A Progress Report on DBOS: A Database-oriented Operating System

Qian Li[1*], Peter Kraft[1*], Kostis Kaffes[1*], Athinagoras Skiadopoulos[1], Deeptaanshu Kumar[3], Jason Li[2], Michael Cafarella[2], Goetz Graefe[4], Jeremy Kepner[2], Christos Kozyrakis[1], Michael Stonebraker[2], Lalith Suresh[5], Matei Zaharia[1]
[1]Stanford, [2]MIT, [3]CMU, [4]Google, [5]VMware

Abstract

Over the last year, a group of us at MIT, Stanford, CMU, Google, and VMware have been designing and implementing a new Operating System (OS) stack for modern hyperscale datacenter environments. This new stack leverages a set of multi-core, multi-node distributed DBMSs near the bottom of the stack to manage a cluster of machines on a public or private cloud. In this paper, we briefly review the rationale for a new OS, present our resulting architecture, and review our progress to date. The meat of our paper is a presentation of the main lessons thus far from this project. Many of these have to do with missing capabilities in multi-node DBMSs that form the guts of our proposal. Finally, we present future research directions in database-oriented operating systems.

1 Why a New OS?

Current OSs like Linux are based on the UNIX design from the 1970s. Back then, hardware resources consisted of a uniprocessor with limited main memory and disk. In the intervening 40 years, resources under OS management have increased by five or six orders of magnitude. The MIT Supercloud [5], for example, has approximately 10,000 cores and a hundred terabytes of main memory in total. OS state (files, tasks, messages, etc.) has increased in size by the same scale factor. Hence, today's massive scale systems are very different from what Linux was designed for.

Over the past year, we have worked on designing a new OS stack for distributed environments called DBOS. We chose to base our stack on DBMS technology as it is known to provide the scale, performance, and reliability required in large-scale systems. For example, increasing network speeds have made fast disaggregated memory a reality [16, 29] as specialized machines host large memory pools used by multiple users and applications. Traditional OSes offer little support for these pools of shared memory, but battle-tested relational DB semantics are a good way to expose them: the DBMS engine can use the underlying hardware to achieve high performance and scalability while offering a better and more intuitive programming model for users. Furthermore, monitoring and debugging large distributed clusters with OSs like UNIX/Linux is notoriously difficult; DBOS aims to simplify these operations greatly using DBMS-based logging and provenance tracking.

Using a DBMS offers many advantages on the software engineering side as well. Many system software experts have come to realize that transactions and high availability are a good idea [11, 14, 30]. Such constructs come built into modern DBMSs. If a DBMS is central to the OS stack, these features can be implemented once and then used by everybody. More importantly, making a DBMS central to the OS stack allows novel capabilities to be easily implemented. Our community has been building cloud systems and implementing new features by layering software either below existing OSes (e.g., hypervisors [3, 10]) or above them (e.g., Borg and Kubernetes [12]). These designs are hard to implement and maintain because developers need to take careful account of interactions between various layers belonging to different stakeholders with potentially conflicting goals. They also often introduce limitations due to incompatibility. By contrast, our DBOS architecture is easy to modify and new cross-stack features can readily be added. For example, we will show how to implement object-level provenance across all layers of the stack.

These issues have motivated us to build a new OS, whose design we briefly summarize in the next section. We have oriented our design toward cloud computing and our first goal is to support a cluster of cloud machines. We discuss our motivation and vision in more detail in [13] and [32].

2 The DBOS Stack

Figure 1 shows the DBOS stack pictorially. At the top level are standard applications that run protected from the rest of the stack as in current systems. One level down are OS utilities such as `ls`, `chdir`, etc. that are currently supported by traditional code in C or C++. In our proposal, these are almost entirely written as stored procedures and user-defined functions, mostly in SQL. OS services such as the file system, interprocess messaging, and scheduling are similarly supported in SQL. At level 2 is a logically centralized, physically distributed polystore system. The polystore system consists of high-performance DBMSs with characteristics suitable for different use cases (e.g., OLTP and OLAP) that all implement SQL. For example, DBOS stores frequently updated system state in a high-performance distributed OLTP DBMS, while processes historical data in a columnar OLAP DBMS for faster analytical queries. The polystore interface provides a universal common SQL interface so that the complexity is transparent to upper levels. Finally, at level 1 is a microkernel with the minimal required facilities needed to run the DBMS. Mostly, this is raw device support, interrupt handlers, and basic communication between nodes.

DBOS centralizes system state and user data in a uniform data model as database tables and executes all operations on state as DBMS transactions, invoked from otherwise stateless processes.

*These authors contributed equally to this paper.

 12th Annual Conference on Innovative Data Systems Research (CIDR '22). January 9-12, 2022, Chaminade, USA.

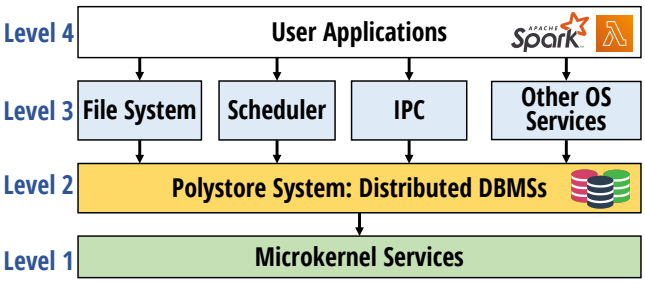


Figure 1: The DBOS stack.

While the idea of managing state in a DBMS has been applied in specific contexts like file system metadata access [11, 27] and cluster managers [33], we propose a new extreme. We believe that all levels of the system stack, from high-level applications down to core services like schedulers, file systems, and monitoring, should manage their state centrally in a distributed transactional DBMS.

This proposal requires a drastic rethinking of how to build a cloud operating system and the applications that run on top of it. A developer must choose schemas, indexes, appropriate transaction isolation levels, and partition keys for tables that store state and implement DBMS stored procedures and SQL queries over these tables for application workflows, analysis, and reporting.

3 Progress To Date

In the past year, we have written a first DBOS prototype entirely in user-space code, running on the MIT Supercloud and in Google Cloud, using VoltDB [9] as the OLTP DBMS in level 2 of our stack (Figure 1). At level 3, our prototype includes a scheduler, a file system, and a messaging system, all implemented in SQL. We have also written a few Linux utilities at level 3, such as `ls` and `ls -r` (list all files and folders in a directory recursively), to demonstrate their performance on DBOS. Our VLDB paper [32] discusses preliminary results from this prototype.

Recently, we have added a serverless environment to support end-to-end applications at level 4, which is described in Section 4. We have also implemented a provenance system that can capture usage information from all layers of the DBOS prototype, also described in Section 4. This required adding a data warehouse DBMS in level 2 of our stack, making DBOS a polystore system.

4 Lessons from the First Year

In this section, we present a collection of observations and lessons from our experience so far. In general, we are very optimistic about the potential and success of database-oriented operating systems.

4.1 Newer SQL DBMSs are fast enough for DBOS

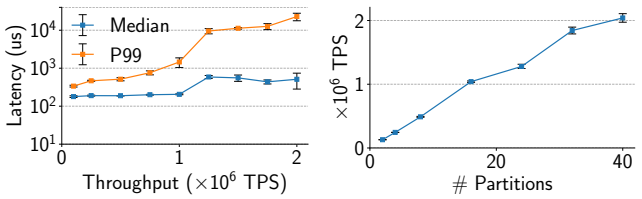
Early on, we made the decision that absolutely **everything** goes in the DBMS. There was grumbling that there must be things that cannot be made fast enough or cannot be general enough. So far, we have found no insurmountable obstacles. In part, this is because fast manipulation of OS state is basically an OLTP problem. VoltDB is ideally suited for this problem, as it is architected for very high performance on short OLTP tasks. There was also grumbling that VoltDB limitations (we present in following sections) would doom the project. However, the feeling among the team is that competitive performance demands something as fast as VoltDB, and we would not trade its limitations for significantly poorer performance.

To demonstrate the performance of VoltDB, in Figure 2, we present the latency and scalability of a FIFO scheduler implemented as a stored procedure:

```
select ID, NumTasks from Worker
  where NumTasks < MaxCap limit 1;
if ID not None:
  update Worker set NumTasks = NumTasks + 1
    where ID = ID;
```

This stored procedure first queries the DBMS for a worker that can run an additional task. If one is found, it increments `NumTasks` for that worker. The scheduler executes the procedure on a randomly chosen partition, iterating until it succeeds. Our prototype is synthetic: it schedules tasks but does not execute them, thus we measure only the scheduling latency. It assumes that all tasks are identical and that any worker can execute a fixed number of tasks (its maximum capacity) simultaneously.

We conducted two experiments: 1) Measure the median and tail scheduling latencies as we vary the system load, and 2) Measure the maximum throughput with an increasing number of VoltDB partitions. Our experiments used forty parallel schedulers on two VoltDB servers and two client machines to generate load, all on MIT Supercloud [5]. All clients and servers have 40-core dual-socket Intel® Xeon® Gold 6248 2.5GHz CPUs, and Mellanox ConnectX-4 25Gbps NIC.



(a) Latencies v.s. load **(b) Throughput v.s. #partitions**

Figure 2: Performance of the distributed FIFO scheduler.

We first measured how latency changes as we increase throughput with a fixed number of table partitions. As we show in Figure 2a, with forty table partitions on two servers we can schedule as many as one million tasks per second with sub-millisecond tail latency, and as many as two million tasks per second with sub-millisecond median latency. Therefore, our simple two-server system could saturate more than 200K cores with 100 ms tasks (a normal estimate for serverless task durations [31]) without incurring a scheduling overhead of more than 1%, easily outperforming most existing distributed schedulers (e.g., a Spark [37] cluster with tens of thousands of nodes can only launch a thousand tasks per second).

We next measured how maximum throughput changes as we increase the number of table partitions. As we show in Figure 2b, throughput scales near-linearly with the number of parallel partitions. This indicates that we can support higher scheduling throughput by adding partitions and database servers to the system.

4.2 Making everything a stored procedure works well

In VoltDB, conversational SQL is compiled into a "one-shot" stored procedure and all recurring tasks are stored procedures. We have decided to implement all of level 3 in DBOS as stored procedures. This means that OS functionality such as file system and scheduling operations are implemented as stored procedures, while user applications may be composed of many stored procedures. As described in the next section, users can implement complex workflows by declaring directed acyclic graphs of stored procedures in our SE. For example, a shopping cart application can have different stored procedures for placing the order, billing, and shipping in a manner similar to a microservices environment. So far, we have not found any show-stoppers to this approach. Stored procedures naturally provide encapsulation and isolation. Making everything a stored procedure dramatically simplifies DBOS, since VoltDB manages the stored procedure library and executes all stored procedures transactionally, simplifying task execution and failure recovery.

The only drawback of the stored-procedure-centric approach is that application developers are constrained by the programming model of the underlying DBMS. For example, VoltDB (and many DBMSs) only supports stored procedures written in Java. Thus, users currently need to implement the non-SQL components of their applications in Java. This constraint makes it hard to use popular frameworks and libraries better suited for other languages such as Tensorflow and PyTorch. We do not view the programming model as a fundamental limitation as we aspire to provide bindings that will allow users to invoke non-Java code from the VoltDB JVM.

4.3 Our serverless environment is surprisingly fast

An important part of the future of cloud programming is clearly serverless environments (SE) such as AWS Lambda [1] or OpenWhisk [2]. Using SE, one divides a computation into subtasks that form an execution graph. These subtasks are executed when ready and consume resources only for the duration of the subtask. In DBOS, we have written our own SE, where subtasks are stored procedures in VoltDB. In other words, we leverage the DBMS wherever possible, and we are committed to "eating our own dog food." Our SE can be faster than both Amazon Lambda and OpenWhisk on data-centric tasks as we can co-locate compute and data, avoiding unnecessary data movements.

The serverless model is a great match for DBOS's characteristics as it allows us to make the architectural decision to avoid sophisticated memory management. A subtask requests its maximum memory requirement at its start, and the stored procedure is not executed until sufficient real memory is available. When a stored procedure finishes, all memory is released. This decision simplifies memory management and task scheduling complexity. As expected, the graph of subtasks for a specific computation is also stored in the database.

4.4 SQL is highly advantageous when requirements change

We have changed the DBOS schema multiple times either because of poor performance or because of new requirements. In every case, recoding the SQL in stored procedures and changing the stored tables was easy and quick. This should be contrasted with traditional systems, where change is difficult, tedious, and error-prone. For instance, in DBOS, new data fields can be easily added by extending the table schema without updating myriads of data structures and interfaces in the code. One of the unforeseen benefits of DBOS is the programmer productivity of SQL in both system evolution and initial system design. For example, as also discussed in [33], making even a small change to the widely used Spark or Kubernetes schedulers is a Herculean task due to ad-hoc data structures and complex code. On the contrary, we were able to implement least-loaded and locality-aware schedulers simply by changing a couple of lines of SQL code [32], dramatically lowering the barrier for policy exploration and experimentation.

4.5 Do everything just once

Modern distributed DBMSs support serializable transactions, replication, and failover to a backup when a failure occurs. Our design puts all OS state into tables, so all OS facilities can use database transaction and high availability features. As such transactions and high availability can be implemented just once inside the DBMS and then used by all OS services as well as user-level tasks. This should be contrasted with current systems that must implement and re-implement such features in various subsystems as needed, often relying on ad-hoc solutions.

4.6 Work around the limitations of the DBMS

When we began the project, there was considerable discussion of the limitations of VoltDB, our chosen OLTP DBMS, along with suggestions that we abandon our "everything in the DBMS" mantra. One issue was the lack of triggers in VoltDB. Since interprocess communication entails the sender adding a row to a `Message` table and the receiver reading and then deleting the row, our implementation suffers from the absence of triggers. However, even when replacing triggers with receiver polling, IPC is still competitively fast compared to widely used gRPC [32].

DBOS also suffers from some aspects of the VoltDB stored procedure (SP) model. In VoltDB, stored procedures are executed as a single transaction within a single partition that runs to completion. This means SPs are isolated from each other, and there is no concurrency, which may cause head-of-line blocking if an SP occupies a partition for a long time.

In addition, it is effectively impossible for one stored procedure to call another one as a subroutine because the outer SP will stall the VoltDB partition which it is associated with while the inner SP runs, presumably on another partition.

Moreover, there are no nested transactions in VoltDB, so appropriate transactional behavior of nested SPs is not supported. As a result, one important programming tool (subroutines) is not available to DBOS. However, while an SP cannot be invoked from an SP, it is possible to invoke an SP that takes in the output of other SPs as its input. Therefore, DBOS provides a programming model where users submit graphs of subtasks and each subtask is executed on its parents' outputs, an interface common in distributed systems [17, 18, 26].

4.7 DBOS had to become a polystore

A few months ago we realized that storing all OS state in tables would allow a powerful and sophisticated provenance system to be constructed. All we needed was to capture all changes to system

tables in a log (also a DB table) and then support SQL provenance queries to the log table.

In theory, this is straightforward; however, a historical provenance database is gigantic and ill-suited to a main memory OLTP DBMS such as VoltDB. Obviously, the requirements for high performance modification of system tables are very different than supporting historical provenance queries. As a result, we added a parallel data warehouse DBMS to level 2 (Figure 1) to support provenance tables. In our case, we chose to run Vertica [8], though other parallel column stores could also be used. Hence, we needed to capture all writes and optionally all reads in VoltDB and spool them transactionally to Vertica. This makes DBOS a polystore system, and our future improvements are discussed in Section 4.8.

Our preliminary experiments [20] show that capturing all object level reads and writes and streaming them into Vertica does not impact system performance until the transaction rate gets high (greater than 50K transactions per second). Even at higher levels, performance degradation is quite modest.

As noted in [20], we described a collection of 10 provenance queries that capture many tasks that security analysts wish to do. These are all readily coded in SQL and execute in small numbers of seconds on very large provenance data tables.

Many of our desired provenance queries require transitive closure. One such query is to find all possible data leakage paths from a given user to somebody else. However, transitive closure is not supported in Vertica, so we had to code the iteration manually (e.g., set the recursion depth in Vertica). Our industrial partners indicate that approximate transitive closure is usually good enough. For example, most data leakage paths are of length one (from perpetrator directly to accomplice). The full transitive closure is not required in most circumstances, and indefinite iteration can be avoided.

We plan to test whether a graph DBMS (e.g. Neo4J, RedisGraph) would be beneficial in provenance queries. A similar exercise five years ago yielded negative results, but hope springs eternal.

4.8 Better polystore support would be very helpful

DBOS needs an OLTP DBMS for storing OS state, a warehouse DBMS for provenance, and spooling data between the two. Although VoltDB supports "change data capture", that only deals with writes, and a different mechanism is needed to capture reads. Also, there are multiple mechanisms to do spooling (JDBC, bulk loader, Kafka, ...). In general, implementing provenance has been a "heavy lift" task. DBOS would appreciate much better polystore support in commercial DBMSs.

As mentioned in [20], we envision a general polystore system in a future DBOS system. In real production applications, users may want to store their data in diverse types of data stores, and execute provenance in different DBMSs. What we desire is automatic object capture on reads and writes at various granularities (event, file, block, record), configurable on a table-by-table basis, and cross-DBMS logging and query optimizations.

Figure 3 demonstrates how a social network application could potentially use such a polystore system. Both application data and system state would be stored in DBOS. First, the online query path is handled by the OLTP DBMS, while read/write provenance capture happened asynchronously in the background, exporting user data to an OLAP DBMS and social graph interactions to a graph DBMS.

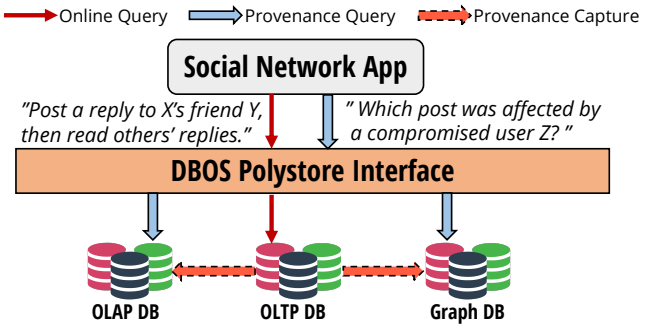


Figure 3: A use case for a general polystore system.

Then, a monitoring system may issue a provenance query to find potentially compromised posts, which is handled jointly by the OLAP and graph DBMSs. Note that the DBOS polystore interface would hide all the complexity from the user application.

4.9 Tuning a multi-core, multi-node DBMS is hard

Our team contains highly skilled programmers and substantial DBMS expertise. Even so, we needed help from the vendor for both VoltDB and Vertica to optimize performance in our environment. Both systems have a myriad of tuning knobs and exhibit non-intuitive behavior, and we have spent many, many hours trying to improve performance and scalability. Moreover, sometimes the issue was caused by machine configuration that was indirectly connected to DBMSs, which makes tuning even harder. In Section 4.13 we have two examples for our tuning experience. As a result, we are a huge fan of self-tuning efforts, such as [35, 38] to make this activity easier.

4.10 DBOS would probably benefit from a different tradeoff concerning multi-partition transactions

The VoltDB architecture makes single-partition transactions blindingly fast, but at the cost of slow multi-partition transactions. DBOS benefits greatly from fast single-partition transactions, but may sometimes require multi-partition transactions; for example, sending a message to multiple receivers or making a scheduling decision based on the state of multiple partitions. We believe that if schemas are designed carefully, 99-99.9% of DBOS transactions will be single-partition, but the slowness of the remainder is problematic.

The most important issue with VoltDB multi-partition transactions is that they obtain a global DBMS lock blocking all partitions, not just those needed for the transaction. As a result, a small percentage of multi-partition transactions can dramatically reduce cluster performance; in one of our experiments 0.1% multi-partition transactions decreased overall throughput by 50%. We would happily accept a concurrency control scheme that reduced single-partition performance by 10-20% but eliminated this global lock. In our opinion, exploring the tradeoff between multi-partition and single partition performance in distributed databases would be a worthwhile research topic.

4.11 Multi-tenant support would be very, very helpful

Obviously, the world is moving toward cloud deployment. Guaranteeing security and isolation is essential in a multi-tenant cloud

environment. Modern DBMSs have already provided some levels of support for security and integrity constraints. For example, authentication to invoke a stored procedure is supported in VoltDB.

However, DBOS has issues that require even stronger multi-tenant support as we need to run untrusted SPs submitted by multiple users. For example, in VoltDB a stored procedure is linked into the VoltDB runtime. As such, a malicious SP can potentially hack into the execution engine, access its entire memory footprint, and decode data records for which it may not be authorized. If SE users share a single DBMS instance, then serious multi-tenancy support will be required.

4.12 Auto-scaling would be very, very helpful

Although VoltDB supports elastic scaling to resize a cluster, it has no capabilities to automatically adjust the size of a cluster as requirements change, and resizing may affect the performance of ongoing transactions. In DBOS, we fully expect the resources devoted to our DBMSs to require automatic adjustment as the load changes or the mix between provenance queries and operations changes. Snowflake [36] has led the way with dynamic adjustment of resources, and everybody is going to need such capabilities.

Furthermore, VoltDB partitions data by the partitioning column value, which may cause hotspot issue if a few partition values are more frequently visited. Since many real-world OLTP workloads are skewed and highly variable [34], we expect the data (re-)partitioning and placement to be handled automatically/dynamically by the DBMS for better load balancing.

4.13 Scale matters

One lesson we learned early on is that testing DBOS at small scale is very different from testing at large scale. Invariably, things that work fine in the small fail for unforeseen reasons at scale. For example, when we tested on a single machine, VoltDB can support millions of transactions per second (TPS). However, we once noticed that VoltDB stopped scaling above 200K TPS while being accessed remotely in a cluster. This was caused by an underlying physical machine configuration that restricted network processing to a single core. Similarly, we had a Vertica scaling issue which was caused by a single "planned concurrency" configuration parameter. The benchmark had decent performance on a single node, but only revealed the issue while testing at a large cluster. Hence, it is crucial to design and test for scale.

Our experiments have so far assumed that the OLTP state of the system and the applications fits in main memory. However, this might not be the case for specific workloads, e.g., video analytics, or at very large scale. Larger-than-memory working sets would currently be handled by using traditional OS mechanisms such as paging. In the future, we plan to handle this case ourselves by spilling state pro-actively to disk and making more informed decisions about which data to evict.

5 The Future of DBOS

In this section we present several research problems we are presently working on.

5.1 Pervasive monitoring

Traditional systems software uses a piecemeal approach to logging and monitoring where products like Splunk [7] and Prometheus [6] capture different information. In DBOS, however, object-level provenance automatically captures OS state, unifying the efforts of these disparate systems and allowing DBOS to act as a single monitor of OS state. We envision using machine learning (ML) to identify undesirable OS states and to identify corrective actions. We expect to have a complete system along these lines in 2022. The same underlying mechanisms will be used for application level monitoring.

5.2 Heterogeneous hardware support

Modern datacenters use heterogeneous hardware such as GPUs, TPUs, FPGAs, as well as multiple types of memory and storage. Currently, OSs and distributed platforms have limited, if any, support for such devices. Therefore, a computation has to specifically invoke such hardware, after figuring out the availability and how to move data in and out. DBOS can do better than that by exploiting the potential opportunity of hardware fungibility, i.e., leveraging heterogeneous hardware to run the same program based on cost/performance trade-off [28] or data location. Since DBOS stores all state in the DBMS, it can effectively make the optimal decision.

At the present time, all stored procedures execute on CPU hardware; however, this is not an architectural requirement. We envision multiple kinds of SPs, one per kind of specialized processor. Of course, this would require substantial extensions to the SP model. Specifically, code in languages other than Java would be required. In addition, the SP would have to be decoupled from a VoltDB partition to avoid blocking the partition. More elegantly, VoltDB could be made to understand partitions that were not associated with stored data. We expect to figure out the best way to support SPs on non-CPU devices.

5.3 Security

Guaranteeing security is essential in a multi-tenant cloud environment. Fortunately, modern DBMSs provide strong support for security and integrity constraints, and are therefore a natural point to enforce them. For example, authentication for the entire system can be done at once using DBMS facilities. Different protection domains, such as file protection, can be implemented using database views [15, 24]; each user or entity acts on a restricted view of the state tables. A centralized extensible security system such as this will hopefully be better at avoiding configuration errors and leaks than the sprawl of configuration tools used today [21]. Moreover, the abundance of structured monitoring data available in DBOS will facilitate modern analytics-based security approaches [4].

As discussed in Section 4.11, DBOS also requires a strong security and isolation guarantee since it needs to run untrusted programs from multiple tenants. We are evaluating different sandboxing mechanisms to improve security of underlying DBMSs.

5.4 Self-adaptivity

"Everything in the DBMS" and "everything is a stored procedure" unlock opportunities for DBOS to be self-adaptive. In particular, our method makes integrating modern machine learning (ML) and reinforcement learning (RL) techniques into a distributed system/application easier as the required data is likely available in the DBMS along with built-in support for the necessary graph analytics and machine learning algorithms. Learning parameters can be more effective than using heuristic [22]; for example, DBMS knobs can be

automatically tuned as workloads change [35, 38]. RL can even be used to learn a system component in DBOS such as a scheduler [23], index structures [19], or device placement policy [25].

6 Conclusion

In this paper, we presented our progress on DBOS, a database-oriented datacenter operating system. We presented the design of DBOS, current progress, lessons learned from the past year, and new research directions both for implementing systems software on top of a DBMS and for improving DBMSs to better support this systems software use case. We believe that database-oriented designs like DBOS will make cloud applications and systems easier to build, maintain, extend, and scale.

References

[1] 2021. Amazon Lambda. https://aws.amazon.com/lambda/.
[2] 2021. Apache OpenWhisk. https://openwhisk.apache.org/.
[3] 2021. gVisor. https://gvisor.dev/.
[4] 2021. Lacework. https://www.lacework.com/.
[5] 2021. MIT Supercloud. https://supercloud.mit.edu/.
[6] 2021. Prometheus - Monitoring system & time series database. https://prometheus.io/.
[7] 2021. Splunk. https://www.splunk.com/.
[8] 2021. Vertica. https://www.vertica.com/.
[9] 2021. VoltDB. https://www.voltdb.com/.
[10] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache
[11] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. 2019. File Systems Unfit as Distributed Storage Backends: Lessons from 10 Years of Ceph Evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 353–369. https://doi.org/10.1145/3341301.3359656
[12] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade. *Queue* 14, 1 (2016), 70–93.
[13] Michael Cafarella, David DeWitt, Vijay Gadepally, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, and Matei Zaharia. 2020. DBOS: A Proposal for a Data-Centric Operating System. arXiv:2007.11112 [cs.OS]
[14] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.
[15] D. E. Denning, S. G. Akl, M. Heckman, T. F. Lunt, M. Morgenstern, P. G. Neumann, and R. R. Schell. 1987. Views for Multilevel Database Security. *IEEE Transactions on Software Engineering* SE-13, 2 (1987), 129–140. https://doi.org/10.1109/TSE.1987.232889
[16] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 401–414. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi{ć}
[17] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 475–488. https://www.usenix.org/conference/atc19/presentation/fouladi
[18] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. 2018. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 213–231. https://www.usenix.org/conference/osdi18/presentation/gjengset
[19] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*. 489–504.
[20] Deeptaanshu Kumar, Qian Li, Jason Li, Peter Kraft, Athinagoras Skiadopoulos, Lalith Suresh, Michael Cafarella, and Michael Stonebraker. 2021. Data Governance in a Database Operating System (DBOS). *Poly'21 workshop* (2021).
[21] T. F. Lunt, D. E. Denning, R. R. Schell, M. Heckman, and W. R. Shockley. 1990. The SeaView security model. *IEEE Transactions on Software Engineering* 16, 6 (1990), 593–607. https://doi.org/10.1109/32.55088
[22] Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, ravichandra addanki, Mehrdad Khani Shirkoohi, Songtao He, Vikram Nathan, Frank Cangialosi, Shaileshh Venkatakrishnan, Wei-Hung Weng, Song Han, Tim Kraska, and Dr.Mohammad Alizadeh. 2019. Park: An Open Platform for Learning-Augmented Computer Systems. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc., 2494–2506. https://proceedings.neurips.cc/paper/2019/file/f69e505b08403ad2298b9f262659929a-Paper.pdf
[23] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*. 270–288.
[24] Alana Marzoev, Lara Timbó Araújo, Malte Schwarzkopf, Samyukta Yagati, Eddie Kohler, Robert Morris, M Frans Kaashoek, and Sam Madden. 2019. Towards multiverse databases. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 88–95.
[25] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. 2018. Hierarchical planning for device placement. (2018).
[26] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. 2011. CIEL: A Universal Execution Engine for Distributed Data-Flow Computing. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/nsdi11/ciel-universal-execution-engine-distributed-data-flow-computing
[27] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. 2017. HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 89–104. https://www.usenix.org/conference/fast17/technical-sessions/presentation/niazi
[28] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 397–411.
[29] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 315–332. https://www.usenix.org/conference/osdi20/presentation/ruan
[30] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, 351–364. http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf
[31] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad
[32] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael Cafarella, Vijay Gadepally, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. 2022. DBOS: A DBMS-oriented Operating System. *To appear at VLDB'22* (2022).
[33] Lalith Suresh, João Loff, Faria Kalim, Sangeetha Abdu Jyothi, Nina Narodytska, Leonid Ryzhyk, Sahan Gamage, Brian Oki, Pranshu Jain, and Michael Gasch. 2020. Building Scalable and Flexible Cluster Managers Using Declarative Programming. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 827–844. https://www.usenix.org/conference/osdi20/presentation/suresh
[34] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment* 8, 3 (2014), 245–256.
[35] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Bilien, and Andrew Pavlo. 2021. An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. *Proceedings of the VLDB Endowment* 14, 7 (2021), 1241–1253.
[36] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 449–462. https://www.usenix.org/conference/nsdi20/presentation/vuppalapati

[37] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX, San Jose, CA, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[38] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*. 415–432.