyperscale.
[3] 2018. Oracle Cloud Database. https://cloud.oracle.com/database/.
[4] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *SIGMOD 1995*.
[5] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. 2011. Hyder - A Transactional Record Manager for Shared Flash. In *CIDR 2011*.
[6] Dhruba Borthakur. 2017. The Birth of RocksDB-Cloud. http://rocksdb.blogspot.com/2017/05/the-birth-of-rocksdb-cloud.html.
[7] Peter Braam, Sean Roberts, Matthew O'Keefe, and David Bonnie. 2016. The Limits of Open Source in Extreme-scale Storage Systems Design. https://docplayer.net/62056362-The-limits-of-open-source-in-extreme\-scale-storage-systems-design.html.
[8] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. 2008. Building a Database on S3. In *SIGMOD 2008*.
[9] Alain Bui and Hacène Fouchal (Eds.). 2002. *OPODIS 2002*. Studia Informatica Universalis, Vol. 3.
[10] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. 2011. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *SOSP 2011*.
[11] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. 2012. Spanner: Google's Globally-Distributed Database. In *OSDI 2012*.
[12] Michael J. Franklin, Björn Þór Jónsson, and Donald Kossmann. 1996. Performance Tradeoffs for Client-Server Query Processing. In *SIGMOD 1996*.
[13] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis E. Shasha. 1996. The Dangers of Replication and a Solution. In *SIGMOD 1996*.
[14] Jim Gray and Andreas Reuter. 1990. *Transaction Processing: Concepts and Techniques*.
[15] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB* 5, 4 (2011).
[16] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. 2015. On the Design and Scalability of Distributed Shared-Data Databases. In *SIGMOD 2015*.
[17] David B. Lomet, Alan Fekete, Gerhard Weikum, and Michael J. Zwilling. 2009. Unbundling Transaction Services in the Cloud. In *CIDR 2009*.
[18] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *TODS* 17, 1 (1992).
[19] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-Structured File System. *TOCS* 10, 1 (1992).
[20] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD 2017*.

	STDEV	Min (us)	Median (us)	Max (us)
XIO	431	2518	3300	36864
DD	167	484	800	39857

Table 6: CDB UpdateLite Latency: XIO vs. DD

	Threads	Log MB/s	CPU %
XIO	128	69	30
DD	16	70	9

Table 7: CDB UpdateLite Log Throughput: XIO vs. DD

A EXPERIMENT 4: TEST CLUSTER, XIO VS. DIRECTDRIVE

The last set of experiments studied the impact of the storage service used to implement the LZ. Azure constantly improves its storage services and Socrates allows SQL DB Hyperscale to take advantage of these innovations.

For this experiment, we ran Socrates in a Test cluster in two different configurations with identical hardware and running the same workload on the same database. The only difference was the implementation of the LZ: We compared an implementation based on XIO (as in all previous experiments) and another implementation based on a new Azure service called DirectDrive (DD for short). When we did these experiments, DD was also in preview. DD aggressively exploits new technology trends such as RDMA. DD uses Win32 and the Windows I/O stack. For this experiment, we used a CDB workload mix with mostly small updates and no read transactions. We reduced the number of clients and measured the *latency* of committing a transaction. In this workload the CPU utilization is low because the clients did not generate enough work to saturate it.

Table 6 shows that DD has significantly better min and median latency. Only for the max latency are XIO and DD the same.

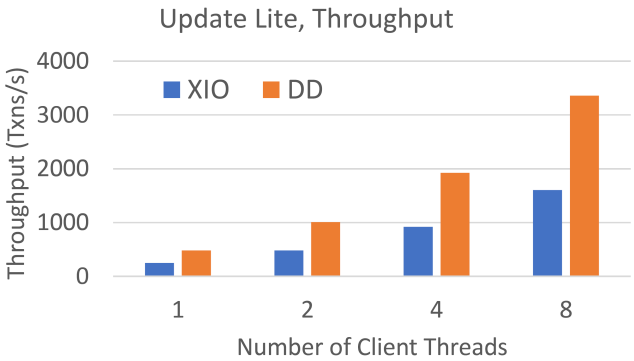

Figure 4: Socrates UpdateLite Throughput vs. Threads

The results of Table 6 were obtained with a single client thread. Figure 4 depicts the throughput with a growing number of concurrent client threads. Obviously, lower latency translates into higher throughput as long as the CPU of the Primary is under-utilized. This rule is workload-dependent and depends, in particular, on the read/write ratio. Nevertheless, DD is the clear winner indicating that Socrates benefits nicely from such innovations as DD.

Finally, Table 7 shows the CPU utilization. In this experiment, we varied the number of client threads such that Socrates with both XIO and DD has roughly the same log throughput of 70 MB per second. Table 7 shows that XIO needs 8x the load to achieve the same log throughput as DD, thereby consuming three times as much CPU in the Primary (and eight times as much CPU in the clients). This way, DD can significantly reduce the cost of Socrates databases as CPU dominates the cost of running a DBaaS offering. Here, one factor is that XIO requires expensive REST calls whereas DD requests go through cheaper Win32 calls. We hope that leveraging DD will also help Socrates to close the performance gap to SQL DB discussed in Experiment 1.

In summary, all these experiments make a simple point: Socrates can leverage storage innovations in ways that traditional, more monolithic DBaaS architectures cannot. These improvements were achieved without changing a single line of code. In the long run, we believe that these improvements outweigh the performance penalties due to reading data from remote servers.