Interactive Analytic DBMSs: Breaching the Scalability Wall

Pedro Pedreira, Amit Dutta, Sergey Pershin, Lin Liu, Sushant Shringarpure
Jialiang Tan, Brian Landers, Ge Gao and Karen Pieper
Facebook Inc.
1 Hacker Way
Menlo Park, CA
{pedroerp, adutta, spershin, codeshower, sushants, jtan6, blanders, gaoge, kpieper}@fb.com

***Abstract*—Analytic DBMSs optimized for query interactivity commonly push the computation down to storage nodes, thus avoiding large network transfers and keeping query execution wall-time to a minimum. In these systems, data is sharded and stored locally by cluster nodes, which must all participate in query execution. As the system scales-out, hardware failures and other non-deterministic sources of tail latency start to dominate, to a point where query latency and success ratio increasingly violate the system's SLA. We refer to this tipping point as the system's *scalability wall*, when sharding data between more nodes only worsens the problem.**

This paper describes how an analytic DBMS optimized for low-latency queries can breach the scalability wall by sharding different tables to different subsets of cluster nodes — a strategy we call *partial sharding* — and reduce the query fan-out. Because partial sharding requires the DBMS to implement many tedious and complex shard management tasks, such as shard mapping, load balancing and fault tolerance, this paper describes how a database system can leverage an external general-purpose shard management service for such tasks. We present a case study based on *Cubrick*, an in-memory analytic DBMS developed at Facebook, highlighting the integration points with a shard management framework called Shard Manager. Finally, we describe the many design decisions, pitfalls and lessons learned during this process, which eventually allowed Cubrick to scale to thousands of nodes.

***Index Terms*—Interactive DBMS, analytics, sharding.**

I. Introduction

A common architectural trend when designing analytical DBMSs and query engines is to decouple compute and storage. This strategy makes database systems easier to scale since compute and storage can be scaled independently, in addition to providing a cleaner separation of responsibilities between these two components. Considering that query engines and storage systems commonly use similar interfaces and connectors, it also favors interoperability and makes these systems more interchangeable and flexible. For example, a particular query engine may be able to integrate with multiple storage systems with different storage characteristics.

When considering *interactive* analytic engines, however, where low query latency is paramount and every millisecond during query execution counts, the most common strategy is to push the computation closer to the data and use the same set of servers for both compute and storage. Although less flexible and harder to scale than decoupled systems, these *tightly coupled* architectures commonly cut down query stalls caused by network transfers, overall increasing data locality and reducing query processing latency. Despite recent improvements in networks, smart caching and push-down strategies, tightly coupled system are still the predominant architecture for latency sensitive analytic workloads [9] [20] [13] [1].

Even though some of the state-of-art DBMSs for interactive analytics are single-node and can only scale vertically [13] [23], analytic DBMSs traditionally scale-out by horizontally sharding tables [15] [6]. In these systems, tables are commonly sharded between all nodes to better utilize the cluster's resources, and queries are broadcast so that all available shards are visited. Each node eventually returns a partial result, which are merged and materialized on a query coordinator node. This strategy works reasonably well for a small number of well behaved hosts, but quickly deteriorates as the system scales-out and tables are sharded between more nodes. The probability of hardware failures and other non-deterministic errors affecting query execution increases as more hosts need to be visited by a query.

Although many techniques have been discussed in the literature to amortize the effect of tail latencies [8], as the system scales-out there is a tipping point where the query success ratio drops below the system's SLAs; at that point, adding more hosts and sharding the dataset even further only worsens the problem. In this paper, we refer to this situation as the system's **scalability wall**. Figure 1 illustrates query success ratio as more nodes need to be visited to complete a query. Assuming that servers have a 0.01% chance of failure at any given time, a system with 99% query success SLA will hit the scalability wall at about 100 servers.

When hitting the scalability wall, a viable strategy to continue scaling-out the system is to break the full fan-out model by *partially sharding* data, where tables and queries are contained to subsets of cluster nodes. This model does not target use cases comprised of a single massively large table, but it is compelling for multi-tenant systems storing a large number of small and medium sized tables. It prevents broadcasts at query time, allowing the system to smoothly scale-out by adding more nodes, which will eventually store partitions/shards of a subset of database tables.

The partial sharding mode allows database systems to scale

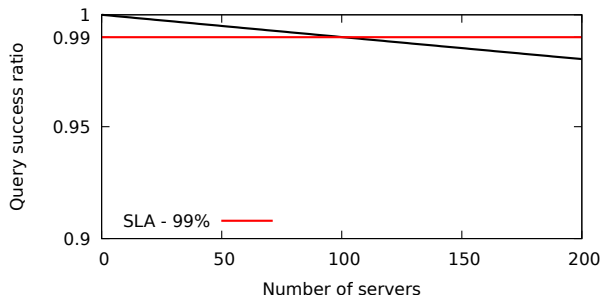


Fig. 1: Theoretical query success ratio as more nodes need to be visited to complete a query, assuming that servers have a 0.01% chance of failure at any given time, and a system with 99% query success SLA.

while pushing compute closer to the data, but it raises many other design questions. For instance, a system needs to decide how tables are mapped to shards, how shards are mapped to servers and how many shards each table should span. It also brings some systems and operational considerations, such as load balancing between shards, shard replication, shard migration between servers, failure models and machine automation. Besides being a tedious effort that can take years to be completed, building a full-fledged shard management system for a DBMS is error-prone, promotes code duplication and often generates suboptimal results [2]. Instead, DBMSs should be able to re-use external general-purpose shard management frameworks for these tasks, thus promoting consolidation, code de-duplication and reliability.

This paper describes how an interactive analytic DBMS can partially shard tables in order to breach the scalability wall. A case study is described in the context of the Cubrick database system [22], an in-memory analytic DBMS optimized for low-latency interactive queries, developed from the ground up at Facebook. Cubrick was adapted to leverage SM (Shard Manager, a sharding-as-a-service framework that powers a multitude of production services at Facebook [10]) for all shard managements tasks, break the full fan-out model and partially shard tables. We highlight the different trade-offs, design choices, pitfalls and lessons learned during this process, which eventually allowed Cubrick to scale to thousands of nodes.

In this paper we make the following contributions:

- We present and characterize a common scalability limitation present in many modern interactive analytic DBMSs.
- We describe how partial sharding can be leveraged to overcome this limitation, discussing the many architectural design trade-offs regarding mapping tables to shards and servers.
- We discuss how complex and error-prone shard management tasks can be decoupled from the DBMS code and leverage an external general-purpose shard management framework, in addition to detailing the API and integration points between these systems.
- We describe a case study based on Cubrick, an in-memory analytic DBMS optimized for interactive queries, and discuss how it was able to leverage an external general-purpose shard management framework, breach the scalability wall and scale to thousands of servers.
- We present operational stats about the current production system and experimental results comparing tables with different fan-out modes.
- We highlight some of the lessons learned, pitfalls and different iterations the system went through, stressing the bottlenecks and design changes.

The remainder of this paper is organized as follows. Section II characterizes the *scalability wall*, a common scalability limitation on many analytic DBMSs. Section III discusses database sharding concepts, in addition to describing the architecture and key features of Facebook's shard management framework, Shard Manager (SM). Section IV presents a case study based on the Cubrick DBMS stressing the design decisions, while Section V discusses some of the lessons learned during this process. Finally, Section VI points to related work and Section VII concludes this paper.

II. Scalability Wall

A. Coupled and Decoupled Systems

There are two main categories of analytic database systems in terms of **compute** and **storage**. In one hand, *tightly coupled* systems use cluster nodes for both storage and compute, and therefore all nodes have some type of local storage and can participate in query execution [9] [20] [13] [1] [22]. On the other hand, *decoupled* systems usually have a pool of worker nodes that are used for query execution, while data is transferred at query time from a remote storage system [24] [14] [25] [3].

Even though many trade-offs need to be to considered when comparing these two architectures, in general, decoupled systems are easier to scale (considering that compute and storage can be scaled and provisioned independently), while tightly coupled systems provide lower query latency since data is already stored locally where the computation happens. A common pattern observed in production services is that decoupled systems (sometimes referred to as *disaggregated* or only *disagg*) are usually bound by network I/O, or sometimes CPU due to heavy compression techniques meant to reduce I/O. Tightly coupled systems, on the other hand, are usually bound by CPU used for actual query processing.

There are many strategies that can be explored with the goal of building systems that are both flexible and low latency. A common technique is to cache frequently accessed data blocks on compute nodes. In order to achieve satisfactory cache hit rates, however, there needs to be a degree of affinity between data blocks and computes nodes. Since the computation needs to happen in a specific host (the one that has affinity for a particular data block), we argue that, essentially, this strategy is equivalent to pushing compute to data — or rather pushing data to compute. Other techniques such as smart predicate push down to storage nodes, in addition to operations such as

aggregates, group bys and joins were also explored by some database systems [3].

Overall, although network improvements and the described techniques are bridging the gap between decoupled and tightly coupled systems, coupled systems are still the main architecture leveraged by analytic systems optimized for query interactivity [13] [9] [22]. Considering this work targets analytics DBMS optimized for interactive and low latency queries, the remainder of this paper focuses on tightly coupled systems.

B. Full Sharding

In the trivial case, tightly coupled systems are single-node and all storage and computation happens in the same server [13] [20]. Single-node systems can still *scale-up* (vertically) and leverage multicore intra-node parallelism, but there are strict physical limits as to how far these systems can scale. When scaling-out tightly coupled systems horizontally, however, there must be a strategy as to how compute and storage are distributed among nodes in a cluster. Since interactive analytic systems optimize for query latency, the predominant strategy is to distribute (horizontally partition, or to *shard*) table storage between all cluster nodes, in order to fully utilize the resources available [6] [15]. Compute is usually pushed down to where data is stored and only smaller partial results are transferred through the network, providing optimal parallelization from a resource utilization perspective, while minimizing the amount of data transferred. Most systems also provide ways to replicate (instead of horizontally partition) tables which are smaller and used more frequently between all clusters nodes, in order to speed up joins with larger distributed tables [6]. We refer to these systems as *fully-sharded*.

Considering that all nodes participate in query execution, fully-sharded system queries need to be broadcast. This strategy works well for a reasonably small number of well-behaved nodes, but reliability quickly deteriorates the more nodes are added. The larger the cluster size, the more likely queries are to be stalled by hardware or software failures, or to be delayed by network instabilities and other causes of tail latency [8]. For instance, assume that the probability of a server failure in a given instant is 0.01%. Figure 1 illustrated the query failure ratio as more nodes were added to a hypothetical cluster. We refer to the tipping point where query failure ratio falls below the system's SLA as the system's **scalability wall**, when adding more servers will make query success rates even lower.

Even though the probability of failure can vary greatly due to hardware reliability characteristics, all fully-sharded systems are bound to hit the scalability wall if enough scale is required. Figure 2 extends the presented model, illustrating the query success curves for larger cluster sizes, given different server failure probabilities.

C. Breaching the Wall

When hitting the scalability wall, there are two strategies that can be used in order to continue scaling a given system. An analytic system can trade query consistency and accuracy for scale, and assume that partial results from servers that do not answer in a given time interval are ignored [1]. This compromise might be acceptable for log analysis, monitoring and other workloads where accuracy is not fundamental, but there are many BI and data analytics workloads where this assumption cannot be made. In such cases, the only strategy left is to prevent tables from being sharded among all cluster nodes, and hence control the query fanout. We refer to the these systems as *partially-sharded*.

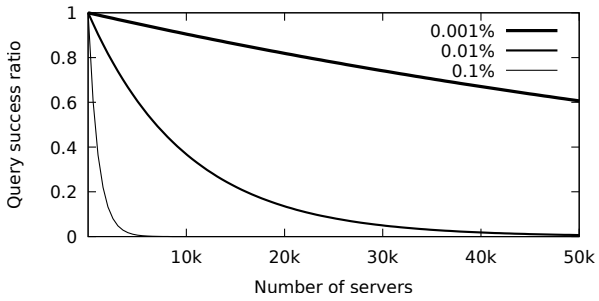


Fig. 2: Theoretical model of query success ratio considering servers with different chances of failure at any given time.

.

In partially sharded systems, each table is allocated to only a few shards, and the number of allocated shards is usually much lower than the total number of cluster nodes. Although this model is not applicable to use cases comprised of a single large table, it is sufficient for multi-tenant systems containing a large number of small and medium sized tables. Furthermore, multi-tenant systems already commonly pose some limitation to the maximum size of tables, in order to prevent single users or tables from monopolizing a large chunk of the cluster's capacity. Section IV-B presents an in-depth discussion about the number of shards assigned for each table.

Considering that all fully-sharded systems are bound to hit the scalability wall, we argue in this paper that all tightly coupled analytical systems *must* be partially-sharded in order to be *scalable*. However, there are many questions and design decisions that need to be considered when building a partially-sharded system. Some of the questions are:

- **Data distribution and location.** How are tables mapped to shards, and how are shards mapped to nodes? Over how many shards is a given table distributed? How is this number determined? How to locate the shards required by a particular query?
- **Load balancing.** How to prevent *hot* shards from being co-located within the same node? Which criteria to utilize when characterizing hot shards? How to move a shard between cluster nodes?
- **Fault tolerance.** How to monitor and detect if shards are alive? How to failover shards between cluster nodes? How to drain hosts?
- **Cluster resize.** How to add and remove cluster nodes on-the-fly, while ensuring the system is properly load balanced?

The remaining of this paper discusses possible answers for

these questions, and the design decisions taken by a real-life partially-sharded production DBMS system.

III. Sharding

Database sharding is a form of horizontal partitioning that splits table rows into smaller and more manageable segments. These segments (or shards) are stored by separate database instances, usually on different hosts in order to provide parallelism and scale-out the system.

There are many *application-specific* decisions that need to be made when designing a sharded database system. For example, deciding how tables and table rows are mapped to shards, and which shards need to be queried given a particular set of filters. However, there are also many *shard management* tasks that need to be performed on a real-life sharded system, such as shard heartbeats, load balancing, shard migrations and shard failover. These operations are complex and can be error-prone if implemented on a per-application basis. To address this issue, a few libraries that abstract shard management tasks were developed in the last decade, leaving applications free to concentrate on business logic and application-specific tasks [2] [12].

In the next sections, we describe Facebook's implementation of a shard management framework, called Shard Manager (SM), leveraged by many infrastructure services at Facebook. Due to space limitations, we only describe the characteristics required to contextualize how a truly scalable in-memory analytics DBMS system can be built on top of such a shard management system; a complete description is outside the scope of this paper.

A. Shard Manager

Shard Manager (SM) [10] is a framework developed at Facebook that provides sharding-as-a-service to distributed applications. SM simplifies the development of distributed sharded applications by abstracting shard management tasks such as shard placement, migration and failover, load balancing, high availability, primary and secondary replica management, resource constraint checks and machine automation stack integration. Applications using SM need to (a) implement a partitioning scheme (mapping application keys to shards), (b) provide system metrics that will be used for load balancing, and (c) specify shard replication and placement configuration.

SM's architecture is comprised of the following components:

- **SM Server.** This is the central SM scheduler that collects shard metrics for all applications and makes shard placement decisions. The server also exposes APIs to allow users to register new applications or change the current configuration.
- **Application Server (AS).** These are the services written by users which, in fact, host the shards. An SM-specific library is linked to the service, providing endpoints that allow SM server to communicate with it, collect counters, add and drop shards.
- **SM Client.** A library used by Application Server clients to interact with Application Server. SM Client learns from a Service Discovery system where a particular shard is located, and dispatches requests to the appropriate servers.
- **Datastore.** Zookeeper[1] is used to store SM server's persistent state and collect heartbeats from Application Server libraries. If heartbeats stop, SM Server gets notified by zookeeper and a shard failover operation might be triggered.
- **Service Discovery System.** A Service Discovery system (also based in Zookeeper) is utilized to expose shard↔server mappings. Facebook's service discovery system is called *Services Management Configuration* (SMC). Since service discovery is heavily used by application clients and the number of clients can be large, SMC uses a multi-level data distribution tree to cache and propagate this data. However, this can add a small delay to how long it takes for client to learn about changes to shard assignment.

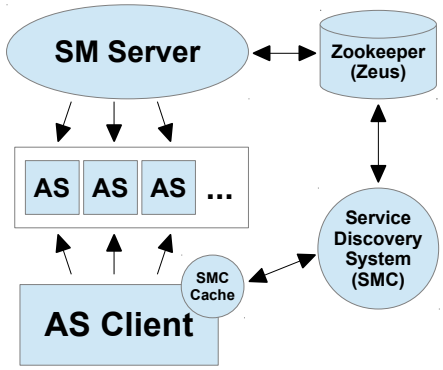


Fig. 3: Overall architecture of a service using SM.

The overall architecture is depicted in Figure 3. An SM Server monitors and coordinates multiple Application Servers (AS), and might instruct them to add or drop shards, according to load balancing rules or external factors (servers being drained, failures, etc). Application Servers are fully responsible for implementing the business logic of $addShard()$ and $dropShard()$ endpoints. On a stateful service, for instance, the $addShard()$ implementation would be responsible for discovering what data needs to be recovered, where to recover it from, and the actual recovery process that copies data and metadata to the new server. This workflow excludes SM Server from the data intensive path, making it more self-contained and easier to scale.

When required to interact with AS, AS clients need to provide a service name and a shard number to SM Client library. SM Client library will resolve the pair *(service, shard)* to a hostname by leveraging the service discovery system

[1]Facebook has its own implementation of a distributed coordination system, called Zeus, which is used in production. Zeus provides full Zookepeer API.

SMC. SMC is backed by Zookeeper and cached by a service running locally on every single server in the fleet, in order to avoid unnecessary network round-trips to reach to a shard.

1) Shard Replication: SM's fault tolerance model is based on shard replication. Shards in SM can have two different roles, *primary* and *secondary*, and at any given time there is at least one primary copy of a shard, and potentially multiple secondary ones. Application server developers can control the number of secondary replicas by specifying a *replication factor* configuration. Moreover, SM also let developers control how replicas of a shard need to be *spread*, and whether failure domains are composed of single servers, racks, or entire regions.

SM provides support for three different replication models:

- **Primary-only**. In primary-only mode, each shard has a single replica and therefore there is no redundancy. This is the case when replication factor is zero.
- **Primary-secondary**. In primary-secondary mode, there is a single primary replica for each shard, and multiple secondary. Primary replicas are commonly responsible for handling writes and coordinating data replication to secondary replicas. Optionally, read-only traffic can be served from secondary replicas.
- **Secondary-only**. Each shard has multiple replicas, and they all play the same role.

No matter the fault tolerance mode being used, it is important to notice that SM only controls shard roles and server assignments. The replication of the actual data stored within shards, as well as handling writes, dealing with conflicts, consensus and which type of traffic to direct to which replica, are all responsibilities of the application.

2) Shard Migration: Shard migration is the process of moving the responsibility over a particular shard from one server to another. SM server periodically monitors all shards and application servers, and might decide to trigger a shard migration based on a variety of reasons, such as load balancing constraints, servers being drained or failures.

There are two types of shard migration: *live* shard migrations and *failovers*. Live shard migrations are triggered when the server that currently hosts the shard is still healthy, commonly used by load balancing or when draining servers and racks. A failover is triggered when the old server is unavailable, usually on hardware failure scenarios.

Shard migrations are coordinated by SM server and executed using the $addShard()$ and $dropShard()$ endpoints implemented by application servers. There are a few workflows that can be used by SM when migrating shards, which depend on the fault tolerance mode being used and the number of primary and secondary replicas existent. For example, primary replica migration on a primary-secondary service works by disabling the primary, electing a secondary to be the new primary, then allocating a new secondary replica. SM also provides a more intricate migration protocol called *graceful shard migration* that allows primary shards to be migrated without any downtime (see Section IV-E). However, a thorough description of these different workflows is outside the scope of this paper.

3) Load Balancing: SM has two goals when distributing shards between servers: (a) to ensure that shards can only be assigned to servers that have enough capacity, and (b) to evenly spread the load between servers. Therefore, an important design decision is to select an appropriate metric to both describe a server's capacity, and the capacity required by a shard — its *size* or *weight*. Since different application running on SM may be throttled by different resources and have different load balancing objectives, SM makes a conscious design choice of decoupling measurement and management, allowing applications to provide their own custom metrics, like CPU, memory usage, disk capacity, queue sizes, or any other internal application counters. Based on the provided metrics, SM server is responsible for handling the load distribution logic, and orchestrating shard migration operations based on its internal load balancing algorithm.

There are a few other important features supported by SM regarding load balancing:

- **Support for asymmetric shards.** In order to support applications where shards may have different *sizes*, metrics collected need to be exported *per-shard*. Having data about the size of individual shards allows SM to take better informed load balancing decisions.
- **Support for dynamic shards**. Since shard sizes can change over time, SM server must periodically collect shard size metrics. If the metric being utilized has a spiky nature (such as CPU usage), it is the application's responsibility to smooth out bursts by using moving averages, for example.
- **Support for multiple load balancing metrics**. SM allows applications to export multiple metrics for load balancing, as long as there is some degree of correlation between the timeseries, and they do not create competing goals.
- **Heterogeneous servers**. Considering that in large clusters it is possible to find servers with different hardware configurations, SM allows application servers to export the total capacity for a particular host. For example, a cluster using memory usage as the load balancing metric might be composed of servers with different memory capacity. In addition, SM also allows application server to periodically export (and change) the current capacity of a host, which might be useful in some complex load balancing scenarios (see Section IV-F).
- **Throttling load balancing migrations**. Lastly, since shard migrations invariably cause some degree of overhead to the system, SM allows application owners to configure and throttle the maximum number of shard migrations allowed on a single load balancing run.

IV. Cubrick - A Case Study

Cubrick is an in-memory analytic DBMS developed from the ground up at Facebook, focused on low-latency OLAP queries to power dashboards and interactive data exploration

tools. Cubrick leverages a novel partitioning technique — called Granular Partitioning [21] — that provides fast and low overhead indexing abilities over multiple columns by range partitioning the dataset on every dimension column. An earlier version of Cubrick's internal partitioning technique was described in [22].

Currently, Cubrick's deployment at Facebook is comprised of thousands of datasets and millions of queries per day, backed by thousands of servers spanning multiple data centers. Despite being used only by internal analytic tools, tens of thousands of people use Cubrick every month.

In the early days, Cubrick was a *fully-sharded* system, in which every table was sharded among all hosts in the cluster. The more datasets (and consequently, servers) were added, the harder it got to maintain the same level of SLAs. Tail-latencies and increasing server failure rates all hurt the provided guarantees, and adding servers only made the problem worse. In other words, the system hit the **scalability wall**.

In order to overcome this issue, in a second generation multiple (but smaller) Cubrick clusters were deployed, one per customer, allowing the system to scale further and onboard new use cases. The more use cases — and therefore, clusters — were added, the harder management became. Deployment of new software versions, capacity provisioning per cluster and resource usage imbalance quickly became problematic.

To address both scalability and manageability concerns, the current generation of Cubrick employs a *partial-sharding* model, where each table is comprised of a few shards, depending on its size, and the SM framework is used for shard management. Decoupling shard management from Cubrick's main business logic removed substantial complexity from the service, which allowed the team to focus on DBMS development (instead of distributed systems problems), in addition to making the service more flexible and reliable. It allowed Cubrick to incorporate dynamic sharding and load balancing, made the system able to adapt to workload changes, and automatically integrated with data center automation tools. Finally, it made sharding easier to monitor and maintain — because it is broadly used at Facebook, SM has full-fledged management consoles and monitoring dashboards.

The next section describes how Cubrick's data model was mapped and adapted to leverage SM's abstractions, some of the pitfalls and lessons learned.

A. DBMS Sharding

Similarly to other distributed DBMSs, Cubrick segments each table into multiple horizontal partitions. The assignment of records to partitions may be done according to some deterministic function or randomly (refer to [22] for more details). Each table partition is mapped to a shard, and each shard ultimately gets mapped to a physical server by SM. When mapping records to partitions, the goal is to minimize the skew between partitions of the same table, so that at query time each server has, on average, the same amount or work to perform. Let's assume a hypothetical table dim_users, containing 4 partitions. Internally, we refer to each partition of this table as: $dim\_users\#0$, $dim\_users\#1$, $dim\_users\#2$ and $dim\_users\#3$ (# is a special character and thus not allowed as part of table names).

SM provides a flat key space for shards — from $[0..maxShards)$ — , where $maxShards$ is configurable. A usual deployment utilizes between 100k and 1M total shards. Furthermore, there must be a function to provide the mapping of application keys (tables names and partitions) to SM shards. Since the number of shards is fixed for a particular service, Cubrick leverages a simple $hash(tbl) \% maxShards$ function to map table partitions to SM shards. In case changing the maximum number of shards had to be supported, a consistent hashing function could have been used instead.

In the example above, the following mapping would have been applied (assuming 100k total shards):

table name	shard
$dim\_users\#0$	15863
$dim\_users\#1$	11617
$dim\_users\#2$	45311
$dim\_users\#3$	20163

1) Collisions: One inherent issue when mapping an arbitrarily large application key space (table names) to a finite number of shards, and a large number of shards to a smaller number of servers, is handling different types of collisions. There are two main types of collisions to avoid: (a) *partition collisions* and (b) *shard collisions*.

Partition collisions. Partition collisions, or partitions from different tables mapped to the same shard, are expected and unavoidable, and only dictate that a particular set of table partitions will always be stored within the same host. If SM decides that a particular shard needs to be migrated, all table partitions mapped to that shard need to be shipped together to the new server. The problem with the naive hashing approach described above is being susceptible to collisions within the same table. For example:

table name	shard
$test\_table\#0$	**25140**
$test\_table\#1$	28396
$test\_table\#2$	**25140**
$test\_table\#3$	37422

in which case, the server to host shard 25140 would always have to perform twice as much work as other partitions, and hence increase query latency. To overcome this problem, Cubrick's current shard mapping function hashes only partition zero, and monotonically increments the remaining partitions. For instance:

table name	shard
$test\_table\#0$	25140
$test\_table\#1$	25141
$test\_table\#2$	25142
$test\_table\#3$	25143

This mapping method prevents collisions within the same table as long as tables have at most $numShards$ number

of partitions, which is always the case in our production deployments.

Shard collisions. A different type of collision happens when different shard containing partitions of the same table are mapped to the same host by SM. Although having the same effect as partition collisions (query latency increases since some servers might need to be perform twice as much work), shard collisions can be resolved by just moving one of the shards to another server.

There are two aspects worth noting about shard collisions. First, if a shard migration is requested by SM and Cubrick detects it will cause a shard collision, *i.e.* the target server already stores a shard that contains a partition of one of the tables within the shard being migrated, Cubrick server throws a *non-retryable* exception. A non-retryable exception alerts SM server that the application server cannot take this particular shard, and that it should try migrating it somewhere else. This approach, however, does not prevent collisions at table creation time, when shards are already allocated and the new table ends up containing shard collisions.

Second, when there is a shard collisions and multiple partitions of the same table are stored within the same server (say, $s1$), $s1$ will use about twice as much resources to execute the same workload. Query latency will increase since $s1$ needs to scan twice as much data; aggregation, group by and join buffers will be larger resulting in higher overall memory usage. Considering that SM server periodically collects resource usage counters per shards, eventually it will request the migration of one of these shards to a new server to smooth out resource usage between servers (load balancing is described in more details in Section IV-F).

Figure 4a illustrates the number of shard and partition collisions found in the current Cubrick deployment. About 7% of tables have shard collision, where different shards containing different partitions of the same table are assigned to the same host by SM; about 3% have partition collision of different tables, where different partitions of different tables are assigned to the same shard by SM; and no tables have partition collision on the same table, which is prevented by design by the shard mapping function described above.

Other approaches. Finally, an alternate approach explored by the team when mapping table partitions to shards (which is in fact used internally by other systems inside Facebook [1]) is to map table partitions to shard *replicas*. Using that strategy, each table is mapped to a single shard, and different table partitions are stored within shard secondary replicas. Even though this approach provides the compelling property of avoiding shard collisions (since SM will never assign different shard replicas to the same host), there are some key limitations. First, in this strategy all tables need to have the exact same number of partitions — limited to the cluster's *replication factor* —, making it harder to deal with table size variability. Second, there is an implicit assumption that shard replicas store, in fact, copies of the exact same data. This strategy would break this invariant, and potentially make it harder to integrate and leverage other SM features in the future.

B. Partitions per Table

From a resource scheduling perspective, having shards with similar sizes facilitates the shard allocation process, and contributes to a more even load balance. In real production workloads though, tables can have very different sizes, and even tables with similar sizes can have very disparate query workload characteristics. In the extreme case, one single shard can never exceed the amount of resources available on a single host. In order to ensure a single shard does not grow too large, and that shards on average have similar sizes, Cubrick adopts a dynamic table partitioning model, where the number of partitions created for a particular table (and hence, the number of shards) changes according to the table size.

Considering that at table creation time the DMBS has little context on how large the table will grow, we found that a good starting point is to use 8 partitions for every newly created table. It provides a good balance between giving tables enough space so that re-partitions are not triggered too frequently, and allowing even small tables to leverage parallel CPU power of 8 servers for query processing.

When a single table partition exceeds a certain size threshold, a *re-partition* operation is triggered. Re-partition operations essentially change the number of partitions a table has (say, from 8 to 16), and allows for a finer control over table partition size; if a partition gets too large, create more partitions; if they get too small, collapse the data into fewer partitions. However, table re-partitions are computationally expensive operations that require data shuffling of part of the table, so its usage must be sporadic.

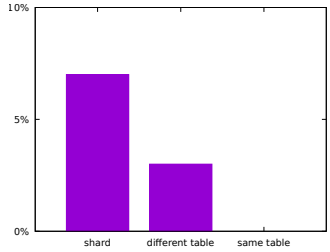
Figure 4b illustrates the number of partitions per table in Cubrick's current deployment. The vast majority of tables in the system are composed of 8 partitions, since they never hit the size threshold in which they are re-partitioned. For the larger tables that are re-partitioned (about 10%), the maximum number of table partitions for a single table is about 60[2].

C. Locating a Table

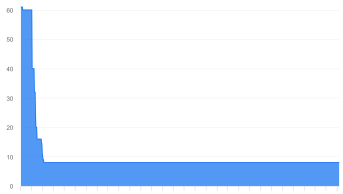
Having tables with different number of partitions brings the following load balancing question: when interacting with a table, which partition should a client connect to? In Cubrick, queries are invariably executed by the hosts that store partitions of a table, always pushing the compute closer to the data. The host that receives the client connection is called a *query coordinator*. A query coordinator is required to run on a host that stores one partition of the target table (or one of the target tables, in case of joins or sub-queries). A query coordinator has additional responsibilities, if compared to other query workers, such as merging partial results, query parsing, compilation and distribution, and coordination of multi-step queries. For an ideal load distribution, query coordinators must be evenly balanced between table partitions.

The following strategies were implemented and used in production before landing on the current approach:

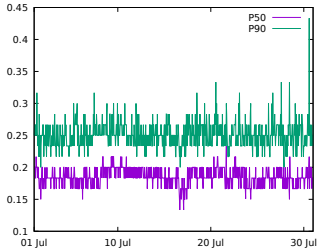
[2]There is no explicit limit on the maximum number of partitions for a given table, but there is a limit on the maximum dataset size that can be currently loaded in Cubrick, which is about 1TB.



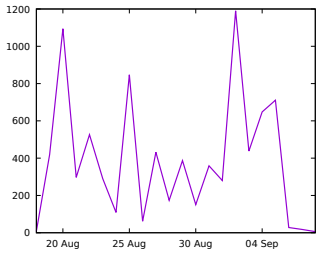
(a) Frequency of different types of shard collisions.



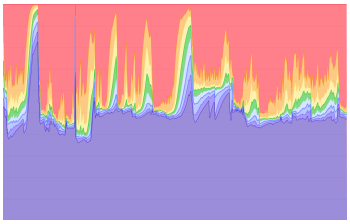
(b) Distribution of number of partitions per table on Cubrick's current production deployment.



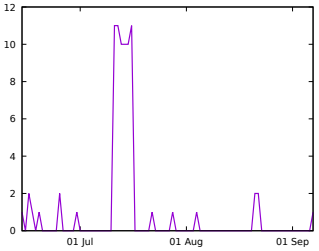
(c) Service discovery system's local proxies propagation delay (in secs).



(d) Number of shard migrations per day.



(e) Distribution of data blocks based on their hot (red) and cold (blue) counters in a production deployment over a week.



(f) Number of hosts sent to repair per day (permanent host failures).

Fig. 4: Operational stats extracted from the current Cubrick deployments.

1) *Always forward queries to partition 0*. In this strategy, clients always append #0 to the table name, apply the mapping function to get a shard number, and use SM client to connect to the appropriate host. This strategy causes resource usage imbalance since the same host is always used as query coordinator.
2) *Forward to partition 0, and from there forward to another random partition*. This strategy balances query coordinators between partitions, but requires an additional network hop. Additional network hops are particularly bad when retrieving large buffers since it requires one extra network transfer.
3) *First retrieve the current number of available partitions, then randomly forward the connection to one of them*. This strategy perfectly balances query coordinators and avoids extra network transfers of query buffers, but requires an extra roundtrip before initiating the query.
4) *Cache number of partitions per table, then forward to a random one*. This is the current strategy used in production deployments.

Cubrick queries are always submitted to a query proxy (see Section IV-D), which handles automatic retries, admission control, blacklisting, logging and other proxy functions. The proxy is responsible for keeping a cache containing the current number of partitions per table, and randomizing the target partition. To avoid extra roundtrips and keep the cache up-to-date, the number of partitions per table is always included as part of query results metadata, and updates the proxy's cache.

D. Fault Tolerance

Cubrick's current production deployment is composed of three regions, each containing a full copy of all tables and partitions. The number of regions was chosen so that even in the event of disasters or codes pushes (when one entire region might be down) there is still redundancy and at least two other regions to load balance queries and support the incoming traffic.

Queries are always submitted to a Cubrick proxy service, which is a stateless service also running in the same three regions. Cubrick proxy is responsible for handling all user queries and deciding which is the most suitable region to dispatch a query to. This decision is based on region availability (since entire regions might be down or drained), and proximity to client in order to reduce latency. Proxies are also responsible for retrying queries which failed due to some types of errors, such as a hardware failures during query execution or target table/partition corrupted. In these cases, the query is transparently retried on a different region and users are unaware of the failure. Finally, the proxy is also responsible for a list of features such as admission control, blacklisting, logging and query tracing.

Once a query is dispatched to be executed in a certain region, all table partitions required by the query are required to be available within that region — there is no cross-region traffic during query execution. If some partition is unavailable, queries will fail and be retried on a different region by Cubrick proxy. Failovers – or shard migrations when the old server is unavailable – are automatically handled by downloading a copy of the failed shard from a healthy region.

Conceptually, the SM fault tolerance mode used by Cubrick

is **secondary-only** (since all shard replicas play the same role), **replication factor** is two (since there are three copies of each shard), and **spread** is defined as a region (since shard replicas cannot be placed in the same region). Even though this model is supported by SM (as described in Section III-A1), for operational simplicity and flexibility Cubrick is currently deployed as three independent *primary-only* services.

E. Shard Migration

One of the benefits when integrating with a shard management framework is the ability to leverage *dynamic sharding*, where shards can be transparently migrated from one server to another. There are many scenarios that might trigger a shard migration, for example: (a) load balancing, (b) server failures (c) data center automation tools or (d) service admin manual intervention. All the aforementioned cases are automatically handled by SM servers (or by tools integrated with SM servers), and will eventually generate calls to the only two endpoints that need to be implemented by application services: $addShard()$ and $dropShard()$.

There are a few steps executed by a Cubrick server on a shard migration: (a) identifying all table partitions that map to the shard being migrated, (b) creating the appropriate table and shard metadata on the new host, and (c) moving the actual table partition data. On a live shard migration, the data is directly copied from the old healthy server, whereas on a failover, data and metadata are copied from a healthy server in a different region (see Section IV-D).

Graceful shard migration. In order to support shard migration without any downtime, SM provides a slightly different workflow called *graceful shard migration* (as described in Section III-A2). The endpoints called by SM to migrate shard $s1$ from *oldServer* to *newServer* are the following:

- **prepareAddShard(s1)**: SM informs *newServer* to prepare for taking over $s1$. From this point onwards, SM server expects *newServer* to be able to answer requests for $s1$, only if they are being forwarded by *oldServer*.
  - Cubrick copies all data and metadata stored within $s1$ from *oldServer* to *newServer*.
- **prepareDropShard(s1)**: SM informs *oldServer* to start forwarding all requests related to $s1$ to *newServer*.
- **addShard(s1)**: SM informs *newServer* that it is now effectively responsible for $s1$. *newServer* starts handling requests related to $s1$ from all sources.
- At this point, SM server instructs the service discovery system (SMC) that $newServer$ is now responsible for shard $s1$. As described in Section III-A, it might take a few seconds until this information is fully propagated to all clients. Figure 4c illustrates the usual propagation delay observed in production clusters.
- **dropShard(s1)**: SM informs *oldServer* to drop all data and metadata related to $s1$.
  - Cubrick waits for a pre-defined number of seconds (SMC's usual propagation delay), and finally, when the number of requests per second to $s1$ in $oldServer$ drops to zero, all data and metadata are deleted.

Naturally, the graceful protocol is only used on live shard migrations; failovers are translated to a single $addShard()$ call in the target server, considering that *oldServer* is down and unavailable to forward requests during the protocol execution. Although hardware and other non-deterministic failures can cause unavailability for a few tables in one single region, these errors are automatically retried by Cubrick proxy on a different regions in a user-transparent manner (as described in Section IV-D). Figure 4d illustrates the number of shard migrations executed daily on a production Cubrick cluster.

F. Load Balancing

As described in Section III-A3, SM provides a flexible way to configure load balancing as long as application owners provide the correct per-shard metrics and server capacity. Over the years, Cubrick went through a few changes regarding how data and shards are stored, which required changes to the metrics reported to SM server. The three different generations are described in the next Subsections.

1) First Generation: As an in-memory analytic DBMS, Cubrick's first generation had an implicit assumption that all data would be available in main memory before query execution. Therefore, the server capacity metric exported to SM was set to the amount of physical memory available on the host — or rather, to 90% of the available memory to save memory for kernel and other basic services running on every host. In addition, the size reported for each shard was set to the sum of the memory footprint of all table partitions inside that shard.

Even though only accounting for table size and excluding metrics such as QPS and CPU usage from load balancing decisions, this approach worked reasonably well in our production deployments for about a year.

2) Second Generation: The first Cubrick change that required adjustments to the metrics used for load balancing was a feature called *adaptive compression*. With adaptive compression, Cubrick maintains *hotness counters* for each data block in the system (also called *brick* in Cubrick terminology), that are incremented once they are needed by a query, and slowly and stochastically decay over time if not used[3]. Considering that access patterns between data blocks are usually skewed (for instance, more recently loaded data is usually queried more frequently than old data), this strategy provides a clear separation between frequently (hot) and seldomly (cold) used data blocks. Figure 4e illustrates the distribution of hot/cold data blocks in a current production deployment.

When there is memory pressure, *i.e.* the host is running low on free memory, a memory monitor procedure is triggered and incrementally compresses data blocks based on their *hotness counter* (from coldest to hottest), eventually freeing up some memory. In the same way, if there is a surplus of available

[3]The strategy used to classify hot and cold data blocks was inspired by [16].

memory, compressed data blocks are uncompressed (from hottest to coldest), reducing the amount of compressed data, and likely the amount of decompressions at query time.

Adaptive compression provides many important benefits, such as only compressing when there is memory pressure, and thus minimizing the impact of decompressions on query runtime, but it breaks the load balancing strategy used in the first generation. Shard sizes (memory footprint) now depend on the current server's resource usage, and might shrink or expand if there is memory shortage or surplus. Therefore, considering that a shard's size can substantially (and non-deterministically) change once it is migrated from one server to another, load balancing becomes a challenging (if not impossible) task.

To overcome this issue, currently, instead of reporting the actual shard memory footprint, Cubrick reports the *decompressed* size per shard, which is the memory footprint this shard would incur if all data were to be decompressed. The motivation behind this change is providing a consistent metric to SM that does not change given the server's current resource usage, but only based on the actual shard size. Shard size can still change, however deterministically, if more data is added to a table partition. Lastly, the capacity of a server, as reported to SM, is now set to the current host's memory capacity multiplied by the average compression ratio observed in production deployments.

3) Third Generation: Currently, we are working on a third generation, which not only compresses, but also evicts data to SSD once enough memory pressure is given. This new model will likely require changes on which metrics are exported for load balancing, considering that now all data for a shard can be evicted and memory footprint could essentially be zero. The strategy currently being explored by the team is to export SSD available space as the hosts capacity, and SSD footprint per shard as shard size. However, this approach does not take into account that shards can have different working sets (sets of *hot* data blocks), and that query latency can deteriorate if a particular host does not have enough memory to keep the working sets of all shards in memory. Since in such cases the amount of I/O requests would increase, the team is also investigating adding number of IOPS as a load balancing metric.

Nonetheless, choosing the optimal load balancing metric for this scenario is still an open problem, and it has been actively investigated by the team. This is one of our focuses for future research.

G. Data Center Automation

One important aspect of large scale distributed systems is their integration with data center automation tools. Having automated workflows to handle different types of machine management requests is essential to avoid manual intervention, facilitate operations and contribute to self-driven cluster management. Other than handling hardware and other non-deterministic failures, which are more frequent the larger the cluster size, there are many other *planned* situations where servers might need to be drained and removed from production clusters. Server decommission events, rack and cluster physical movement, power, network and server maintenances, and disaster preparedness exercises are all examples of normal events that need to be automatically handled when operating large scale clusters.

Considering that SM is used by many services at Facebook, it provides out of the box integration with all data center automation systems. By providing a centralized control plane for all maintenance and machine management requests, SM is able to execute multiple safety checks and, for example, (a) control whether requests can be fulfilled without compromising the application's fault tolerance model, (b) check for conflicts with load balancing operations, and (c) ensure there will be enough capacity left to operate the cluster once the request is finished. Figure 4f illustrates the number of host failures that are automatically handled per day on Cubrick hosts by datacenter automation tools. All these workflows are integrated with SM and required no human intervention.

H. Fan-out Experiment

In order to measure the variability added by a larger number of hosts participating in query execution, we conducted an experiment where the same simple query was executed every 500ms for about one week in a production cluster, over tables with varying fan-out levels (resulting in more than 1M queries per table). The results are illustrated in Figure 5 (y-axis on a log scale), showing how, in practice, higher fan-out queries are more susceptible to non-deterministic sources of tail latencies.

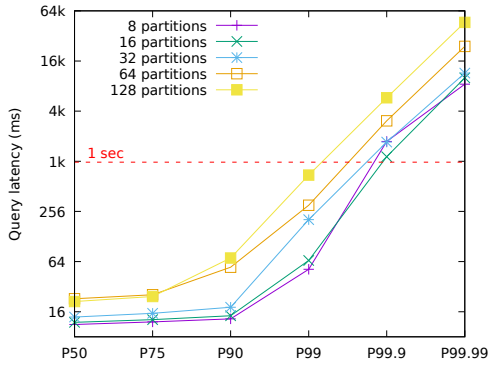


Fig. 5: Query latency for varying fan-out levels.

V. Lessons Learned

In this Section, we highlight some of the important lessons learned while developing Cubrick and integrating it with an external shard management framework, in addition to general lessons about operating a reliable system at scale.

A. Flexibility

Other than allowing the Cubrick team to remove large pieces of ad-hoc shard management code, the integration with an external shard management framework made the system more reliable and better coupled with other machine management

systems. However, the trade-off is *flexibility*. All workflow nuances need to be carefully mapped to what is supported by the underlying framework, and cases where these workflows are incompatible and the framework needs to be extended are much harder to handle, since they depend on agreements between different teams that might have different priorities.

From our experience developing and managing operations for large scale production clusters, the benefits of this integration largely outweigh the flexibility costs, nonetheless.

B. Load Balancing Metrics

Even though the pattern observed in most metrics change over time, considering that new service features are added, code paths are optimized and new workloads are onboarded into the system, there is usually one (or a few metrics) that invariably describe the load and health of a particular shard. The optimal metric commonly goes beyond simple CPU and memory usage counters, so it is important to actively monitor the system and ensure that the metrics that correctly describe server load are identified.

C. Note on Reliability

As systems scale-out, hardware failures and other forms of server unavailability become the norm rather than the exception. Therefore, work needs to be put in place to ensure that different failure modes are automated and do not require any manual intervention from system admins. Also, that automated tests are built to regularly exercise the failover code paths and avoid regressions.

However, there are also other types of larger scale failures that happen more sporadically, such as rack, main switch board or even full region/datacenter failures. These failures are usually a lot more disastrous to a service's availability. Other than taken into account when designing the service, these failure modes need to be documented and well understood. Since testing these failure modes is usually more complicated and harder to simulate with simple unit or integration tests, we found that by regularly simulating disasters scenarios, for instance, taking racks and even full regions offline deliberately, the different fail modes are better understood and tested, and overall more reliable.

Lastly, careful reviews of the dependencies of your system need to be conducted regularly, highlighting what their failure modes are, and what is the impact to your service health. Questions like *"can my service be available if dependency X is down?"*, or *"are there degraded modes my service can provide, to ensure my system remains available if service X is down?"*, need to be asked and discussed, in order to prevent large scale outages and domino effects, and expose possible circular dependencies.

In the case study presented in this paper, for example, the Cubrick service was consciously designed in such a way as to survive scenarios where Shard Manager is unavailable. If SM server is down, metrics won't be collected and no load balancing or shard migration decision will be made, but the Cubrick service is still available for loads and queries. Likewise, clients would still be able to resolve shard ids into hostnames since the mappings are propagated and cached locally by the service discovery system.

VI. Related Work

There are a variety of systems that make the architectural decision of decoupling compute and storage. Presto [24] is an open source query engine developed at Facebook that can execute SQL queries and act as a federation layer between different storage engines. Dremel [17] and Procella [5] are both distributed SQL query engines developed by Google that are able to process data stored in a distributed filesystem called Colossus. Snowflake [7] is a cloud-based database system that segregates compute and storage and provides data warehouse as a service to users. Redshift [11] is a popular database system from Amazon that tightly couples compute and storage, but also provides an external connector to data stored on Amazon S3. Lastly, there are many other traditional query engines that took similar trade-offs and decoupled compute and storage, such as Impala [14], developed by Cloudera, Spark SQL [3] and Hive [25], a query engine based on Hadoop map-reduce jobs.

On the other hand, there is a class of tightly coupled database systems that use the same set of hosts for both compute and storage, in order to reduce network traffic and query latency. Scuba [1] is a system focused on log analysis workloads developed at Facebook that fans-out queries to storage nodes, ignoring answers from dead or slow hosts, thus trading consistency for efficiency. SAP Hana [9] and MemSQL [6] are both relational distributed DBMSs that shard tables between all hosts in a cluster in order to scale-out. They both support two types of tables, one distributed among all nodes in the cluster, and one replicated to all nodes, used by smaller dimension tables frequently joined to larger tables. Vertica [15] is another example of distributed column-oriented DBMS that takes similar trade-offs. To the best of our knowledge, none of the described systems support the partial sharding model described in this paper.

There is also a class of tightly coupled analytic DBMSs optimized for low latency queries that do not provide scale-out capabilities (yet), such as Hyper [13] and Peloton [20]. DuckDB [23] is a new embedded analytic engine distributed as a library, with similar characteristics — tightly coupled, single node.

Other shard management libraries and frameworks have been proposed in the past. Slicer [2] is a general purpose sharding service developed by Google, which took similar design decisions and trade-offs as Shard Manager. Microsoft Orleans [4] is a framework for writing distributed applications, but it relies on consistent hashing for shard placement, limiting the system's flexibility. Azure Service Fabric (ASF) [18] is another framework developed by Microsoft, but supports a less flexible consistency model if compared to Shard Manager.

Lastly, Azure's SQL Elastic Database [19] enables users to partially shard databases on Azure's cloud. Although similar in many aspects to the work presented on this paper, Azure's

Elastic Database focuses on sharding full databases, instead of tables inside the database, like proposed in this work. In addition, it does not highlights a clear separation of responsibilities between shard management and application's business logic, or describes how load balancing and live shard migrations could be used on a real production database system.

VII. Conclusions

Interactive analytic DBMSs commonly leverage tightly coupled architectures, where cluster nodes are responsible for both storage and compute, to provide low query latency guarantees. This paper characterized a common scalability limitation on these systems, referred to as *scalability wall*, which most sharded analytic DBMSs are bound to hit when enough scale is required. A strategy that can overcome this problem, called partial-sharding, was presented, as well as discussions about the many design decisions that need to be considered when building a partially sharded system.

This paper also presented a case study based on Cubrick, a production interactive analytic DBMS developed at Facebook, and how partially sharding tables allowed it to scale to thousands of nodes, while still providing tight query latency and success SLAs. Cubrick leverages a shard management framework called Shard Manager, also developed at Facebook, to offload the responsibilities and complexity of shard management tasks.

Even though Cubrick has already been running in production for many years, there are still areas of active research. The main focus are on optimal load balancing metrics to better reflect host utilization, and prevention of shard collisions at table creation time.

References

[1] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, and et al. Scuba: Diving into data at Facebook. *Proc. VLDB Endow.*, 6(11), Aug. 2013.

[2] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, and et al. Slicer: Auto-sharding for datacenter applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI16, page 739753, USA, 2016. USENIX Association.

[3] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD 15, page 13831394, New York, NY, USA, 2015. Association for Computing Machinery.

[4] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: Cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC 11, New York, NY, USA, 2011. Association for Computing Machinery.

[5] B. Chattopadhyay, P. Dutta, W. Liu, O. Tinn, A. Mccormick, A. Mokashi, P. Harvey, H. Gonzalez, D. Lomax, S. Mittal, and et al. Procella: Unifying serving and analytical data at youtube. *Proc. VLDB Endow.*, 12(12):20222034, Aug. 2019.

[6] J. Chen, S. Jindel, R. Walzer, R. Sen, N. Jimsheleishvilli, and M. Andrews. The memsql query optimizer: A modern optimizer for real-time analytics in a distributed database. *Proc. VLDB Endow.*, 9(13):14011412, Sept. 2016.

[7] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, and et al. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD 16, page 215226, New York, NY, USA, 2016. Association for Computing Machinery.

[8] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[9] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. Sap hana database: Data management for modern business applications. *SIGMOD Rec.*, 40(4):4551, Jan. 2012.

[10] T. K. Gerald Guo. Scaling services with shard manager. https://engineering.fb.com/production-engineering/scaling-services-with-shard-manager/, 2020. Accessed: 2020-09-07.

[11] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD 15, page 19171923, New York, NY, USA, 2015. Association for Computing Machinery.

[12] G. Kakivaya, L. Xun, R. Hasha, S. B. Ahsan, T. Pfleiger, R. Sinha, A. Gupta, M. Tarta, M. Fussell, V. Modi, and et al. Service fabric: A distributed platform for building microservices in the cloud. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys 18, New York, NY, USA, 2018. Association for Computing Machinery.

[13] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE 11, page 195206, USA, 2011. IEEE Computer Society.

[14] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-milne, and M. Yoder. Impala: A modern, open-source sql engine for hadoop. In *In Proc. CIDR15*, 2015.

[15] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *Proc. VLDB Endow.*, 5(12):17901801, Aug. 2012.

[16] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann. Leanstore: In-memory data management beyond main memory. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 185–196, 2018.

[17] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3(12):330339, Sept. 2010.

[18] Microsoft. Azure service fabric documentation. https://docs.microsoft.com/en-us/azure/service-fabric/, 2020. Accessed: 2020-02-25.

[19] Microsoft. Scaling out with azure sql database. https://docs.microsoft.com/en-us/azure/sql-database/sql-database-elastic-scale-introduction, 2020. Accessed: 2020-02-25.

[20] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. Self-driving database management systems. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017.

[21] P. Pedreira. *On indexing highly dynamic multidimensional datasets for interactive analytics*. PhD thesis, Federal University of Paraná, Curitiba, Brazil, 2016.

[22] P. Pedreira, C. Croswhite, and L. C. E. D. Bona. Cubrick: Indexing millions of records per second for interactive analytics. *PVLDB*, 9(13):1305–1316, 2016.

[23] M. Raasveldt and H. Mühleisen. Duckdb: An embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD 19, page 19811984, New York, NY, USA, 2019. Association for Computing Machinery.

[24] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner. Presto: SQL on everything. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 1802–1813, 2019.

[25] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 996–1005, March 2010.